# SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging

Svetlana Sagadeeva*
Graz University of Technology

Matthias Boehm
Graz University of Technology

## ABSTRACT

Slice finding—a recent work on debugging machine learning (ML) models—aims to find the top-K data slices (e.g., conjunctions of predicates such as gender female and degree PhD), where a trained model performs significantly worse than on the entire training/test data. These slices may be used to acquire more data for the problematic subset, add rules, or otherwise improve the model. In contrast to decision trees, the general slice finding problem allows for overlapping slices. The resulting search space is huge as it covers all subsets of features and their distinct values. Hence, existing work primarily relies on heuristics and focuses on small datasets that fit in memory of a single node. In this paper, we address these scalability limitations of slice finding in a holistic manner from both algorithmic and system perspectives. We leverage monotonicity properties of slice sizes, errors and resulting scores to facilitate effective pruning. Additionally, we present an elegant linear-algebra-based enumeration algorithm, which allows for fast enumeration and automatic parallelization on top of existing ML systems. Experiments with different real-world regression and classification datasets show that effective pruning and efficient sparse linear algebra renders exact enumeration feasible, even for datasets with many features, correlations, and data sizes beyond single node memory.

**ACM Reference Format:**
Svetlana Sagadeeva and Matthias Boehm. 2021. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3448016.3457323

## 1 INTRODUCTION

Machine Learning (ML) and data-driven applications fundamentally change many aspects of the IT landscape from user-facing applications, over backend decision systems, to the optimization of the software and hardware stack [21, 36, 56]. Crucial steps in the process of developing and deploying ML pipelines for production are the tasks of data validation (analyzing input data characteristics) [56, 59] and model debugging (analyzing valid ML model characteristics) [22, 56, 60]. Aspects to consider are data errors (e.g., heterogeneity, human error, measurement error), lack of model generalization (e.g., overfitting, imbalance, out-of-domain prediction), as well as systematic bias and missing fairness. A lack of model validation and debugging might cause silent but severe problems [56]. Examples are race-biased jail risk assessment [6], wolf detection based on snow cover [58], and horse detection based on image watermarks [38]. Model debugging aims to identify such issues.

**Model Debugging Techniques:** Apart from basic data debugging and validation [56, 59], model accuracy monitoring and comparison during serving [56, 60], and manual model error analysis via confusion matrices (e.g., matrix visualization of correct versus predicated labels), there exist several advanced model debugging techniques. Examples from the field of computer vision are saliency maps [30, 63, 70], layer-wise relevance propagation [8, 38], and occlusion-based explanations [75], which all aim to find input image areas that significantly influence the prediction. The data management community recently contributed efficient means of incremental computation for occlusion-based explanations [47, 48] by exploiting the inherent overlap in such computations. For structured data and prediction tasks—with continuous and categorical features—the literature is, however, relatively sparse. Existing work includes *explanation tables* [25] (with a primary focus on data summarization), and *slice finder* [18, 19], which aims to find the top-K data slices (e.g., conjunctions of predicates such as gender female and degree PhD), where a trained model performs significantly worse than on the entire dataset. Finding such problematic slices is very useful for understanding a lack of training data or model bias, but also as a path towards model improvements.

**Limitations of Existing Work:** Given an input matrix of $n$ binary features—e.g., obtained via binning, recoding or feature hashing, and subsequent one-hot encoding—slice finding considers a search space of $O(2^n)$ slices (all subsets of binary features, except combinations within original features). Due to this exponential search space, explanation tables [25] rely on greedy heuristics and sampling, while slice finder [18, 19] proposes clustering, decision trees (for non-overlapping slices), and lattice searching with a heuristic, level-wise termination condition of K slices found. For scalability, slice finder uses a queue-based, task-parallel approach and sampling. None of the existing approaches provide exact slice enumeration that guarantees finding the real top-K problematic slices. This debugging uncertainty creates trust concerns and directly motivates our work. Additionally, there is a lack of effective distributed parallelization for both, slices and data.

**Contributions:** Our primary contribution is *SliceLine*, an exact—yet fast and practical—enumeration algorithm for finding the top-K problematic data slices, where an ML model performs worse than overall. We exploit—inspired by frequent itemset mining algorithms [61]—monotonicity for effective pruning, and provide an elegant sparse linear algebra implementation of slice enumeration that ML

*Work done as a visiting student at Graz University of Technology, Austria, and a subsequent master thesis at Saint-Petersburg Polytechnic University, Russia.


*SIGMOD '21, June 20–25, 2021, Virtual Event, China*

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457323

systems can compile into efficient local or distributed execution plans. Our detailed technical contributions are:

- *Problem Formulation:* As a conceptual basis, we redefine the slice finding problem formulation, and derive an intuitive and flexible scoring function in Section 2.
- *Properties and Pruning:* We then establish upper and lower bounds for slice sizes, errors, and scores; and introduce related pruning techniques in Section 3.
- *Enumeration Algorithm:* Putting it altogether, we present a linear-algebra-based enumeration algorithm in Section 4. This simple algorithm prunes, enumerates, and evaluates all slice candidates per level with coarse-grained matrix multiplications, aggregations, and element-wise operations.
- *Experiments:* Finally, we present experimental results for a variety of real datasets, regression and classification tasks, as well as local and distributed execution in Section 5.

## 2 PROBLEM FORMULATION

Our slice finding problem formulation is inspired by the SliceFinder [18] problem. However, to simplify the usage and interpretation, we redefine the problem as a constrained, score-based top-K search. In this section, we define the used notation and the optimization problem, and discuss differences to related problem formulations.

### 2.1 Notation

**Data, Model, and Errors:** Assume an $n \times m$ input feature matrix $\mathbf{X}$ with integer features $\{F_1, F_2, \ldots, F_m\}$ and a continuous or categorical $n \times 1$ label vector $\mathbf{y}$. An input feature $F_j$ has a domain $d_j$ (distinct values) of $\{1, 2, \ldots, d_j\}$, which might have been derived by recoding or hashing categorical features (categories to integers), or binning continuous features (floating point values to integers). Furthermore, assume a classification or regression model $\mathbf{M}$ trained over $\mathbf{X}$ and $\mathbf{y}$. Applying $\mathbf{M}$ on $\mathbf{X}$ yields row-aligned vectors of predictions $\hat{\mathbf{y}}$ and errors $\mathbf{e} = \text{err}(\mathbf{y}, \hat{\mathbf{y}})$ with $\mathbf{e} \geq 0$. Common user-defined error functions err() are classification (in-)accuracy $\mathbf{e} = (\mathbf{y} \neq \hat{\mathbf{y}})$, squared loss $\mathbf{e} = (\mathbf{y} - \hat{\mathbf{y}})^2$ for regression, and algorithm-specific loss functions. The same definitions apply to train, validation, and test splits of $\mathbf{X}$ and $\mathbf{y}$ ($\mathbf{M}$ always created on the train dataset), which provides users with sufficient flexibility of model debugging.

**Slices:** A data slice $S$ is defined as a conjunction (AND-combination) of 1 to $m$ equivalence predicates $F_j = v$ with $v \in d_j$ and at most one predicate per feature $F_j$. Thus, a slice is a subset of rows of $S \subseteq \mathbf{X}$ and the slice size $|S|$ is bounded by $[0, n]$. Furthermore, a slice definition can be represented as a fixed-size, one-hot encoded vector $\mathbf{s}$ of size $1 \times \Sigma_{j=1}^{m} d_j$. Finally, for reasoning about slice errors, we need some additional notation. First, we define $\mathbf{es}$ as a row-aligned $|S| \times 1$ vector, holding the errors $\mathbf{es}_i$ of slice rows $S_i$. Second, let $\overline{e} = \Sigma_{i=1}^{n} \mathbf{e}_i / n$ be the average error, $se = \Sigma_{i=1}^{|S|} \mathbf{es}_i$ be the total slice error, $sm = \max_{i=1}^{|S|} \mathbf{es}_i$ be the maximum slice tuple error, and $\overline{se} = se/|S|$ be the average slice error.

### 2.2 Slice Finding Problem

With the introduced notation, we now define our score-based slice finding problem. Similar to SliceFinder, we aim to find sufficiently large slices (high impact on the overall model) with substantial errors (high negative impact on sub-group/model), but encode these two objectives in an intuitive, linearized scoring function.

