Assessing and Remedying Coverage for a Given Dataset

Abolfazl Asudeh, Zhongjun Jin, H. V. Jagadish
University of Michigan
{*asudeh, markjin, jag*}*@umich.edu*

***Abstract*—Data analysis impacts virtually every aspect of our society today. Often, this analysis is performed on an existing dataset, possibly collected through a process that the data scientists had limited control over. The existing data analyzed may not include the complete universe, but it is expected to cover the diversity of items in the universe. Lack of adequate coverage in the dataset can result in undesirable outcomes such as biased decisions and algorithmic racism, as well as creating vulnerabilities such as opening up room for adversarial attacks.**

In this paper, we assess the coverage of a given dataset over multiple categorical attributes. We first provide efficient techniques for traversing the combinatorial explosion of value combinations to *identify* any regions of attribute space not adequately covered by the data. Then, we determine the least amount of additional data that must be *obtained* to resolve this lack of adequate coverage. We confirm the value of our proposal through both theoretical analyses and comprehensive experiments on real data.

I. Introduction

In the current age of data science, it is commonplace to have a learning algorithm trained based on some dataset. This dataset could be collected prospectively, such as through a survey or a scientific experiment. In such a case, a data scientist may be able to specify requirements such as representation and coverage. However, more often than not, analyses are done with data that has been acquired independently, possibly through a process on which the data scientist has limited, or no, control. This is often called "found data" in the data science context. It is generally understood that the training dataset must be representative of the distribution from which the actual test/production data will be drawn. More recently, it has been recognized that it is not enough for the training data to be representative: it must include enough examples from less popular "categories", if these categories are to be handled well by the trained system. Perhaps the best known story underlining the importance of this inclusion is the case of the "google gorilla" [1]. An early image recognition algorithm released by Google had not been trained on enough dark-skinned faces. When presented with an image of a dark African American, the algorithm labeled her as a "gorilla". While Google very quickly patched the software as soon as the story broke, the question is what it could have done beforehand to avoid such a mistake in the first place.

The Google incident is not unique: there have been many other such incidents. For example, Nikon introduced a camera feature to detect whether humans in the image have their eyes open – to help avoid the all-too-common situation of the camera-subject blinking when the flash goes off resulting in an image with eyes closed. Paradoxically for a Japanese company, their training data did not include enough East Asians, so that the software classified many (naturally narrow) Asian eyes as closed even when they were open [2]. Similarly, HP webcams were not able to detect black faces [3] due to inadequate coverage in the training data [4].

The problem becomes critical when it comes to *data-driven algorithmic decision making*. For example, judges, probation and parole officers are increasingly using algorithms to assess a criminal defendant's likelihood to re-offend [5]. Consider a tool designed to help the judges in sentencing criminals by predicting how likely an individual is to re-offend. Such a tool can provide insightful signals for the judge and have the potential to make society safer. On the other hand, a wrong signal can have devastating effects on individuals' lives. So it is important to make sure that the tool is trained on data that includes adequate representation of individuals similar to each criminal that will be scored by it. In § V-B, we study a real dataset of criminals used for building such a tool, published by Propublica [5]. We shall show how inadequate representation might result, for example, in predicting every widowed Hispanic female as highly likely to re-offend.

While Google's resolution to the gorilla incident was to *"ban gorillas"* [6], a better solution is to ensure that the training data has enough entries in each category. Referring to the issue as "disparate predictive accuracy", [7] also highlights that the problem often is due to the insufficient or skewed sample sizes. If the only category of interest were race, as in (most of) the examples above, there are only a handful of categories and this problem is easy. However, in general, objects can have tens of attributes of interest, all of which could potentially be used to categorize the objects. For example, survey scientists use multiple demographical variables to characterize respondents, including race, sex, age, economic status, and geographic location. Whatever be the mode of data collection for the analysis task at hand, we must ensure that there are enough entries in the dataset for each object category. Drawing inspiration from the literature on diversity [8], we refer to this concept as *coverage*.

Note that the mentioned examples, including the Google incident, are surely not sampling cases where the data scientists poorly chose the samples from a large database. Rather, they somehow collected, or acquired, a dataset, and then failed to realize the lack of coverage for dark-skinned faces.

Lack of coverage in a dataset also opens up the room for adversarial attacks [9]. The goal in an adversarial attack is to generate examples that are misclassified by a trained model. Poorly covered regions in the training data provide the adversary with opportunities to create such examples.

Our goal in this paper is two-fold. First, we would like to help the dataset users to be able to assess the coverage, as a characterization, of a given dataset, in order to understand such vulnerabilities. For example, we propose to use information about lack of coverage as a widget in the nutritional label [10] of a dataset. Once the lack of coverage has been identified, next we would like to help data owners improve coverage by identifying the smallest number of additional data points needed to hit all the "large uncovered spaces".

Given multiple attributes, each with multiple possible values, we have a combinatorial number of possible *patterns*, as we call combinations of values for some or all attributes. Depending on the size and skew in the dataset, the coverage of the patterns will vary. Given a dataset, our first problem is to efficiently identify patterns that do not have sufficient coverage (the learned model may perform poorly in portions of the attribute space corresponding to these patterns of attribute values). It is straightforward to do this using space and time proportional to the total number of possible patterns. Often, the number of patterns with insufficient coverage may be far fewer. In this paper, we develop techniques, inspired from set enumeration [11] and association rule mining (*apriori*) [12], to make this determination efficient. We shall further discuss this and the related work in § VI.

A more interesting question for the dataset owners is what they can do about lack of coverage. Given a list of patterns with insufficient coverage, they may try to fix these, for example by acquiring additional data. In the ideal case, they will be able to acquire enough additional data to get sufficient coverage for all patterns. However, acquiring data has costs, for data collection, integration, transformation, storage, etc. Given the combinatorial number of patterns, it may just not be feasible to cover all of them in practice. Therefore, we may seek to make sure that we have adequate coverage for at least any pattern of ℓ attributes, where we call ℓ the *maximum covered level*. Alternatively, we could identify important pattern combinations by means of a *value count*, indicating how many combinations of attribute values match that pattern. Hence, our goal becomes to determine the patterns for the minimum number of items we must add to the dataset to reach a desired maximum covered level or to cover all patterns with at least a specified minimum value count. Since a single item could contribute to the coverage of multiple patterns, we shall show that this problem translates to a hitting set [13] instance. Given the combinatorial number of possible value combinations, the direct implementation of hitting set techniques can be very expensive. We present an approximate solution technique that can cheaply provide good results.

We note that not all combinations of attribute values are of interest. Some may be extremely unlikely, or even infeasible. For example, we may find few people with attribute age as "teen" and attribute education as "graduate degree". A human expert, with sufficient domain knowledge, is required to be in the loop for (i) identifying the attributes of interest, over which coverage is studied, (ii) setting up a *validation oracle* that identifies the value combinations that are not realistic, and (iii) identifying the uncovered patterns and the granularity of patterns that should get resolved during the coverage enhancement.

Summary of contributions. In summary, our contributions in this paper are as follows:

- We formalize the novel notion of *maximal uncovered patterns (MUP)*, to show the lack of coverage with regard to multiple attributes. We define (i) the MUP Identification problem, as well as (ii) Coverage Enhancement problem for resolving the lack of coverage. We prove that no polynomial-time algorithm can exists for (i) and that (ii) is NP-hard.
- Introducing the pattern graph for modeling the space of possible patterns, we provide three algorithms PATTERN-BREAKER, PATTERN-COMBINER, and DEEPDIVER for efficiently discovering the MUPS.
- For dataset owners, we formulate the problem of additional data collection, connect the problem to hitting set, and propose an efficient implementation of the greedy approximation algorithm for the problem.
- We use empirical evaluation on real datasets to validate our proposal, and to demonstrate the efficiency of the proposed techniques. Besides the performance evaluations, we investigate the lack of coverage in a dataset of criminals' records and discuss how a tool built using it may generate wrong signals for sentencing criminals. We show that a classifier with an acceptable performance on a random test set, may have a bad performance over the minority groups. We also show that remedying the lack of coverage improves the performance of the model for the minorities.

II. PRELIMINARIES

We consider a dataset $\mathcal{D}$ with d low-dimensional categorical attributes, $\mathcal{A} = \{A_1, A_2, ..., A_d\}$. Where attributes are continuous valued or of high cardinalities, we consider using techniques such as (a) bucketization: putting similar values into the same bucket, or (b) considering the hierarchy of attributes in the data cube for reducing the cardinalities. Each tuple $t \in \mathcal{D}$ is a vector with the value of A_i being $t[i]$ for all $i = 1, ...d$. In addition, the dataset also contains the "label attributes" $Y = \{y_1, \cdots, y_{d'}\}$ that contain the target values. The label attributes are not considered for the coverage problem. In practice, a user may be interested in studying the coverage over a subset of "attributes of interest". In such cases, the problem is limited to those attributes. For instance, in a dataset of criminals, attributes such as `sex`, `race`, and `age` can be attributes of interest while the label attribute shows whether or not the criminal has re-offended. In the rest of the paper, we assume A_1 to A_d are the attributes of interest and simply name them as the set of attributes. The cardinality of an attribute A_i is c_i. Hence, the total number of value combinations is $\Pi_{k=1}^{d} c_k$. For a subset of attributes $\mathcal{A}_i \subseteq \mathcal{A}$, we use the notation

$c_{\mathcal{A}_i} = \Pi_{\forall A_j \in \mathcal{A}_i} c_j$ to show the number of value combinations for $\mathcal{A}_i$.

Definition 1 (Pattern)**.** *A pattern P is a vector of size d, in which $P[i]$ is either X (meaning that its value is unspecified) or is a value of attribute A_i. We name the elements with value X as non-deterministic and the others as deterministic.*

An item t *matches* a pattern P (written as $M(t, P) = \top$), if for all i for which $P[i]$ is deterministic, $t[i]$ is equal to $P[i]$. Formally:

$$M(t,P) = \begin{cases} \top, & \forall i \in [1,d] : P[i] = X \text{ or } P[i] = t[i] \\ \bot, & \text{otherwise} \end{cases} \quad (1)$$

For example, consider the pattern $P = X1X0$ on four binary attributes A_1 to A_4. It describes the value combinations that have the value 1 on A_2 and 0 on A_4. Hence, for example, $t_1 = [1, 1, 0, 0]$ and $t_2 = [0, 1, 1, 0]$ match P, as their values on all deterministic elements of P (i.e., A_2 and A_4) match the ones of P. On the other hand, $t_3 = [1, 0, 1, 0]$ does not match the pattern P. That is because $P[2] = 1$ while $t_3[2] = 0$.

Using the patterns to describe the space of value combinations, we now define the coverage notion as follows:

Definition 2 (Coverage)**.** *Given a dataset $\mathcal{D}$ over d attributes with cardinalities $c = \{c_1 \cdots c_d\}$, and a Pattern P based on c and d, the coverage of P is the number of items in $\mathcal{D}$ that match P. Formally: $cov(P, \mathcal{D}) = |\{t \in \mathcal{D} \mid M(t, P) = \top\}|$.*

When $\mathcal{D}$ is known, we can simplify $cov(P, \mathcal{D})$ with $cov(P)$.

