Incremental SliceLine for Iterative ML Model Debugging under Updates

Frederic Caspar Zoepffel [1], Christina Dionysio [2], and Matthias Boehm [2]

Abstract: SliceLine is a model debugging technique for finding the top-K worst slices (in terms of conjunctions of attributes) where a trained machine learning (ML) model performs significantly worse than on average. In contrast to other slice finding techniques, our prior work SliceLine introduced an intuitive scoring function, effective pruning strategies, and fast linear-algebra-based evaluation strategies. Together, SliceLine is able to find the exact top-K worst slices in the full lattice of possible conjunctions of attributes in reasonable time. Recently, we observe a major trend towards iterative algorithms that incrementally update the dataset (e.g., selecting samples, augmenting with new instances) and ML model. Fully computing SliceLine from scratch for every update is, however, unnecessarily wasteful. In this paper, we introduce an incremental problem formulation of SliceLine, new pruning strategies that leverage state of previous slice finding runs on a modified dataset, and an extended linear-algebra-based enumeration algorithm. Our experiments show that incremental SliceLine yields robust runtime improvements of up to an order of magnitude faster than full SliceLine, while still allowing effective parallelization in local, distributed, and federated environments.

Keywords: ML Model Debugging, Slice Finding, Linear Algebra, Large-scale Data Processing, Incremental Maintenance

1 Introduction

Machine learning (ML) models are increasingly deployed in data-driven applications of various domains such as science, transportation, finance, energy, earth-observation, and health-care [Bo20, SB21]. Crucial steps for ensuring reliable predictions are *input data validation* [Po17, Sc18], *data provenance and lineage tracing* [CCT09, PRB21], *data cleaning* [Ne21, SKB23], *ML model debugging* [Ch19, Ch20, SB21], and *explanations and fairness* [Ge14, SSW19, NKP20, Zh21]. Especially model debugging can help in finding silent but severe problems of the deployed ML models.

Slice Finding: Slice finding [Ch19] is a very effective model debugging technique which seeks to find the top-K worst slices (i.e., conjunctions of attributes like `{venue=BTW AND location=Bamberg}`), where a trained ML model substantially under-performs. These slices or sub-groups can then be scrutinized and improved through additional data acquisition, rules, or specialized models [FBL11, Hi23]. Existing work includes explanation tables [Ge14] for data summarization, the original Slice Finder [Ch19, Ch20], and SliceTuner [TW21] for goal-oriented data acquisition. More general work on identifying insufficient

[1] TU Berlin, zoepffel@campus.tu-berlin.de, https://orcid.org/0009-0007-1046-3585
[2] TU Berlin & BIFOLD, dionysio@tu-berlin.de, https://orcid.org/0009-0002-1295-0460; matthias.boehm@tu-berlin.de, https://orcid.org/0000-0003-1344-3663

data coverage [AJJ19, Li20, Ji20] share several enumeration principles. In contrast to other slice finding techniques, our SliceLine [SB21] approach introduced an intuitive scoring function, effective pruning strategies, and fast linear-algebra-based evaluation strategies. However, all existing work perform slice finding once from scratch for a given dataset.

Data-centric ML Pipelines: Over the last couple of years we have seen a clear development towards data-centric ML pipelines. The premise of data-centric ML is that creating high-quality datasets with good coverage of the target domain is often more important for model utility than devising new ML algorithms and neural architectures. Accordingly, data-centric ML pipelines extend traditional ML pipelines—of feature engineering, hyper-parameter tuning, and model training—by additional outer loops for data engineering such as data validation, data cleaning, and data augmentation. The same characteristic can be observed for large language models (LLMs), where new state-of-the-art, pretrained models mostly keep the transformer architecture constant and innovate on scaling and differently composing the training data [Br20, An23]. In the context of data-centric ML, there is another major trend towards goal-oriented data discovery in data lakes [NAJ21, LYK21, Ch22] as well as learned sampling and augmentation [Wa18, Cu19, Ki23]. Many of these new techniques are iterative and thus, require repeated model training and model debugging.

Example 1 (Distribution Tailoring) *A representative example of iterative model debugging is data distribution tailoring for ML in* PLUTUS *[Ch24]. In this demo, we combined (1) existing data acquisition techniques to enrich the training dataset under some desired distribution requirements, with (2) SliceLine for obtaining slices that would benefit from additional training examples. Iteratively exploring such refinements requires repeated model re-training and slice finding. Since model re-training can simply perform warm-starting (start from the old model and fine-tune this model with the new data), repeatedly running SliceLine from scratch became the new major bottleneck.*

Contributions: Our primary contribution is making a case for *incremental slice finding* and introducing a first incremental SliceLine algorithm that solves this new problem. Our detailed technical contributions are:

- **Problem Formulation:** We extend the SliceLine problem formulation for incremental slice finding (see Section 2).
- **Incremental SliceLine:** As a basis for incremental SliceLine, we then devise new min-support and score pruning strategies (see Section 3), and integrate state collection and pruning into the SliceLine algorithm (see Section 4). Intuitively, we exploit the knowledge about the sizes, errors, and scores of previously enumerated and (almost) unchanged slices as well as the previous top-K slices.
- **Experiments:** Finally, we share the results of an extensive experimental evaluation of micro benchmarks and end-to-end iterative dataset refinement (see Section 5). The full reproducibility package is available at https://github.com/damslab/reproducibility/tree/master/btw2025-incsliceline-p111.

2 Background and Problem Formulation

As a basis for incremental SliceLine, we first summarize the SliceLine problem, and then introduce an extended problem formulation for incremental slice finding.

2.1 SliceLine

For the sake of a self-contained presentation, in the following, we briefly summarize the SliceLine [SB21] notation, scoring function, optimization objective, and algorithm sketch. The original SliceLine is implemented as a built-in function in Apache SystemDS[3] [Bo20] but meanwhile there also exists an independent Python library[4] by other authors.

Notation: Let $\mathbf{X}$ be an $n \times m$ input feature matrix with recoded and binned (i.e., integer) features $\{F_1, F_2, \ldots, F_m\}$ and $\mathbf{y}$ be a continuous or categorical $n \times 1$ label vector. Every feature F_j has a number of distinct values (domain) d_j. We then train an arbitrary classification or regression model $\mathbf{M}$ on $\mathbf{X}_{\text{train}}$ and $\mathbf{y}_{\text{train}}$. Applying $\mathbf{M}$ on $\mathbf{X}$ (train or test) yields the predictions $\hat{\mathbf{y}}$ and errors $\mathbf{e} = \text{err}(\mathbf{y}, \hat{\mathbf{y}})$. Common error functions err() are (in-)accuracy $\mathbf{e} = (\mathbf{y} \neq \hat{\mathbf{y}})$ for classification and squared loss $\mathbf{e} = (\mathbf{y} - \hat{\mathbf{y}})^2$ for regression. Finally, a data slice S is defined as conjunctions of equivalence predicates $F_j = v$ with $v \in d_j$ and thus, a subset of rows of $S \subseteq \mathbf{X}$ of size $|S|$. We further define $\mathbf{es}$ as a vector of slice tuple errors, $se = \Sigma_{i=1}^{|S|} \mathbf{es}_i$ as the total error of a slice, $sm = \max_{i=1}^{|S|} \mathbf{es}_i$ as the maximum tuple error of a slice, and $\overline{se} = se/|S|$ as the average slice error.

Scoring Function: In order to balance slice sizes and errors, we further use the following scoring function for ranking slices (with $\alpha \in (0, 1]$ as a weighing parameter):

$$sc = \alpha \left(\frac{\overline{se}}{\overline{e}} - 1 \right) - (1 - \alpha) \left(\frac{n}{|S|} - 1 \right), \tag{1}$$

which is defined for non-empty slices and otherwise assumed negative. The score of $\mathbf{X}$ is zero, and all positive scores are interesting problematic slices.

Slice Finding Problem: Given the input feature matrix $\mathbf{X}$, the error vector $\mathbf{e}$, and a user-provided integer K, the problem is then to find the top-K slices $\mathbf{TS}$ (ordered by sc) from the lattice $\mathcal{L}$ of all slices. Inspired by frequent itemset mining, we establish the additional minimum-support constraint $|S| \geq \sigma$.

