Slice Finder: Automated Data Slicing for Model Validation

Yeounoh Chung*
Brown University
yeounoh_chung@brown.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

Neoklis Polyzotis
Google Research
npolyzotis@google.com

Steven Euijong Whang*
KAIST
swhang@kaist.ac.kr

arXiv:1807.06068v2 [cs.DB] 1 Aug 2018

***Abstract*—As machine learning (ML) systems become democratized, it becomes increasingly important to help users easily debug their models. However, current data tools are still primitive when it comes to helping users trace model performance problems all the way to the data. We focus on the particular problem of slicing data to identify subsets of the validation data where the model performs poorly. This is an important problem in model validation because the overall model performance can fail to reflect that of the smaller subsets, and slicing allows users to analyze the model performance on a more granular-level. Unlike general techniques (e.g., clustering) that can find arbitrary slices, our goal is to find interpretable slices (which are easier to take action compared to arbitrary subsets) that are problematic and large. We propose** Slice Finder**, which is an interactive framework for identifying such slices using statistical techniques. Applications include diagnosing model fairness and fraud detection, where identifying slices that are interpretable to humans is crucial.**

***Index Terms*—data slicing, model validation**

I. Introduction

Machine learning (ML) systems [8] are becoming more prevalent thanks to a vast number of success stories. However, the data tools for interpreting and debugging models have not caught up yet and many important challenges exist to improve our model understanding after training [14]. One such key problem is to understand if a model performs poorly on certain parts of the data, hereafter also referred to as a *slice*.

Example 1. *Consider a Random Forest classifier that predicts whether a person's income is above or below $50,000 (UCI Census data [29]). Looking at Table I, the overall metrics may be considered acceptable, since the overall log loss (a widely-used loss metric for binary classification problem) is low for all the data (see the "All" row). However, the individual slices tell a different story. When slicing data by gender, the model is more accurate for* Female *than* Male *(the* effect size *defined in Section II captures this relation by measuring the normalized loss metric difference between the* Male *slice and its counterpart, the* Female *slice). The* Local-gov White *slice is interesting because the average loss metric is on par with* Male*, but the effect size is much smaller (by convention, $d \leq 0.3$ is small). A small effect size means that the loss metric on* Local-gov White *is similar to the loss metric on other demographics (defined as* counterparts *in Section II). Hence, if the log loss of a slice and that of the counterpart are not acceptable, then it is likely that the model is bad overall, not just on a particular subset. Lastly, we see that people with higher education degrees (*Bachelors*,* Masters*,* Doctorate*) suffer from worse model performance and their losses are higher than their counterparts and thus have higher error concentration. Thus, slices with high effect size are important for model validation, to make sure that the model do not under-perform on certain parts of the data.*

Slice	Log Loss	Size	Effect Size
All	0.35	30k	n/a
Sex = Male	0.41	20k	0.47
Sex = Female	0.21	10k	-0.47
Workclass = Local-gov Race = White	0.43	1.7k	0.19
Education = HS-grad	0.32	9.8k	-0.09
Education = Bachelors	0.44	0.5k	0.27
Education = Masters	0.49	1.6k	0.40
Education = Doctorate	0.47	5k	0.32

TABLE I: UCI Census data slices for Example 1

The problem is that the overall model performance can fail to reflect that of smaller data slices. Thus, it is important that the performance of a model is analyzed on a more granular level. While a well-known problem [31], current techniques to determine under-performing slices largely rely on domain experts to define important sub-populations (or at least specify a feature dimension to slice by) [4], [23]. Unfortunately, ML practitioners do not necessary have the domain expertise to know all important under-performing slices in advance, even after spending a significant amount of time exploring the data. In this problem context, enumerating all possible data slices and validating model performance for each is not practical due to the sheer number of possible slices. Worse yet, simply searching for the most under-performing slices can be misleading because the model performance on smaller slices can be noisy, and without any safeguard, this leads to slices that are too small for meaningful impact on the model quality or that are false discoveries (i.e., non-problematic slices appearing as problematic). Ideally, we want to identify the largest and true problematic slices from the smaller slices that are not fully reflected on by the overall model performance metric.

There are more generic clustering-based algorithms in

*Work done at Google Research.

model understanding [27], [32], [33] that group similar examples together as clusters and analyze model behavior locally within each cluster. Similarly, we can cluster similar examples and treat each cluster as an arbitrary data slice; if a model under-performs on any of the slices, then the user can analyze the examples within. However, clusters of similar examples can still have high variance and high cardinality of feature values, which are hard to summarize and interpret. In comparison, a data slice with a few common feature values (e.g., Female slice contains all examples with Sex = Female) is much easier to interpret. In practice, validating and reporting model performance on interpretable slices are much more useful than validating on arbitrary (non-interpretable) slices (e.g., a cluster of similar examples with mixed properties).

A good technique to detect problematic slices for model validation thus needs to find easy-to-understand subsets of data and ensure that the model performance on the subsets is meaningful and not attributed to chance. Each problematic slice should be immediately understandable to a human without the guesswork. The problematic slices should also be large enough so that their impact on the overall model quality is non-negligible. Since the model may have a high variance in its prediction quality, we also need to be careful not to choose slices that are false discoveries. Finally, since the slices have an exponentially large search space, it is infeasible to manually go though each slice. Instead, we would like to guide the user to a handful of slices that satisfy the conditions above. In this paper we propose Slice Finder, which efficiently discovers large possibly-overlapping slices that are both interpretable and actually problematic.

A slice is defined as a conjunction of feature-value pairs where having fewer features is considered more interpretable. A problematic slice is identified based on testing of a significant difference of model performance metrics (e.g., loss function) of the slice and its counterpart. That is, we treat each problematic slice as a hypothesis and perform a principled hypothesis testing to check if it is a true problematic slice and not a false discovery by chance. We discuss the details in Section II. One problem with performing many statistical tests (due to a large number of candidate slices) is an increased number of false positives. This is what is also known as Multiple Comparisons Problem (MCP) [9]: imagine a test of Type-I error (false positive: recommending a non-problematic slice as problematic) rate of 0.05 (a common α-level for statistical significance testing); the probability of having any false positives blows up exponentially with the number of comparisons (e.g., $1-(1-0.05)^8 = 0.34$, even for just 8 tests, but then, we may end up exploring hundreds and thousands of slices even for a modest number of examples). We address this issue in Section III-B.

In addition to testing, the slices found by Slice Finder can be used to evaluate model fairness or in applications such as fraud detection, business analytics, and anomaly detection, to name a few. While there are many definitions for fairness, a common one is that a model performs poorly (e.g., lower accuracy) on certain sensitive features (which define the slices), but not on others. Fraud detection also involves identifying classes of activities where a model is not performing as well as it previously did. For example, some fraudsters may have gamed the system with unauthorized transactions. In business analytics, finding the most promising marketing cohorts can be viewed as a data slicing problem. Although Slice Finder evaluates each slice based on its losses on a model, we can also generalize the data slicing problem where we assume a general scoring function to assess the significance of a slice. For example, data validation is the process of identifying training or validation examples that contain errors (e.g., values are out of range, features are missing, and so on). By scoring each slice based on the number or type of errors it contains, it is possible to summarize the data errors through a few interpretable slices rather than showing users an exhaustive list of all erroneous examples.

In summary, we make the following contributions:

- We define the data slicing problem and the use of hypothesis testing for problematic slice identification (Section II) and false discovery control (Section III-B).
- We describe the Slice Finder system and propose three automated data slicing approaches, including a naïve clustering-based approach as a baseline for automated data slicing (Section III).
- We present model fairness as a representative use case for Slice Finder (Section IV).
- We evaluate the three automated data slicing approaches using real datasets (Section V).