Definition 1 (Scoring Function). *Let $\alpha \in (0, 1]$ be a weight parameter for the importance of the average slice error. Then, we define the score $sc$ of a slice $S$ as*

$$\begin{aligned} sc &= \alpha \left( \frac{\overline{se}}{\overline{e}} - 1 \right) - (1 - \alpha) \left( \frac{n}{|S|} - 1 \right) \\ &= \alpha \left( \frac{n}{|S|} \cdot \frac{\Sigma_{i=1}^{|S|} \mathbf{es}_i}{\Sigma_{i=1}^{n} \mathbf{e}_i} - 1 \right) - (1 - \alpha) \left( \frac{n}{|S|} - 1 \right), \end{aligned} \tag{1}$$

*which is defined for non-empty slices $S$ and otherwise assumed negative*[1]*. For finding problematic slices, we aim to maximize $sc$.*

**Scoring Function Interpretation:** In principle, we linearize the errors and sizes by including the ratio of average slice error to average overall error, and subtracting the ratio of overall size to slice size, while weighting these components by the user parameter $\alpha$. This scoring function has several compelling properties. First, the components are balanced under $\alpha = 0.5$. A slice with twice the relative error but half the size of another slice, has exactly the same score. This characteristic is crucial for an intuitive influence of the $\alpha$ parameter. Second, independent of $\alpha$, the score of the original $\mathbf{X}$ is always $sc = 0$. Third, for $\alpha = 0$ (all weight on size), the maximum feasible score is 0, and no slice smaller than $\mathbf{X}$ can reach it. For this reason, we defined $\alpha \in (0, 1]$. Fourth and finally, all components of Equation (1) are either constants, or slice errors and sizes, which makes this function amenable to pruning.

**Slice Finding Problem:** The above scoring function allows identifying interesting slices that show larger than average errors with $sc > 0$. In order to ensure statistical significance, we additionally establish—inspired by frequent itemset mining—a minimum support threshold $|S| \geq \sigma$ [35] (by default $\sigma = \max(32, n/100)$). The score-based slice finding problem is then defined as follows:

Definition 2 (Score-based Slice Finding). *Given the input feature matrix $\mathbf{X}$, and error vector $\mathbf{e}$ (derived from a model $\mathbf{M}$, $\mathbf{X}$ and label vector $\mathbf{y}$), as well as an integer $K$, find the top-$K$ slices $\mathbf{TS}$—from the lattice $\mathcal{L}$ of all slices—that satisfy the following condition:*

$$\begin{aligned} \mathbf{TS} = \arg\max_{\mathcal{S} \subset \mathcal{L}} & \; \Sigma_{k=1}^{K} sc(\mathcal{S}_k) \\ s.t. \; & \; \forall k \in [1, K] : |\mathcal{S}_k| \geq \sigma \wedge sc(\mathcal{S}_k) > 0, \end{aligned} \tag{2}$$

*returned sorted in descending order of $sc(\mathcal{S}_k)$.*

### 2.3 Related Problems

Our score-based slice finding is related to several other data mining problems. In the following, we summarize these problems, and describe how our problem formulation differs.

**SliceFinder:** The SliceFinder problem [18, 19] is naturally the closest to our formulation. It also aims to find the top-K slices under an ordering by "increasing number of literals, decreasing slice size, and decreasing effect size" [18] subject to (1) a minimum effect size threshold $T$, (2) statistical significance, and (3) a dominance constraint (no coarser slice satisfies 1 and 2). The effect size measures the difference of distributions $S$ and $\neg S$, while hypothesis testing via Welch's t-test checks that errors for $S$ are significantly larger than

[1] E.g., replacing $|S|$ with $\max(|S|, 1)$ yields $sc = -2\alpha - n + \alpha n + 1$ for empty slices.

for $\neg S$. While this work is very inspiring and provides statistically robust outputs—including false discovery control [77]—the dominance constraint might hide the most interesting slices with large errors, and we believe a simpler, more intuitive, score-based formulation is necessary in practice. The top-K setting, typically large minimum support constraints, and manual inspection of results (aided by statistical tests) also render false discoveries unlikely.

**Data Coverage:** Recent work on identifying patterns with insufficient data coverage in databases [7, 33, 42] is also closely related. A pattern $\mathcal{P}$ is similar to our slice definition $S$, and its coverage $\text{cov}(\mathcal{P}, \mathcal{D})$ on a—single- or multi-table—dataset $\mathcal{D}$ is equivalent to the slice size $|S|$. The maximal uncovered patterns (MUP) identification problem [7] then aims to find all uncovered patterns (i.e., $\mathcal{P}$ with $\text{cov}(\mathcal{P}, \mathcal{D}) < \tau$) that are not dominated by other uncovered, but coarser patterns. This problem follows a similar intuition of statistical significance as our minimum support constraint, but aims to find the maximum slices that *do not* satisfy this constraint. In contrast, we consider both errors and sizes as part of our scoring function and aim to find the top-K worst slices.

**Frequent Itemset Mining:** Frequent itemset mining, and association rule mining, have been studied extensively in the literature [4]. Given a database of transactions $\mathcal{D} = \{T_1, T_2, \ldots, T_n\}$, each comprising a set of items, the goal is to find the itemsets that satisfy a minimum support threshold $\sigma$ (i.e., count of transactions they appear in) [61]. Major algorithm classes include Apriori [5], Eclat [74], and FP-Growth [28], which all exploit the monotonicity of itemset frequencies—an itemset can only be frequent if all its subsets are frequent—for effective pruning. Specialized algorithms with additional pruning exist for finding maximal frequent itemsets [15], and discovering functional dependencies [54]. These ideas also inspired our pruning techniques for fast slice finding. In contrast to frequent itemset mining though, we focus on both slice errors and sizes, joint effects on the scoring function, and we aim to find the top-K worst slices, allowing for additional pruning.

## 3 PROPERTIES AND PRUNING

As a basis for slice enumeration, in this section, we first discuss basic properties and upper bounds of slice errors and sizes, and then derive an upper bound for the overall scoring function. Finally, we present several effective pruning techniques.

### 3.1 Basic Properties and Bounds

Figure 1 shows a simplified example lattice of slices for $\mathbf{X}$ with $m = 4$ unary features. We use this example to describe the slice finding search space, introduce terminology, and analyze monotonicity properties of relevant data characteristics.

**Search Space of Slices:** For the special case of our example, the lattice contains $O(2^m)$ nodes, organized in $m + 1$ levels. The top level (level 0) represents no predicates and thus, the original dataset $\mathbf{X}$ of size n and total error $e = \Sigma_{i=1}^{n} \mathbf{e}_i$. A node in level $i$ is a conjunction of $i$ predicates, has $i$ parents, and $m - i$ children. For example, $cd$ has two parents $c$ and $d$, and two children $acd$, and $bcd$. For the general case, of $m$ integer features ($l$ one-hot encoded features, with $l \geq m$), the lattice has fewer than $2^l$ nodes because conjunctions of the same original feature are invalid. In this case, we get $O(2^l - \Sigma_{j=1}^{m} 2^{d_j} + l + m)$ nodes (full lattice except invalid).

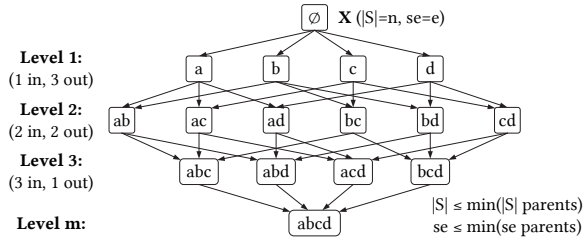



**Figure 1: Example Lattice and Slice Properties.**

**Monotonicity:** Similar to the monotonicity of itemset frequencies, we observe that both slice sizes and slice errors are monotonically decreasing along a directed path in the lattice. Due to the focus on conjunctions, a slice is defined as the intersection (i.e., subset) of its parents. Three properties follow from this observation:

- *Slice Size:* The size of a slice $|S|$ is upper bounded by $\lceil |S| \rceil$, the minimum size of all its parent slices (count of tuples).
- *Absolute Slice Error:* The total slice error $se$ (sum of tuple errors) is upper bounded by $\lceil se \rceil$, the minimal absolute error $se$ of all its parent slices, and the minimal maximum tuple error $sm$ of all its parent slices times $\lceil |S| \rceil$.
- *Relative Slice Error:* The relative error $\overline{se} = se/|S|$ is *non-monotonic* because $|S|$ appears in the denominator.