SliceLine Algorithm: The SliceLine enumeration algorithm then exploits the monotonicity property of slice sizes and total errors in order to efficiently find the exact solution to the above search problem. For example, the size of the slice `{venue=BTW AND location=Bamberg}` is guaranteed to be smaller than or equal to the minimum size of

[3] Apache SystemDS SliceLine: https://github.com/apache/systemds/blob/main/scripts/builtin/sliceLine.dml
[4] Sliceline Python library: https://github.com/DataDome/sliceline

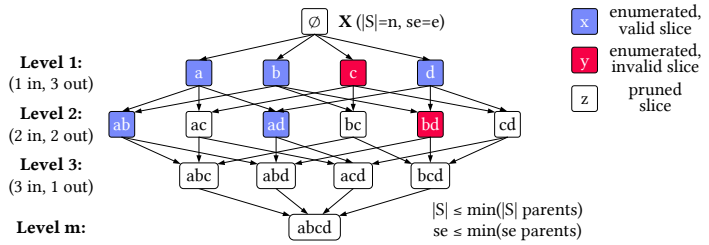


Fig. 1: Example Lattice, Slice Properties, and Pruning (blue and red indicate enumerated and evaluated slices, while white slices—and all slices reachable from them—are pruned before evaluation).

`{venue=BTW}` or `{location=Bamberg}`. The linear-algebra-based algorithm—which uses cumulative aggregates [BER19] for preprocessing—then iterates through the m levels of the lattice $\mathcal{L}$ enumerates and prunes candidate slices, and evaluates all candidates per lattice level in a single matrix multiplication with the one-hot-encoded $\mathbf{X}$. Figure 1 shows an example lattice with highlighted enumerated candidate slices (blue and red) as well as actually valid slices (blue). Invalid slices are either violating the min-support threshold or the upper-bound scores (and thus, their reachable slices cannot qualify for the top-K either).

2.2 Incremental Slice Finding

Incremental slice finding addresses the challenge of repeatedly executing SliceLine on slightly updated datasets and models. Figure 2 shows the overall pipeline, where we aim to support both insertions of tuples (e.g., iterative data acquisition) as well as deletion of tuples (e.g., learned sampling, dropping dirty data).

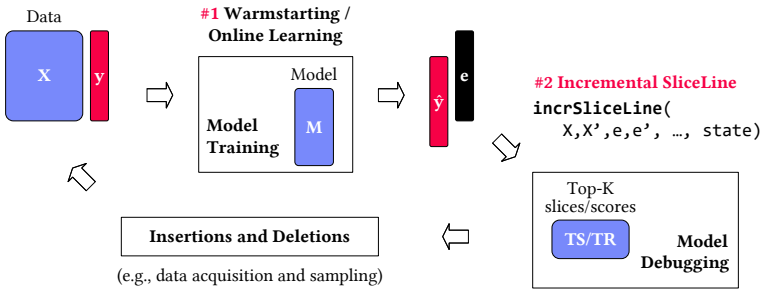


Fig. 2: Context of Repeatedly Updated Data and Models (first invocation trains the model and performs slice finding from scratch, subsequent invocations incrementally adjust the model and top-k slices).

Incremental Slice Finding Problem: Similar to the basic slice finding problem, our goal for incremental slice finding is to find the top-K slices **TS** (ordered by sc) from the lattice $\mathcal{L}$ of all slices. Additionally, we want to minimize the number of unnecessarily evaluated

slices. In order to provide all context information of changes, we pass the original feature matrix $\mathbf{X}$, the original error vector $\mathbf{e}$, the modified feature matrix $\mathbf{X}'$ (after data changes) and the modified errors $\mathbf{e}'$ (after model retraining), as well as a list for state from previous SliceLine invocations on $\mathbf{X}$. To deal with insertions and deletions, we define $\mathbf{X}'$ and $\mathbf{e}'$ as:

$$\mathbf{X}' = \begin{bmatrix} \Delta^- \mathbf{X} \\ \Delta^c \mathbf{X} \\ \Delta^+ \mathbf{X} \end{bmatrix} \qquad \mathbf{e}' = \begin{bmatrix} \Delta^- \mathbf{e} \\ \Delta^c \mathbf{e} \\ \Delta^+ \mathbf{e} \end{bmatrix} \tag{2}$$

where $\Delta^- \mathbf{X}$ comprises all deleted rows, $\Delta^c \mathbf{X}$ comprises all unchanged rows, $\Delta^+ \mathbf{X}$ comprises all added rows. Accordingly, the original feature matrix $\mathbf{X}$ is equivalent to the concatenation of $\Delta^- \mathbf{X}$ and $\Delta^c \mathbf{X}$ (modulo changes in positions which we do consistently for $\mathbf{X}$ and $\mathbf{X}'$). The same notation also applies for the error vector $\mathbf{e}'$. Note that after model adjustments, the errors of unchanged tuples $\Delta^c \mathbf{e}$ might have changed as well.

Two-Phase Algorithm Semantics: Executing an incremental slice finding algorithm should yield exactly the same top-K results as the non-incremental algorithm. Furthermore, we have to distinguish two phases of incremental slice finding:

1. **Initialization:** On the first invocation of incremental slice finding for a dataset, the algorithm needs to perform a full slice finding from scratch, but simultaneous collect intermediate results of the enumeration process as state.
2. **Exploitation:** For further invocations of incremental slice finding, the passed state can be exploited for more efficiently computing the top-K set. While exploiting the state, new state of the current run needs to be collected for future invocations.

3 Pruning Strategies

In this section, we first introduce our pruning principles and discuss the pruning strategies separately before integrating them into the overall incremental SliceLine algorithm.

3.1 Pruning Principles

Our overall pruning principles—which largely influence the design of incremental SliceLine—are summarized as follows:

- **Correctness:** All pruning strategies that are applied by default should be lossless, that is, incremental SliceLine must produce exactly the same results as SliceLine from scratch. Accordingly, we only prune slices (and slices reachable from there) if we can guarantee that the pruning does not affect the top-K set.
- **Incomplete State:** The original SliceLine algorithm—and incremental SliceLine during initialization—already applies a variety of pruning strategies (by size, by

upper-bound scores, by number of parent nodes) [SB21], which leads to the fact of having only a small subset of slices actually being evaluated. During state collection, we gather the enumerated slices and their statistics, but after updates of the dataset need to robustly handle this incomplete state.

- **Lightweight Pruning:** Fundamentally, our goal is not to exploit all pruning opportunities possible but aim for a lightweight pruning, which is robust (in terms of correctness and incomplete state) and practical (in terms of yielding end-to-end runtime improvements compared to the fast, linear-algebra-based SliceLine).

Honoring these principles, we made the *major design decision* to use the skeleton of the original SliceLine algorithm (with its existing pruning strategies and robust enumeration of non-evaluated slices) and extend this algorithm with additional pruning strategies that can be efficiently evaluated on $\Delta^-\mathbf{X}$, $\Delta^+\mathbf{X}$, $\Delta^-\mathbf{e}$, and $\Delta^+\mathbf{e}$ (without unnecessarily accessing the main, potentially very large, dataset $\Delta^c\mathbf{X}$) as part of the normal enumeration.

3.2 Extended Min-Support Pruning

The original min-support pruning exploits the apriori property that a slice can only be frequent if all its parents are frequent (i.e., slice sizes are above the minimum-support constraint). For basic (i.e., single-feature) slices, we simply compute these exactly via $\mathbf{ss}_0 = \text{colSums}(\mathbf{X})^\top$ and prune every slice where $\mathbf{ss}_0 < \sigma$. For enumerated slices of further lattice levels, we compute the upper bound slice sizes $\lceil |S| \rceil = \min_{p \in \mathcal{P} | S_p |}$ (as minimum of its parent slice sizes), and prune slices below the min-support before slice evaluation.

P1 – Unchanged Slice Sizes: In addition to this existing size pruning, we exploit the exactly known sizes of unchanged slices. Given the added tuples $\Delta^+\mathbf{X}$, newly enumerated slice candidates $\mathbf{S}$ at lattice level l, previously enumerated slices $\mathcal{S}$ (list of slice candidate matrices per level of previous invocation), and statistics $\mathcal{R}$ (list of slice statistics matrices per level of previous invocation), we prune candidates as follows (with $\odot$ being a matrix multiplication, $\mathbf{pS} = \mathcal{S}[l]$ the previous slice candidates, and $\mathbf{pR} = \mathcal{R}[l]$ the previous slice statistics):

$$\begin{aligned}
&\mathbf{i1} = \text{colSums}((\Delta^+\mathbf{X} \odot \mathbf{pS}^\top) == l)^\top + \mathbf{pR}[4] < \sigma \quad \textbf{// improved over BTW'25} \\
&\mathbf{i2} = \text{rowMaxs}((\mathbf{S} \odot \mathbf{pS}[\mathbf{i1}]^\top) == l) == 0 \\
&\mathbf{S}' = \mathbf{S}[\mathbf{i2}]
\end{aligned} \tag{3}$$

Intuitively, in the first line, we compute which previous slices are unaffected (i.e., do not appear) in $\Delta^+\mathbf{X}$ and previously did not satisfy the min-support constraint. Utilizing this indicator vector $\mathbf{i1}$, we gather only these pruning candidates, and match them to new slice candidates in $\mathbf{i2}$. Finally, we simply select the subset of slice candidates that cannot be pruned. Note that this pruning by unchanged size (or better not increased sizes) accesses only the small delta data matrix $\Delta^+\mathbf{X}$ but not the large original data matrix $\Delta^c\mathbf{X}$.