We would like a high enough coverage for each pattern, to make sure it is adequately represented. How high is enough is expected as an input to our problem, and is expected to be determined through statistical analyses. There is a long tradition of computing the "power" of an experiment design, to determine the subject pool size (corresponding to coverage) required to obtain statistically meaningful results. Borrowing the concept from statistics and central limit theorem, the rule of thumb suggests the number of representatives to be around 30. For example, Sudman [14] suggests that for each "minor subgroup" a minimum of 20 to 50 samples is necessary. This is what we also observed in our experiments (§ V-B). Using such, or other, techniques, we will assume that a *Coverage threshold*, τ, has been established for each pattern.

Definition 3 (Covered/Uncovered Pattern)**.** *A pattern P is said to be* covered *in a dataset $\mathcal{D}$ if its coverage is greater than or equal to the specified coverage threshold: $cov(P, \mathcal{D}) \geq \tau$. Otherwise, the pattern P is said to be* uncovered*.*

Each pattern describes a region in the space of value combinations, constrained by its deterministic elements. We define the level of each pattern P, shown as $\ell(P)$, as the number of deterministic elements in it. Patterns with fewer deterministic elements (smaller level) are more general. For example, consider two patterns P_1 =1XXX and P_2 =10X1 on four binary attributes A_1 to A_4. $\ell(P_1) = 1$ and $\ell(P_2) = 3$. While only the value combinations 1001 and 1011 match P_2, any value combination with value 1 on A_1 matches P_1.

The set of value combinations that match a pattern P may be a subset of the ones that match a more general pattern P'. We say that P is *dominated by* P' (or P' dominates P). For example, the pattern P_2 =10X1 is dominated by the pattern P_1 =1XXX.

Definition 4 (Parent/Child Pattern)**.** *A pattern P_1 is a parent of a pattern P_2 if P_1 can be obtained by replacing one of the deterministic elements in P_2 (say $P_2[i]$) with X. We can equivalently say that P_2 is a child of pattern P_1.*

In general, patterns can each have multiple parents and multiple children. A pattern with all elements being non-deterministic has no parent and a pattern with all elements being deterministic has no child.

If a pattern is uncovered, all of its children, and their children, recursively, must also be uncovered. When identifying uncovered patterns, it is redundant to list all these dominated uncovered patterns: doing so just makes the output much larger, and much harder for a human to digest and address. Therefore, our goal is to identify the uncovered patterns that are not dominated by more general uncovered ones.

Definition 5 (Maximal Uncovered Pattern (MUP))**.** *Given a threshold τ, a pattern P is maximal uncovered, if $cov(P) < \tau$, while for any pattern P' parent of P, $cov(P') \geq \tau$.*

With these definitions, we formally state our first problem as:

Problem 1 (MUP Identification Problem)**.** *Given a dataset $\mathcal{D}$ defined over d attributes with cardinalities c, as well as the coverage threshold τ, find all maximal uncovered patterns $\mathcal{M}$.*

While there usually are far fewer MUPs than uncovered patterns, the worst case remains bad, as we show next.

Theorem 1. *No Polynomial time algorithm can guarantee the enumeration of the set of maximal uncovered patterns.*

The proof is by construction of an example with an exponential number of MUPs.

Proof. We prove the theorem by construction. Consider a dataset $\mathcal{D}$ (shown in the following) with n items and $d = n$ binary attributes in which only the values of the elements on the diagonal are one and the rest are zero. That is, $\forall i \in [1, n] :\ t_i[i] = 1$ and $\forall j \neq i :\ t_i[j] = 0$.

	A_1	A_2	$\cdots$	A_n
t_1	1	0	$\cdots$	0
t_2	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	0	0	$\cdots$	1

Let the threshold τ be $\frac{n}{2} + 1$.

First, any pattern with $(n - 1)$ non-deterministic element and one deterministic element with value 1 is a MUP. That is because t_i is the only tuple matching such a pattern where its deterministic value (1) appears at i-th element. Since these patterns have only one non-deterministic element, their parent is the pattern $XX \cdots X$ in which there is no deterministic

element. All the items in the dataset match this pattern and, hence, its coverage is n. As a result, all such patterns with only one deterministic element 1 are MUPs. The number of such patterns in n. Also, since any pattern with more than one deterministic element where one of them is 1 is dominated by these MUPs, there cannot exist such a MUP. It means all the deterministic values of the remaining MUPs should be 0.

Next, consider a pattern P with $m \leq n$ deterministic elements with values 0. Let I be the indices of the deterministic elements. For all values $i \in I$, t_i does not match P; simply because $t_i[i] = 1$ while $P[i]$ is 0. Note that all other tuples in $\mathcal{D}$, t_i for $i \notin I$, match P. Thus, the coverage of such a pattern is $n - m$. Now, let m be $\frac{n}{2}$. The coverage of any such pattern is $\frac{n}{2} < \tau$. Similarly, the coverage of any of its parents of such nodes is $\frac{n}{2} + 1 = \tau$ (because it is a pattern with m - 1 deterministic elements with value 0). As a result, every pattern P with $\frac{n}{2}$ deterministic elements, all being 0, is a MUP. The number of such patterns are $\binom{n}{n/2}$. Therefore, the total number of MUPs in $\mathcal{D}$ is:

$$|\mathcal{M}| = n + \binom{n}{n/2} > 2^n$$

As a result, any algorithm enumerating over them is exponential. □

Not all MUPs are problematic. For example, if some combination of attribute values is known to be infeasible, the corresponding pattern will necessarily be uncovered. A domain expert can examine a list of MUPs and identify the ones that can safely be ignored. The remaining are considered *material*.

In many situations, large uncovered regions in the dataset are more harmful than narrow uncovered regions. Following this observation, for a dataset $\mathcal{D}$, we define the *maximum covered level* as the maximum level up until which there is enough coverage in the dataset. Formally:

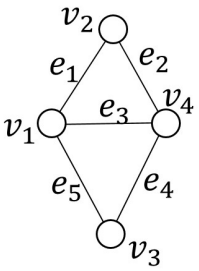
Definition 6 (Maximum Covered Level)**.** *Let $\mathcal{M}$ be the material MUPs for a dataset $\mathcal{D}$. Then, the maximum covered level of $\mathcal{D}$ is the maximum level λ such that $\forall P \in \mathcal{M}, \ell(P) > \lambda$.*

In light of the above, we would like to have as large a maximum covered level as possible for a dataset.

Problem 2 (Coverage Enhancement Problem)**.** *Given a dataset $\mathcal{D}$, its set of material MUPs $\mathcal{M}_\mathcal{D}$, and a positive integer number λ, determine the minimum set of additional tuples to collect such that, after the data collection, the maximum covered level of $\mathcal{D}$ is at least λ.*

We can consider variants of the coverage enhancement problem where we seek to attain some other coverage property rather than satisfy a maximum coverage level. For example, instead of the level of a pattern P, one could consider the number of value combination matching it.

Definition 7 (Value Count)**.** *Let $\mathcal{A}_P$ be the set of corresponding attributes for non-deterministic elements of a pattern P. The value count of P is the number of value combinations matching P. That is, $c_{\mathcal{A}_P} = \Pi_{\forall A_j \in \mathcal{A}_P} c_j$.*



(a) A sample graph for the vertex cover problem.

	A_1	A_2	A_3	A_4	A_5
t_1	1	0	1	0	1
t_2	1	1	0	0	0
t_3	0	0	0	1	1
t_4	0	1	1	1	0
t_5	0	0	0	0	0
t_6	0	0	0	0	0
t_7	0	0	0	0	0
P_1	1	X	X	X	X
P_2	X	1	X	X	X
P_3	X	X	1	X	X
P_4	X	X	X	1	X
P_5	X	X	X	X	1

(b) The constructed dataset (and its MUPs) for graph of Figure 1a.

Fig. 1: Illustration of the reduction from vertex cover to coverage enhancement problem.

For example, consider the pattern $P = X1X0$ over binary attributes $\mathcal{A} = \{A_1, \cdots, A_4\}$. $\mathcal{A}_P = \{A_1, A_3\}$. Hence, the number of value combinations matching P is $c_{\mathcal{A}_P} = 2 \times 2 = 4$. The coverage enhancement problem can be modified to require that every pattern P in $\mathcal{D}$ be covered if the value count of P is at least v. As further explained in § IV, the proposed solution can easily be extended for such alternative measures.

Next, in Theorem 2, we study the complexity of the coverage enhancement problem.

Theorem 2. *The Coverage Enhancement Problem is NP-hard.*

Proof. The decision version of the Coverage Enhancement (CE) Problem is as follows: given a dataset $\mathcal{D}$, its set of material MUPs $\mathcal{M}_\mathcal{D}$, a positive integer number λ, and a decision value m, is there a set of m additional tuples to collect such that after the data collection the maximum coverage level of $\mathcal{D}$ is at least λ. We use the polynomial-time reduction from the *vertex cover* (VC) problem [13]. The decision version VC, given an unweighted undirected graph $G(V, E)$ and a decision variable m, decides if there is a subset of m vertices such that each edge is incident to at least one vertex of the set.

The reduction is as follows: Given a graph $G(V, E)$ for VC, construct the dataset $\mathcal{D}$ with $n = |V| + 3$ items and $d = |E|$ attributes (Figure 1b shows the reduction for the sample graph of Figure 1a). For every edge $e_j \in E$ add the attributes A_j to $\mathcal{D}$. For every vertex $v_i \in V$, add the item t_i to the dataset. For every edge e_j that is connected to v_i, set $t_i[j]$ to 1; set the rest of attribute values of t_i as 0. Add three items $t_{|v|+1}$, $t_{|v|+2}$, and $t_{|v|+3}$ all with attribute values being 0. Set the coverage threshold to $\tau = 3$ and the maximum coverage level to $\lambda = 1$. In this dataset, any pattern with only one deterministic element with value 1 is a MUP. That is because, (i) for any such pattern P_j having 1 in its j-th element, the corresponding items with the two vertices in G that are connected to e_j are the only items matching it; also, (ii) the only parent of such patterns, $XX \cdots X$ has the coverage above the threshold, as all the items match it. Consequently, any pattern with a more than one deterministic element with at least one 1 is dominated by a MUP and, therefore, is not a MUP. Moreover, any pattern with some deterministic elements with values 0 is covered by $t_{|v|+1}$, $t_{|v|+2}$, and $t_{|v|+3}$ and, thus, is not a MUP. As a result, the patterns P_j (with only one deterministic element with value

1 in the j-th element) are the collection of MUPs for $\mathcal{D}$. There totally are $|E|$ of those patterns (pattern P_j is associated with e_j). Having the maximum coverage level as $\lambda = 1$, CE needs to cover all of these patterns; hence, if there exists a subset of size m of items covering these patterns, their equivalent vertices are the subset of m vertices such that each edge is incident to at least one vertex of the subset.
Given this reduction, there is no polynomial-time algorithm for the CE Problem, unless $P = NP$. □

III. MUP Identification

In this section, we study Problem 1, MUP identification, and propose efficient search and pruning strategies for it.

A. Naïve

A single pass over the dataset can suffice, with one counter for each pattern. With one pass, we obtain the count for each pattern, and can determine which patterns are uncovered. We can then compare each pair of uncovered patterns, $\{P_i,P_j\}$. If P_i dominates P_j, then the latter is not maximal, and can be removed from the list of uncovered patterns discovered. After all pairs of uncovered patterns have been processed, and the ones not maximal eliminated, then the remaining uncovered patterns are the desired maximal uncovered patterns.