Note that the number of parents $|\mathcal{P}|$ increases as we descend the lattice, which makes the minimum bounds increasingly effective.

**Scoring Function Upper Bound:** Despite the non-monotonic relative slice error, we aim to upper bound the slice score $sc$ because it would be very helpful for pruning. Revisiting Equation (1) shows the challenge: $|S|$ appears in the denominators of positive and negative terms, which does not allow deriving an upper bound for $sc$ by plugging in upper bounds for $|S|$ and $se$. However, relevant scores are guaranteed to have a slice size $|S| \in [\sigma, \lceil |S| \rceil]$. We leverage this property, and derive the upper bound score $\lceil sc \rceil$ by solving for the $|S|$ that maximizes the score in this interval:

$$\begin{aligned} \lceil sc \rceil = \max_{|S| \in [\sigma, \lceil |S| \rceil]} & \alpha \left( \frac{n}{|S|} \cdot \frac{\lceil se \rceil}{\Sigma_{i=1}^{n} \mathbf{e}_i} - 1 \right) - (1 - \alpha) \left( \frac{n}{|S|} - 1 \right) \\ \text{with} \quad & \lceil se \rceil = \min(\min_{p \in \mathcal{P}} se_p, |S| \cdot \min_{p \in \mathcal{P}} sm_p) \\ \text{and} \quad & \lceil |S| \rceil = \min_{p \in \mathcal{P}} |S_p|. \end{aligned} \tag{3}$$

Depending on the parameters, this function is either monotonically increasing or decreasing with increasing $|S| \in [\sigma, \lceil |S| \rceil]$. This fact allows for a very simple solution: we compute $\lceil sc \rceil$ as the maximum scores of the interesting points $\sigma$, $\max(se/sm, \sigma)$, and $\lceil |S| \rceil$.

### 3.2 Pruning Techniques

Regarding the score-based slice finding problem, we can now leverage the upper bounds for simple, yet very effective pruning.

**Size Pruning:** Every slice must satisfy $|\mathcal{S}| \geq \sigma$. Hence, we can prune a lattice node without data access—and due to monotonically decreasing slice sizes all reachable children—if $\lceil |\mathcal{S}| \rceil < \sigma$.

**Score Pruning:** Similarly, with the upper bound score $\lceil sc \rceil$, we get two additional pruning opportunities. First, as every slice must satisfy $sc(S) > 0$, we can prune a node and all reachable children if $\lceil sc(S) \rceil \leq 0$. Children can be pruned because the upper bound represent the highest possible score of any subset (i.e., any reachable

child). Second, when maintaining the set of top-K slices $\mathcal{S}$, we can leverage $sc$ of $\mathcal{S}_k$ as a monotonically increasing lower bound $sc_k$ for pruning any slice (and its children) where $\lceil sc(S) \rceil \leq sc_k$.

**Handling of Pruned Slices:** All three pruning techniques eliminate nodes and all their children without data access. However, it is important to keep track of pruned nodes when computing upper bounds for following levels. Looking back to the lattice search space in Figure 1 reveals that we can simply discard these pruned nodes, and assume $|\mathcal{S}| = 0$ and $\lceil sc \rceil = 0$ if $|\mathcal{P}| < i$, i.e., the number of enumerated parents $|\mathcal{P}|$ of a slice at level $i$ is less than $i$.

**Summary Pruning Impact:** Together, these pruning techniques are very effective. Slice sizes and errors are monotonically decreasing, the lower bound score $sc_k$ is monotonically increasing, and the number of parent nodes (and thus, the scope of minimum conditions) is monotonically increasing as well.

## 4 ENUMERATION ALGORITHM

A major design goal of SliceLine was to provide fast and scalable model debugging close to where the models are trained. In this section, we describe a vectorized linear algebra implementation of slice finding, which can be implemented on a variety of ML systems. Enumerating and pruning is done via sparse linear algebra. We first outline the overall algorithm, and then describe initialization, pair enumeration, slice evaluation, and top-K maintenance.

### 4.1 Overall Algorithm

The basic structure of the overall enumeration algorithm, as shown in Algorithm 1, is very simple. Here, we first describe the inputs, outputs, data preparation, general structure, and its termination.

**Inputs and Outputs:** The $n \times m$ input feature matrix $\mathbf{X}_0$ is expected in an integer-encoded form (1-based, continuous integer range), representing categories and bins. Additional inputs are the error vector $\mathbf{e}$, and the parameters $K$ (top-K threshold), $\sigma$ (min support threshold), $\alpha$ (error/size weight), and $\lceil L \rceil$ (maximum lattice level). Our SliceLine algorithm then computes the top-K slices, and returns **TS**, as a $K \times m$ integer-encoded matrix with one row per slice where zeros represent free features, and **TR** as the aligned, corresponding slice statistics (scores, errors, sizes).

**Data Preparation:** In a first step, we prepare the data by computing feature offsets and one-hot encoding the entire dataset in lines 1-5. We compute the domain (i.e., number of distinct items) per features as colMaxs($\mathbf{X}_0$), which exploits the continuous integer codes. We then compute the start and end offset of each feature in one-hot encoded representation via a simple cumsum(**fdom**), i.e., as a cumulative or prefix sum of the feature domains. Finally, we perform one-hot encoding of the shifted integer codes $\mathbf{X}_0 + \mathbf{fb}$. This element-wise matrix-vector addition creates global integer codes for one-hot encoding via the following vectorized expression

```
rix = matrix(seq(1,m) %*% matrix(1,1,n), m*n, 1)
cix = matrix(X0 + fb, m*n, 1);
X = table(rix, cix);  # contingency table
```

The vectors **rix** and **cix** act as row and column indexes, and table(**rix**, **cix**) counts each unique pair by adding 1 to the respective row-column output positions. Here, all pairs are unique, yielding the—likely sparse—one-hot-encoded 0/1 matrix **X** in Line 5.

**Algorithm 1** SliceLine Enumeration Algorithm

**Input:** Feature matrix $\mathbf{X}_0$, errors $\mathbf{e}$, $K = 4$, $\sigma = 32$, $\alpha = 0.5$, $\lceil L \rceil = \infty$
**Output:** Top-K slices **TS**, Top-K scores, errors, sizes **TR**

1: *// a) data preparation (one-hot encoding X)*
2: **fdom** ← colMaxs($\mathbf{X}_0$) *// $1 \times m$ matrix*
3: **fb** ← cumsum(**fdom**) − **fdom**,
4: **fe** ← cumsum(**fdom**)
5: **X** ← onehot($\mathbf{X}_0$ + **fb**) *// $n \times l$ matrix*
6: *// b) initialization (statistics, basic slices, initial top-K)*
7: $\bar{e} \leftarrow$ sum(**e**)/$n$
8: [**S**, **R**, **cI**] ← createAndScoreBasicSlices(**X**, **e**, $\bar{e}$, $\sigma$, $\alpha$)
9: [**TS**, **TR**] ← maintainTopK(**S**, **R**, **0**, **0**, $K$, $\sigma$)
10: *// c) level-wise lattice enumeration*
11: $L \leftarrow 1$, $\lceil L \rceil \leftarrow \min(m, \lceil L \rceil)$ *// current/max lattice levels*
12: **X** ← **X**[, **cI**] *// select features satisfying* $\mathbf{ss}_0 \geq \sigma \wedge \mathbf{se}_0 > 0$
13: **while** nrow(S) $> 0 \wedge L < \lceil L \rceil$ **do**
14: $L \leftarrow L + 1$
15: **S** ← getPairCandidates(**S**, **R**, **TS**, **TR**, $K$, $L$, $\bar{e}$, $\sigma$, $\alpha$, **fb**, **fe**)
16: **R** ← matrix(0, nrow(S), 4), **S2** ← S[, **cI**]
17: **for** $i$ **in** nrow(S) **do** *// parallel for*
18: $R_i \leftarrow$ evalSlices(**X**, **e**, $\bar{e}$, $\mathbf{S2}_i^\top$, $L$, $\alpha$)
19: [**TS**, **TR**] = maintainTopK(**S**, **R**, **TS**, **TR**, $K$, $\sigma$)
20: **return** decodeTopK(**TS**, **fb**, **fe**), **TR**

**Key Primitives:** In a second step in Lines 6-8, we compute basic statistics, and evaluate the basic, 1-predicate slices (Section 4.2). After maintaining the top-K slices, we enter the while loop for level-wise lattice enumeration. Each iteration, then generates paired candidates from previous-level valid slices and pruning (Section 4.3), evaluates the scores, errors, and sizes of non-pruned slice candidates (Section 4.4), and maintains the top-K slices (Section 4.5).