Algorithm 1 GetMaxScoreAllFeatures

Input: num rows n, num features m, previous slices $\mathcal{S}$, previous stats $\mathcal{R}$, previous meta $\mathcal{M}$
Output: vector of maximal scores $\mathbf{sc}_{\max}$

1: $\mathbf{sc}_{\max} \leftarrow$ matrix(val = $-\infty$, rows = m, cols = 1)
2: **for all** i **in** 1 **to** $|\mathcal{S}|$ **do**
3: $\quad \mathbf{pS} \leftarrow$ ENCODE($\mathcal{S}[i], \mathcal{M}[i]$)
4: $\quad \mathbf{sc} \leftarrow$ SCORE($\mathcal{R}[i,4], \mathcal{R}[i,2], \bar{\mathbf{e}}, \alpha, n$)
5: $\quad \mathbf{sc}_{\max} = \max(\mathbf{sc}_{\max}, \text{colMaxs}(\mathbf{pS} \cdot \mathbf{sc})^\top)$
6: **return** $\mathbf{sc}_{\max}$

3.3 Extended Score Pruning

Furthermore, we make two extensions to the existing upper-bound score pruning where upper bounds of slice sizes and errors are used to derive an upper bound score of a slice (and transitively for any reachable slice).

P2 – Existing Top-K Scores: The original SliceLine uses the continuously improving top-K set **TK** and its statistics **TR** for pruning by comparing the upper bound score of slice candidates with the minimum score in the top-K set. As we find better slices, this score pruning becomes more and more effective. Our extension is very simple. We take the previous top-K slices and evaluate these few slices (but very good candidates) on the modified dataset $\mathbf{X}'$. This approach ensures much more effective pruning for basic (i.e., single-feature) slices and slice candidates in early iterations, where we otherwise would still need to find good top-K slices. Pruning such early slices is very effective because they eliminate all reachable slices along with them. For a seamless integration, we make the old top-K set our initial top-K, which requires a modification of the MAINTAINTOPK function to handle duplicates. Our vectorized operations sort the candidates descending by score and slice ID, and subsequently eliminate adjacent duplicates via a shifted comparison.

P3 – Unchanged Slice Scores: Our second score pruning utilizes all previous intermediate slice candidates. We aim to compute for every unchanged basic (single-feature) slice the maximum score previously reached across all lattice levels. Algorithm 1 shows the GETMAXSCOREALLFEATURES function, which iterates over lattice levels, and computes these maximum scores. First, we encode the (potentially ID-coded) slices into their one-hot encoded representations, and score these slices using their size and error statistics—but also the new average error $\bar{\mathbf{e}}$ and number of rows $|\mathbf{X}'|$—in a vectorized manner. Second, we broadcast these scores **sc** via an element-wise matrix-vector multiplication to all columns (ones in the one-hot encoded slice representation) in order to finally, compute the max value per column (feature) over all slices of this level. Finally, we utilize this $\mathbf{sc}_{\max}$ vector in order to prune all unaffected ($(\text{colSums}(\Delta^{\pm}\mathbf{X}) == 0)^\top$) basic slices if this maximum score is below 0 (uninteresting slice) or the minimum top-K score.

3.4 Approximate Score Pruning

Inspired by the effectiveness of the pruning by maximally-reachable scores of unchanged slices (**P3**), we devise another pruning strategy that generally applies to all modified slices as well. However, this strategy is approximate as it only computes high-probability maximally-reachable scores of changed, previously enumerated slices, but does not give hard guarantees of not missing a newly qualifying slice (e.g., a slice that was previously pruned due to not satisfying the min-support constraint).

P4 – Changed Slice Scores: In detail, we iterate over the previously enumerated lattice levels like in Algorithm 1 in order to compute these "upper-bounds" $\mathbf{sc}_{\overline{\max}}$. In contrast to the search for unchanged slices, we explicitly take the added and removed tuples and errors (i.e., $\Delta^+\mathbf{X}$, $\Delta^-\mathbf{X}$, $\Delta^+\mathbf{e}$, and $\Delta^-\mathbf{e}$) into account. We compute the size and error deltas as follows:

$$\begin{aligned}
\Delta^+\mathbf{ss} &= \text{rowSums}((\mathbf{pS} \odot \Delta^+\mathbf{X}^\top) == l) \\
\Delta^-\mathbf{ss} &= \text{rowSums}((\mathbf{pS} \odot \Delta^-\mathbf{X}^\top) == l) \\
\Delta^+\mathbf{se} &= \text{rowSums}(((\mathbf{pS} \odot \Delta^+\mathbf{X}^\top) == l) \cdot \Delta^+\mathbf{e}^\top) \\
\Delta^-\mathbf{se} &= \text{rowSums}(((\mathbf{pS} \odot \Delta^-\mathbf{X}^\top) == l) \cdot \Delta^-\mathbf{e}^\top)
\end{aligned} \tag{4}$$

Subsequently, we compute the scores at the interesting points $(\mathbf{ss} - \Delta^-\mathbf{ss}, \mathbf{se} + \Delta^+\mathbf{se})$, $(\mathbf{ss}, \mathbf{se} + \Delta^+\mathbf{ss})$, $(\mathbf{ss} + \Delta^+\mathbf{ss}, \mathbf{se} + \Delta^+\mathbf{se})$, $(\mathbf{ss} + \Delta^+\mathbf{ss} - \Delta^-\mathbf{ss}, \mathbf{se} + \Delta^+\mathbf{se} - \Delta^-\mathbf{se})$ (exact slice), as well as pick the maximum resulting score, and map the slice scores to maximum scores per feature as in Algorithm 1, Line 5. Finally, we also include the true upper bound score of the positive delta $\mathbf{ss} = \text{colSums}(\Delta^+\mathbf{X})^\top$ and $\mathbf{se} = (\Delta^+\mathbf{e}^\top \odot \Delta^+\mathbf{X})^\top$ of basic (single-feature) slices. Together, this high-probability upper bound represents previously enumerated slices exactly and close neighbors approximately. In order to give hard guarantees, we would need to also collect all previously pruned slices and evaluate if these decisions still hold. Due to many different pruning strategies and thus, complex integration, we made the design decision to provide this approximate pruning strategy (as an optional configuration, not enabled by default to ensure correctness) instead.

4 Incremental SliceLine Algorithm

Putting it altogether, we now describe the incremental SliceLine[5] algorithm. Algorithm 2 shows the skeleton of the original SliceLine algorithm with the extension points for incremental computation highlighted in blue. Note that the two-phase semantics of incremental SliceLine—namely, *initialization* for initial state collection and *exploitation* of this state for additional pruning—correspond to the first and subsequent invocations of this algorithm.