Suppose there are d attributes. Each element of a pattern can either be non-deterministic, or a value from the corresponding attribute. As such, there are $c_i + 1$ choices for each attribute A_i, resulting in a total of $c^+_{\mathcal{A}} = \Pi^d_{k=1}(c_i + 1)$ patterns. We need one counter for each pattern, or a total space of $O(c^+_{\mathcal{A}})$. The time taken to find all uncovered patterns is $O(n \times c^+_{\mathcal{A}})$, where there are n tuples in the dataset. Let the total number of uncovered patterns found be u. Then an additional $O(u^2)$ time is required to find the maximal uncovered patterns from among these. Thus, the total time required is $O(n\ c^+_{\mathcal{A}} + u^2)$. While the additional time due to the second term will usually be smaller than the first term, we note that u could be as large as $c^+_{\mathcal{A}}$, and is usually much larger than the number of maximal uncovered patterns. As a toy example, consider Example 1.

Example 1. *Consider a dataset $\mathcal{D}$ with binary attributes A_1, A_2, and A_3, containing the tuples $t_1 : 010$, $t_2 : 001$, $t_3 : 000$, $t_4 : 011$, and $t_5 : 001$. Let the coverage threshold be $\tau = 1$.*

The dataset in Example 1 has one MUP 1XX. In addition to the MUP, the other 8 uncovered patterns (dominated by the only MUP) are 1X0, 1X1, 10X, 11X, 100, 101, 110, and 111.

B. Pattern Graph

In the naïve algorithm, we computed all uncovered patterns, only to eliminate those that were not maximal. It would appear that we could do less work if we could exploit relationships between patterns. Specifically, patterns have parent/child relationships, as discussed in § II. We can represent relationships between patterns by means of a pattern graph, and use this data structure to find better algorithms.

Definition 8 (Pattern Graph)**.** *Let $\mathcal{P}$ be the set of all possible patterns defined over d attributes with cardinalities c. Pattern graph of $\mathcal{P}$ is the graph $G(V, E)$ where $V = \mathcal{P}$. There is an edge between every pair of nodes P and P' that have a parent-child relationship. Every edge is between two nodes at adjacent levels, the parent node being one level smaller than the child node.*

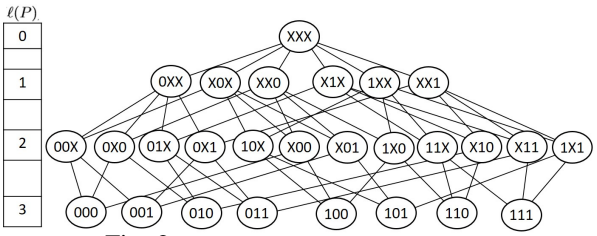


Fig. 2: The pattern graph for three binary attributes

Figure 2 shows the pattern graph for Example 1. The value of $P[i]$ for a node in the pattern graph is either X or one of the values the corresponding attribute can take. Hence, the total number of nodes in a pattern graph defined over d attributes is $c^+_{\mathcal{A}} = \Pi^d_{k=1}(c_k + 1)$. For instance, the pattern graph in Figure 2 contains $(2 + 1)^3 = 27$ nodes. Any pattern graph has only one node $XX\cdots X$ at level 0. In Figure 2, the patterns at level 1 are directly connected to XXX as those are its children. The pattern $0XX$, the left-most node at level 1, is connected to the patterns $00X$, $0X0$, $01X$, and $0X1$ at level 2. That is because, those have exactly one less X and their value for the first attribute is 0. As explained in § II, the number of value combinations matching P (with non-deterministic attributes $\mathcal{A}_P$) is $c_{\mathcal{A}_P} = \Pi_{\forall A_j \in \mathcal{A}_P} c_j$. The number of non-deterministic attributes of a pattern P with level $\ell(P)$ is $d-\ell(P)$. For example, in Figure 2, every pattern at level 1 contains $3-1 = 2$ non-deterministic elements. Since the cardinality of all attributes is $c_i = 2$, the number of value combinations matching each pattern at level 1 is $2 \times 2 = 4$. Hence, in general, the patterns with smaller levels are more general, i.e., more value combinations match them. Each node at level ℓ contains $d-l$ non-deterministic elements. There are $\binom{d}{\ell}$ such combinations in all. The deterministic elements can take any value in the cardinality of the corresponding attribute. Hence, for the special case where all attributes have the same cardinality $c_i = c$, total number of nodes at level ℓ are $\binom{d}{\ell}c^\ell$. For example, in Figure 2, there are $\binom{3}{1}2^1 = 6$ nodes at level 1 and $\binom{3}{2}2^2 = 12$ nodes at level 2. The node of a pattern P in the pattern graph has $\sum_{\forall A_i \in \mathcal{A}_P} c_i$ edges to the nodes at level $\ell(P) + 1$. If all attributes have equal cardinalities of $c_i = c$, each node at level ℓ has $c(d-\ell)$ edges to nodes at level $\ell+1$. Hence, the total number of edges in such a graph is:

$$\sum_{\ell=0}^{d-1} c(d-\ell)\binom{d}{\ell}c^\ell = c \times d \times (c+1)^{d-1}$$

This is confirmed in Figure 2, where there are totally 54 edges.

C. PATTERN-BREAKER: *The top-down algorithm*

The root of the pattern graph, level 0, contains a single node, the most general pattern that matches every value combination. The lower levels of the graph contain more specific patterns

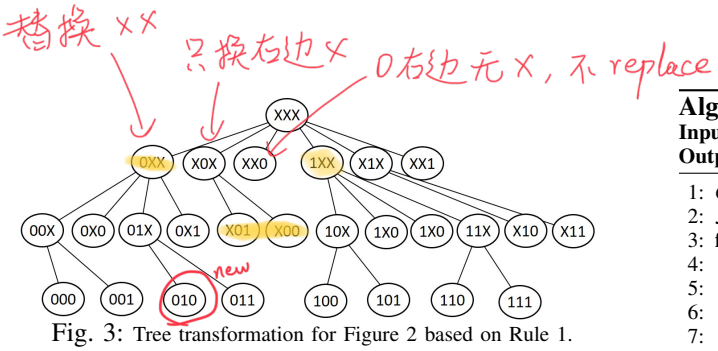


Fig. 3: Tree transformation for Figure 2 based on Rule 1.

that match fewer combinations of values. The PATTERN-BREAKER algorithm starts from the general patterns at the top of the pattern graph and moves down by breaking them down to more specific ones. It uses the *"monotonicity"* property of coverage to prune some parts of the pattern graph. That is, if a pattern P is uncovered, all of its children and descendants (the nodes at level greater than $\ell(P)$ that have a path to it) are also uncovered. Also, none of those children and descendants can be a MUP, even if it has a parent that is covered. Hence, this subgraph of the pattern graph can immediately get pruned.

The top-down BFS traversal of the pattern graph starts from the top of the graph and uses a queue to check the nodes level by level. For every node in the queue, if its coverage is greater than the threshold, it adds each of its unvisited children to the queue. This method, however, generates each node multiple times by all of its parents. Therefore, PATTERN-BREAKER uses the following rule:

Rule 1. A node P with the coverage more than the threshold τ, generates the candidate nodes at level $\ell(P)+1$ by replacing the non-deterministic elements in the right-hand side of its right-most deterministic element with an attribute value.

Theorem 3. *Enforcing Rule 1 guarantees that each MUP candidate is generated exactly once.*

Proof. Every node P (other than the root) in the pattern graph has exactly one parent node P' that is in charge of generating it based on Rule 1. This parent can be found by replacing the right-most deterministic element of P with X.

Also, based on Definition 5, since the coverage of all of the parents of every MUP $P \in \mathcal{M}$ is more than the threshold τ, the coverage of the generator of P based on Rule 1 (and every node in the path to root) is also more than the threshold. Therefore, following Rule 1 guarantees each MUP candidate once and only once. □

By enforcing Rule 1, every node in the pattern graph gets generated at most once, and hence, the graph is transformed to a *tree*. For example, Figure 3 shows the corresponding tree (generated by following Rule 1) for the pattern graph of Figure 2. In this figure, for instance, consider the node $0XX$ while considering Example 1. $cov(0XX) = 5 \geq \tau = 1$. The right-most deterministic element is element 1 with value 0. Thus, it replaces the non-deterministic elements at positions 2 and 3 one at a time, and generates the nodes $0X0$, $0X1$, $00X$, and $01X$. Similarly, the node $X1X$ generates the nodes $X10$ and $X11$. That is because the right-most non-deterministic element in this pattern is element 2 with value 1. So it replaces the X at position 3, with attribute values.

Algorithm 1 PATTERN-BREAKER
Input: Dataset $\mathcal{D}$ with d attributes having cardinalities c and threshold τ
Output: Maximal uncovered patterns $\mathcal{M}$

```
1: Q = {XX···X} // start from the root
2: M = {}; Q_p = {}
3: for /* each level of the graph */ l = 0 to d do
4:     if Q is empty then break
5:     Q_n = {}
6:     for P ∈ Q do
7:         flag = false
8:         for P' in parents(P) do
9:             if P' ∉ Q_p or P' ∈ M then flag = true; break
10:        end for
11:        if flag then continue
12:        cnt = cov(P, D)
13:        if cnt < τ then
14:            add P to M
15:        else
16:            add children of P based on Rule 1 to Q_n
17:        end if
18:    end for
19:    Q_p = Q; Q = Q_n
20: end for
21: return M
```

Starting from the root, the algorithm moves level by level, checking the candidate patterns at each level. In addition to the list of nodes at the current level, it maintains the nodes at previous level, and constructs the nodes at the next level. For every candidate node at the current level, it first checks if any of its parents is uncovered; if so, it marks the node as uncovered without computing its coverage. Otherwise, the node is added to the set of MUPs if its coverage is less than the threshold. If the node satisfies the coverage threshold, its children, while enforcing Rule 1, are added to the list of candidate nodes of the next level. We use inverted indices [15] for computing the coverage of a pattern. Further details about the efficient coverage computation are provided in Appendix A. Using the monotonicity property of the pattern graph, PATTERN-BREAKER (Algorithm 1) tries to gain performance by pruning some parts of the graph. A problem with PATTERN-BREAKER is that it traverses over the *covered* regions of the graph while we are looking for the uncovered nodes as the output. This makes its running time dependent on the size of the covered part of the graph. Moreover, the algorithm may mistakenly generate a large number of candidate nodes from the pruned regions. For example, in Figure 3, consider a case where $\tau = 1$ and the dataset does not have any items matching the pattern XX1, but contains the items $t_1 = 000$ and $t_2 = 010$. In this example, XX1 is a MUP. Following Rule 1, PATTERN-BREAKER generates the pattern 0XX, and its children in Figure 3 as both t_1 and t_2 match it (i.e., its coverage is more than the threshold). One of its children (0X1) has the coverage 0 (below the threshold), yet is not a MUP, because it is dominated by the MUP XX1.

D. PATTERN-COMBINER*: The bottom-up algorithm*

As observed in the analysis of PATTERN-BREAKER, its problem is that it explores the covered regions of the pattern graph while the objective is to discover the uncovered ones. As a result, for the cases that there are only a few uncovered

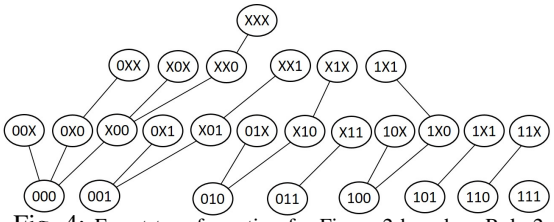


Fig. 4: Forest transformation for Figure 2 based on Rule 2.

patterns, it needs to explore a large portion of the exponential-size graph. Also, for each candidate node that it cannot prune, the algorithm needs to compute the coverage.