**Termination:** The overall algorithm terminates if there are no more slice candidates (nrow(S) = 0), or we reach the last lattice level, which is equal to the number of original features $m$, or $\lceil L \rceil$.

### 4.2 Initialization

As a starting point for initialization, we compute several interesting statistics, as well as create and score all basic slices. First, we materialize the average error on **X** with $\bar{e} = \Sigma_{i=1}^{n} \mathbf{e}/n$ because it is repeatedly used for slice scoring (Equation (1)). Second, in createAndScoreBasicSlices, we compute all basic slice sizes and errors in a vectorized form via:

$$\begin{aligned} \mathbf{ss}_0 &= \text{colSums}(\mathbf{X})^\top \quad \text{(slice sizes)} \\ \mathbf{se}_0 &= (\mathbf{e}^\top \odot \mathbf{X})^\top \quad \text{(slice errors).} \end{aligned} \tag{4}$$

This fast and convenient form is enabled by the one-hot-encoding of **X**, where column sums represent the counts per feature value, and the vector-matrix multiplication $(\mathbf{e}^\top \odot \mathbf{X})^\top$ joins and scales rows with its errors, and computes sums per feature value. We then obtain an indicator of valid slices with $\mathbf{cI} = \mathbf{ss}_0 \geq \sigma \wedge \mathbf{se}_0 > 0$ for selecting one-hot-encoded slice representations **S**, and corresponding slice sizes **ss** and errors **se**, **sm** (e.g., via **se** = removeEmpty(**cI** · $\mathbf{se}_0$)). Finally, we can compute Equation (1) in a vectorized manner

$$\mathbf{sc} = \alpha((\mathbf{se}/\mathbf{ss})/\bar{e} - 1) - (1 - \alpha)(n/\mathbf{ss} - 1) \tag{5}$$

and return the slices **S** and statistics **R** = cbind(**sc**, **se**, **sm**, **ss**).

## 4.3 Pair Enumeration

A central component of SliceLine is the enumeration of slice candidates for evaluation. This enumeration applies the pruning techniques from Section 3.2, and discards invalid or pruned candidates.

**Pair Construction:** Slice candidates for level $L$ are generated as pairs of evaluated slices $\mathbf{S}$ from level $L-1$. This basic idea is inspired by frequent itemset mining, specifically the join of $L_{i-1}$ itemsets to generate $L_i$ itemsets in Apriori [5]. For example, *abc* can be generated from *ab* and *ac* (common *a*), *ab* and *bc* (common *b*), or *ac* and *bc* (common *c*). However, constructing pairs in linear algebra is challenging. Our implementation has four steps. First, we prune invalid input slices by the minimum support and non-zero error constraints $\mathbf{S} = \text{removeEmpty}(\mathbf{S} \cdot (\mathbf{R}_{:,4} \geq \sigma \wedge \mathbf{R}_{:,2} > 0))$. This pruning reduces the input size $n$ of the $O(n^2)$ pair generation, and it does not jeopardize overall pruning because we handle missing parents. Second, we join compatible slices via a self-join on $\mathbf{S}$:

$$\mathbf{I} = \text{upper.tri}((\mathbf{S} \odot \mathbf{S}^\top) = (L-2), \text{values=TRUE}). \tag{6}$$

We compare the matrix multiplication output with $(L-2)$ to ensure compatibility (e.g., $L-2=1$ matches for level $L=3$, which checks that *ab* and *ac* have 1-item overlap for generating *abc*), and extract the upper triangular matrix because $\mathbf{S} \odot \mathbf{S}^\top$ is symmetric[2]. Third, we create the combined slices by converting $\mathbf{I}$ to row-column index pairs, and creating extraction matrices $\mathbf{P1}$ and $\mathbf{P2}$ as follows:

```
rix = matrix(I * seq(1,nr), nr*nc, 1);
rix = removeEmpty(target=rix, margin="rows");
P1 = table(seq(1,nrow(rix)), rix, nrow(rix), nrow(S));
```

The combined slices are then merged via $\mathbf{P} = ((\mathbf{P1} \odot \mathbf{S}) + (\mathbf{P2} \odot \mathbf{S})) \neq 0$. Similarly, we extract combined sizes **ss**, total errors **se**, and maximum errors **sm** as the minimum of parent slices with

$$\mathbf{ss} = \min(\mathbf{P1} \odot \mathbf{R}_{:,4}, \mathbf{P2} \odot \mathbf{R}_{:,4}). \tag{7}$$

Fourth, we discard invalid slices with multiple assignments per feature. With the feature offsets **fb** and **fe**, we scan over $\mathbf{P}$, check $\mathbf{I} = \mathbf{I} \wedge (\text{rowSums}(\mathbf{P}_{:,\text{beg:end}}) \leq 1)$ for each original feature, and retain only rows in $\mathbf{P}$ where no feature assignment is violated.

**Candidate Deduplication:** At this point, we have valid slices for level $L$, but there are duplicates. For example, we get three *abc* slices from joining *ab*-*ac*, *ab*-*bc*, and *ac*-*bc*. The multiple parents enable effective pruning but create exponentially increasing redundancy. We address this challenge by deduplication via slice IDs. Interpreting the one-hot vectors as binary integers, however, already overflows for a moderate number of columns. Instead, we use the domain of features $\mathbf{dom} = \mathbf{fe} - \mathbf{fb} + 1$ and compute the IDs like an ND-array index. We scan over $\mathbf{P}$, and compute the sum of feature contributions by $\mathbf{ID} = \mathbf{ID} + \text{scale} \cdot \text{rowIndexMax}(\mathbf{P}_{:,\text{beg:end}}) \cdot \text{rowMaxs}(\mathbf{P}_{:,\text{beg:end}})$, where scale is the feature entry from cumprod(**dom**). Duplicate slices now map to the same ID and can be eliminated. Since the domain can still be very large, we transform the IDs via frame recoding to consecutive integers. For pruning and deduplication, we materialize this mapping as $\mathbf{M} = \text{table}(\mathbf{ID}, \text{seq}(1, \text{nrow}(\mathbf{P})))$ and deduplicate via $\mathbf{P} = \mathbf{M} \odot \mathbf{P}$.

**Candidate Pruning:** Before final deduplication, we further apply all pruning techniques from Section 3.2 with respect to all parents of a slice. As a basis for pruning, we first compute the upper bound slice sizes **ss**, upper bound errors **se** and **sm** (minimum of all parents), and the number of parents **np** as follows:

$$\begin{aligned} \lceil \mathbf{ss} \rceil &= 1/\text{rowMaxs}(\mathbf{M} \cdot (1/\mathbf{ss}^\top)) \\ \mathbf{np} &= \text{rowSums}((\mathbf{M} \odot (P1 + P2)) \neq 0) \end{aligned} \tag{8}$$

We minimize by maximizing the reciprocal (and replace $\infty$ with 0) for accounting only existing parents, while avoiding large dense intermediates. With these inputs, Equation (3) computes the upper bound scores $\lceil \mathbf{sc} \rceil$, and all pruning becomes a simple filter on $\mathbf{M}$:

$$\mathbf{M} = \mathbf{M} \cdot (\lceil \mathbf{ss} \rceil \geq \sigma \wedge \lceil \mathbf{sc} \rceil > sc_k \wedge \lceil \mathbf{sc} \rceil \geq 0 \wedge \mathbf{np} = L). \tag{9}$$

Finally, we discard empty rows in $\mathbf{M}$ to get $\mathbf{M}'$, deduplicate slices with $\mathbf{S} = \mathbf{M}' \odot \mathbf{P}$ (at implementation level with $\mathbf{S} = \mathbf{P}[, \text{rowIndexMax}(\mathbf{M}')]$), and return $\mathbf{S}$ as new slice candidates.