State Collection: During the first invocation of incremental SliceLine (intialization), there are no major differences to the original SliceLine, except for state collection in Line 24. In

[5] SystemDS incSliceLine: https://github.com/apache/systemds/blob/main/scripts/builtin/incSliceLine.dml

Algorithm 2 incSliceLine 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** $\leftarrow$ colMaxs$(\mathbf{X}_0)^\top$ *// $m \times 1$ matrix*
3: **fb** $\leftarrow$ cumsum(**fdom**) $-$ **fdom**,
4: **fe** $\leftarrow$ cumsum(**fdom**)
5: $\mathbf{X} \leftarrow$ onehot$(\mathbf{X}_0 + \mathbf{fb})$ *// $n \times l$ matrix*
6: *// b) initialization (statistics, basic slices, initial top-K)*
7: $\overline{e} \leftarrow$ sum(**e**)$/n$
8: $\text{tk}_{\min} \leftarrow$ COMPUTELOWESTPREVTK$(\mathbf{X}', \text{state}, \alpha)$
9: $[\mathcal{S}, \mathcal{R}, \mathcal{M}] \leftarrow$ EXTRACT$(state)$
10: $\mathbf{sc}_{\max} \leftarrow$ GETMAXSCOREALLFEATURES$(n, m, \mathcal{S}, \mathcal{R}, \mathcal{M})$
11: $\mathbf{sc}_{\overline{\max}} \leftarrow$ GETMAXCHANGEDSCOREALLFEATURES$(n, m, \mathbf{X}', \mathbf{e}', \mathcal{S}, \mathcal{R}, \mathcal{M})$
12: $[\mathbf{S}, \mathbf{R}, \mathbf{cI}] \leftarrow$ CREATEANDSCOREBASICSLICES$(\mathbf{X}', \mathbf{e}', \overline{e'}, \sigma, \alpha, \mathbf{sc}_{\max}, \text{tk}_{\min})$
13: $[\mathbf{TS}, \mathbf{TR}] \leftarrow$ MAINTAINTOPK$(\mathbf{S}, \mathbf{R}, \mathbf{TS}, \mathbf{TR}, K, \sigma)$
14: *// c) level-wise lattice enumeration*
15: $L \leftarrow 1, \quad \lceil L \rceil \leftarrow \min(m, \lceil L \rceil)$ *// current/max lattice levels*
16: $\mathbf{X} \leftarrow \mathbf{X}[, \mathbf{cI}]$ *// select features satisfying $\mathbf{ss}_0 \geq \sigma \wedge \mathbf{se}_0 > 0$*
17: **while** nrow(**S**) $> 0 \wedge L < \lceil L \rceil$ **do**
18: $L \leftarrow L + 1$
19: $\mathbf{S} \leftarrow$ GETPAIRCANDIDATES$(\mathbf{S}, \mathbf{R}, \mathbf{TS}, \mathbf{TR}, K, L, \overline{e}, \sigma, \alpha, \mathbf{fb}, \mathbf{fe})$
20: $\mathbf{S} \leftarrow$ PRUNEUNCHANGEDSLICES$(\mathbf{S}, \mathcal{S}, \mathcal{R}, \Delta^{\pm}\mathbf{X}, \sigma, L)$
21: $\mathbf{R} \leftarrow$ matrix$(0, \text{nrow}(\mathbf{S}), 4), \quad \mathbf{S2} \leftarrow \mathbf{S}[, \mathbf{cI}]$
22: **for** i **in** nrow(**S**) **do** *// parallel for*
23: $R_i \leftarrow$ EVALSLICES$(\mathbf{X}, \mathbf{e}, \overline{e}, \mathbf{S2}_i^\top, L, \alpha)$
24: state$' \leftarrow$ APPEND$(state, \mathbf{S}, \mathbf{R}, \text{meta})$ *// state collection incl. pruned slices*
25: $[\mathbf{TS}, \mathbf{TR}] =$ MAINTAINTOPK$(\mathbf{S}, \mathbf{R}, \mathbf{TS}, \mathbf{TR}, K, \sigma)$
26: **return** DECODETOPK$(\mathbf{TS}, \mathbf{fb}, \mathbf{fe})$, **TR**

detail, at every lattice level, we collect the slice candidates **S**, their meta data, and their statistics **R**, where the latter contains slice scores **sc**, errors **se**, maximum errors **sm**, and sizes **ss**. Not shown in the algorithm, the slices can be collected as shallow-copies of sparse matrices (with l non-zeros per slice at level l) or in ID-encoded form (single non-zero per slice). Encoding during state collection and decoding slices during exploitation of this state increases computational overhead but reduces memory consumption. The handling of this state is robust to different domain sizes of $\mathbf{X}'$. During subsequent invocations of incremental SliceLine, we utilize this state for additional pruning. In order to facilitate multiple subsequent invocations of incremental SliceLine, we collect both evaluated and pruned slices with their respective exact or upper-bound statistics.

Min-Support Pruning: Besides the existing min-support pruning of basic slices in Line 12, we added the extended min-support pruning of unchanged slices (**P1**) in Line 20. This pruning strategy brings—as observed during experimentation with many datasets and configurations—the number of evaluated slice candidates very close to the number of valid slice candidates that pass the min-support threshold.

Tab. 1: Dataset Characteristics (n rows, m features, l one-hot encoded features).

Dataset	n (nrow($\mathbf{X}_0$))	m (ncol($\mathbf{X}_0$))	l (ncol($\mathbf{X}$))	ML Task
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

Score Pruning: The extended score pruning is integrated in multiple parts of the algorithm. First, we evaluate the old top-K slices (**P2**) in Line 8, exploit them (via $\text{tk}_{\min}$) in CREATEANDSCOREBASICSLICES for more effective upper-bound pruning in Line 12, make them the initial top-K set in Line 13, and adapt the maintenance of the top-K set in Line 25 accordingly. Second, we compute the maximum reachable score of individual features (**P3**) in Line 10 for very effective, additional pruning in CREATEANDSCOREBASICSLICES, which early on, eliminates many possible feature combinations that we otherwise would need to enumerate. Finally, if optionally enabled, we compute high-probability upper-bound scores $\mathbf{sc}_{\overline{\max}}$ for all features via GETMAXCHANGEDSCOREALLFEATURES (**P4**) in Line 11, which then allows pruning basic slices with $\mathbf{sc}_{\overline{\max}} < \max(0, \text{tk}_{\min})$.

5 Experiments

Our experiments study the end-to-end performance improvements of incremental slice finding as well as details of the pruning effectiveness and robustness to different extends of changes to the feature matrix and model. Overall, we find that our simple extended pruning strategies together with efficient vectorized linear algebra operations yield substantial runtime improvements of more than an order of magnitude.

5.1 Experimental Setting

HW Environment: We ran all experiments on a scale-out cluster node equipped with an AMD EPYC 7443P CPU @ 2.8 – 4.0 GHz (24 physical/48 virtual cores), 256 GB DDR4 RAM @ 3.2 GHz (with 190 GB/s memory bandwidth), and 1×480 GB SATA SSD (system) as well as 8×2 TB SATA HDDs (data). The software stack comprises Ubuntu 20.04.6, Apache Hadoop 3.3, Apache Spark 3.5, OpenJDK 11 (with `-Xmx200g -Xms200g`), and Apache SystemDS from the main branch (as of Jan 03, 2025).

Datasets and Algorithms: For seamless comparison and reproducibility, we utilize a subset of the SliceLine datasets. Table 1 shows these datasets and their basic metadata in terms of number of rows, number of columns, number of columns after one-hot encoding, and the type of the ML task. For classification, we use multinomial logistic regression (`multiLogReg`)

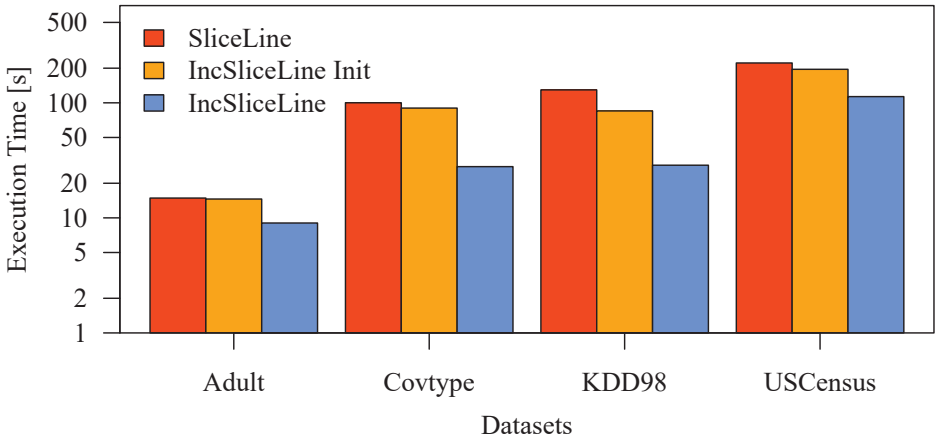


Fig. 3: Runtime Comparison of Incremental Slice Finding with Exact Pruning.

whereas for regression, we use linear regression (`lm`), which internally dispatches to `lmDS` or `lmCG` if ncol($\mathbf{X}$) > 1,024. As preprocessing steps, we recode categorical features into integer domain, bin numerical features with 10 bins each, and drop all ID columns.

5.2 End-to-end Slice Finding

Before studying the different components and pruning effectiveness of incremental SliceLine in detail, we first conduct end-to-end runtime performance experiments on the different datasets to validate that incremental computation indeed yields runtime improvements. We report the average runtime of 10 repetitions.