Therefore, we propose PATTERN-COMBINER, an algorithm that performs a bottom-up traversal of the pattern graph. It uses an observation that the coverage of a node at level ℓ of the pattern graph can be computed using the coverage values of its children at level $\ell + 1$. Consider a pattern P and a non-deterministic element $P[i]$ in it. Note that the pattern can contain more than one non-deterministic element. Consider the children of P in which element i is deterministic. These children create c_i *disjoint partitions* of the matches of P (c_i is the cardinality of attribute A_i). Thus, one can compute the coverage of P as the sum of the coverages of those children. *E.g.*, in Figure 2, $cov(1XX) = cov(1X0) + cov(1X1)$.

The algorithm PATTERN-COMBINER also uses the monotonicity of the coverage to prevent the complete traversal of the graph. That is, the coverage of a node is not less than any of the coverage of any of its children. It starts from the most specific patterns, i.e., the patterns at level d of the pattern graph, computes the coverage of each by passing over the data once. The algorithm then, keeps combining the uncovered patterns at each level to get the coverage of the candidate nodes at level $\ell - 1$. The uncovered nodes at level ℓ that all of their parents at level $\ell - 1$ are covered are MUPs.

PATTERN-COMBINER transforms the pattern graph to a *forest*, in order to make sure every node is generated once. Rule 2 guarantees the transformation.

Rule 2. A node P with the coverage less than the threshold τ, generates the candidate nodes at level $\ell(P) - 1$ by replacing the deterministic elements with value 0 in the right-hand side of its right-most non-deterministic element with X.[1]

For example, consider the node $P = X01$ in Figure 2. Its element $P[1]$, with value X, is the right-most non-deterministic element in P. The only deterministic element with value 0 in the right hand-side of element 1 is element 2. Therefore, this node generates the node $XX1$ in the bottom-up traversal of the graph. As another example, consider the node with pattern $P = 000$. It does not have any non-deterministic element and therefore, 0 is considered as the index of the right-most X. elements 1, 2, and 3 have values 0 and therefore, this node generates the nodes $00X$, $0X0$, and $X00$. Figure 4 shows the transformation of Figure 2 to a forest, based on Rule 2.

Theorem 4. *Enforcing Rule 2 guarantees each MUP candidate is generated exactly once.*

[1]Note that this rule is not specific to the binary attributes. All we require is that one of the values of each attribute is mapped to 0.

Proof. The proof is based on the fact that every non-leaf node P in the pattern graph has exactly one child node P' that is in charge of generating it based on Rule 2. This child can be found by replacing the right-most non-deterministic element of P with 0.

Also, based on the monotonicity of coverage, all of the children of every MUP $P \in \mathcal{M}$ has the coverage less than τ. Therefore, the coverage of the generator of P (and every node in the path to the starting leaf node) based on Rule 2 is also less than the threshold. Consequently, following Rule 2 guarantees to generate each MUP candidate exactly once. □

PATTERN-COMBINER prunes a branch once it finds a covered node. Due to the monotonicity of coverage if a child node is covered, all of its parents (and ancestors) are also covered. Therefore, a node is not pruned by the algorithm unless it is covered. This, together with Theorem 4, provide the assurance that all MUPs are discovered by PATTERN-COMBINER.

Starting from the bottom of the pattern graph, the algorithm first iterates over the dataset to compute the coverage of each node at level $\ell = d$. Maintaining the record of the uncovered nodes at each level, the algorithm generates the nodes at level $\ell-1$ by applying Rule 2 on the uncovered nodes of level ℓ. For each node P' at level $\ell-1$, PATTERN-COMBINER finds a set of nodes at level ℓ that create disjoint partitions of the matches of P'. To do so, it finds the right-most non-deterministic element i in P' and considers the children of P' that have a value on element i. The coverage of P' is the summation of the coverages of those nodes. If the coverage P' is less than τ its record is maintained as a candidate MUP. After finding the uncovered nodes at level $\ell - 1$, the algorithm adds the MUPs at level ℓ to the output set. The pseudocode of PATTERN-COMBINER is in Algorithm 2.

E. DEEPDIVER*: Fast search space pruner*

PATTERN-BREAKER traverses over the covered regions of the pattern graph before it visits the uncovered patterns. Therefore, it does not perform well when a large portion of the pattern graph is covered. Conversely, PATTERN-COMBINER traverses over the uncovered nodes first; so it will not perform well if most of the nodes in the graph are uncovered. When most MUPs are in the middle of the graph, both algorithms do poorly because they have to traverse about half of the graph. In this subsection, we propose DEEPDIVER, an algorithm that tries to quickly identify some MUPs and use them to prune the search space.

The monotonicity property creates an opportunity to prune the search space: *none of the ancestors or descendants of a given MUP can be MUP*. PATTERN-BREAKER tends to traverse level by level over the covered regions of the pattern graph before it visits the uncovered patterns. As a result, the moment it reaches out to a MUP, it already has visited its ancestors and does not take the advantage of pruning the nodes dominating the MUPs. PATTERN-COMBINER, on the other hand, starts off in the uncovered regions; initially, the nodes being visited early are at the bottom of the pattern

Algorithm 2 PATTERN-COMBINER
Input: Dataset $\mathcal{D}$ with d attributes having cardinalities c and threshold τ
Output: Maximal uncovered patterns $\mathcal{M}$

count = *new* hash()
for $\forall P \in \{v_1 v_2 \cdots v_d \mid v_i \in c[A_i]\}$ **do**
  cnt= $cov(P, \mathcal{D})$
  if cnt< τ **then** count[P] = cnt
end for
if count is empty **then return** $\emptyset$
for $\ell = 0$ to d **do**
  nextCount = *new* hash()
  for P in count.keys **do**
    $\mathcal{P}'$ = generates nodes at $\ell - 1$ based on Rule 2 on P
    for $P' \in \mathcal{P}'$ **do**
      i = the index of right-most X in P'
      $\mathcal{P}'' = \{P'' | \forall j \neq i : P''[j] = P'$ and $P''[i] \in c[A_i]\}$
      $cnt = \sum_{\forall P'' \in \mathcal{P}''}$ (count[P''] **if** $P'' \in$ count.keys **else** τ)
      if $cnt < \tau$ **then** nextCount[P'] = cnt
    end for
  end for
  for P in count.keys **do**
    if parents(P)$\cap$ nextCount.keys $= \emptyset$ **then**
      add P to $\mathcal{M}$
    end if
  end for
  if nextCount is empty **then break**
  count = nextCount
end for
return $\mathcal{M}$

Algorithm 3 DEEPDIVER
Input: Dataset $\mathcal{D}$ with d attributes having cardinalities c and threshold τ
Output: Maximal uncovered patterns $\mathcal{M}$

Let S = an empty stack
push $X \cdots X$ to S
while S is not empty **do**
  P = pop a node from S
  $uncoveredFlag$ = a flag indicating if P is uncovered
  if P is dominated by $\mathcal{M}$ **then**
    continue
  else if P dominates $\mathcal{M}$ **then**
    $uncoveredFlag = true$
  else
    $cnt = cov(P, \mathcal{D})$
    $uncoveredFlag = cnt < \tau$
  end if
  if $uncoveredFlag$ is $true$ **then**
    Let S' = an empty stack
    while S' is not empty **do**
      P' = pop a node from S'
      $\mathcal{P}'$ = generates parent nodes of P' by replacing one deterministic cell with X.
      for $P'' \in \mathcal{P}'$ **do**
        $cnt' = cov(P'', \mathcal{D})$
        if $cnt' < \tau$ **then** push P'' to S'; **break**
      end for
      add P to $\mathcal{M}$
    end while
  else
    Q = generate nodes on P and c based on Rule 1
    push Q to S
  end if
end while
return $\mathcal{M}$

graph. It gradually moves up level by level until it hits the MUPs. Therefore, when the MUPs are discovered the descendants have already been visited and, as a result, PATTERN-COMBINER does not take the advantage of pruning the nodes dominated by MUPs.

With the above observations, we propose DEEPDIVER, a search algorithm that tends to quickly find MUPs, and use them to limit the search space by pruning the nodes both dominating and dominated by the discovered MUPs. Since each MUP is the child of a covered node, instead of scanning through the covered/uncovered patterns level by level, DEEPDIVER takes a path down to find an uncovered node.

Initially, DEEPDIVER(Algorithm 3), following a DFS strategy, takes a path down until it reaches into an uncovered region in the graph. However, the discovered uncovered pattern is not necessarily a MUP, as some of its other parents (other than its generator) might also be uncovered. For instance in Example 1, assume that in the first iteration, the algorithm take the path XXX $\rightarrow$ X0X $\rightarrow$ 10X. The nodes XXX and X0X are covered, but 10X is not. Still the uncovered node 10X is not a MUP as it has the uncovered parent 1XX. Therefore, after finding an uncovered node, DEEPDIVER changes direction and starts moving up to find a MUP. To do so, it checks the parents of the current node to see if any of them are uncovered. If there exists such a parent, it moves to the parent and continues until it finds a MUP. Upon discovering a MUP, DEEPDIVER prunes all of its ancestors and descendants, and continues the search for other MUPs in the regions that are still not pruned. The algorithm stops when all of the nodes in the pattern graph are pruned.

Here we extend the notion pattern dominance to MUP dominance, as follows:

Definition 9 (MUP Dominance). *Given a pattern P and a set of MUPs $\mathcal{M}$, P is dominated by $\mathcal{M}$, if there exists a pattern $P' \in \mathcal{M}$ such that P is dominated by P'. Similarly, P dominates $\mathcal{M}$, if there exists a pattern $P' \in \mathcal{M}$ such that P dominates P'.*

Based on the Definition 9, a node being dominated by MUPs is not a MUP. DEEPDIVER uses this property to limit the search space by pruning all descendants of the MUPs. Similarly, the nodes that dominate MUPs are out of the search space. We use inverted indices for efficiently checking MUP dominance. See Appendix B for details.

IV. COVERAGE ENHANCEMENT

So far in this paper, we discussed how to address Problem 1, by discovering the maximal uncovered patterns. A natural question, after discovering the MUPs is how to collect the data based on them. In this section, we seek to answer this question, i.e., Problem 2.

Every MUP represents a part of the value combinations space for which there are not enough observations in the dataset. For example, consider a dataset defined over three ternary attributes A_1, A_2, and A_3, in which MUPs are XX1, 0XX, and 20X. Figure 5 shows the matches for the patterns XX1, 0XX, and 20X, as the red, green, and blue cubes, respectively. The more general MUPs show larger uncovered regions in the data. For example in Figure 5, the cube of the

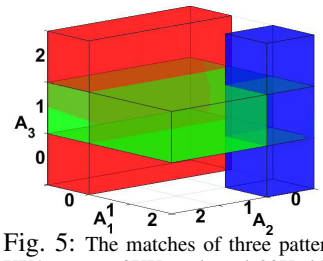


Fig. 5: The matches of three patterns XX1: green, 0XX: red, and 20X: blue in a data set with ternary attributes A_1, A_2, and A_3.