## 4.4 Slice Evaluation

All slice candidates are then evaluated, which requires scanning $\mathbf{X}$ for qualifying rows, and computing slice sizes, errors, and scores.

**Vectorized Evaluation:** A key observation is that we can evaluate all slices $\mathbf{S}$ on $\mathbf{X}$ (both one-hot encoded, see Figure 2) in a single matrix multiplication and additional element-wise operations:

$$\begin{gathered} \mathbf{I} = ((\mathbf{X} \odot \mathbf{S}^\top) = L) \\ \mathbf{ss} = \text{colSums}(\mathbf{I})^\top \quad \mathbf{se} = (\mathbf{e}^\top \odot \mathbf{I})^\top \quad \mathbf{sm} = \text{colMaxs}(\mathbf{I} \cdot \mathbf{e})^\top \end{gathered} \tag{10}$$

Here, we compute **ss**, **se** and **sm** similar to Equation (4) but on $\mathbf{I}$ instead of $\mathbf{X}$. The scores **sc** are again computed with Equation (5). Figure 2 gives the intuition by example. The $n \times$ #slices output is itself a 0/1 matrix and can be treated like the encoded matrix $\mathbf{X}$.

**Data- or Task-Parallelism:** This vectorized slice evaluation is executed either with local or distributed, data-parallel matrix multiplications [12], and thus, scales to larger data sizes, many slices, or both. For the common case of moderate numbers of slices per level, this boils down to efficient broadcast-based, distributed matrix multiplications, where we broadcast $\mathbf{S}$ to all nodes and scan $\mathbf{X}$ in a data-local manner. However, in ML systems with limited sparse-sparse operation support and limited sparsity-exploitation across operations, the two intermediates $(\mathbf{X} \odot \mathbf{S}^\top)$, and $(\mathbf{I})$ might create problems. An alternative task-parallel [13, 50] formulation—which only creates vector intermediates—is shown in Algorithm 1 Lines 16-18. Here, evalSlices is called for a slice of $\mathbf{S}$ at a time. Naturally, hybrid strategies with blocks of slices for scan sharing—similar to model batching [65, 76]—or row partitions of $\mathbf{X}$—similar to mini-batch or federated processing [14, 26, 34]—are possible.

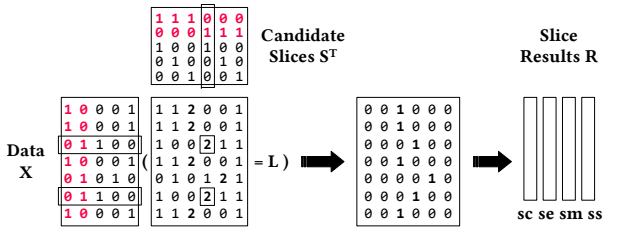



**Figure 2: Example Vectorized Slice Evaluation** (The matrix $\mathbf{X}$ has two red/black features, with 2/3 distinct values. The matrix multiplication $((\mathbf{X} \odot \mathbf{S}^\top)$ evaluates predicates by multiplying the one-hot vectors and counting matching predicates. By checking for $((\mathbf{X} \odot \mathbf{S}^\top) = L)$, we get rows that match all $L$ slice predicates)**.**

[2]This symmetry is also exploited by the related BLAS library calls (e.g., `cblas_dsyrk`).

**Table 1: Dataset Characteristics** ($n$ rows, $m$ columns before binning/one-hot encoding, $l$ columns after one-hot encoding).

| Dataset | $n$ (nrow($X_0$)) | $m$ (ncol($X_0$)) | $l$ (ncol(X)) | ML Alg. |
|---|---|---|---|---|
| Adult | 32,561 | 14 | 162 | 2-Class |
| Covtype | 581,012 | 54 | 188 | 7-Class |
| KDD 98 | 95,412 | 469 | 8,378 | Reg. |
| US Census | 2,458,285 | 68 | 378 | 4-Class |
| US Census10x | 24,582,850 | 68 | 378 | 4-Class |
| CriteoD21 | 192,215,183 | 39 | 75,573,541 | 2-Class |
| Salaries | 397 | 5 | 27 | Reg. |

## 4.5 Top-K Maintenance

We maintain the top-K slices **TS** and their scores **TR** once for each lattice level. Given the new slices **S** and their resulting scores **R**, we create an indicator for qualifying slices with $\mathbf{I} = \mathbf{R}_{:,1} > 0 \wedge \mathbf{R}_{:,4} \geq \sigma$ (min sores and min support), and select $\mathbf{S} = \text{removeEmpty}(\mathbf{I} \cdot \mathbf{S})$ and similarly **R**. Furthermore, we concatenate the old top-K—or zero row matrices for initial top-K selection—and new slices (e.g., $\mathbf{S} = \text{rbind}(\mathbf{TS}, \mathbf{S})$), and extract the new top-K as follows:

```
IX = order(R, by=1, decreasing=TRUE, index.return=TRUE);
IX = IX[1:min(K,nrow(IX)),];
P = table(seq(1,nrow(IX)), IX, nrow(IX), nrow(S));
```

Here, **IX** holds the original row indexes of the sorted, concatenated scores. We take the first K rows, and construct a selection matrix **P** to extract the top-K slices and scores via $\mathbf{TS} = \mathbf{P} \odot \mathbf{S}$ and $\mathbf{TR} = \mathbf{P} \odot \mathbf{R}$.

# 5 EXPERIMENTS

Our experiments study the pruning effectiveness, top-K characteristics, end-to-end runtime efficiency, and scalability of SliceLine. We find that the combination of effective pruning and vectorized evaluation yields very good performance on a variety of real datasets.

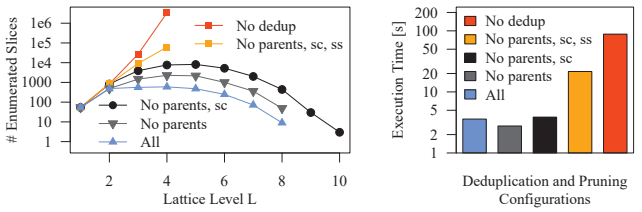
## 5.1 Experimental Setting

**HW Environment:** Most of our experiments are conducted on a *scale-up node* with two Intel Xeon Gold 6238 CPUs @ 2.2-2.5 GHz (56 physical/112 virtual cores), 768 GB DDR4 RAM at 2.933 GHz balanced across 6 memory channels per socket, 2×480 GB SATA SDDs (system/home), and 12 × 2 TB SATA SDDs (data). For scalability experiments, we use an additional *scale-out cluster* of 1+12 nodes, each having a single AMD EPYC 7302 CPU at 3.0−3.3 GHz (16 physical/32 virtual cores), 128 GB DDR4 RAM at 2.933 GHz balanced across 8 memory channels, 2 × 480 GB SATA SDDs (system/home), 12 × 2 TB SATA HDDs (data), and 2 × 10Gb Ethernet. Furthermore, we use Ubuntu 20.04.1, OpenJDK Java 1.8.0_265 with `-Xmx600g -Xms600g`, Apache Hadoop 2.7.7, and Apache Spark 2.4.7.

**Implementation:** We implemented SliceLine on different ML systems—specifically R 4.0.4 with the `doMC` package [52], and Apache SystemDS [11] 2.0.0 (as of 03/2021)[3]—to show SliceLine's general applicability. On both systems, these linear algebra programs require only 200-300 lines of code. We primarily employ SystemDS for its good sparse linear algebra support [67], and hybrid runtime plans of local and distributed operations [12].

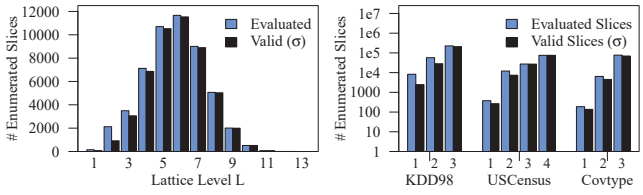
**Datasets and ML Algorithms:** Efficient slice finding heavily relies on effective pruning, and thus, data characteristics. Accordingly, we evaluate SliceLine on real datasets from the UCI repository [23] and Criteo [37]. Table 1 summaries the datasets, their dimensions, and number of features after one-hot encoding. Salaries [55] is a very small dataset, used for an ablation study of pruning and deduplication. The rightmost column also indicates the regression or classification tasks, where we use linear regression (`lm`), and multinomial logistic regression (`mlogit`). We pre-process these datasets by recoding categorical features, binning continuous features (except labels) into 10 equi-width bins, and dropping ID columns. For USCensus—which does not have labels—we derive artificial labels by K-Means clustering. These integer-encoded feature matrices $\mathbf{X}_0$ and error vectors **e** (squared loss for regression, inaccuracy for classification) are then materialized, and we measure the end-to-end runtime of slice finding, including I/O and one-hot encoding.