Runtime Comparison: In order to measure runtime performance, we compare full SliceLine from scratch, incremental SliceLine without previous state (initialization), and incremental SliceLine with previous state. The latter incremental SliceLine invocation uses the same dataset with a dozen additional tuples (last 12), which we held out on the first invocation. For all datasets, we used the parameters $k = 10$, $\alpha = 0.95$, $\sigma = \max(n/100, 8)$, and tpSize = 16 (blocking for task-parallel slice evaluation). The maximum lattice level was unrestricted for Adult, but set to 3 for Covtype and KDD98 and 4 for USCensus, respectively. Figure 3 shows the results as a mean of 10 repetitions (ignoring a first repetition for warmup and reading the data). We observe that the initialization of incremental SliceLine shows very similar performance to the original SliceLine due to lightweight state collection, and in some cases slightly faster due to 12 tuples smaller data and few more refined script-level implementation details. Most importantly, incremental SliceLine with state exploitation (on the full dataset) then shows good runtime improvements from 1.7× (on Adult) to ≈ 4× (on Covtype and KDD98). The most effective pruning strategies are the extended upper-bound score pruning with the initial top-K set (**P2**) as well as the pruning of unchanged basic

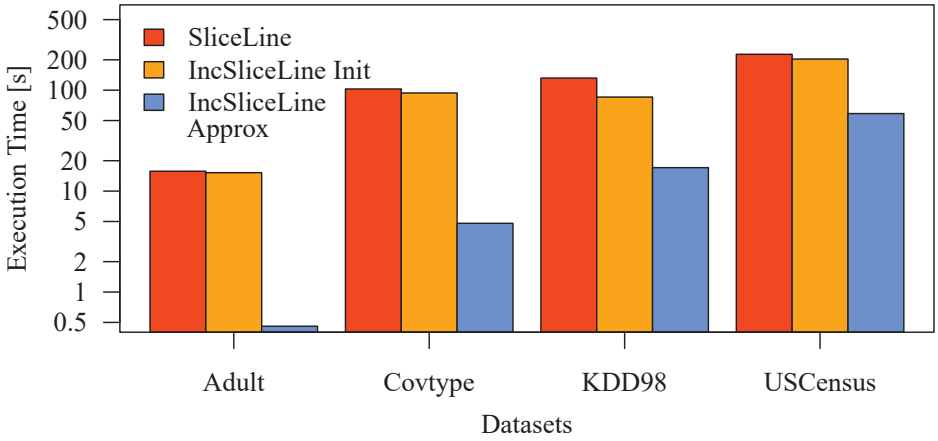


Fig. 4: Runtime Comparison of Incremental Slice Finding with Approximate Pruning.

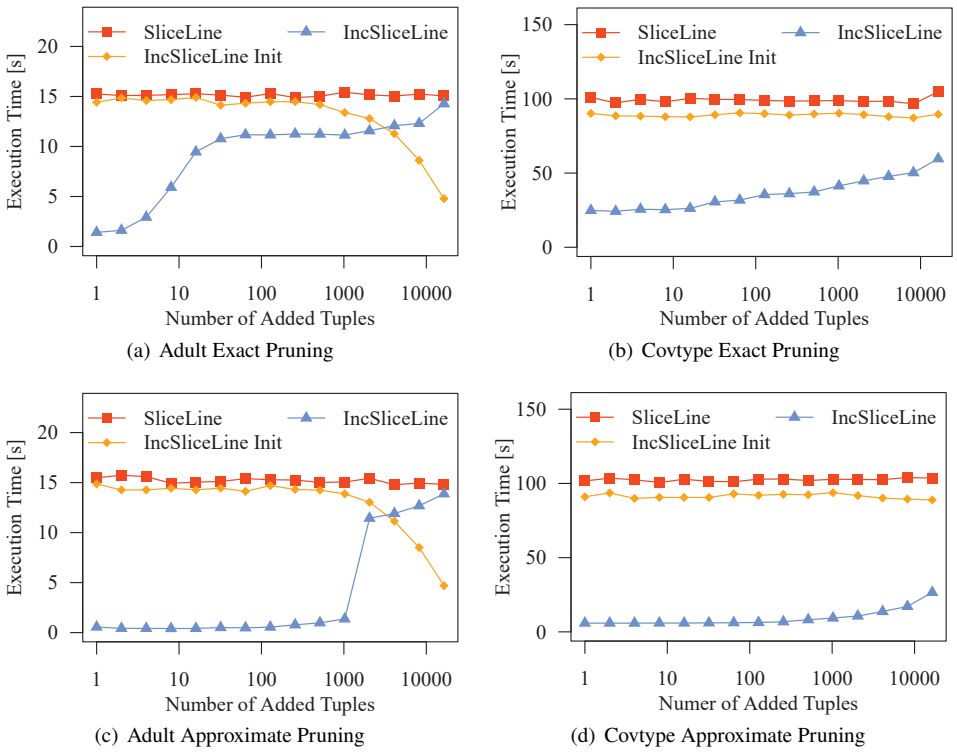
slices according to their maximum reachable scores at all levels (**P3**). Interestingly, on even smaller deltas (2 instead of 12 additional tuples), incremental SliceLine yields an order of magnitude improvement on the Adult dataset (from 15s to 1.6s).

Approximate Pruning: For comparison, we also study the performance with enabled approximate score pruning (**P4**), while all other parameters remained the same. Figure 4 shows the results, where we observe major improvements of incremental SliceLine from 3.9× (on USCensus) to 34× (on Adult). In these experiments and hundreds of unit tests, approximate score pruning always yields the same top-K set (due to its high-probability upper bounds), but this pruning strategy does not provide hard guarantees.

5.3 Micro-Benchmarks

We perform additional micro benchmarks to understand the properties of incremental SliceLine regarding the impact of added and removed tuples as well as different pruning strategies. These experiments are conducted on the smaller datasets Adult and Covtype, default parameters remain unchanged, and we report the average runtimes of 5 repetitions.

Impact of Added/Removed Ratio: We evaluate the impact of the number of added rows (or the ratio of `addedX`/`totalX`) on the total execution time of incremental SliceLine. We increase this ratio while keeping the total size of the input matrix constant. For few added tuples, we expect major runtime improvements by incremental computation. Figure 5 shows the results for both the Adult and Covtype datasets as well as exact and approximate pruning. On the x-axis, we increase the number of added tuples exponentially in $[1, 16\text{K}]$. First, for Adult with all exact pruning techniques (see Figure 5(a)), we observe constant execution time by SliceLine, substantial runtime improvements for very few added tuples by incremental



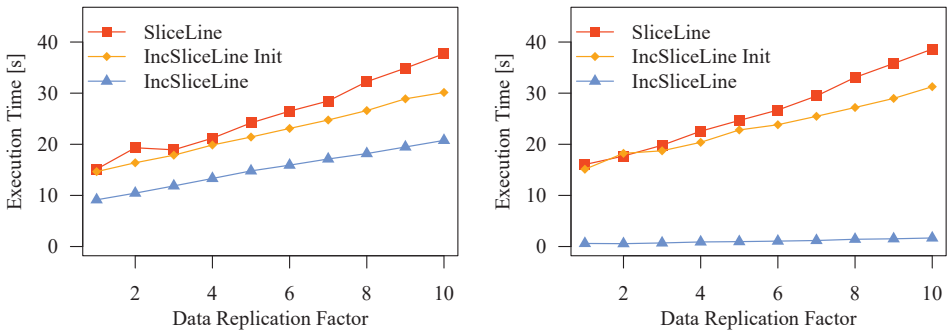
(a) Adult Exact Pruning (b) Covtype Exact Pruning

(c) Adult Approximate Pruning (d) Covtype Approximate Pruning

Fig. 5: Runtime Comparison with Varying Number of Added Tuples.

SliceLine, and robust improvements up to thousands of added tuples. The initialization phase of incremental SliceLine gets faster as we add more tuples because here the initial dataset gets smaller (constant total size). On half the dataset, we see a drop to a third of the execution time because of constant minimum support thresholds and thus, more effective pruning. Second, for Covtype with exact pruning (see Figure 5(b)), we observe similar characteristics except for the final phase because Covtype has more than an order of magnitude more rows. Third, for both datasets with approximate pruning (see Figures 5(c) and 5(d)) there are substantial additional improvements by incremental SliceLine. For example, on Adult, the very fast execution for few added tuples extends until 1,024 tuples. Finally, changing the number of removed tuples shows very similar characteristics compared to these results.