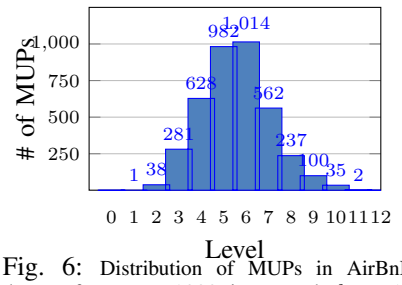


Fig. 6: Distribution of MUPs in AirBnB dataset for $n = 1000$ items and $d = 13$ attributes, while $\tau = 50$.

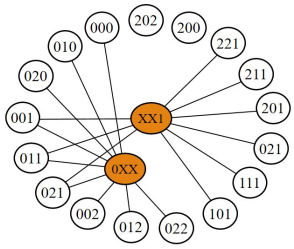


Fig. 7: The bipartite graph for $\lambda = 1$ for Figure 5.

P_1	XX01X
P_2	1X20X
P_3	XXXX1
P_4	02XXX
P_5	XX11X
P_6	111XX
P_7	X020X

Fig. 8: MUPs of Example 2.

more general patterns XX1 and 0XX contain 9 combinations, whereas the one for the pattern 20X contains 3 combinations. While we may be willing to leave some small regions uncovered, we would like to cover at least the large ones. For example, not having enough representatives in a dataset for single black males over the age of sixty may be less of a problem than not having enough black males. Figure 6 shows the distribution of the levels of the MUPs for a real experiment on our AirBnB dataset (c.f. § V) with $n = 1000$ items and $d = 13$ attributes, while $\tau = 50$. There are several thousand MUPs in this setting. This indicates the high expense of covering them all. However, as the distribution has a bell-curve shape, while most MUPs appear at levels 5 and 6, there is only one MUP at level one and less than forty in level two.

Data acquisition is usually costly. If the data are obtained from some third party, there may be direct monetary payment. If the data are directly collected, there may be a data collection cost. In all cases, there is a cost to cleaning, storing, and indexing the data. To minimize these costs, we would like to acquire as few additional tuples as possible to meet our coverage objective.

Before discussing further technical details, we would like to emphasize the necessity of *human-in-the-loop* after the MUP discovery. Not all the MUPs that are discovered are meaningful and some of them may even be invalid. Therefore after the MUP discovery, a domain expert should evaluate and mark out the MUPs that are not problematic. In addition, we require the expert to set up a *validation oracle* as a set of rules that identifies if a value combination is semantically correct or not. For example, any value combination that contains {gender=Male, isPregnant=True} is semantically incorrect.

Definition 10 (Validation Rule)**.** *A validation rule is a set of pairs* $\{\langle A_i, V_i\rangle, \cdots\}$*, where* A_i *is an attribute and* V_i *is a set of values for* A_i*. Given a pattern* P *and a validation rule* R*, we say* P *satisfies* R*, if* $\forall\langle A_i, V_i\rangle \in R$*:* $P[i] \in V_i$*.*

Definition 11 (Validation Oracle)**.** *A validation oracle contains a collection of validation of rules. Given a pattern* P*, the oracle returns true if* P *satisfies* none *of its validation rules. It returns false otherwise.*

The human expert sets up the validation oracle by identifying the collection of validation rules. Later on, in this section, we call the validation oracle to enforce the rules that result in the semantic appropriateness (validity) of the output of the coverage enhancement algorithm.

As formally defined in § II as Problem 2, for a given value λ, our objective is to collect the minimum number of additional tuples such that after the data collection the maximum covered level of $\mathcal{D}$ is at least λ. It is not enough to cover the MUPs with levels $\ell \leq \lambda$, we must cover all uncovered patterns (not necessarily maximal) with level $\ell(P) = \lambda$. We use M_λ to refer to the set of uncovered patterns at level λ. Finding this set is not difficult: details in Appendix C.

We take Example 2 as a running example in this section.

Example 2. *Consider a dataset* $\mathcal{D}$ *with 5 attributes* A_1 *to* A_5 *where* A_2 *and* A_3 *are ternary attributes while the other attributes are binary. Suppose the maximal uncovered patterns are as shown in Figure 8.*

Let λ be 2. Uncovered patterns of Example 2 with level 2, i.e. M_λ, are P_1 to P_6. Our objective is to cover all these patterns.

If we use an alternative problem formulation, the set of patterns to cover may be different. For example, if we wish to cover all patterns with value count of at least v, we must enumerate uncovered patterns that meet this value count criterion. Once this (straightforward) enumeration is completed, thereafter the alternative problem formulation can be solved in exactly the same way. A potential naive idea may be to acquire enough additional tuples separately for each pattern we are required to cover. However, this "solution" acquires much more than the minimum data required, because each tuple may contribute to the coverage of multiple patterns. What we need is to choose tuples carefully to find the minimum number needed to cover all the uncovered patterns of interest. This problem translates to a hitting set [13] instance.

A. Transformation to hitting set

Hitting Set Problem: Given a set $\mathcal{U}$ of elements and a collection $\mathcal{S}$ of non-empty subsets of $\mathcal{U}$, the objective is to find the smallest subset of elements $C \subseteq \mathcal{U}$ such that $\forall S \in \mathcal{S},\ \exists\ e \in C$ where $e \in S$.

The transformation is as follows:

- $\mathcal{U}$: The set of possible value combinations translates to the universe of items $\mathcal{U}$.
- $\mathcal{S}$: Each uncovered pattern in M_λ is the representative of the set of value combinations matching it. Hence, $\mathcal{S}$ is the collection of sets represented by the uncovered patterns.

This can be viewed as a bipartite graph with the value combinations in the first part and the uncovered patterns in

	P_1	P_2	P_3	P_4	P_5	P_6
$A_1 = 0$	1	0	1	1	1	0
$A_1 = 1$	1	1	1	0	1	1
$A_2 = 0$	1	1	1	0	1	0
$A_2 = 1$	1	1	1	0	1	1
$A_2 = 2$	1	1	1	1	1	0

Fig. 9: The inverted indices for values of attributes A_1 and A_2 in Example 2.

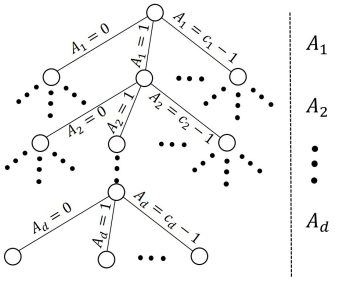


Fig. 10: The tree data structure used for computing the number of patterns a value combination hits.

the second part. There is an edge between a combination and a pattern if the combination matches the pattern. The objective is to select the minimum number of nodes in the first part that hit all the patterns in the second part. Figure 7 shows the bipartite graph for $\lambda = 1$ in Figure 5.

While the hitting set problem is NP-complete, the greedy approach guarantees a logarithmic approximation ratio for it [13]. At every iteration, the greedy approximation algorithm selects the item (value combination) that hits the maximum number of un-hit sets (patterns). It continues until all the sets get hit. In Figure 7, for instance, a run of the greedy algorithm picks 001 as it hits both patterns and then stops.

At every iteration, the algorithm needs to find the value combination that hits the maximum number of un-hit patterns. This is inefficient due to the exponentially large number of the value combinations and potentially exponential number of the patterns to hit. Hence, in the following, we develop an efficient implementation of the greedy algorithm.

B. Efficient implementation of the greedy algorithm

Consider the set of patterns we want to hit by the value combinations. We use inverted indices to keep track of the uncovered patterns. The i-th element of each pattern is either a value of A_i or is non-deterministic. For each attribute value v_j of A_i, we create an inverted index to point to the patterns with either a non-deterministic element or an element with value v_j in the i-th position. We use this to filter out the patterns that do not match a value combination with value v_j on A_i. For example, Figure 9 shows the indices for the values of A_1 and A_2 for P_1 to P_6 in Example 2. The first row shows the inverted index for $A_1 = 0$. All columns of the row, except P_2 and P_6 are 1; that is because a value combination having $A_1 = 0$ will not match P_2 or P_6, but still can match other patterns. Having the inverted indices for the attribute values, we use a tree data structure and design a greedy threshold-based algorithm to find the value combination that hits the maximum number of remaining patterns.

Consider a full tree data structure (Figure 10) with depth d (the number of attributes). Let m be the number of patterns that we want to hit. The root has c_1 children, each showing a value for attribute A_1. Similarly, every node at level i has c_i children, each representing an attribute value of A_i. For instance, in Example 2, the depth of the tree is $d = 5$; the root node has two children standing for $A_1 = 0$ and $A_1 = 1$. Since $c_2 = 3$, each of these two nodes will have three children $A_2 = 0$, $A_2 = 1$, and $A_2 = 2$.

Consider a path in the tree from the root to a leaf; note that it shows a value combination. We associate a bit vector of size m with the root. Initially, all the values in the bit vector are 1, meaning that all patterns still remain to get hit. To find the number of remaining patterns a value combination hits, one can follow the path from the root to it and, along the way, for every edge $A_i = v_j$ update the bit vector showing the patterns it may still hit. This is done efficiently by applying a binary AND operation between the inverted index of $A_i = v_j$ and the current bit vector (initially the bit vector of the root). The number of 1's in the final bit vector after reaching to the leaf node is the number of patterns it hit. For instance, in Example 2 (for patterns P_1 to P_6), assume that still none of the patterns are hit. Thus, the bit vector associated with the root is 111111, showing that none of P_1 to P_6 is so far hit. Now, consider the value combination 12110. In the tree data structure, following from the root to the leaf node, we first reach to the edge $A_1 = 1$. From Figure 9, the inverted index for this is 111011. Hence, after the first AND operation, the patterns it may still hit are: $111111 \wedge 111011 = 111011$. The next value is $A_2 = 2$; having the inverse index of this as 111110, the bit vector gets updated as $111011 \wedge 111110 = 111010$. Following this on the inverse indices of $A_3 = 1$, $A_4 = 1$, and $A_5 = 0$, the final result is 000010, that is 12110 only hits P_5. Using this structure, we design the threshold-based algorithm GREEDY that uses the hit-count of its best-known value combination to prune the tree. The algorithm traverses through the tree data structure in a DFS manner and, starting from the root, it calls the validation oracle before generating each child of a node, to make sure it is semantically meaningful. As a result, it will output only the value combination that are valid. The algorithm computes the bit vector of valid children of a node by applying the binary AND operation between the current bit vector ($filter$) and the inverted index of each of its children. The algorithm uses the best-known value combination as a lower-bound threshold to prune the tree. If the children of the current node are leaf nodes and the best of them hits more patterns than the best-known value combination, the best option gets updated. For the other nodes, the algorithm prioritizes the children of the current node based on the number of 1's in their bit-vectors. Then, starting from the one with the max count, until the number of 1's in the bit-vectors is more than the best-known hit count, it recursively checks if it can find a better value combination in the subtrees of the children. Following this algorithm for patterns P_1 to P_6 in Example 2, a value combination that hits the maximum number of patterns is 02011, hitting the patterns P_1, P_3, and P_4.

Now, using this algorithm, we can implement the GREEDY approximation algorithm (Algorithm 5). The inputs to the algorithm are the set of uncovered patterns at level λ, i.e., M_λ, and it returns the set of value combinations to collect. It keeps collecting the value combinations that hit the maximum number of remained patterns until all of the patterns in M_λ

get hit. Following the greedy algorithm on Example 2 on P_1 to P_6, it suggests collecting three value combinations 02011, 02111, and 10201.