[3] https://github.com/apache/systemds/blob/master/scripts/builtin/slicefinder.dml




(a) Slice Enumeration per Level (b) Total Runtime

**Figure 3: Pruning Techniques on Salaries** $2 \times 2$.




(a) Adult (b) KDD98, USCensus, Covtype

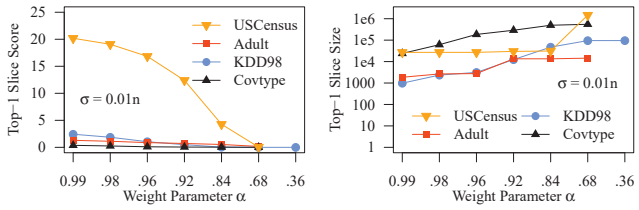
**Figure 4: Dataset Slice Enumeration** (# slices at level).

## 5.2 Pruning Effectiveness

As a basis for understanding the end-to-end runtime experiments, we first evaluate the pruning effectiveness and resulting enumeration characteristics with defaults $\alpha = 0.95$ and $\sigma = \lceil n/100 \rceil$.

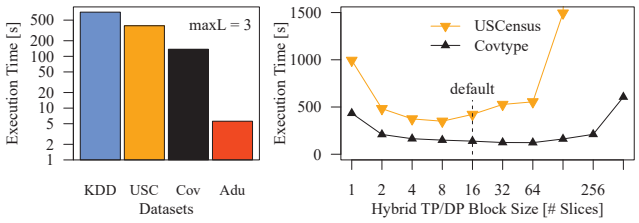
**Pruning Techniques:** We conduct an ablation study with the very small Salaries dataset, with 2x replicated rows and columns to create additional correlation. Figure 3(a) shows the number of enumerated slices per level ($m = 10$, and thus, $L \leq 10$) with (1) all pruning techniques, (2) no parent handling, (3) no parent handling, no score pruning, (4) no parent handling, no score and size pruning, and (5) no pruning and no deduplication. We observe that even on this tiny dataset, pruning and deduplication are crucial to avoid enumerating the exponential candidate space. Furthermore, *all* pruning techniques (by min support, top-K scores, and missing parents) contribute to reducing the number of enumerated slices. Figure 3(b) shows that this pruning effectiveness often directly translates to runtime improvements as well. Configurations without deduplication or pruning ran out-of-memory after 4 levels.

**Different Datasets:** Our evaluation uses datasets with different data and enumeration characteristics. Figure 4 shows the evaluated slices with all pruning techniques enabled. For the Adult dataset (Figure 4(a))—with a mix of large and small slices, which was already used in SliceFinder [18, 19]—we see good pruning effectiveness, a



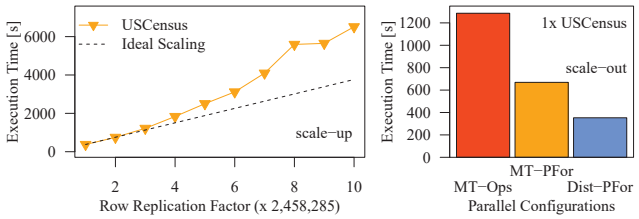

(a) Weight $\alpha \rightarrow sc$ (b) Weight $\alpha \rightarrow |S|$

**Figure 5: Scores with Varying Scoring Parameters.**




(a) All Datasets (b) Hybrid Task-/Data-parallel Execution

**Figure 6: Local End-to-End Runtime.**




(a) Data Size (Scale-up) (b) Local/Distributed (Scale-out)

**Figure 7: Scalability with Data Size and Parallelism.**

moderate number of slices per level, and early termination after level 12 of 14. In contrast, for the other datasets in Figure 4(b), we see more extreme behavior. KDD98 has thousands of qualifying basic slices, while USCensus and Covtype are known to exhibit correlations [24, 39]. There are several correlated column groups, where even conjunctions of many features yield large slices. This characteristic makes these datasets challenging for exact enumeration. Accordingly, we had to limit $\lceil L \rceil$ to 3 or 4 levels, respectively. However, on all datasets, the number of candidate slices closely matches the number of valid slices still exceeding $\sigma$. As we descend the lattice, the differences are also decreasing. These observations indicate that our pruning techniques are indeed very effective.

## 5.3 Scoring Parameters

The scoring function has several parameters and constraints. Here, we study their impact on the resulting top-K slice scores and sizes.

**Weight Parameter $\alpha$:** First, we fix $\sigma = n/100$, and vary the weight $\alpha$ in $\{0.36, 0.68, 0.84, 0.92, 0.96, 0.98, 0.99\}$. Figures 5(a) and 5(b) show the scores and sizes of the top-1 slice with $\lceil L \rceil = 3$. With increasing $\alpha$, we see increasing scores and decreasing sizes because the error term in Equation (1) gets more weight. The score change varies across datasets though. As expected, the slice sizes are decreasing because the sizes have less impact. Additionally, we observed that the score difference between levels increases with increasing $\alpha$ because smaller slices are found in deeper levels. Similarly, the score differences between the Top-1 and Top-K (i.e., score ranges) decrease with increasing $\alpha$ because there are more small slices with similar errors. With increasing $\alpha$, the runtime is constant for Covtype, increasing for KDD98 (1.3x), and decreasing for Adult (1.5x) and USCensus (3x) due to better pruning.

**Varying the $\sigma$ Constraint:** In contrast, the minimum support constraint $\sigma$ has less impact. We ran an experiment varying the $\sigma \in [10^{-4}n, 10^{-1}n]$ with $\alpha = 0.95$, $K = 10$ and $\lceil L \rceil = 3$. As expected, for high $\sigma$ the scores reduce as some slices do not satisfy $|S| \geq \sigma$ anymore. However, even with very small $\sigma$, the scores remained similar to the above. The reason is the size term in Equation (1), which also counteracts too small slices. In contrast to varying $\alpha$, $\sigma$ has significant impact on the runtime though. As we reduce the minimum support constraint $\sigma$, we see increasing runtime of more than an order of magnitude for some of the datasets.

## 5.4 End-to-end Runtime

**Local Runtime:** Figure 6(a) shows the total runtime for all datasets with defaults $\sigma = n/100$, $\alpha = 0.95$, and again $\lceil L \rceil = 3$. We observe very good performance even for datasets with many rows (USCensus), many features (KDD98), and strong correlations (USCensus, Covtype). Figure 6(a) further analyzes the hybrid slice evaluation (Section 4.4). The configurable block size $b$ generalizes both task-parallel ($b = 1$) and data-parallel ($b = \text{nrow}(\mathbf{S})$) execution. Increasing block sizes allow for scan sharing, yielding a 2.8x improvement on USCensus. However, once the block size—and thus, size of intermediates ($\text{nrow}(\mathbf{X}) \times b$)—gets too large, blocking causes overhead for allocation and eviction, rendering it slower than pure task-parallel execution. This effect is more pronounced on USCensus because it has more rows (and thus, larger intermediates), and more slices are evaluated (so blocking has a larger impact on end-to-end runtime). Our default is $b = 16$, which offers a good balance.

**ML Systems Comparison:** For our new problem formulation, there is unfortunately no baseline for direct comparison. However, we can share two relevant data points. First, we also implemented SliceLine in R, which takes 200.4s on the Adult dataset with default SliceLine and `doMC` parameters, and $\lceil L \rceil = 3$. In contrast, our primary implementation in SystemDS DML takes 5.6s with the same configuration, likely due to more efficient sparse linear algebra operations. Second, the SliceFinder paper also used the Adult dataset and reported >100s [19] for a hand-crafted single-worker lattice search with level-wise termination. In all our experiments—even with $\lceil L \rceil = \infty$ and different $\sigma$, $\alpha$—SliceLine never exceeded 68s.