Varying Input Sizes: In additional to the ratio of added/removed tuples, the benefits of incremental computation also depend on the total size of the input feature matrix. Accordingly, Figure 6 shows the execution time of SliceLine and incremental SliceLine for increasing data sizes. We replicated the Adult dataset, and keep the ratio of added tuples and relative min-support constant. Here, we observe that the relative improvements



(a) Adult Exact Pruning (b) Adult Approximate Pruning

Fig. 6: Runtime Comparison with Increasing Data Size.

of incremental SliceLine slowly increase as we scale from replication factor 1 to 10. We attribute this scaling to the fact that the pruning techniques only perform computation in the size of slices and feature lattice, but can eliminate unnecessary computation (pruned slice evaluation) on the full dataset. The slight improvements of the initialization run of incremental SliceLine is again due to a few more refined script-level implementation details and efficient state collection in lists which require only shallow-copies.

Enumerated and Valid Slices: We further compare the pruning capabilities of SliceLine and incremental SliceLine in terms of enumerated and valid slices across all lattice levels. The number of enumerated slices is measured right after the evaluation of slices. Afterward, a pruning step filters out slices not adhering to the error and size pruning conditions. This comparison of the evaluated and valid slices allows understanding the pruning potential and pruning effectiveness. Figure 7 shows for different numbers of added tuples (from top to bottom), for each lattice level (from left to right) how many slices have been enumerated (and are thus, evaluated on the dataset) and how many of these enumerated slices are in fact valid slices that cannot be eliminated. Together the score (**P2**) and max-score (**P3**) pruning techniques of incremental SliceLine eliminate slices early on, which also eliminates reachable slices in subsequent lattice levels, and thus, substantially reduces the total number of enumerated slices. Furthermore, we see a strong correlation between the number of enumerated slices and yielded runtime improvements (compare Figure 5 from before), indicating that additional computation for pruning does not cause major overheads.

Impact of Pruning on Different Datasets: Finally, we return to our end-to-end benchmarks with all four datasets and ML tasks and conduct an ablation study of pruning techniques. Figure 8 shows the runtime performance in log-scale for SliceLine, incremental SliceLine initialization, as well as six pruning configurations. From left to right, we compare (1) no pruning, (2) only top-K score pruning (**P2**), (3) only min-support/size pruning (**P1**), (4) only max-score pruning (**P3**), (5) all exact pruning techniques (**P1**-**P3**), and (6) all pruning including approximate (**P1**-**P4**). We see a similar pattern as in the micro benchmarks:

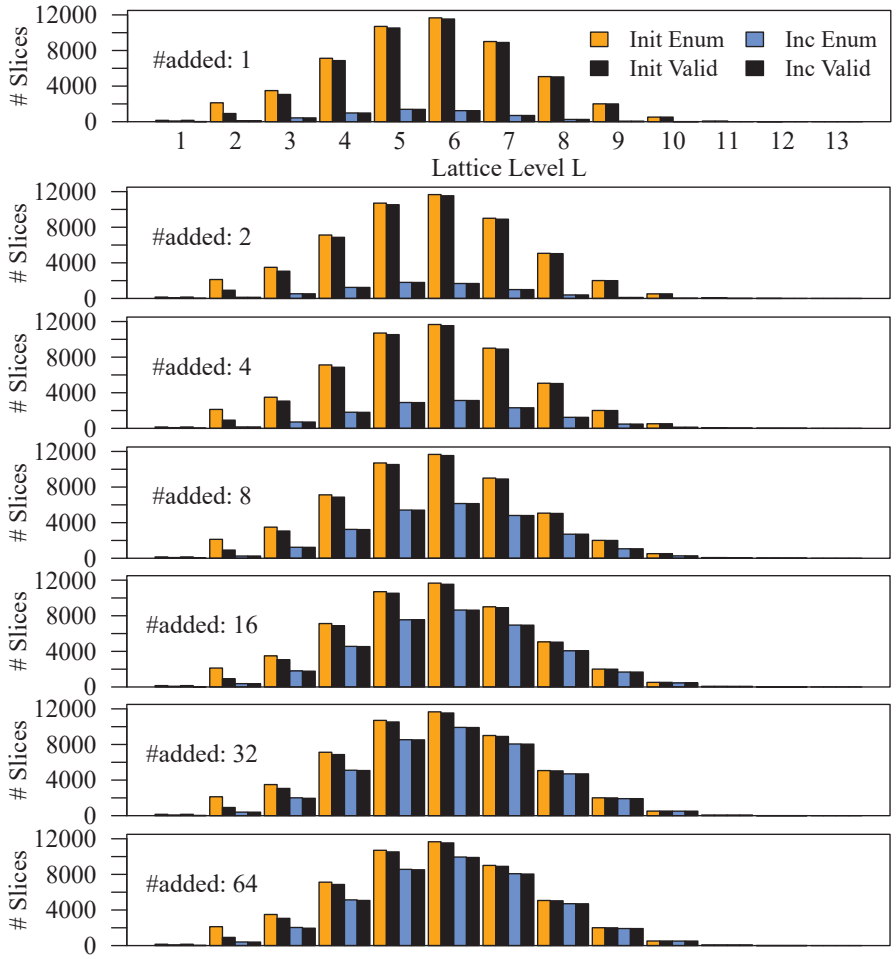


Fig. 7: Numbers of Enumerated and Valid Slices for Adult with Exact Pruning (Init refers to IncSliceLine initialization which enumerates as many slices as SliceLine, while Inc refers to Incremental SliceLine).

- **P1:** The extended min-support pruning usually only yields minor improvements because this technique only skips the evaluation of individual enumerated slices.
- **P2:** The extended score pruning is usually very effective (all datasets).
- **P3:** The pruning by maximum unchanged slice scores provides substantial additional improvements on multiple datasets (Adult, Covtype, and KDD).
- **P4:** The approximate pruning gives additional major improvements on all datasets, but does not provide correctness guarantees.

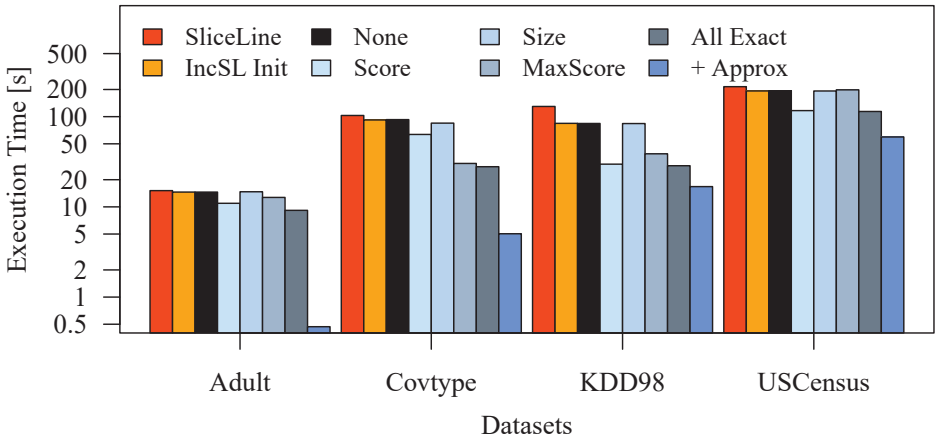


Fig. 8: Runtime Comparison of Incremental Slice Finding with Different Pruning Strategies.

Accordingly, Incremental SliceLine uses by default all exact pruning strategies, and allows users to optionally also enable the approximate pruning. If the number of changes are small, we recommend to use approximate pruning without any hesitation.

6 Related Work

Beyond the already covered work on slice finding and data coverage, incremental SliceLine is related to incremental query optimization, incremental view maintenance, incremental linear algebra operations, incremental frequent itemset mining, and clustered model specialization.

Incremental Query Optimization: For multi-objective optimization problems (e.g., query execution time, time to first tuple, and memory consumption), there is also work on incremental query optimization [TK15]. In this work, intermediate solutions of incrementally refined optimization problems are reused across invocations. In contrast to changed optimization problems, incremental slice finding deals with changed datasets.

Incremental View Maintenance: In the context of data warehousing [Le03], the incremental maintenance of materialized views is a well-studied problem [BLT86, GL95, Ka15, Sv23]. Traditionally, we distinguish eager, lazy, and deferred maintenance [ZLE07], where eager and lazy require incremental maintenance. The delta tuples are first propagated to a view delta and subsequently applied (joined and updated) to the view [Le03]. Recent work utilizes viewlet transforms and parallelization [Ah12], ID delta sets [Ka15] and schema information [Sv23] for faster incremental maintenance. In contrast to propagating base table changes to materialized views, we deal with a more complex top-K search in the lattice of slices.