II. Data Slicing Problem

A. Preliminaries

We assume a dataset D with n examples and a model h that needs to be tested. Following common practice, we assume that each example $x_F^{(i)}$ contains features $F = \{F_1, F_2, ..., F_m\}$ where each feature F_j (e.g., country) has a list of values (e.g., {US, DE}) or discretized numeric value ranges (e.g., {[0, 50), [50, 100)}). We also have a ground truth label $y^{(i)}$ for each example, such that $D = \{(x_F^{(1)}, y^{(1)}), (x_F^{(2)}, y^{(2)}), ..., (x_F^{(n)}, y^{(n)})\}$. The test model h is an arbitrary function that maps an input example to a prediction, and the goal is to validate if h is working properly for different subsets of the data. For ease of exposition, we focus on a binary classification problem (e.g., UCI Census income classification) with h that takes an example $x_F^{(i)}$ and outputs a prediction $h(x_F^{(i)})$ of the true label $y^{(i)} \in \{0, 1\}$ (e.g., a person's income is above or below $50,000).

A *slice* S is a subset of examples in D with common features and can be described as a conjunction of the common feature-value pairs $\bigwedge_j F_j \ op \ v_j$ where the F_j's are distinct (e.g., country = DE $\wedge$ gender = Male), and op can be one of $=, <, \leq, \geq$, or $>$. For numeric features, we can discretize their values (e.g., quantiles or equi-height bins) and generate ranges so that they are effectively categorical features (e.g., age = [20,30)). Numeric features with large domains tend to have fewer examples per value, and hence do not appear

as significant. By discretizing numeric features into a set of continuous ranges, we can effectively avoid searching through tiny slices of minimal impact on model quality and group them to more sizable and meaningful slices.

We also assume a classification loss function $\psi(S, h)$ that returns a performance score for a set of examples by comparing h's prediction $h(x_F^{(i)})$ with the true label $y^{(i)}$. A common classification loss function is logarithmic loss (*log loss*), which in case of binary classification is defined as:

$$-\frac{1}{n}\sum_{(x_F^{(i)},y^{(i)})\in S}[y^{(i)}\ \ln\ h(x_F^{(i)}) + (1-y^{(i)})\ \ln\ (1-h(x_F^{(i)}))]$$

The log loss is non-negative and grows with the number of classification errors. A perfect classifier h would have log loss of zero, and a random-guesser ($h(x) = 0.5$) log loss of $-ln(0.5) = 0.693$. Also note that our techniques and the problem setup can easily generalize to other ML problem types (e,g., multi-class classification, regression, etc.) with proper loss functions/performance metrics.

B. Problematic Slice as Hypothesis

We define a slice to be *problematic* if the classification loss function takes vastly different values between the slice and its counterpart. The counterpart slice serves as a reference to which we measure how problematic is S, and the definition depends on the problem in hand. For instance, in the most general case where user wants to validate if the model underperforms on any data slices, we define the counterpart as the complement of S ($S' = D - S$) and consider the difference $\psi(S, h) - \psi(S', h)$ assuming ψ is a loss function, such as a log loss. (The definition of counterpart can change in other scenarios as we explain later.) This effectively allows us to identify S with a higher error concentration for h (i.e., most erroneous examples are contained in S and not in S'), which should deserve the user's attention for deeper analysis.

Finding real problematic slices for model validation is non-trivial, mainly because it requires to balance between the magnitude of the difference in loss function values and the size of the slice. That is, a problematic slice must contain more erroneous examples (i.e., model performs worse) in relation to the rest of data, and it should also be large enough to have a meaningful impact on model quality. In some applications, each example may also have a weight, which reflects its importance. As a result, a slice with few examples can still be considered important due to its large weight sum. In the remainder of the paper, we will assume that weights are always 1, but extending to varying weights is straightforward. Interestingly, larger slices tend to have performance metric (i.e., loss function value) similar to that of the overall dataset with a smaller variance; thus, the difference tends to be smaller. Notice that we are looking at a one-sided difference $\psi(S, h) - \psi(S', h)$, so any large negative difference values with extreme counterpart $\psi(S', h)$ are not of interest. On the other hand, if a larger slice has high positive $\psi(S, h) - \psi(S', h)$, then the signal is more likely to be real and deserves the user's attention.

Based on the previous points, one possible approach to identifying problematic slices would be to rank each slice based on some heuristic combination of its size and difference in average losses. However, such a heuristic is hard to tune and not even practical, assuming that we want a solution that can work with any validation data, model and loss functions.

Our solution is to instead treat each problematic slice as a hypothesis and perform a testing for the strength of the signal ($\psi(S, h) - \psi(S', h)$) and its statistical significance: *is the observed difference simply by chance or for real?*. The definition of problematic slices with respect to its counterparts is general and thus applicable across domains. In addition, the definition naturally translates into hypothesis testing with a null and an alternative hypothesis:

$$H_o : \psi(S, h) <= \psi(S', h)$$

$$H_a : \psi(S, h) > \psi(S', h)$$

The test accepts S as problematic if it has a large difference and large enough support (the number of examples). The testing is performed based on a standardized score ϕ of the difference by the pooled standard deviations of $\psi(S, h)$ and $\psi(S', h)$, σ_S and $\sigma_{S'}$ respectively (a.k.a., *effect size* [1]):

$$\sqrt{2} \times \frac{\psi(S, h) - \psi(S', h)}{\sqrt{\sigma_S^2 + \sigma_{S'}^2}} \tag{1}$$

The effect size directly measures the strength of the signal (i.e., how problematic the slice is) with respect to the distribution of the loss differences, and the testing ensures that the observed signal is not by chance. The effect size is also a standardized score, for which we consider 0.2 to be small, 0.5 medium, 0.8 large, and 1.3 very large (*Cohen's convention* [12]). Slice Finder brings the user's attention to a handful of the largest problematic slices, by taking all problematic slices S with effect size $\phi > T$ and ranking them by size (number of examples). Slice Finder provides a slider for the effect size threshold T for user to explore slices with different degrees of problematic-ness (Section III-C). It is also important to note that the power (i.e., *probability of detecting false positives*) of testing depletes quickly as we perform numerous tests (a lot of candidate slices); we address this issue in Section III-B.

Lastly, our definition of problematic slice is also applicable to another common scenario, where a modeler wants to check if any sub-populations would experience degraded performance if she switches h to h' (i.e., is a new model h' safe to push?). In this case, we simply evaluate S with two different models and consider $\psi(S, h') - \psi(S, h)$, with an alternative hypothesis, $H_a : \psi(S, h') > \psi(S, h)$. Here the counterpart of S using model h is the same slice S using model h'.

C. Data Slicing Problem

The goal of Slice Finder is to identify a handful (e.g., top-K) of the largest problematic slices. Larger problematic slices are preferable because they carry more examples for illustrating the model quality issue, and thus, have more

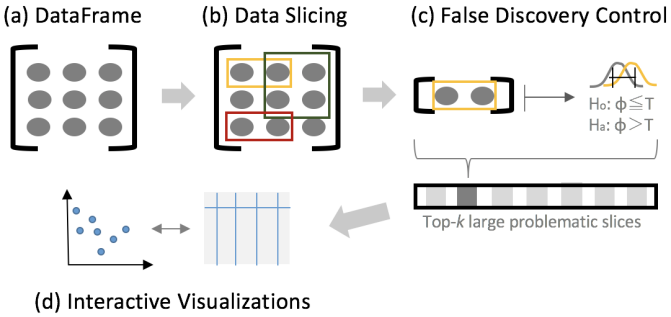


Fig. 1: The Slice Finder architecture: (a) Data is loaded into a Pandas DataFrame, and (b) Slice Finder perform automated data slicing to find top-k large problematic slices. (c) Slice Finder uses a false discovery control procedure to include only statistically significant problematic slices (d) for interactive visualizations.

impact on the model quality. On the other hand, the model performance on a tiny slice does not provide much information since it may well be statistically insignificant (i.e., due to noise) and debugging the model on such a tiny slice would not change much the overall model quality. In addition, fewer features are preferred to make the problematic slices more interpretable. For example, country = DE is more interpretable than country = DE $\wedge$ age = 20-40 $\wedge$ zip = 12345.