**Scalability with # Rows:** For evaluating the scalability of SliceLine, we replicated the USCensus dataset up to 10 times row-wise. This setup preserves the slice enumeration characteristics due to the relative min-support constraint $\sigma = n/100$ and relative slice errors. SliceLine evaluates 12,021 slices in level 2, and 27,288 slices in level 3 ($\lceil L \rceil = 3$). Figure 7(a) shows the scalability (on the scale-up node with 112 vcores) with increasing data size, where *ideal scaling* refers to the USCensus runtime multiplied by the replication factor. With constant block size $b = 4$, we see a moderate deterioration due to larger intermediates and increasing memory pressure (garbage collection). Figure 7(b) compares different parallelization strategies (on the scale-out cluster with $12 \times 32 = 384$ vcores) using HDFS and Spark [73]. Comparing MT-Ops (multi-threaded operations) and MT-PFor (multi-threaded operations and parallel for-loops), we

Table 2: Criteo Slice Enumeration Statistics.

| Lattice Level: | 1 (Init) | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Candidates:** | 75,573,541 | 1,209 | 1,644 | 4,305 | 8,801 | 13,248 |
| **Valid Slices:** | 209 | 668 | 1,622 | 4,305 | 8,801 | 13,248 |
| **Elapsed Time:** | 933s | 1,219s | 1,314s | 1,496s | 1,906s | 2,527s |

observe a 2x improvement with MT-PFor because it avoids barriers per operation and thus, reaches higher utilization. With distributed slice evaluation (Dist-PFor), we get an additional 1.9x improvement because all 12 nodes are utilized, but there is overhead for Spark context creation, data and slice broadcasting, result aggregation, and a non-negligible serial fraction outside slice evaluation.

**Scalability with # Columns:** Additionally, we use 1 of 24 days from Criteo [37]—which we call CriteoD21—for studying SliceLine on a large dataset of 192M rows and 76M columns, with distributed data-parallel operations in our scale-out cluster of 1+12 nodes. After one-hot encoding the dataset becomes ultra-sparse (density 4.9e-7), which is challenging for distributed operations on block-partitioned (1K × 1K) matrices. Table 2 shows the enumeration statistics including elapsed time up to lattice level 6. Due to many categories per original feature, only 209 of 75,573,541 features satisfy the minimum support constraint. Pruning then keeps the number of pair-candidates moderately small and close to the true number of valid slices. Similar to USCensus and Covtype, we observe again correlations which hinders early termination. Due to SystemDS' hybrid runtime plans of local and distributed operations, and dynamic recompilation of appropriate physical operators across iterations, we see good scalability and a reasonable total elapsed time <45min.

## 6 ADDITIONAL RELATED WORK

Besides model debugging (Section 1) and data mining problems (Section 2.3), SliceLine is also related to tree models, maximum inner product search, data slicing in ML systems, and feature selection.

**Decision Trees:** Our initial hand-coded implementations of SliceLine were inspired by work on distributed training of decision trees. PLANET [53] trains a decision tree via MapReduce using a breadth-first, queue-based node expansion approach. Large nodes are added to an MR queue (for shared scans), while small nodes are added to an in-memory queue (for processing entire subtrees). Later, Yggdrasil [1] introduced a vertical data partitioning by features. In contrast to greedy decision tree training, slice finding considers overlapping slices and enumerates the entire lattice. Our vectorized slice evaluation is similar to the breadth-first enumeration and shared scans in PLANET [53]. However, SliceLine relies on linear algebra and distributed matrix multiplications [12, 29, 71] for scaling to large data, many slices, or both. Recent work on recursive model training in SimSQL [31, 32], and the compilation of tree models into tensor operations in Hummingbird [49, 51], follow similar goals of scaling to large data or models, and vectorized operations.

**Maximum Inner Product Search:** Naïve user recommendations with low-rank matrix factorization models require scoring all items, and selecting the top-K. This maximum inner product search (MIPS) problem has been addressed via dedicated index structures—such as LEMP [66] and FLEXIPRO [40]—to provide recommendations in sub-linear time. SliceLine could similarly benefit from specialization (e.g., $(\mathbf{S} \odot \mathbf{S}^\top) = (L-2)$, or $(\mathbf{X} \odot \mathbf{S}^\top) = L$). However, we aim for a simple design, mostly regular data access, and good parallelization via linear algebra. Interesting, OPTIMUS [2] made a similar observation that plain matrix multiplication can outperform existing MIPS index structures, and thus, used a hybrid approach. Similar, SWOLE [20] uses predicate pull-ups to yield better data access patterns in compiled queries.

**Slicing in ML Systems:** In recent years, several systems added support for data slicing and model analysis. The original SliceFinder work [18, 19] was developed in the context of Google TensorFlow TFX [10] and apparently a form of it was added to TFX Model Analysis, executed on top of Apache Beam [22]. Other systems with shared goals like Amazon SageMaker [41] have similar components for data and model validation. These systems often pair data slicing with interactive visualizations, where users can explore slices manually. Finally, slice finding also relates to systems for data augmentation and weak supervision such as Snorkel [9, 57]. Recent work added so-called slicing functions to identify critical subsets of data for label generation or slice-aware predictions [16].

**Feature Selection and Explainability:** Finding problematic slices as conjunctions of features is also related to feature engineering. First, sparse n-gram token featurization [27] also derives counts from lattice parent nodes for pruning rare n-grams. Feature selection methods like forward selection [69] and Lasso [68] share the goal of selecting feature subsets with high influence. However, these techniques focus on overall model performance. Second, techniques for discovering features and feature interactions analyze joint feature effects (e.g., SAFE [62], BP-FIS [17]), or weighted embeddings of feature combinations (e.g., AutoCTR [64], AutoFIS [43]). In contrast to our exact enumeration, these techniques focus on pair interactions or rely on greedy enumeration. Third, model explainability derives explanations from features to model predictions [46]. Common techniques include interpretable models like decision trees, causal learning and interpretability [45], and generic techniques like SHAP (shapely additive explanations) [44]. For efficiency, these techniques often make additional assumptions.

## 7 CONCLUSIONS

We introduced SliceLine, a model debugging technique for finding data slices, where a trained ML model performs worse than overall. Inspired by ideas from slice finding and frequent itemset mining, SliceLine leverages monotonicity properties and upper bounds for effective pruning. Working with slice finding for almost two years, we draw three conclusions. First, slice finding is a valuable debugging technique for both model understanding and data sourcing. Second, the combination of effective pruning techniques, and a vectorized linear algebra implementation makes slice finding practical in both local or distributed environments, despite its exponential search space. Third, SliceLine is a proof-of-concept for building even complex, enumeration-based data mining algorithms on top of ML systems. This is a profound outcome because it enables a seamless integration in data science workflows without boundary crossing, and makes such tasks amenable to parallelization and HW acceleration. Interesting future work includes priority-based enumeration (e.g., based on errors or classes) [33], slice finding for bias and fairness (instead of accuracy) [3, 72], and adopting the SliceLine enumeration ideas to other data mining problems.

# REFERENCES


[1] Firas Abuzaid, Joseph K. Bradley, Feynman T. Liang, Andrew Feng, Lee Yang, Matei Zaharia, and Ameet S. Talwalkar. 2016. Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale. In *NeurIPS*. 3810–3818.

[2] Firas Abuzaid, Geet Sethi, Peter Bailis, and Matei Zaharia. 2019. To Index or Not to Index: Optimizing Exact Maximum Inner Product Search. In *ICDE*. 1250–1261.

[3] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna M. Wallach. 2018. A Reductions Approach to Fair Classification. In *ICML (PMLR)*, Vol. 80. 60–69.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. In *SIGMOD*. 207–216.

[5] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*. 487–499.

[6] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine Bias – There's software used across the country to predict future criminals. And it's biased against blacks. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

[7] Abolfazl Asudeh, Zhongjun Jin, and H. V. Jagadish. 2019. Assessing and Remedying Coverage for a Given Dataset. In *ICDE*. 554–565.

[8] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLOS ONE* 10, 7 (2015), 1–46.

[9] Stephen H. Bach, Daniel Rodriguez, Yintao Liu, Chong Luo, Haidong Shao, Cassandra Xia, Souvik Sen, Alexander Ratner, Braden Hancock, Houman Alborzi, Rahul Kuchhal, Christopher Ré, and Rob Malkin. 2019. Snorkel DryBell: A Case Study in Deploying Weak Supervision at Industrial Scale. In *SIGMOD*. 362–375.