Incremental Linear Algebra: Ideas of incremental maintenance have also been applied to linear algebra programs. LinVIEW [NEK14] preserves key intermediates—of algorithms

like ordinary least squares and matrix powers—in order to facilitate incremental updates for delta matrices. LIMA [PRB21] supports lineage-based, full and partial reuse of intermediates in linear algebra programs, which enables incremental processing in iterative algorithms like step-wise feature selection. Finally, some of these incremental maintenance ideas (positive deltas) have been applied for machine unlearning (negative deltas) [Sc20]. Although partial reuse would also apply to incremental slice finding, such reuse is general-purpose and thus, unable to exploit the full potential (e.g., extended score pruning).

Incremental Slicing Alternatives: The existing work on finding slices of tabular data where a model substantially underperforms [Ch19, Ch20, TW21, SB21] do not allow for incremental computation. However, alternatives to full-fledged slice finding on the lattice of attribute conjunctions are decision trees [Ch19, Ch20] and clustering according to prediction errors. These alternatives do not allow for overlapping slices—where an instance can appear in multiple slices—but existing incremental strategies apply. First, in the context of machine unlearning from above [Sc20], the follow-up work HedgeCut [SGD21] extended extremely randomized trees (ERTs) to incrementally remove tuples by quantifying the robustness of split decisions. Such ideas can be extended for incremental refinements (added/removed tuples) of widely-used decision trees and random forests. Second, work on discovering and explaining slices of multi-model data (e.g., images, audio) aim to find coherent clusters where the model makes systematic prediction errors [Ey22, d'22].

Incremental Frequent Itemset Mining: Frequent itemset mining, and the related association rule mining, have been well-studied [AIS93, Sc14]. From a database of transactions (each containing items), we aim to find the itemsets that frequently appear together and satisfy a minimum support threshold. The major algorithm classes are Apriori [AS94] (horizontal data layout), Eclat [Za97] (vertical data layout), and FP-Growth [HPY00] (tree-based layout). These algorithms exploit the apriori property that itemsets can only be frequent if all their subsets are frequent. Recent work on incremental frequent itemset mining and association rule mining [TK21] leverages FP-Growth and existing index structures in order to quickly index and mine new transactions. In contrast, we focus on data in matrix representations and linear-algebra-based slice finding for easy parallelization and deployment.

Clustered Model Specialization: A loosely connected area is clustered model specialization. Instead of repeatedly running data acquisition, model training, and model debugging, one could split the data into groups and learn specialized models per group. We discuss two examples. First, clustered classification [An20, Tr23, Hi23] uses clustering to group training instances, and then trains specialized models for individual groups. This method can yield very good results, but requires hyper-parameter tuning and building and deploying an ensemble of models. Second, clustered federated learning [SMS21] trains a model on all client data to convergence (gradient close to zero), and as a post-processing step, clusters gradients of clients in order to find groups with similar data distribution, and then fine-tunes separate models for these subgroups. In contrast to these ensemble strategies, we aim provide incremental model debugging for building a single model.

7 Conclusions

To summarize, we introduce incremental SliceLine an efficient incremental algorithm for finding the top-K worst slices (where an ML model underperforms) under iterative updates of the dataset and model. Key strategies include systematic state collection, additional min-support and score pruning strategies, and an extended linear-algebra-based slice finding algorithm. We draw three fundamental conclusions. First, the additional pruning yields substantial reductions in the number of enumerated slices and thus, execution time. Second, together warm-starting for ML model retraining and incremental slice finding enable many new use cases of iterative model debugging during data discovery and sampling. Third, the cleanly extended linear-algebra-based SliceLine algorithm preserves its vectorized formulation which makes it easy to parallelize and deploy in local, distributed, and federated environments. Interesting future work includes tuning SliceLine and incremental SliceLine for federated environments [Ba21] and modern HW accelerators; as well as deploying and specializing them in data-centric ML pipelines with data discovery, data cleaning, and learned sampling, augmentation, and quantization.

Acknowledgments

We thank Jiwon Chang and Fatemeh Nargesian for early discussions of incremental data discovery and slice finding (during our work on PLUTUS [Ch24]), which inspired our incremental SliceLine. Furthermore, we also gratefully acknowledge funding from the German Federal Ministry of Education and Research (under research grant BIFOLD24B).

Bibliography

[Ah12] Ahmad, Yanif; Kennedy, Oliver; Koch, Christoph; Nikolic, Milos: DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. PVLDB, 5(10):968–979, 2012.

[AIS93] Agrawal, Rakesh; Imielinski, Tomasz; Swami, Arun N.: Mining Association Rules between Sets of Items in Large Databases. In: SIGMOD. pp. 207–216, 1993.

[AJJ19] Asudeh, Abolfazl; Jin, Zhongjun; Jagadish, H. V.: Assessing and Remedying Coverage for a Given Dataset. In: ICDE. pp. 554–565, 2019.

[An20] Andreeva, Olga; Li, Wei; Ding, Wei; Kuijjer, Marieke L.; Quackenbush, John; Chen, Ping: Catalysis Clustering with GAN by Incorporating Domain Knowledge. In: KDD. pp. 1344–1352, 2020.

[An23] Anil, Rohan et al.: Gemini: A Family of Highly Capable Multimodal Models. CoRR, abs/2312.11805, 2023.

[AS94] Agrawal, Rakesh; Srikant, Ramakrishnan: Fast Algorithms for Mining Association Rules in Large Databases. In: VLDB. pp. 487–499, 1994.

[Ba21] Baunsgaard, Sebastian; Boehm, Matthias; Chaudhary, Ankit; Derakhshan, Behrouz; Geißelsöder, Stefan; Grulich, Philipp M.; Hildebrand, Michael; Innerebner, Kevin; Markl, Volker; Neubauer, Claus; Osterburg, Sarah; Ovcharenko, Olga; Redyuk, Sergey; Rieger, Tobias; Mahdiraji, Alireza Rezaei; Wrede, Sebastian Benjamin; Zeuch, Steffen: ExDRa: Exploratory Data Science on Federated Raw Data. In: SIGMOD. pp. 2450–2463, 2021.

[BER19] Boehm, Matthias; Evfimievski, Alexandre V.; Reinwald, Berthold: Efficient Data-Parallel Cumulative Aggregates for Large-Scale Machine Learning. In: BTW. pp. 267–286, 2019.

[BLT86] Blakeley, José A.; Larson, Per-Åke; Tompa, Frank Wm.: Efficiently Updating Materialized Views. In: SIGMOD. pp. 61–71, 1986.

[Bo20] Boehm, Matthias; Antonov, Iulian; Baunsgaard, Sebastian; Dokter, Mark; Ginthör, Robert; Innerebner, Kevin; Klezin, Florijan; Lindstaedt, Stefanie N.; Phani, Arnab; Rath, Benjamin; Reinwald, Berthold; Siddiqui, Shafaq; Wrede, Sebastian Benjamin: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In: CIDR. 2020.

[Br20] Brown, Tom B. et al.: Language Models are Few-Shot Learners. In: NeurIPS. 2020.

[CCT09] Cheney, James; Chiticariu, Laura; Tan, Wang Chiew: Provenance in Databases: Why, How, and Where. Found. Trends Databases, 1(4):379–474, 2009.

[Ch19] Chung, Yeounoh; Kraska, Tim; Polyzotis, Neoklis; Tae, Ki Hyun; Whang, Steven Euijong: Slice Finder: Automated Data Slicing for Model Validation. In: ICDE. pp. 1550–1553, 2019.

[Ch20] Chung, Yeounoh; Kraska, Tim; Polyzotis, Neoklis; Tae, Ki Hyun; Whang, Steven Euijong: Automated Data Slicing for Model Validation: A Big Data - AI Integration Approach. IEEE Trans. Knowl. Data Eng., 32(12):2284–2296, 2020.

[Ch22] Chai, Chengliang; Liu, Jiabin; Tang, Nan; Li, Guoliang; Luo, Yuyu: Selective Data Acquisition in the Wild for Model Charging. Proc. VLDB Endow., 15(7):1466–1478, 2022.

[Ch24] Chang, Jiwon; Dionysio, Christina; Nargesian, Fatemeh; Boehm, Matthias: PLUTUS: Understanding Data Distribution Tailoring for Machine Learning. In: SIGMOD. pp. 528–531, 2024.