Problem 1. *Given a positive integer K and threshold T, the data slicing problem is defined as finding the top-K largest slices such that:*

- *Each slice has an effect size at least T,*
- *The effect size is statistically significant,*
- *No slice can be replaced with another with the same size, but with fewer features.*

Note that the top-K slices do not have to be distinct, e.g., country = DE and education = Bachelors overlap in the demographic of Germany with a Bachelors degree.

III. System Architecture

Underlying the Slice Finder system is an extensible architecture that combines automated data slicing and interactive visualization tools. The system is implemented in Python (for a single node processing and the front-end) and C++ (to run the Slice Finder lattice search on a distributed processing framework such as Flume [10]).

Slice Finder loads the validation data set into a Pandas DataFrame [30]. The DataFrame supports indexing individual examples, and each data slice keeps a subset of indices instead of a copy of the actual data examples. Slice Finder provides basic slice operators (e.g., intersect and union) based on the indices; only when evaluating the ML model on a given slice Slice Finder accesses the actual data by the indices to test the model. The Pandas library also provides a number of options to deal with dirty data and missing values, and for the work presented here, we dropped *NaN* (missing values) or any values that deviate from the column types.

Once data is loaded into a DataFrame, Slice Finder processes the data to identify the problematic slices and allow the user to explore them. This process comprises three major components, summarized below.

Slice Finder searches for problematic slices either by training a CART decision tree around mis-classified examples or by performing a more exhaustive search on a lattice of slices. Both search strategies progress in a top-down manner until they find top-k large problematic slices with $\phi \geq T$. The decision tree approach materializes the tree model and traverses to extract nodes for different request queries (with different k and T). In lattice searching, Slice Finder materializes all the candidate slices, even non-problematic slices. This allows Slice Finder to quickly respond to a new request with different T or continue searching with more filter clauses.

As Slice Finder searches through a large number of slices, some slices might appear problematic by chance (i.e., multiple comparisons problem [17]). Slice Finder controls such a risk, by applying a marginal false discovery rate (mFDR) controlling procedure [17]. Slice Finder compiles a final top-k recommendation list with only statistically significant problematic slices.

Lastly, even a handful of problematic slices can still be overwhelmingly large, since the user needs to take an action (e.g., deeper analyses or model debugging) on each slice. Hence, it is important to enable the user to quickly browse through the slices by their impacts (size) and scores (effect size). To this end, Slice Finder allows the user to explore the recommended slices with interactive visualization tools.

The following subsections describe each component in detail.

A. Automated Data Slicing

As mentioned earlier, the goal of this component is to automatically identify problematic slices for model validation. To motivate the development of the two techniques that we mentioned (decision trees and lattice search), let us first consider a simple baseline approach that identifies the problematic slices through clustering. And then, we discuss two automated data slicing approaches used in Slice Finder that improve on the clustering approach.

1) Clustering: The idea is to cluster similar examples together and take each cluster as an arbitrary data slice. If a test model fails on any of the slices, then the user can examine the data examples within or run a more complex analysis to fix the problem. This is an intuitive way to understand the model and its behavior (e.g., predictions) [27], [32], [33]; we can take a similar approach to the automated data slicing problem. The hope is that similar examples would behave similarly even in terms of data or model issues.

Clustering is a reasonable baseline due to its ease of use, but it has major drawbacks: first, it is hard to cluster and explain high dimensional data. We can reduce the dimensionality using principled component analysis (PCA) before clustering, but many features of clustered examples (in its original feature vector) still have high variance or high cardinality of values.

Unlike an actual data slice filtered by certain features, this is hard to interpret unless the user can manually go through the examples and summarize the data in a meaningful way. Second, the user has to specify the number of clusters, which affects crucially the quality of clusters in both metrics and size. As we want slices that are problematic and large (more impact for model quality), this is a key parameter which is hard to tune.

The two techniques that we present next overcome these deficiencies of clustering. The first technique is based on decision-trees that capture the distribution of classification results. Here the effect sizes are large, but the slices may be smaller as a result. In contrast, the second technique, called lattice searching, focuses on slices that are neither too small nor large, but have large-enough effect sizes.

2) Decision Tree Training: To identify more interpretable problematic slices, we train a decision tree that can classify which slices are problematic. The output is a partitioning of the examples into the slices defined by the tree. For example, a decision tree could produce the slices $\{A > v,\ A \leq v\ \&\ B > w,\ A \leq v\ \&\ B \leq w\}$. For numeric features, this kind of partitioning is natural. For categorical features, a common approach is to use one-hot encoding where all possible values are mapped to columns, and the selected value results in the corresponding column to have a value 1.

To use a decision tree, we first identify the bottom-most problematic slices (leaves) with the highest effect size (i.e., highest error concentration). Then we can go up the decision tree to find larger (and more interpretable) slices that generalize the problematic slices, which still have effect size larger than a user-specified effect size threshold, T.

The advantage of decision trees is that they have a natural interpretation, since the leaves correspond directly to slices. The downside of using a tree is that it only finds non-overlapping slices that are problematic. In addition, if the decision tree gets too deep with many levels, then it starts to become uninterpretable as well [18].

The Decision Tree approach can be viewed as "greedy" because it optimizes on the classification results and is thus not designed to exhaustively find all problematic slices according to Definition 1. For example, if some feature is split on the root node, then it will be difficult to find single-feature slices for other features. In addition, a decision tree always partitions the data, so even if there are two problematic slices that overlap, at most one of them will be found. Hence, a more exhaustive approach is needed to ensure all possibly-overlapping problematic slices are found.

3) Lattice Searching: The lattice searching approach considers a larger search space where the slices form a lattice, and problematic slices can overlap with one another. We assume that slices only have equality predicates, e.g., $\bigwedge_i F_i = v_i$. In contrast with the decision tree training approach, lattice searching can be more expensive because it searches overlapping slices.

Figure 2 illustrates how the slices are organized as a lattice. The key intuition is to perform a breadth-first search and efficiently identify problematic slices as shown in Algorithm 1.

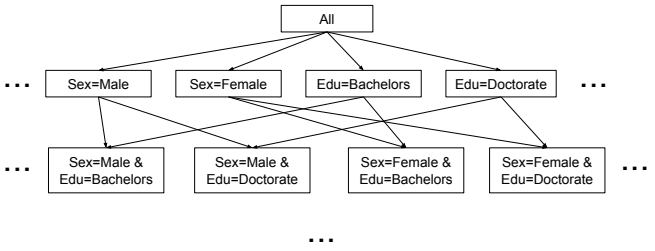


Fig. 2: A lattice hierarchy of slices. In contrast with a decision tree, the search is more exhaustive and covers all possible feature combinations.