As an implementation note, when the algorithm selects a value combination, it takes the intersection of the patterns it hits to find a more general pattern that any of its matching value combinations hit the same set of patterns. It provides more freedom to the user in the data collection.

V. Experimental Evaluation

We conducted comprehensive experiments on real data to both validate our proposal and to study the efficiency of the proposed algorithms in practice.

A. Experimental Setup

Datasets. Three real datasets were used for the experiments:

- *COMPAS*[2]*:* ProPublica is a nonprofit organization that produces investigative journalism. They collected and published the COMPAS dataset as part of their investigation into racial bias in criminal risk assessment. The dataset contains demographics, recidivism scores, and criminal offense information for 6,889 individuals. We used the attributes `sex` (0: male and 1: female), `age` (0: under 20, 1: between 20 and 39, 2: between 40 and 59, and 3: above 60), `race` (0: African-American, 1: Caucasian, 2: Hispanic, and 3: other races), and `marital status` (0: single, 1: married, 2: separated, 3: widowed, 4: significant other, 5: divorced, and 6: unknown) for studying the coverage.
- *AirBnB*[3]*:* AirBnB is a popular online peer to peer travel marketplace that provides a framework for people to lease or rent short-term lodging. We use a collection of the information of approximately 2 million *real* properties enlisted in AirBnB. The website provides 41 attributes for each property, out of which 36 are boolean attributes, such as `TV`, `internet`, `washer`, and `dryer`.
- *BlueNile*[4]*:* Blue Nile is the largest online diamond retailer globally. We collected its catalog containing 116,300 diamonds at the time of access. The dataset has 7 categorical attributes for the diamonds, namely `shape`, `cut`, `color`, `clarity`, `polish`, `symmetry`, and `florescence` with cardinalities 10, 4, 7, 8, 3, 3, and 5, respectively.

[2]www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis

[3]www.airbnb.com

Algorithm 4 hit-count

Input: The bit vector $filter$, best-known hit-count c_{max}, inverse indices I, and the current level i

```
for value v_j in c_i do
    bv[v_j] = filter ∧ I_{A_i=v_j}
    cnt[v_j] = number of 1's in bv[v_j]
end for
if i == d then
    v_max = argmax cnt[j]
    return max(cnt[v_max], c_max), v_max
end if
sort values v_j in c_i based on cnt[j], descendingly
for j = 1 to c_i do
    if cnt[v_j] < c_max then
        break
    end if
    tmp_cnt, tmp = hit-count(bv[v_j], c_max, I, i + 1)
    if tmp_cnt > c_max then
        retval = join(tmp, v_j)
        c_max = tmp_cnt
    end if
end for
return c_max, retval
```

Hardware and Platform. The experiments were conducted on a Linux machine with a 3.8 GHz Intel Xeon processor and 64 GB memory. The algorithms were implemented in Java.

Experiments plan. We want to study coverage in real data. Are there indeed uncovered patterns? Are these likely to cause errors in prediction or analysis? We also want to study the performance of the proposed algorithms, both for MUP identification and for coverage enhancement. We studied both these sets of questions on all three datasets. In the interests of space, we report only the most salient results. In particular, we focus on the COMPAS dataset for the first set of questions, since the negative consequences of lack of coverage are potentially more severe than for other datasets where the impact may be limited to errors in analytical results. We focus on the AirBnB dataset for the performance questions, since this is the largest of the three datasets. Since attributes of AirBnB are binary, we supplement with the BlueNile dataset to highlight situations where its much higher cardinality of attribute values matters.

B. Validation

1) Issues with Coverage in Real Data: Consider four demographical attributes `sex`, `age`, `race`, and `marital status`, as the attributes of interest in the COMPAS dataset. We investigate the lack of coverage in this dataset with regard to these four attributes to show the risks of using it for important tasks such as assessing a criminals' likelihood to reoffend and sentencing them accordingly. Setting the threshold to 10, all the single attribute values contain more instances than the threshold. Still, there totally are 65 MUPs in this dataset, out of which 19 are in level $\ell = 2$, 23 in level $\ell = 3$, and 23 in level $\ell = 4$. Besides other MUPs, the existence of 19 level two MUPs in the dataset emphasizes the potential of bad predictions for large spaces in the data cube. To highlight one example, the MUP XX23 shows the lack of coverage for *widowed Hispanics*. The dataset contains

[4]www.bluenile.com/diamond-search?

Algorithm 5 GREEDY

Input: The set of uncovered patterns to hit M

Output: The set of value combinations to collect

```
filter = a bit vector of size |M| with all bits being 1
I = inverted indices of attribute values to M
V = {}
while ∃ 1 ≤ j ≤ |M| s.t. filter[j] = 1 do
    c_max, v = hit-count(filter, 0, I, 1)
    add v to V and update the filter accordingly
end while
return V
```

only two instances matching this pattern and interestingly both of them have offended multiple times. In the absence of enough representatives for the minority subgroup, the trained model, will likely generalize, not sticking to the couple of examples it has seen for the minority subgroup. However, the generalization becomes problematic when the "behavior" in the subgroup is different and the generalization is misleading. This means that the model *may* not do a good job in modeling the behaviour of minority sub-groups. Of course, we use the MUP identification to raise a signal for these lack of coverage cases. Whether or not it is problematic, needs the human expert in the loop. Lack of coverage in this dataset shows the risk of using it for predicting the behavior of individuals for under-represented groups; it, therefore, questions the decisions made based on such predictions. To show case an effect of the lack of coverage, next we use this dataset for training a classifier.

2) Lack of coverage's effect: After showing the lack of coverage in the COMPAS dataset, next we conduct an experiment to show its effect on accuracy of a prediction task. Using the scikit-learn package (version 0.20) on Python, we trained a *decision tree* as the classifier, while using `sex`, `age`, `race`, and `marital status` as the observation attributes. Using the attribute `prior-count`, we created the binary label attribute that shows if a criminal has re-offended. First, using the cross-validation, we observed that the trained model has acceptable accuracy and f1 measures of 0.76 and 0.7 over a random test set. Relying on these numbers, a data scientist may consider using this model for predicting the behaviour of criminals. However, in this experiment we show that these measures does not necessarily show the good performance for the minorities. We focus on the minority class of Hispanic Females (HF), as there are only 100 of those in the dataset. We chose this group, specifically because we were limited to the records in the dataset, and were not able to collect additional data points. Hispanic Females was (i) a minority subgroup small enough that removing its instances would not noticeably change the size of the training data, while (ii) there were "enough" tuples of this group in the dataset (100 tuples) that we could show case the impact of additional data collection. We considered a randomized set of 20 (out of the 100) HF as the test set for studying the prediction over this group. Since we not only wanted to study the effect of the lack of coverage, but also the coverage enhancement, we created 5 training sets (using the remaining 80 HF criminals), containing $\{0, 20, 40, 60, 80\}$ HF plus all other records not in this demographic. We used these datasets for training the classifier and calculated its accuracy and f1 measure for predicting our test set of 20 HF. Figure 11 shows the results. The x-axis shows the datasets, while the left and right y-axes show the accuracy and f1 measure for each setting. Collecting the additional data points should result in more accurate for the under-represented groups, as those are to provide a better understanding of those, while not having a major impact in the overall accuracy. This is confirmed in the figure, as the overall accuracy remained on 76% in all settings. We also observed that overall f1-measure did not change from 0.7. First, one can see that the dataset that does not have any HF, has an unacceptable performance for this class, as its accuracy is less than 50%. The next observation is that the accuracy and f1 measures improve as the lack of coverage is resolved by adding more HF to the training data. The reduction in the slope of the accuracy curve around 40 suggests that it can be a good choice for the coverage threshold. Interestingly, this is aligned with the central limit theorem's rule of thumb of 30. In a similar experiment, we considered two other minority subgroups, (1. Female - Other Races (FO) and 2. Male - Other Races (MO)) for which there existed at least 20 records in the dataset that we could consider as the test data. Removing the records of these demographics from the training data, the accuracy of the model was 39% for FO and 59% for MO. The accuracy different between the two groups shows the higher similarity in the "behaviour" of MO to other records in the training data.

3) Coverage Enhancement Quality: In previous experiment, we showed the quality of coverage enhancement in the sense that it increases the model performance for the under-represented groups, while not impacting the overall performance of the model. In this experiment, we show the role of human-in-the-loop by setting up the validation oracle and identifying the MUPs to be covered. Enforcing the rules of validation oracle while expanding the tree data structure used by the coverage enhancement algorithm GREEDY, the semantic appropriateness (validity) of the output of the coverage enhancement algorithm is guaranteed. We consider the MUPs discovered in § V-B1 while targeting to satisfy the coverage level of 2. In the validation oracle, we rule out (a) the combinations with marital status being unknown and (b) the age group below 20 being not single. Coverage enhancement suggests to collect {over 60, other races, widowed}, {between 20 and 40, Hispanic, widowed}, {over 60, significant other}, {other races, divorced}, and {other races, widowed}.

C. Performance Evaluation

We evaluate the performance of (i) the three MUP identification algorithms PATTERN-BREAKER, PATTERN-COMBINER, and DEEPDIVER, as well as (ii) the coverage enhancement algorithm. The Naïve algorithm for MUP identification (§ III-A) did not finish for any of the settings within the time limit. Therefore, we did not include it in the results. For the coverage enhancement problem, we compare the GREEDY algorithm (§ IV-B) with the direct implementation of the hitting set's approximation algorithm (naïve). We use our largest dataset, i.e. AirBnB, as the default and test the algorithms' performances under various coverage threshold (τ) on both AirBnB and BlueNile. We varied the number of attributes (d) and the size of the dataset (n) on our largest dataset, that is AirBnB. In addition to the proposed algorithms, for MUP identification, we also consider comparing with APRIORI, the following adaptation of apriori algorithm [12]: we consider each ⟨attribute,value⟩ as an item and find the frequent item-sets. For each such item-set we find its parents (the item-sets that include the item-set and one more item). For each such parent, if all of its children are frequent, we find the

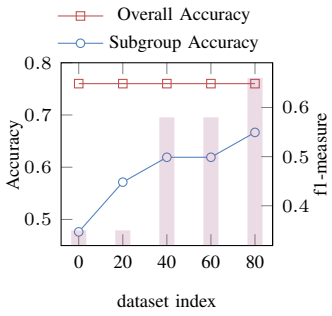


Fig. 11: The effect of lack of coverage on classification: accuracy, f1 measure

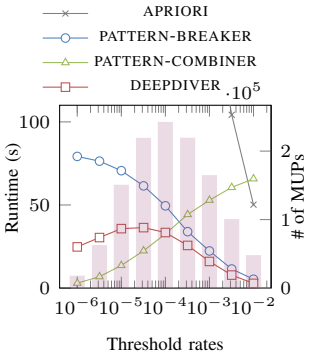


Fig. 12: AirBnB: MUP identication, varying threshold (n = 1M, d = 15)

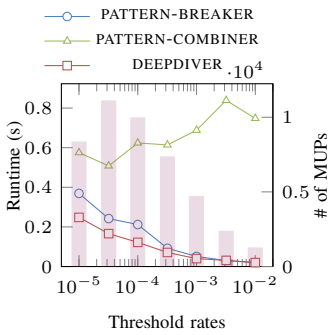


Fig. 13: BlueNile: MUP identication, varying threshold (n = 116,300, d = 7)