[Cu19] Cubuk, Ekin D.; Zoph, Barret; Mané, Dandelion; Vasudevan, Vijay; Le, Quoc V.: AutoAugment: Learning Augmentation Strategies From Data. In: CVPR. pp. 113–123, 2019.

[d'22] d'Eon, Greg; d'Eon, Jason; Wright, James R.; Leyton-Brown, Kevin: The Spotlight: A General Method for Discovering Systematic Errors in Deep Learning Models. In: FAccT. pp. 1962–1981, 2022.

[Ey22] Eyuboglu, Sabri; Varma, Maya; Saab, Khaled Kamal; Delbrouck, Jean-Benoit; Lee-Messer, Christopher; Dunnmon, Jared; Zou, James; Ré, Christopher: Domino: Discovering Systematic Errors with Cross-Modal Embeddings. In: ICLR. 2022.

[FBL11] Fischer, Ulrike; Boehm, Matthias; Lehner, Wolfgang: Offline Design Tuning for Hierarchies of Forecast Models. In: BTW. pp. 167–186, 2011.

[Ge14] Gebaly, Kareem El; Agrawal, Parag; Golab, Lukasz; Korn, Flip; Srivastava, Divesh: Interpretable and Informative Explanations of Outcomes. PVLDB, 8(1):61–72, 2014.

[GL95] Griffin, Timothy; Libkin, Leonid: Incremental Maintenance of Views with Duplicates. In: SIGMOD. pp. 328–339, 1995.

[Hi23] Hirsch, Vitali; Reimann, Peter; Treder-Tschechlov, Dennis; Schwarz, Holger; Mitschang, Bernhard: Exploiting domain knowledge to address class imbalance and a heterogeneous feature space in multi-class classification. VLDB J., 32(5):1037–1064, 2023.

[HPY00] Han, Jiawei; Pei, Jian; Yin, Yiwen: Mining Frequent Patterns without Candidate Generation. In: SIGMOD. pp. 1–12, 2000.

[Ji20] Jin, Zhongjun; Xu, Mengjing; Sun, Chenkai; Asudeh, Abolfazl; Jagadish, H. V.: MithraCoverage: A System for Investigating Population Bias for Intersectional Fairness. In: SIGMOD. pp. 2721–2724, 2020.

[Ka15] Katsis, Yannis; Ong, Kian Win; Papakonstantinou, Yannis; Zhao, Kevin Keliang: Utilizing IDs to Accelerate Incremental View Maintenance. In: SIGMOD. pp. 1985–2000, 2015.

[Ki23] Killamsetty, KrishnaTeja; Evfimievski, Alexandre V.; Pedapati, Tejaswini; Kate, Kiran; Popa, Lucian; Iyer, Rishabh K.: MILO: Model-Agnostic Subset Selection Framework for Efficient Model Training and Tuning. CoRR, abs/2301.13287, 2023.

[Le03] Lehner, Wolfgang: Datenbanktechnologie für Data-Warehouse-Systeme. Konzepte und Methoden. Dpunkt Verlag, 2003.

[Li20] Lin, Yin; Guan, Yifan; Asudeh, Abolfazl; Jagadish, H. V.: Identifying Insufficient Data Coverage in Databases with Multiple Relations. PVLDB, 13(11):2229–2242, 2020.

[LYK21] Li, Yifan; Yu, Xiaohui; Koudas, Nick: Data Acquisition for Improving Machine Learning Models. Proc. VLDB Endow., 14(10):1832–1844, 2021.

[NAJ21] Nargesian, Fatemeh; Asudeh, Abolfazl; Jagadish, H. V.: Tailoring Data Source Distributions for Fairness-aware Data Integration. Proc. VLDB Endow., 14(11):2519–2532, 2021.

[Ne21] Neutatz, Felix; Chen, Binger; Abedjan, Ziawasch; Wu, Eugene: From Cleaning before ML to Cleaning for ML. IEEE Data Eng. Bull., 44(1):24–41, 2021.

[NEK14] Nikolic, Milos; Elseidy, Mohammed; Koch, Christoph: LINVIEW: incremental view maintenance for complex analytical queries. In: SIGMOD. pp. 253–264, 2014.

[NKP20] Nakandala, Supun; Kumar, Arun; Papakonstantinou, Yannis: Query Optimization for Faster Deep CNN Explanations. SIGMOD Rec., 49(1):61–68, 2020.

[Po17] Polyzotis, Neoklis; Roy, Sudip; Whang, Steven Euijong; Zinkevich, Martin: Data Management Challenges in Production Machine Learning. In: SIGMOD. pp. 1723–1726, 2017.

[PRB21] Phani, Arnab; Rath, Benjamin; Boehm, Matthias: LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In: SIGMOD. pp. 1426–1439, 2021.

[SB21] Sagadeeva, Svetlana; Boehm, Matthias: SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In: SIGMOD. pp. 2290–2299, 2021.

[Sc14] Schlegel, Benjamin: Frequent Itemset Mining on Multiprocessor Systems. PhD thesis, Dresden University of Technology, 2014.

[Sc18] Schelter, Sebastian; Lange, Dustin; Schmidt, Philipp; Celikel, Meltem; Bießmann, Felix; Grafberger, Andreas: Automating Large-Scale Data Quality Verification. PVLDB, 11(12):1781–1794, 2018.

[Sc20] Schelter, Sebastian: Ämnesia Machine Learning Models That Can Forget User Data Very Fast. In: CIDR. 2020.

[SGD21] Schelter, Sebastian; Grafberger, Stefan; Dunning, Ted: HedgeCut: Maintaining Randomised Trees for Low-Latency Machine Unlearning. In: SIGMOD. pp. 1545–1557, 2021.

[SKB23] Siddiqi, Shafaq; Kern, Roman; Boehm, Matthias: SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. Proc. ACM Manag. Data, 1(3):218:1–218:26, 2023.

[SMS21] Sattler, Felix; Müller, Klaus-Robert; Samek, Wojciech: Clustered Federated Learning: Model-Agnostic Distributed Multitask Optimization Under Privacy Constraints. IEEE Trans. Neural Networks Learn. Syst., 32(8):3710–3722, 2021.

[SSW19] Scherzinger, Stefanie; Seifert, Christin; Wiese, Lena: The Best of Both Worlds: Challenges in Linking Provenance and Explainability in Distributed Machine Learning. In: ICDCS. pp. 1620–1629, 2019.

[Sv23] Svingos, Christoforos; Hernich, André; Gildhoff, Hinnerk; Papakonstantinou, Yannis; Ioannidis, Yannis E.: Foreign Keys Open the Door for Faster Incremental View Maintenance. Proc. ACM Manag. Data, 1(1):40:1–40:25, 2023.

[TK15] Trummer, Immanuel; Koch, Christoph: An Incremental Anytime Algorithm for Multi-Objective Query Optimization. In: SIGMOD. pp. 1941–1953, 2015.

[TK21] Thurachon, Wannasiri; Kreesuradej, Worapoj: Incremental Association Rule Mining With a Fast Incremental Updating Frequent Pattern Growth Algorithm. IEEE Access, 9:55726–55741, 2021.

[Tr23] Treder-Tschechlov, Dennis; Reimann, Peter; Schwarz, Holger; Mitschang, Bernhard: Approach to Synthetic Data Generation for Imbalanced Multi-class Problems with Heterogeneous Groups. In: BTW. pp. 329–351, 2023.

[TW21] Tae, Ki Hyun; Whang, Steven Euijong: Slice Tuner: A Selective Data Acquisition Framework for Accurate and Fair Machine Learning Models. In: SIGMOD. pp. 1771–1783, 2021.

[Wa18] Wang, Tongzhou; Zhu, Jun-Yan; Torralba, Antonio; Efros, Alexei A.: Dataset Distillation. CoRR, abs/1811.10959, 2018.

[Za97] Zaki, Mohammed Javeed; Parthasarathy, Srinivasan; Ogihara, Mitsunori; Li, Wei: New Algorithms for Fast Discovery of Association Rules. In: SIGKDD. pp. 283–286, 1997.

[Zh21] Zhang, Hantian; Chu, Xu; Asudeh, Abolfazl; Navathe, Shamkant B.: OmniFair: A Declarative System for Model-Agnostic Group Fairness in Machine Learning. In: SIGMOD. pp. 2076–2088, 2021.

[ZLE07] Zhou, Jingren; Larson, Per-Åke; Elmongui, Hicham G.: Lazy Maintenance of Materialized Views. In: VLDB. pp. 231–242, 2007.