ALGORITHM 1: Lattice Searching Algorithm

Input : Lattice L, maximum number of slices to return K, effect size threshold T, the set of all possible features F
Output: Problematic slices S

```
1  S = [];                                   /* problematic slices */
2  Q = PriorityQueue();   /* priority queue sorted by
     descending size and ascending number of
     features */
3  Q.push(L.root);
4  while |S| ≤ K and Q not empty do
5      s = Q.pop();                          /* s = ⋀_{i∈I} F_i = v_i */
6      if Effect_size(s) ≥ T then
7          S.append(s);
8      end
9      else
10         Q.push({⋀_{i∈I} F_i = v_i ∧ G = v | G ∈
             F − {F_1, ..., F_|I|}, v ∈ G's values});
11     end
12 end
13 return S;
```

The input is the training data, a model, and an effect size threshold T. As a pre-processing step, Slice Finder takes the training data and discretizes numeric features. For categorical features that contain too many values (e.g., IDs are unique for each example), Slice Finder uses a heuristic where it considers up to the N most frequent values and places the rest into an "other values" bucket. The possible slices of these features form a lattice where a slice S is a parent of every S with exactly one more feature-value pair.

Slice Finder finds the top-K largest problematic slices by traversing the slice lattice in a breadth-first manner using a priority queue. The priority queue contains the current slices being considered sorted by descending size and then by ascending number of features. For each slice $\bigwedge_{i \in I} F_i = v_i$ that is popped, Slice Finder checks if it has an effect size at least T. If so, the slice is added to the top-K list. Otherwise, the slice is expanded where the slices $\{\bigwedge_{i \in I} F_i = v_i \wedge G = v | G \in F - \{F_1, \ldots, F_{|I|}\}, v \in G's \text{ values}\}$ are added to the queue. Slice Finder optimizes this traversal by avoiding slices that are subsets of previously identified problematic slices. The intuition is that any subsumed (expanded) slice contains a

subset of the same exact examples of its parent and is smaller with more filter predicates (less interpretable); thus, we do not expand larger and already problematic slices. By starting from the base slices (with single filter predicate/clause) and expanding only non-problematic slices with one additional predicate at a time (i.e., top-down search from lower order slices to higher order slices), we can generate a superset of all candidate slices. This is similar to *Apriori* fast frequent itemset mining algorithm [5], where only large $(d-1)$-itemsets are joined together to generate a superset of all large d-itemsets. This process repeats until either the top-K slices have been found or there are no more slices to explore.

Example 2. *Suppose there are three features A, B, and C with the possible values* $\{a_1\}$ *and* $\{b_1, b_2\}$*, and* $\{c_1\}$*, respectively. Also say* $K = 2$*, and the effect size threshold is* T*. Initially, the priority queue* Q *contains the entire slice. This slice is popped and expanded to the slices* $A = a_1$*,* $B = b_1$*,* $B = b_2$*, and* $C = c_1$*, which are inserted back into the queue. Among them, suppose* $A = a_1$ *is the largest slice with an effect size at least* T*. Then this slice is popped from* Q *and added to the top-*K *result. Suppose that no other slice has an effect size at least* T*, but* $B = b_1$ *is the largest. This slice is then expanded to* $B = b_1 \wedge C = c_1$ *(notice that* $B = b_1 \wedge A = a_1$ *is unnecessary because it is a subset of* $A = a_1$*). If this slice has an effect size at least* T*, then the final result is [*$A = a_1$*,* $B = b_1 \wedge C = c_1$*].*

The following theorem formalizes the correctness of this algorithm for the slice-identification problem.

Theorem 1. *The* Slice Finder *slices identified by Algorithm 1 satisfy Definition 1.*

Proof. Since we only add slices with effect size at least T to the priority queue, the first condition is satisfied trivially. The second condition can be proven to hold using contradiction. Suppose a slice S that is popped from the queue has a large enough effect size, but there is another slice S' that has not yet been added to the result, but has the same size with fewer features and should have been added to the result first. However, the ancestors of this slice must have been all popped and expanded before S was popped. In addition, since S' has fewer features than S, it should have been placed before S in the queue (hence the contradiction). □

4) Scalability: Slice Finder optimizes its search by expanding the filter predicate by one additional feature/value at a time (top-down strategy). Unfortunately, this does not solve the scalability issue of the data slicing problem completely, and Slice Finder could still search through an exponential number of slices, especially for big high-dimensional data sets. To this end, Slice Finder implements two approaches that can speed up the search.

Parallelization: For lattice searching, evaluating a given model on a large number of slices one-by-one (sequentially) can be very expensive. So instead, Slice Finder distributes the slice evaluation jobs (lines 5–10 in Algorithm 1) by keeping separate priority queues Q_d for the different number of filter predicates d. The idea is that workers take slices from the current Q_d in a round-robin fashion and evaluate them asynchronously; the workers push the next candidate slices $\{\bigwedge_{i \in I} F_i = v_i \wedge G = v | G \in F - \{F_1, \ldots, F_{|I|}\}, v \in G's \text{ values}\}$ with one additional filter clause G to Q_{d+1} as they finish evaluating the slices. Once done with Q_d (i.e., Q_d is empty and $|S| \leq K$), Slice Finder moves onto the next queue Q_{d+1} and continue searching until $|S| \geq K$. Keeping slices of different d in separate queues allows multiple workers to evaluate multiple slices in parallel, without having to worry about redundant discoveries because only slices with $d+1$ predicates can be subsumed by slices with d predicates. The added memory and communication overheads are negligible, especially, with respect to the slice evaluation time.

On the other hand, for DT, our current implementation does not support parallel learning algorithms for constructing trees. But, there exist a number of highly parallelizable learning processes for decision trees [35], which Slice Finder could implement to make DT more scalable.

Sampling: We take a smaller sample to run Slice Finder if the original data set is too large. Note that the run time is linearly proportional to the size of sample, assuming that the run time for the test model is constant for each example. Taking a sample, however, comes with a cost. Namely, we run the risk of false positives (non-problematic slices that appear problematic) and false negatives (problematic slices that appear non-problematic or completely disappear from the sample) due to a decreased number of examples. Since we are interested in large slices that are more impactful to model quality, we can disregard false negatives that disappeared from the sample. Furthermore, we perform significance testing to filter slices that falsely appear as problematic or non-problematic (Section III-B).

B. False Discovery Control

As Slice Finder finds more slices for testing, there is also the danger of finding more "false positives," which are slices that are not statistically significant. Slice Finder controls false positives (Type-1 errors) in a principled fashion using α-investing [17]. Given an alpha-wealth (overall Type I error rate) α, α-investing spends this over multiple comparisons, while increasing the budget α towards the subsequent tests with each rejected hypothesis. This so called pay-out (increase in α) helps the procedure become less conservative and puts more weight on more likely to be faulty null hypotheses. More specifically, an alpha-investing rule determines the wealth for the next test in a sequence of tests. This effectively controls marginal false discovery rate at level α:

$$\frac{\mathbb{E}(V)}{\mathbb{E}(R)} \leq \alpha \tag{2}$$

Here, V is the number of false discoveries and R the number of total discoveries returned by the procedure. Slice Finder uses α-investing, mainly because it allows more interactive multiple hypothesis error control, namely, with an unspecified

number of tests in any order. On the contrary, more restricted multiple hypothesis error control techniques, such as Bonferroni correction and Benjamini-Hochberg procedure [9] fall short as they require the total number of tests m in advance or become too conservative as m grows large.

There are different α-investing policies for testing a sequence of hypotheses. In particular, our exploration strategy orders slices by their significance (t-score) and test hypotheses believed most likely to be rejected. This is called Best-foot-forward policy; we test the seemingly more significant slices with more power, and continue testing the rest only if we have left over α-wealth. The successful discovery of significant slices earns extra testing power (alpha-wealth), helping us to continue testing until there is no remaining wealth.

C. Interactive Visualization Tool

Slice Finder interacts with users through the GUI in Figure 3. **A**: On the left side is a scatter plot that shows the (size, effect size) coordinates of all slices. This gives a nice overview of top-k problematic slices, which allows the user to quickly browse through large and also problematic slices and compare slices to each other. **B**: Whenever the user hovers a mouse over a dot, the slice description, size, effect size, and metric (e.g., log loss) are displayed next to it. If a set of slices are selected, their details appear on the table on the right-hand side, **C**: On the table view, the user can sort slices by any metrics on the table.