[10] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *SIGKDD*. 1387–1395.

[11] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*.

[12] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.

[13] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564.

[14] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konecný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards Federated Learning at Scale: System Design. In *MLSys*.

[15] Douglas Burdick, Manuel Calimlim, Jason Flannick, Johannes Gehrke, and Tomi Yiu. 2005. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE Trans. Knowl. Data Eng.* 17, 11 (2005), 1490–1504.

[16] Vincent S. Chen, Sen Wu, Alexander J. Ratner, Jen Weng, and Christopher Ré. 2019. Slice-based Learning: A Programming Model for Residual Learning in Critical Data Slices. In *NeurIPS*. 9392–9402.

[17] Yifan Chen, Pengjie Ren, Yang Wang, and Maarten de Rijke. 2019. Bayesian Personalized Feature Interaction Selection for Factorization Machines. In *SIGIR*. 665–674.

[18] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. 2019. Slice Finder: Automated Data Slicing for Model Validation. In *ICDE*. 1550–1553.

[19] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. 2020. Automated Data Slicing for Model Validation: A Big Data - AI Integration Approach. *IEEE Trans. Knowl. Data Eng.* 32, 12 (2020), 2284–2296.

[20] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *ICDE*. 1273–1284.

[21] Jeff Dean. 2016. Building Machine Learning Systems that Understand. In *SIGMOD*. 1.

[22] Mike Dreves, Gene Huang, Zhuo Peng, Neoklis Polyzotis, Evan Rosen, and Paul Suganthan C. G. 2020. From Data to Models and Back. In *DEEM@SIGMOD*.

[23] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[24] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB* 9, 12 (2016), 960–971.

[25] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *PVLDB* 8, 1 (2014), 61–72.

[26] Google. 2020. TensorFlow Federated: Machine Learning on Decentralized Data . https://www.tensorflow.org/federated

[27] John Hallman. 2021. Efficient Featurization of Common N-grams via Dynamic Programming. https://sisudata.com/blog/efficient-featurization-common-n-grams-via-dynamic-programming

[28] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD*. 1–12.

[29] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*. 1–12.

[30] Martin Jägersand. 1995. Saliency Maps and Attention Selection in Scale and Spatial Coordinates: An Information Theoretic Approach. In *ICCV*. 195–202.

[31] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS. *PVLDB* 12, 7 (2019), 822–835.

[32] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2020. Declarative Recursive Computation on an RDBMS. *SIGMOD Rec.* 49, 1 (2020), 43–50.

[33] Zhongjun Jin, Mengjing Xu, Chenkai Sun, Abolfazl Asudeh, and H. V. Jagadish. 2020. MithraCoverage: A System for Investigating Population Bias for Intersectional Fairness. In *SIGMOD*. 2721–2724.

[34] Peter Kairouz, Brendan McMahan, and Virginia Smith. 2020. Federated Learning Tutorial. In *NeurIPS*. https://slideslive.com/38935813/federated-learning-tutorial

[35] Adam Kirsch, Michael Mitzenmacher, Andrea Pietracaprina, Geppino Pucci, Eli Upfal, and Fabio Vandin. 2009. An Efficient Rigorous Approach for Identifying Statistically Significant Frequent Itemsets. In *PODS*. 117–126.

[36] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*. 1717–1722.

[37] Criteo AI Lab. 2020. Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[38] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. 2016. Analyzing Classifiers: Fisher Vectors and Deep Neural Networks. In *CVPR*. 2912–2920.

[39] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. 2019. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD*. 1517–1534.

[40] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *SIGMOD*. 835–850.

[41] Edo Liberty, Zohar S. Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Sundar Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. 731–737.

[42] Yin Lin, Yifan Guan, Abolfazl Asudeh, and H. V. Jagadish. 2020. Identifying Insufficient Data Coverage in Databases with Multiple Relations. *PVLDB* 13, 11 (2020), 2229–2242.

[43] Bin Liu, Chenxu Zhu, Guilin Li, Weinan Zhang, Jincai Lai, Ruiming Tang, Xiuqiang He, Zhenguo Li, and Yong Yu. 2020. AutoFIS: Automatic Feature Interaction Selection in Factorization Models for Click-Through Rate Prediction. In *KDD*. 2636–2645.

[44] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NeurIPS*. 4765–4774.

[45] Marloes H. Maathuis. 2020. Causal Learning (Breiman Lecture). In *NeurIPS*.

[46] Raha Moraffah, Mansooreh Karami, Ruocheng Guo, Adrienne Raglin, and Huan Liu. 2020. Causal Interpretability for Machine Learning - Problems, Methods and Evaluation. *SIGKDD Explor.* 22, 1 (2020), 18–33.

[47] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations. In *SIGMOD*. 1589–1606.

[48] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2020. Query Optimization for Faster Deep CNN Explanations. *SIGMOD Rec.* 49, 1 (2020), 61–68.

[49] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *OSDI*. 899–917.

[50] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *PVLDB* 13, 11 (2020), 2159–2173.

[51] Supun Nakandala1, Gyeong-In Yu, Markus Weimer, and Matteo Interlandi. 2019. Compiling Classical ML Pipelines into TensorComputations for One-size-fits-all Prediction Serving. In *Systems for ML@NeurIPS*.

[52] Hong Ooi and Steve Weston. 2019. doMC: Foreach Parallel Adaptor for 'parallel'. https://cran.r-project.org/web/packages/doMC/index.html

[53] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce.

*PVLDB* 2, 2 (2009), 1426–1437.

[54] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB* 8, 10 (2015), 1082–1093.

[55] Nisha Patel. 2018. Exploratory Analysis of the Salaries Dataset. https://rstudio-pubs-static.s3.amazonaws.com/399031_67d0656319ae4a288083cfea7a6a4200.html http://forge.scilab.org/index.php/p/rdataset/source/tree/master/csv/car/Salaries.csv.

[56] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. In *SIGMOD*. 1723–1726.

[57] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB* 11, 3 (2017), 269–282.

[58] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *KDD*. 1135–1144.

[59] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018), 1781–1794.

[60] Sebastian Schelter, Tammo Rukat, and Felix Bießmann. 2020. Learning to Validate the Predictions of Black Box Classifiers on Unseen Data. In *SIGMOD*. 1289–1299.

[61] Benjamin Schlegel. 2014. *Frequent Itemset Mining on Multiprocessor Systems*. Ph.D. Dissertation. Dresden University of Technology.

[62] Qitao Shi, Ya-Lin Zhang, Longfei Li, Xinxing Yang, Meng Li, and Jun Zhou. 2020. SAFE: Scalable Automatic Feature Engineering Framework for Industrial Tasks. In *ICDE*. 1645–1656.

[63] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *ICLR Workshops*.

[64] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. 2020. Towards Automated Neural Interaction Discovery for Click-Through Rate Prediction. In *KDD*. 945–955.

[65] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. 2015. Automating Model Search for Large Scale Machine Learning. In *SoCC*. 368–380.

[66] Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. 2015. LEMP: Fast Retrieval of Large Entries in a Matrix Product. In *SIGMOD*. 107–122.

[67] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *PVLDB* 11, 13 (2018), 2168–2182.

[68] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58, 1 (1996), 267–288.

[69] William N. Venables and Brian D. Ripley. 2002. *Modern applied statistics with S, 4th Ed.* Springer.

[70] Anna Volokitin, Michael Gygli, and Xavier Boix. 2016. Predicting When Saliency Maps are Accurate and Eye Fixations Consistent. In *CVPR*. 544–552.

[71] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. 2016. Matrix Computations and Optimization in Apache Spark. In *SIGKDD*. 31–38.

[72] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez-Rodriguez, and Krishna P. Gummadi. 2017. Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment. In *WWW*. 1171–1180.

[73] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.

[74] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. 1997. New Algorithms for Fast Discovery of Association Rules. In *SIGKDD*. 283–286.

[75] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *ECCV*, David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.), Vol. 8689. 818–833.

[76] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*. 265–276.

[77] Jing Zhou, Dean P. Foster, Robert A. Stine, and Lyle H. Ungar. 2005. Streaming Feature Selection using Alpha-investing. In *SIGKDD*. 384–393.