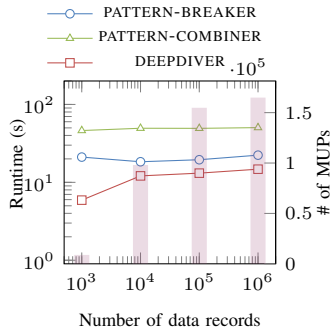


Fig. 14: AirBnB: MUP identication, varying data size (τ = 0.1%, d = 15)

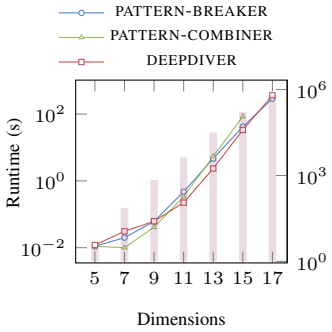


Fig. 15: AirBnB: MUP identication, varying dimension (n = 1M, τ = 0.1%)

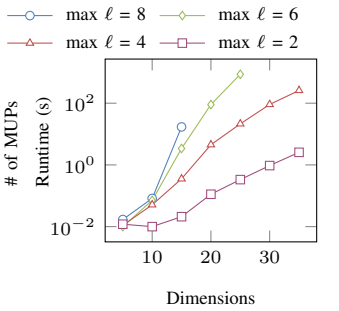


Fig. 16: MUPs identification with various dimensions using DEEPDIVER (AirBnB, n = 1M, τ = 0.1%)

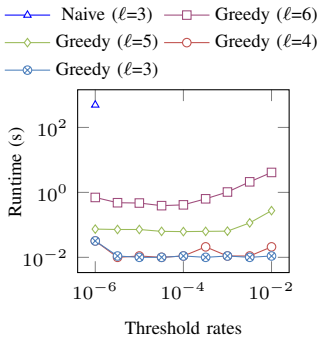


Fig. 17: Coverage Enhancement with various thresholds (AirBnB, n = 1M, d = 13)

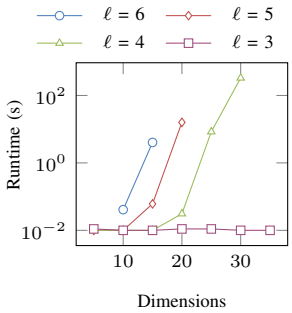


Fig. 18: Coverage Enhancement with various dimensions using Greedy (AirBnB, n = 1M, τ = 0.1%)

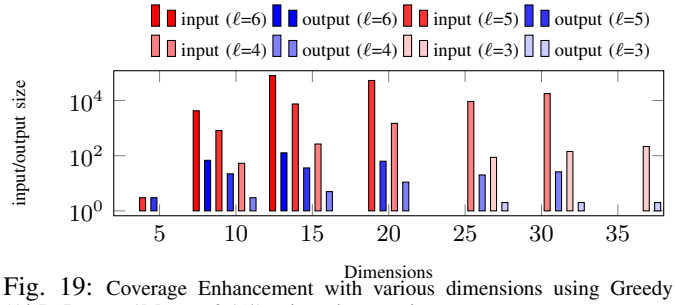


Fig. 19: Coverage Enhancement with various dimensions using Greedy (AirBnB, n = 1M, τ = 0.1%) – input/output sizes

corresponding pattern and add it to the set of MUPs. We understand that because the items are considered independently in the frequent item-set mining, not all item-set represent a valid pattern. For instance, consider the items $I_1 = \langle A_1, 0\rangle$ $I_2 = \langle A_1, 1\rangle$. Then the item-set $\{I_1, I_2\}$ does not represent a valid pattern. Furthermore, this algorithm considers a much larger search space (lattice) to explore, compared to the pattern graph our algorithms explore. For instance, consider a case where there are 10 attributes, each with cardinality 5. The size of the pattern graph is $(5+1)^{10}$, around 60 million nodes, whereas after considering each attribute-value as an item, the size of the defined lattice is $2^{5\times 10}$, around 10^{15}.

1) MUP identification - varying threshold: For AirBnB, we varied the coverage threshold from 0.0001% (most patterns are covered) to 1% (most patterns are uncovered). The dataset size was set to one million, and the number of attributes was set to 15. For BlueNile, we had 7 attributes and 116,300 records. We varied coverage from 0.001% (threshold = 1) to 1%.

Results. The runtimes are shown in Figure 12 (AirBnb) and Figure 13 (BlueNile). The x-axis denotes the different threshold rate values. The left-y-axis is the runtime in seconds, while the right axis and the bars show the output size (number of MUPs). In addition, Figure 12 also contains the results for APRIORI, the adaptation of the apriori algorithm for discovering the MUPs. As explained in § V-C, this algorithm suffers from multiple facts that makes it unsuitable for MUP discovery. First, the lattice data structure it has to explore can be extensively larger than the pattern graph. Second, it needs to generate the parents of the frequent item-sets to find the infrequent item-sets that all of their children are frequent. Finally, not all the discovered item-sets represent valid MUPs. This is confirmed in this experiment where it only finished for one settings in less than 100 seconds. For instance for threshold of 0.001% it took 516 sec. to finish. As expected, we observed the same behaviour in other experiments as well. Hence, in the rest of experiments, we only focus on evaluating the algorithms we proposed in this paper. When the threshold increases, larger regions in the space become uncovered and more general MUPs with smaller levels appear in the results. This is the reason for the drop in the runtime of PATTERN-BREAKER in Figure 12 and Figure 13. Recall that PATTERN-BREAKER is a top-down search algorithm, and generally returns faster when the MUPs are higher in the pattern graph (having small levels). In contrast, PATTERN-COMBINER's runtime increases as the PATTERN-COMBINER

is a bottom-up search algorithm and, hence, terminates faster when the MUPs are low in the pattern graph (when the space is mostly covered) as shown in Figure 12 and Figure 13. In tests with AirBnB, these two algorithms have similar speeds when the threshold is around 0.01%, in which case, most MUPs appear in the middle of the graph. Meanwhile, Figure 12 also shows that DEEPDIVER is as fast, if not faster, as the other two algorithms in all situations. This suggests that the efficiency of DEEPDIVER is more robust to the actual data coverage status. As for BlueNile, Figure 13 also suggests DEEPDIVER is the best in all cases, whereas PATTERN-COMBINER is always slower. Still the gap between PATTERN-COMBINER with the two other algorithms is larger. The high cardinality of the attributes in BN is the key to this behavior. In this situation, the width of the pattern graph quickly increases. The lowest level (level 7) of the pattern graph in this case has more than 100K nodes, whereas for 7 binary attributes, it is 128. Therefore, due to the significant width of the graph in the bottom-level, PATTERN-COMBINER (the bottom-up algorithm) loses its efficiency.

2) MUP identification - varying data size: Setting the number of attributes to 15 and threshold to 1%, we evaluated the three MUP identification algorithms on data samples of various sizes from 10K to 1M and measured the runtime.

Results. Figure 14 shows the runtime plots. The x-axis denotes the size of test dataset; the left-y-axis denotes the runtime in seconds and the right-y-axis (and the bars) show the number of MUPs. All three algorithms had running time only slightly impacted by data set size, taking less than 100 seconds in all settings. The effort is driven more by the number of patterns, which is independent of data set size. The PATTERN-COMBINER algorithm checks the actual dataset only for the bottom layer of the pattern graph and so the data set size has no effect on most of its computation. PATTERN-BREAKER and DEEPDIVER need to check the data for computing the coverage of the intermediate nodes, so data set size does matter. However, the use of inverted indices limits the impact.

3) MUP identification - varying data dimensions: Similarly, we evaluate the scalability of the proposed algorithm as the number of attributes (d) increases. With a dataset size of one million records and the threshold set at 1%, we measured the overall runtime of all three algorithms with the dataset projected down to between 5 and 17 dimensions.

Results. In Figure 15, the x-axis denotes the number of attributes, while the left-y-axis and right-y-axis (the bars) denote the runtime in seconds and the output size, respectively. The size of the pattern graph increases exponentially with the number of attributes. The number of MUPs and the algorithm running times also increase exponentially. Still, all algorithms managed to finish in a reasonable time (under two minutes) for up to 17 attributes.

As the number of attributes increases, the number of MUPs increases exponentially, but those become the combination of more attributes. While the MUPs with fewer are harmful and important to discover, the MUPs with more attributes are too specific, and hence, less interesting. For example, while lack of coverage for Hispanic males in a dataset is an important fact to discover, not having enough married Hispanic males under the age of 20 is less harmful. Limiting the exploration level to a certain number, allows the MUP identification algorithms to scale for datasets with tens of attributes and still finding the risky MUPs. We evaluated this by limiting the MUP discovery level in Figure 16 while using DEEPDIVER for the identification. As observed in the figure, the algorithm was able to quickly find MUPs of up to level 2 (the MUPs that are the combinations of one or two attributes) for even 35 attributes in around 10 sec.

4) Coverage enhancement - varying threshold: Recall that the objective is to identify the minimum additional data to collect, such that after the data collection the maximum coverage level is not less than λ, i.e. there are no uncovered patterns on or above a given level λ. Setting the number of items to 1M in the AirBnB dataset and number of attributes to 13, we vary the threshold rate from 10^{-6} to 0.01 while choosing different maximum coverage levels from 3 to 6.

Results. Figure 17 represents the experiment results. The x-axis shows threshold and the y-axis provides the runtime in seconds. First, the single blue triangular tick mark in the top-left of the plot shows the only setting for which the naïve algorithm finished within the time limit. GREEDY, on the other hand, finished in a few seconds for all settings. The next observation is that, as expected, the runtime of the GREEDY algorithm increases by the level; that is because it needs to collect more data points to ensure that there is no uncovered pattern on or above level λ, i.e., $\forall P \in \mathcal{M} : \ \ell(P) \geq \lambda$. Also, as the threshold rate increases the MUPs move to the top of the pattern graph. Therefore, more regions in the space become uncovered and more data points are required to guarantee the given maximum coverage level. As a result, the algorithm's runtime increases by the threshold.

5) Coverage enhancement - varying data dimensions: Lastly, we study the effect of the number of attributes on the performance of GREEDY, as well as input and output sizes. Using the AirBnB dataset, while setting the number of items to 1M and the threshold to %1, we vary the number of attributes from 5 to 35, and the max. coverage level from 3 to 6.

Results. Figure 18 shows the runtime of the algorithm, while Figure 19 provides information about the input and output sizes. Here, by the input size, we refer to the number of uncovered patterns (to cover) at the given level λ while the output size is the number of additional data points to collect. First, as explained above, increasing the maximum coverage level increases the runtime of the algorithm, as the output size increases. This is also reflected in Figure 19, as for a fixed number of attributes, both the input and output size increase in orders of magnitude. Similarly, increasing the number of attributes increases the size of the pattern graph exponentially, and also does the algorithm runtime (Figure 18) and the output size (Figure 19). Still, recall that lack of coverage for the patterns that are the combination of a few attribute

values (having smaller levels) is more harmful than the ones in the form of the combination of several attribute values. Looking at Figure 18, while solving the coverage enhancement problem for larger levels takes more time, the algorithm has a reasonable performance for resolving the lack of coverage for smaller values of maximum coverage level. Finally, in Figure 19, applying the greedy approximation algorithm, the output sizes are significantly smaller than the input sizes for each setting. That is because every value combination in the output hits multiple uncovered patterns in the input.