On the bottom, **D**: Slice Finder provides configurable sliders for adjusting k and T. Slice Finder materializes all the problematic slices ($\phi \geq T$) as well the non-problematic slices ($\phi < T$) searched already. If T decreases, then we just need to reiterate the slices explored until now to find the top-K slices. If T increases, then the current slices may not be sufficient, depending on k, so we continue searching the slice lattice. This is possible because Slice Finder looks for top-k problematic slices in a top-down manner.

IV. Using Slice Finder for Model Fairness

In this section, we look at model fairness as a use case of Slice Finder where identifying problematic slices can be a preprocessing step before more sophisticated analysis on fairness on the slices.

As machine learning models are increasingly used in sensitive applications, such as predicting whether individuals will default on loans [21], commit crime [2], or survive intensive hospital care [19], it is essential to make sure the model performs equally well for all demographics to avoid discrimination. However, models may fail this property for various reasons: bias in data collection, insufficient data for certain slices, limitations in the model training, to name a few cases.

Model fairness has various definitions depending on the application and is thus non-trivial to formalize (see recent tutorial [6]). While many metrics have been proposed [15], [16], [21], [24], there is no widely-accepted standard, and some definitions are even at odds. In this paper, we focus on a relatively common definition, which is to find of data where the model performs relatively worse using some of these metrics, which fits nicely into the Slice Finder framework.

Using our definition of fairness, Slice Finder can be used to quickly identify interpretable slices that have fairness issues without having to specify the sensitive features in advance. Here, we demonstrate how Slice Finder can be used to find any unfairness of the model with equalized odds [21]. Namely, we explain how our definition of problematic slice using effect size also conforms to the definition of equalized odds. Slice Finder is also generic and supports any fairness metric that can be expressed as a scoring function. Any subsequent analysis of fairness on these slices can be done afterwards.

Equalized odds requires a predictor $\hat{Y}$ (e.g., a classification model h in our case) to be independent of protected or sensitive feature values $A \in \{0, 1\}$ (e.g., gender = Male or gender = Female) conditional on the true outcome Y [21]. In binary classification ($y \in \{0, 1\}$), this is equivalent to:

$$Pr\{\hat{Y} = 1 | A = 0, Y = y\} = Pr\{\hat{Y} = 1 | A = 1, Y = y\} \tag{3}$$

Notice that equalized odds is essentially matching true positive rates (*tpr*) in case of $y = 1$ or false negative rates (*fnr*) otherwise.

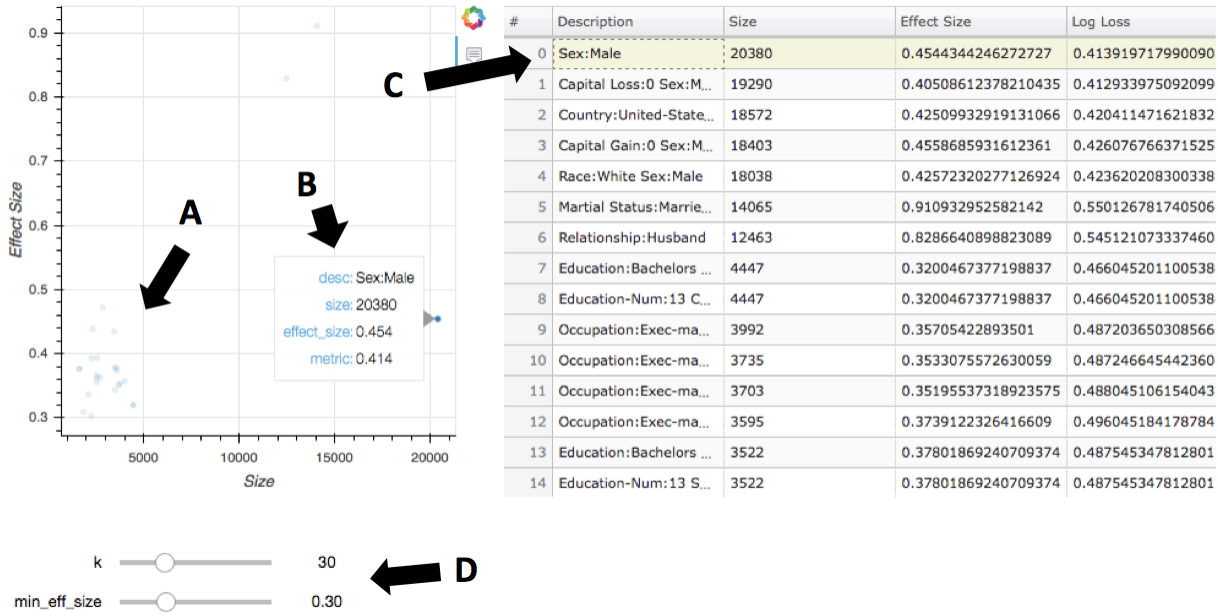
Slice Finder can be used to identify slices where the model is potentially discriminatory; an ML practitioner can easily identify feature dimensions of the data, without having to manually consider all feature value pair combinations, on which a deeper analysis and potential model fairness adjustments are needed. The problematic slices with $\phi > T$ suffer from higher loss (lower model accuracy in case of log loss) compared to the counterparts. If one group is enjoying a better rate of accuracy over the other, then it is a good indication that the model is biased. Namely, accuracy is a weighted sum of *tpr* and *fnr* by their proportions, and thus, a difference in accuracy means there are differences in *tpr* and false positive rate ($fpr = 1 - tpr$), assuming there are any positive examples. As equalized odds requires matching *tpr* and *fpr* between the two demographics (a slice and its counterpart), Slice Finder using log loss ψ can identify slices to show that the model is potentially discriminatory. In case of the gender = Male slice above, we flag this as a signal for discriminatory model behavior because the slice is defined over a sensitive feature and has a high effect size.

There are other standards, but equalized odds ensures that the prediction is non-discriminatory with respect to a specified protected attribute (e.g., gender), without sacrificing the target utility (i.e., maximizing model performance) too much [21].

V. Experiments

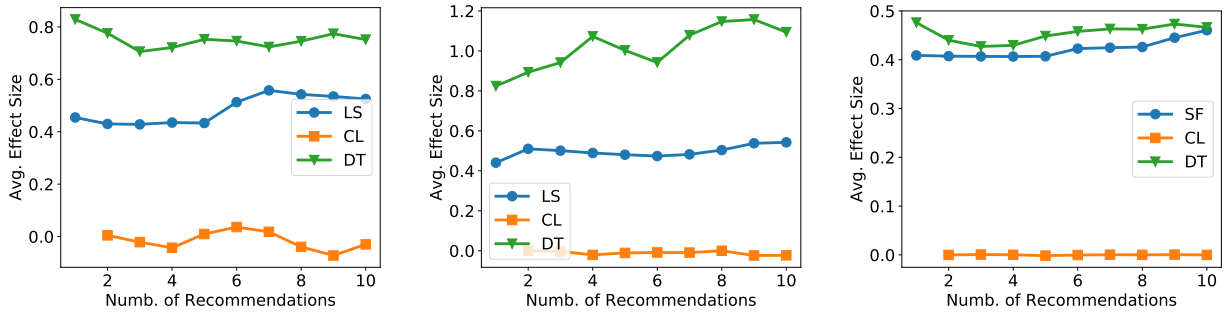
In this section, we compare the two Slice Finder approaches (decision tree and lattice search) with the baseline (clustering-based approach). We address the following key questions using both real-world and simulated ML problems:

- What are the trade-offs between the three automated slicing approaches?



#	Description	Size	Effect Size	Log Loss
0	Sex:Male	20380	0.4544344246272727	0.413919717990090
1	Capital Loss:0 Sex:M...	19290	0.40508612378210435	0.412933975092099
2	Country:United-State...	18572	0.42509932919131066	0.420411471621832
3	Capital Gain:0 Sex:M...	18403	0.4558685931612361	0.426076766371525
4	Race:White Sex:Male	18038	0.42572320277126924	0.423620208300338
5	Martial Status:Marrie...	14065	0.910932952582142	0.550126781740506
6	Relationship:Husband	12463	0.8286640898823089	0.545121073337460
7	Education:Bachelors ...	4447	0.3200467377198837	0.466045201100538
8	Education-Num:13 C...	4447	0.3200467377198837	0.466045201100538
9	Occupation:Exec-ma...	3992	0.35705422893501	0.487203650308566
10	Occupation:Exec-ma...	3735	0.3533075572630059	0.487246645442360
11	Occupation:Exec-ma...	3703	0.35195537318923575	0.488045106154043
12	Occupation:Exec-ma...	3595	0.3739122326416609	0.496045184178784
13	Education:Bachelors ...	3522	0.37801869240709374	0.487545347812801
14	Education-Num:13 S...	3522	0.37801869240709374	0.487545347812801

Fig. 3: Slice Finder visualization tools help the user quickly browse through problematic slices by effect size and by slice size on a scatter plot (A) and see slice summary by hovering over any point (B); the user can sort slices by any metrics and select on the scatter plot view or on the table view. The selections are highlighted on the linked views (C). The user can also explore top-k large problematic slices by different effect size threshold using the slider (*min_eff_size*) on the bottom left corner (D).



(a) Census Income Classification (b) Credit Card Fraud Detection (c) Click-Through-Rate Prediction

Fig. 4: Effect size comparisons between different data slicing approaches ($\phi \geq 0.4$). The baseline (CL) fails to produce meaningful slices for model validation/debugging; Slice Finder (LS, DT) identifies problematic slices with effect size above T. The number of recommendations is the number of clusters for CL.

- What do we gain for being more exhaustive and searching for overlapping slices (lattice search)?
- Are the slices interpretable and actionable?
- How efficient are the techniques?

A. Experimental Setup

We used the following three problems with different datasets and models to compare how three different automated slicing techniques perform in terms of recommended slice quality as well as their interpretability. For all experiments, we run the k-means, decision tree, and lattice search algorithms to recommend top-k slices for model validation with the full data set as described/processed below, except for the scalability experiments (Section V-D, where we used samples):

Census Income Classification: We trained a random forest classifier (Example 1) to predict whether the income exceeds $50K/yr based on UCI census data [26]. There are 15 features and 30K examples.

Credit Card Fraud Detection: We trained a random forest classifier to predict fraudulent transactions among credit card transactions [13]. This dataset contains transactions that occurred over two days, where we have 492 frauds out of 284k transactions (examples), each with 29 features. Because the data set is heavily imbalanced, we first undersample non-fraudulent transactions to balance the data. This leaves a total of 984 transactions in the balanced dataset.

Slice	Size	Effect Size
Lattice search results		
V10 = 0.22 – 0.81	99	0.44
V17 = 0.50 – 1.06	99	0.57
V18 = 0.50 – 1.00	99	0.48
Decision tree results		
root → V4 ≥ 0.76	568	0.82
root → V10 ≥ 0.76 → V14 ≥ -0.80	181	0.96
root → V4 ≥ 0.76 → Amount < 320.0 → V1 ≥ -0.08	21	1.04

TABLE II: Top-3 largest problematic slices by lattice search and decision tree approaches for credit card fraud detection problem. The anonymized features (V1, V2, ...) are all standardized to a range [-1, 1].

Click-Through-Rate Prediction: This dataset is proprietary and is used to train neural models for predicting user clicks on an app store. There are several hundred features, but we take a subset of 28 features and train on 50K examples.

B. Large Problematic Slices

Figure 4 and Figure 5 show how lattice search (LS) and decision tree (DT) approaches outperform the baseline (CL) in terms of slice size and effect size. The clustering baseline approach produces large clusters that have very low effect size. When comparing DT and LS, LS produces larger slices with lower effect sizes (all above the minimum effect size threshold, $T = 0.4$). This result indicates that LS is good at finding large slices with enough effect sizes whereas DT finds smaller slices with very high effect sizes. Notice that the average effect size of CL recommended slices are around 0.0 (and sometimes even negative, which means the slices are not problematic), which illustrates that grouping similar examples does not guide users to problematic data slices.

LS considers all the possible slices above a effect size threshold T from top to bottom (i.e., searches slices with a fewer filter clauses first); LS will continue searching until it finds all k problematic slices (or it runs out of candidates). As the search progresses, LS looks at smaller slices with more filter clauses, and this is why LS tends to recommend larger slices just above T. On the other hand, DT slices data in a way that explains misclassified examples best. That is, decision boundaries are formed just around any groups of misclassified examples as long as their size is above the minimum leaf size; this behavior allows for high effect size slices.

It is interesting to see in Figure 5(b) that DT yields much larger slices than LS. This is because the dataset consists of only numeric (continuous) features. Table II shows the top-3 largest problematic slices among the 10 recommended slices by LS and DT. LS discretize the numeric features into continuous ranges (e.g., 10 quantiles), whereas DT simply groups misclassified examples at discontinuous value ranges (value ranges are more dynamic at a finer and varying granularity). In general, LS recommends larger slices and DT more problematic slices by overfitting the decision boundaries with more complex filter predicates. Note that the Credit Card Fraud Detection slices are not easily interpretable because all the feature names are encrypted (e.g., V1, V2, ...).

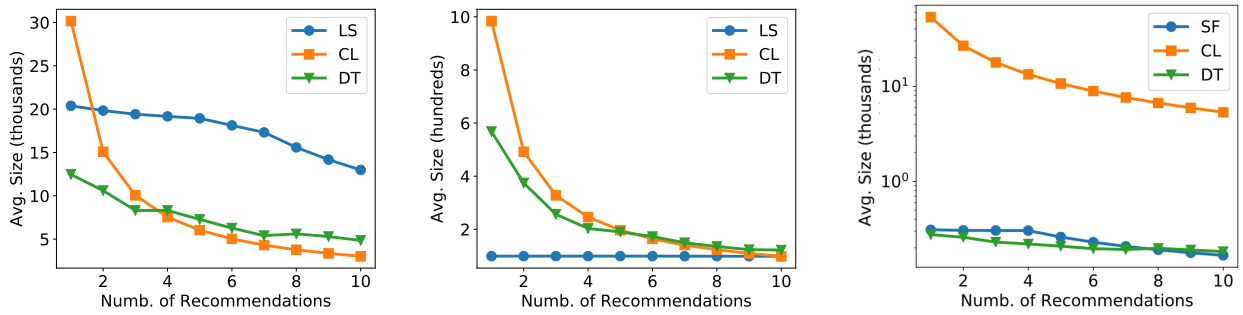
C. Adjusting T

We show performance results for updating the top-K results when the effect size threshold T is adjusted using the slider. It is important to note that we do not need to retrain CL or DT (assuming that we grew the tree to a great enough depth; here, we grew DT with maximum depth of 20 and minimum leaf size of 10). In case of LS, we first run an initial lattice search (e.g., with $T = 0.5$ and $k = 30$) and materialize all the rejected candidate slices. In this way, we can simply look through the materialized slices for top-k largest slices with effect size above any $T < 0.5$ (increasing T may require additional lattice search). Figure 6 shows how average effect size and average slice size of LS and DT change over different T values. LS is much more sensitive to T because it tends to identify larger slices just above the minimum effect size threshold. On the other hand, DT generally recommends high effect size slices, thus, the recommendations are the same for the most part ($0.1 \leq T \leq 0.6$). We exclude CL because T is not enforced on clusters.