VI. Related Work

Diversity, as a general term for capturing the quality of a collection of items on the variety of its constituent elements [8], is an important issue in a wide range of contexts, including social science [16], political science [17], information retrieval [18], and big data environments and data ethics [19], [8]. Facility dispersion problems [20] tend to disperse a set of points such that the minimum or average distance between the pair of points is maximized. Also, techniques such as determinantal point process (DPP) have been used for diverse sampling [21], [22]. A recent work [23] considers diversity as the entropy over one discrete low-cardinality attribute. Our definition of coverage can be seen as a generalization of this, defined over combinations of multiple attributes.

The rich body of work on sampling, especially in the database community, aims to draw samples from a large database [24], [25]. Our goal in this paper is to ensure that a given dataset (often called as "found data") is appropriate for use in a data science task. The dataset could be collected independently, through a process on which the data scientist have limited, or no, control. This is different from sampling.

Technically speaking, there are similarities between the algorithms provided in this paper and the classical powerset lattice and combinatorial set enumeration problems [11], such as data cube modeling [26], frequent item-sets and association rule mining [12], data profiling [27], recommendation systems [28], and data cleaning [29]. While such work, and the algorithms such as apriori, traverse over the powerset lattice, our problem is modeled as the traversal over the pattern graph which has a different structure (and properties) compared to a powerset lattice. Hence, those techniques cannot be directly applied here. We provided some rules for traversing the pattern graph that are inspired from the set enumeration tree [11], one-to-all broadcast in a hypercube [30], and lattice traversal heuristic proposed in [27]. In § IV, we modeled the data collection problem as a hitting set instance (an equivalent of the set cover problem). Further details about this fundamental problem can be found in references such as [13], [31].

VII. Final Remarks

In this paper, we studied lack of coverage as a risk to using a dataset for analysis. Lack of coverage in the dataset may cause errors in outcomes, including algorithmic racism. Defining the coverage over multiple categorical attributes, we developed techniques for identifying the spots not properly covered by data to help the dataset users; we also proposed techniques to help the dataset owners resolve the coverage issues by additional data collection. Comprehensive experiments over real datasets demonstrated the validity of our proposal.

Following ideas such as [14], in MUP identification problem, we considered a fixed threshold across different value combinations, representing "minor subgroups". We consider further investigations on identifying threshold value and minor subgroups, as well as other alternatives for future work.

VIII. Acknowledgements

This work was supported by NSF Grant No. 1741022. We are grateful to the University of Toronto, Department of Computer Science, and Dr. Nick Koudas for the AirBnB dataset.

References

[1] M. Mulshine. A major flaw in google's algorithm allegedly tagged two black people's faces with the word 'gorillas'. Business Insider, 2015.
[2] Adam Rose. Are face-detection cameras racist? Time Business, 2010.
[3] Mallory Simon. HP looking into claim webcams can't see black people. CNN, 2009.
[4] Tess Townsend. Most engineers are white and so are the faces they use to train software. Recode, 2017.
[5] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. Machine bias: Risk assessments in criminal sentencing. ProPublica, 5/23/2016.
[6] Alex Hern. Google's solution to accidental algorithmic racism: ban gorillas. The Guardian, 2018.
[7] Irene Chen, Fredrik D Johansson, and David Sontag. Why is my classifier discriminatory? In *NeurIPS*, 2018.
[8] Marina Drosou, HV Jagadish, Evaggelia Pitoura, and Julia Stoyanovich. Diversity in big data: A review. *Big data*, 5(2), 2017.
[9] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *ECML PKDD*, 2013.
[10] Ke Yang, Julia Stoyanovich, Abolfazl Asudeh, Bill Howe, HV Jagadish, and Gerome Miklau. A nutritional label for rankings. In *SIGMOD*, 2018.
[11] Ron Rymon. Search through systematic set enumeration. Technical report, University of Pennsylvania, 1992.
[12] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.
[13] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.
[14] Seymour Sudman. Applied sampling. *Academic Press New York*, 1976.
[15] Doug Cutting and Jan Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR*, 1989.
[16] Edward H Simpson. Measurement of diversity. *Nature*, 163(4148), 1949.
[17] James Surowiecki. *The wisdom of crowds*. Anchor, 2005.
[18] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *WSDM*, pages 5–14. ACM, 2009.
[19] Solon Barocas and Andrew D Selbst. Big data's disparate impact. *Cal. L. Rev.*, 104:671, 2016.
[20] SS Ravi, Daniel J Rosenkrantz, and Giri Kumar Tayi. Facility dispersion problems: Heuristics and special cases. In *WADS*. Springer, 1991.
[21] Alex Kulesza, Ben Taskar, et al. Determinantal point processes for machine learning. *Foundations and Trends in ML*, 5(2–3), 2012.
[22] N. Anari, Sh. O. Gharan, and A. Rezaei. Monte carlo markov chain algorithms for sampling strongly rayleigh distributions and determinantal point processes. In *COLT*, pages 103–115, 2016.
[23] L Elisa Celis, Amit Deshpande, Tarun Kathuria, and Nisheeth K Vishnoi. How to be fair and diverse? *CoRR*, abs/1610.07183, 2016.
[24] Frank Olken and Doron Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42, 1995.
[25] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2011.
[26] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.
[27] A. Heise, J.A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 2013.

[28] Abolfazl Asudeh, Azade Nazi, Nick Koudas, and Gautam Das. Assisting service providers in peer-to-peer marketplaces: Maximizing gain over flexible attributes. *CoRR*, abs/1705.03028, 2017.
[29] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, 2016.
[30] D.P. Bertsekas, C. Özveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. *JPDC*, 11(4), 1991.
[31] Dorit S Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

APPENDIX

A. Coverage Computation

Computing the coverage of a pattern is a key operation for identifying the MUPs. Therefore, in this subsection, we design the coverage oracle cov, that given a pattern P (as well as the dataset $\mathcal{D}$), returns $cov(P)$.

The direct implementation of the oracle passes through the dataset once, and follows Definition 2, literally. Instead, we use inverted indices [15]. For every value v_j for an attribute A_i, we consider a bit vector $I_{i,j}$. Also, we aggregate the items with the same value combinations. Let D be the set of unique value combinations in $\mathcal{D}$ while for each entry in D we maintain the number of items in $\mathcal{D}$ with that value combination. The k-th bit of a vector $I_{i,j}$ is 1, if $D_k[i] = v_j$ (and is 0 otherwise). In order to compute the coverage of a pattern P, it applies the binary AND operations between the corresponding vectors for the deterministic elements in P and takes the dot product of the result vector with the count vector that shows the number of items matching each value combination. For instance, consider Example 1. The following are the bit vectors and the count vector for this example:

vid	000	001	010	011	vid	000	001	010	011	vid	000	001	010	011
$v_{1,0}$	1	1	1	1	$v_{2,0}$	1	1	0	0	$v_{3,0}$	1	0	1	0
$v_{1,1}$	0	0	0	0	$v_{2,1}$	0	0	1	1	$v_{3,1}$	0	1	0	1
cnt	1	2	1	1										

Then the coverage of the pattern $0X1$, for example, is calculated by applying the binary AND operation on the vectors $v_{1,1}$ and $v_{3,0}$ and taking the dot product of the result with the *cnt* vector. As a result, $cov(0X1) = 3$.

The total number of bit vectors are cd, as one vector is maintained for each attribute value. Since the length of each bit vector is in $O(n)$, the storage requirement for the bit vectors is $O(cdn)$.

B. Efficient Dominance Checking

Due to the large amount of node visits, efficient MUP dominance checking is critical. A naïve design of this operation is a simple explorative search among all MUPs; this, however, can be expensive given the potentially large number of MUPs.

Instead, similar to Appendix A, we use inverted indices. We create an inverted index for each value of each attribute A_i. For each attribute we also consider an inverted index for the patterns that have non-deterministic elements on that attribute. For each attribute value, we consider a bit vector of size of the current set of discovered MUPs. When a new MUP is identified, a new bit of 1 is added to the bit vectors corresponding to the values of the elements of the new MUP; the rest of bit vectors are appended by 0.

To check if a pattern P dominates the current set of MUPs $\mathcal{M}$, we iterate through each value of pattern P and skip the non-deterministic elements as the patterns dominated by P can have any value on those. Applying the binary AND operation between the bit vectors for the values of the deterministic elements identify if P dominates $\mathcal{M}$: if there is a non-zero element in the result vector, there exists a pattern P' that is dominated by P and, therefore, P dominates $\mathcal{M}$.

Similarly, to check if a pattern P is dominated by the current MUPs $\mathcal{M}$, we parse each value of P but collect a different set of bit vectors: (i) for the non-deterministic elements (X values) in P, we collect the corresponding bit vectors for the non-deterministic elements and (ii) for the deterministic elements in P, we collect the result of bitwise OR operation between the bit vector for this attribute value and the bit vector for value X in this attribute. In the end, we perform the binary AND operation over the collected vectors: if the result is all zero, $\mathcal{M}$ does not dominate P; otherwise, $\mathcal{M}$ dominates P.

In addition, since we are only interested to know if there is a 1 in the result of the AND operations, we apply an early stop strategy by conducting the operation word by word and terminating it as soon as a 1 is observed in the results.

C. Uncovered patterns to hit for maximum coverage level assurance

Here we discuss the set of patterns we need to hit (c.f. § IV) in order to guarantee the maximum coverage level for a user-specified value λ. In the first glance, applying the hitting set on the MUPs with level at most λ will cover them all and, thus, assures the maximum coverage level of λ. But, this is not correct. To further explain it, consider again Example 2 and the collection of its MUPs P_1 to P_7. One may notice that the value combinations 02011, 02111, and 10201 (discovered by the greedy hitting set for P_1 to P_6) also cover the pattern P_7 : X020X, which means all of the MUPs in the example are covered. Consider the pattern P : 1X11X in level $\ell = 3$. First, this is uncovered, as it is a child of the MUP P_5 : XX11X. None of the combinations 02011, 02111, or 10201 match P. This means that there exists at least one pattern at level $\ell = 3$ that remains uncovered. This contradicts the claim that the maximum coverage level is $\ell = 3$.

In fact, every MUP represents the set of uncovered patterns at lower levels that are connected to it. Covering the MUP may not satisfy covering those patterns as well. As a result, even though the MUP itself is covered, some of its children may still not be.

On the other hand, if all the (not necessarily maximal) uncovered patterns at a level λ are covered, all the more general patterns with level $\leq \lambda$ are also covered. That is because every (more general) pattern P at a level less than λ represents a set of value combinations that are the superset of the matches for at least one uncovered pattern P' at level ℓ (for the ease of explanation, we say P' is a "subset" pattern for P). Thus, collecting a value combination that matches P' will also match P.

As a result, in order to guarantee the coverage at level λ, it is enough to apply the greedy hitting set as explained in § IV on the set of uncovered patterns at level λ. But, we first need

to generate all uncovered patterns at level λ. To do so, for every MUP $P \in \mathcal{M}$ where $\ell(P) \leq \lambda$, we find its descendants at level λ by replacing $(\lambda - \ell)$ non-deterministic elements (X's) with deterministic values. For example, in Example 2, the subset patterns for P_1 : XX01X at level $\ell = 3$ are: 0X01X, 1X01X, X001X, X101X, X201X, XX010, and XX011.