D. Scalability

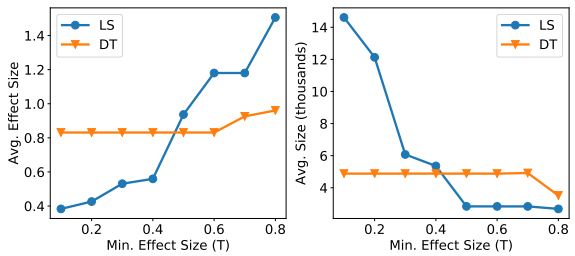
Slice Finder uses sampling (DT and LS) and parallelization (LS) to be more scalable, especially with big, high-dimensional data sets. Figure 7(a) illustrates how Slice Finder scales with increasing sample sizes (using a single node/worker). The original Census Income Classification data set contains 30K examples with 15 features (both continuous and categorical). The run time of LS increases almost linearly with the increasing sample size. DT also runs faster with a smaller sample, but runs slower than LS because DT always grows a max-depth tree (a modified version of Classification & Regression Tree algorithm) before traversing it for top-k problematic slices. We also look at recall, which measures how many of the top-10 large problematic slices based on the full data set are missing from each top-10 slices from a smaller sample. LS retains more than a half of the top-10 large problematic slices even with a 10% sample, and this is acceptable as the goal of Slice Finder is to surface a handful of large problematic slices to users for deeper analysis. As DT's decision boundaries are formed to best explain groups of mis-classified examples, the boundaries (i.e., filter predicates for slices) vary for different samples; the recall is 0 (no match) for all samples and 1 (perfect match) for the full data set.

Figure 8(a) illustrates how Slice Finder can scale with parallelization. LS can distribute the evaluation (e.g., effect size computation) of the slices with the same number of filter predicates to multiple workers, and for the same Census Income Classification data set (sampling fraction$= 1.0$) increasing the number of workers results in better run-time. Notice that the marginal run-time improvement decreases as we add more workers. The reported results are not DT is not shown here because the current implementation does not support parallel DT model training.

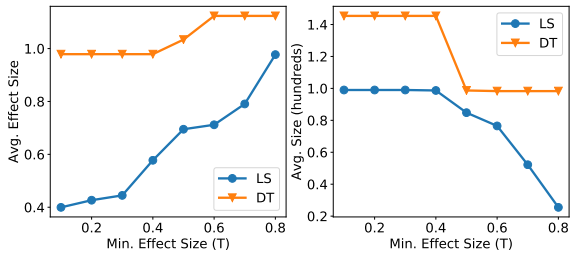


(a) Census Income Classification (b) Credit Card Fraud Detection (c) Click-Through-Rate Prediction

Fig. 5: Slice size comparisons between different data slicing approaches ($\phi \geq 0.4$). In general, LS produces larger slices with simpler slice filter predicates; however, in (b), LS fails to produce larger problematic slices due to poor discretization of numeric features. Notice that CL starts with the entire dataset with the number of clusters is just 1 (the number of recommendations).



(a) Census Income Classification



(b) Credit Card Fraud Detection

Fig. 6: Minimum effect size threshold T and recommended top-10 slice quality (effect size and slice size).

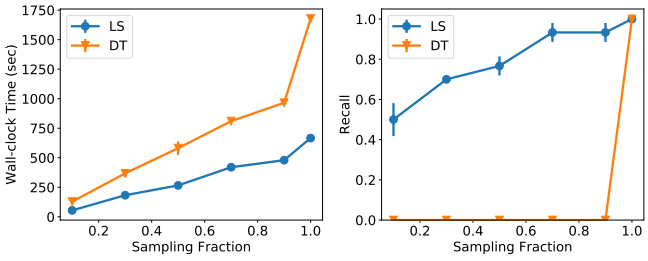
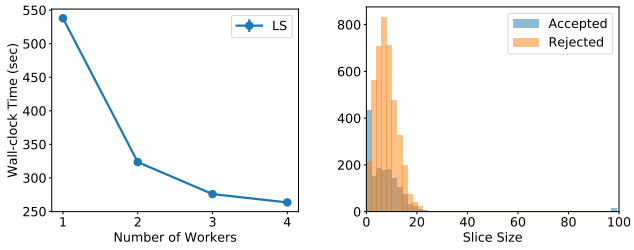


Fig. 7: Slice Finder (LS, DT) run-time on a single node and recall with sampling.



(a) Parallelization (b) Multi-hypotheses testing

Fig. 8: (a) Slice Finder (LS) run-time with increasing number of workers. (b) Size distribution of significant (accepted) and non-significant (rejected) problematic slices ($\phi \geq 0.4$).

E. False Discovery Control

Even for a small data set (or sample), there can be an overwhelming number of problematic slices. The goal of Slice Finder is to bring the user's attention to a handful of large problematic slices; however, if the sample size is small, most slices would contain a fewer examples, and thus, it is likely that a lot of slices and their effect size measures are seen by chance. In such a case, it is important to prevent false discoveries (e.g., non-problematic slices appear as problematic ($\phi \geq T$) due to sampling bias). Figure 8(b) illustrates this in credit card fraud detection problem where most problematic slices are similar in sizes, and yet, many of them are false discoveries (the rejected). The average size for accepted (statistically significant) problematic slices was 7.86 and 8.36 for the rejected. Therefore, without a proper control over false discoveries, Slice Finder can recommend falsely identified problematic slices.

F. Interpretability

Users want to see data slices that are easy to understand with a few common features. In other words, the performance metrics or presenting a cluster of mis-classified examples are

Slice	Size	Effect Size
Census Income Classification		
Sex = Male	20380	0.47
Marital Status = Married-civ-spouse	14065	0.92
Relationship = Husband	12463	0.84
Capital Gain = 0, Occupation = Exec-manag.	3453	0.46
Capital Gain = 0, Hours per week = 50	2333	0.47
Relationship = Wife	1406	0.63
Capital Gain = 0, Education = Masters	1368	0.47
Capital Gain = 0, Education-Num = 14	1368	0.47
Capital Gain = 0, Hours per week = 60	1225	0.48
Workclass = Self-emp-inc	1074	0.41

TABLE III: Top-10 largest problematic slices in Census Income Classification; the slices are easy to interpret with a fewer number of common features. High-dimensional data examples, e.g., clustered by their similarities, are hard to interpret and reason about the possible cause of model performance degradation.

not sufficient to understand and describe the model behavior. In practice, a user often goes through all the mis-classified examples (or clusters of them) manually to describe/understand the problem. To this end, Slice Finder can be used as a preprocessing step to quickly identify data slices where the model might be biased or failing, and the slices are easy to describe with a few number of common features. Table III shows top-10 largest problematic slices in the Census Income Classification; the slices are easy to interpret with a few number of common features. We see that Sex = Male slice has effect size above $T = 0.4$ and contains a lot of examples indicating that the model can use improvements for this slice. It is also interesting to see that the model fails for some sub-demographics of Capital Gain = 0, especially those who are likely to make more money (e.g., work overtime, exec-managerial or self-employed). We also see that slices associated with high education degrees tend to be problematic down the list (not shown in the top-10, except Captial Gain = 0, Education=Masters = or Education-Num = 14). For Click-Through-Rate Prediction also, Slice Finder shows human-readable feature descriptions of problematic slices that partition the data in a way that the slice contains more mis-classified examples than the rest of the data (counterpart). We do not show the slice descriptions because the information is proprietary.

VI. Related Work

In practice, the overall performance metrics can mask the issues on a more granular-level, and it is important to validate the model accordingly on smaller subsets/sub-populations of data (slices). While a well-known problem, the existing tools are still primitive in that they rely on domain experts to pre-define important slices. State-of-art tools for ML model validation include Facets [3], which can be used to discover bias in the data, TensorFlow Model Analysis (TFMA), which slices data by an input feature dimension for a more granular performance analysis [4], and MLCube [23], which provides manual exploration of slices and can both evaluate a single model or compare two models. While the above tools are manual, Slice Finder complements them by automatically finding slices useful for model validation.

There are also several other relevant lines of work related to this problem, and here we list the most relevant work to Slice Finder.

Data Exploration: Online Analytical Processing (OLAP) has been tackling the problem of slicing data for analysis, and the techniques deal with the problem of large search space (i.e., how to efficiently identify data slices with certain properties). For example, Smart Drilldown [22] proposes an OLAP drill down process that returns the top-K most "interesting" rules such that the rules cover as many records as possible while being as specific as possible. Intelligent rollups [34] goes the other direction where the goal is to find the broadest cube that share the same characteristics of a problematic record. In comparison, Slice Finder finds slices, on which the model under-performs, without having to evaluate the model on all the possible slices. This is different from general OLAP operations based on cubes with pre-summarized aggregates, and the OLAP algorithms cannot be directly used.

Model Understanding: Understanding a model and its behavior is a broad topic that is being studied extensively [7], [18], [28], [32], [33], [36]. For example, LIME [32] trains interpretable linear models on local data and random noise to see which feature are prominent. Anchors [33] are high-precision rules that provide local and sufficient conditions for a black-box model to make predictions. In comparison, Slice Finder is a complementary tool to provide part of the data where the model is performing relatively worse than other parts. As a result, there are certain applications (e.g., model fairness) that benefit more from slices. PALM [27] isolates a small set of training examples that have the greatest influence on the prediction by approximating a complex model into an interpretable meta-model that partitions the training data and a set of sub-models that approximate the patterns within each pattern. PALM expects as input the problematic example and a set of features that are explainable to the user. In comparison, Slice Finder finds slices with high effective sizes and does not require any user input. Influence functions [25] have been used to compute how each example affects model behavior. In comparison, Slice Finder identifies interpretable slices instead of individual examples. An interesting direction is to extend influence functions to slices, to quantify the impact of each slice on the overall model quality.

Feature Selection: Slice Finder is a model validation tool, which comes after model training. It is important to note that this is different from feature selection [11], [20] in model training, where the goal is often to identify and (re-)train on the most correlated features (dimensions) to the target label (i.e., finding representative features that best explain model predictions). Instead, Slice Finder identifies a few common feature values that describe subsets of data with significantly high error concentration for a given model; this, in turn, could help the user to interpret hidden model performance issues that are masked by good overall model performance metrics.

VII. Conclusion

We have proposed Slice Finder as a tool for efficiently finding large, problematic, and interpretable slices. The techniques are relevant to model validation in general, but also to model fairness and fraud detection where human interpretability is critical to understand model behavior. We have proposed two methods for automated data slicing for model validation: decision tree training, which is efficient and finds slices defined as ranges of values, and slice lattice search, which can find overlapping slices and are more effective for categorical features. We also provide an interactive visualization front-end to help user quickly browse through a handful of problematic slices.

In the future, we would like to improve Slice Finder to better discretize numeric features and support the merging of slices. We would also like to deploy SliceFinder to products and conduct a user study on how helpful the slices are for explaining and debugging models.

References

[1] Effect size. https://en.wikipedia.org/wiki/Effect_size.

[2] Machine bias. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing, 2016.

[3] Facets overview. https://research.googleblog.com/2017/07/facets-open-source-visualization-tool.html, 2017.

[4] Introducing tensorflow model analysis. https://medium.com/tensorflow/introducing-tensorflow-model-analysis-scaleable-sliced-and-full-pass-metrics-5cde7baf0b7b, 2018.

[5] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, volume 1215, pages 487–499, 1994.

[6] S. Barocas and M. Hardt. Fairness in machine learning. *NIPS Tutorial*, 2017.

[7] O. Bastani, C. Kim, and H. Bastani. Interpreting blackbox models via model extraction. *CoRR*, abs/1705.08504, 2017.

[8] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD*, pages 1387–1395, 2017.

[9] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series B (Methodological)*, 57(1):289–300, 1995.

[10] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[11] M. Charikar, V. Guruswami, R. Kumar, S. Rajagopalan, and A. Sahai. Combinatorial feature selection problems. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 631–640. IEEE, 2000.

[12] J. Cohen. Statistical power analysis for the behavioral sciences. 2nd, 1988.

[13] A. Dal Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. Calibrating probability with undersampling for unbalanced classification. In *SSCI*, pages 159–166. IEEE, 2015.

[14] F. Doshi-Velez and B. Kim. Towards A Rigorous Science of Interpretable Machine Learning. *ArXiv e-prints*, Feb. 2017.

[15] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel. Fairness through awareness. In *ITCS*, pages 214–226, New York, NY, USA, 2012. ACM.

[16] M. Feldman, S. A. Friedler, J. Moeller, C. Scheidegger, and S. Venkatasubramanian. Certifying and removing disparate impact. In *KDD*, pages 259–268, 2015.

[17] D. Foster and B. Stine. Alpha-investing: A procedure for sequential control of expected false discoveries. *Journal of the Royal Statistical Society Series B (Methodological)*, 70(2):429–444, 2008.

[18] A. A. Freitas. Comprehensible classification models: A position paper. *SIGKDD Explor. Newsl.*, 15(1):1–10, Mar. 2014.

[19] M. Ghassemi, T. Naumann, F. Doshi-Velez, N. Brimmer, R. Joshi, A. Rumshisky, and P. Szolovits. Unfolding physiological state: Mortality modelling in intensive care units. In *KDD*, KDD '14, pages 75–84, New York, NY, USA, 2014. ACM.

[20] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[21] M. Hardt, E. Price, and N. Srebro. Equality of opportunity in supervised learning. In *NIPS*, pages 3315–3323, 2016.

[22] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. Interactive data exploration with smart drill-down. In *ICDE*, pages 906–917. IEEE, 2016.

[23] M. Kahng, D. Fang, and D. H. P. Chau. Visual exploration of machine learning results using data cube analysis. In *HILDA*, page 1. ACM, 2016.

[24] J. M. Kleinberg, S. Mullainathan, and M. Raghavan. Inherent trade-offs in the fair determination of risk scores. In *ITCS*, pages 43:1–43:23, 2017.

[25] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *ICML*, pages 1885–1894, 2017.

[26] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, volume 96, pages 202–207, 1996.

[27] S. Krishnan and E. Wu. Palm: Machine learning explanations for iterative debugging. In *HILDA*, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[28] H. Lakkaraju, E. Kamar, R. Caruana, and J. Leskovec. Interpretable & explorable approximations of black box models. *CoRR*, abs/1707.01154, 2017.

[29] M. Lichman. UCI machine learning repository, 2013.

[30] W. McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[31] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, et al. Ad click prediction: a view from the trenches. In *KDD*, pages 1222–1230, 2013.

[32] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *KDD*, pages 1135–1144, 2016.

[33] M. T. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-precision model-agnostic explanations. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[34] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[35] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. In *High Performance Data Mining*, pages 237–261. Springer, 1999.

[36] P. Tamagnini, J. Krause, A. Dasgupta, and E. Bertini. Interpreting black-box classifiers using instance-level visual explanations. In *HILDA*, pages 6:1–6:6, New York, NY, USA, 2017. ACM.