AutoSlicer: Scalable Automated Data Slicing for ML Model Analysis

Zifan Liu[*]
University of Wisconsin-Madison
zliu676@wisc.edu

Evan Rosen
Google Inc.
embr@google.com

Paul Suganthan G. C.
Google Inc.
paulgc@google.com

Abstract

Automated slicing aims to identify subsets of evaluation data where a trained model performs anomalously. This is an important problem for machine learning pipelines in production since it plays a key role in model debugging and comparison, as well as the diagnosis of fairness issues. Scalability has become a critical requirement for any automated slicing system due to the large search space of possible slices and the growing scale of data. We present AutoSlicer, a scalable system that searches for problematic slices through distributed metric computation and hypothesis testing. We develop an efficient strategy that reduces the search space through pruning and prioritization. In the experiments, we show that our search strategy finds most of the anomalous slices by inspecting a small portion of the search space.

1 Introduction

In production settings, machine learning (ML) practitioners have to consider whether the input data have errors, whether a new version is good enough to replace the model currently used by the downstream stack, how to debug and improve model performance etc. Those require reliable model evaluation that can aid with validating, debugging and improving model performance.

Example 1: *Consider a continuous ML pipeline that trains on new data arriving in batches every day, to obtain a fresh model that needs to be pushed to the serving infrastructure. The ML engineer needs to validate that the new model does not perform worse than the previous one.*

Model metrics computed on the whole evaluation dataset can mask interesting or significant deviations of the same metrics computed on data slices that correspond to meaningful sub-populations. Thus, it is often desired to identify slices of data where the model performs poorly.

Example 2: *Consider the same pipeline in Example 1. To ensure fairness and model quality, we want to avoid pushing the new model to serving if it performs poorly on sensitive data slices.*

Example 3: *Consider an ML engineer who is building the first model for a task. The engineer tries to iteratively debug and improve the model. Knowing on which data slices the current model is performing poorly would aid the engineer to take actions towards improving the accuracy on them.*

To this end, we focus on the problem of automated slicing, aiming at identifying problematic data slices automatically. As part of an ML platform in production, the automated slicing system should satisfy the following requirements: (*Scalability*) The system should be able to operate on distributed clusters and the search method should be efficient, since modern ML datasets are often too large to fit in memory, and enumerating the number all possible slices is infeasible. (*Reliability*) The system should provide uncertainty measures that tell the user how reliable the results are, since some slices may appear problematic by chance due to sampling errors. (*Generality*) The system should be general and flexible to support common types of models and metrics.

[*] Work done while interning at Google.

NeurIPS 2022 Workshop on Challenges in Deploying and Monitoring Machine Learning Systems.

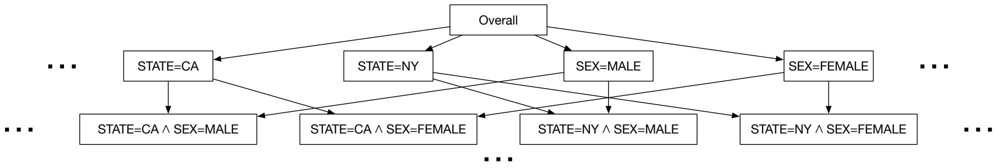

Figure 1: An example of the search space.

SliceFinder [11] identifies problematic slices by lattice search and hypothesis testing. However, SliceFinder focuses on the single-machine setting, failing to scale on large datasets. SliceLine [21] addresses the scalability issues by formulating automated slicing as linear-algebra problems, and solving them through distributed matrix computing. However, SliceLine does not provide uncertainty measures for the computed metric. In addition, SliceFinder and SliceLine only support metrics that are mean values over individual examples (e.g., average loss, accuracy), while we want a general framework that supports all common metrics in ML evaluation (e.g., AUC, F1 score).

In this paper we describe AutoSlicer, a scalable and reliable solution to the automatic slicing problem. Our technical contributions include: 1. A distributed computing framework that performs sliced evaluation efficiently for a general family of ML metrics (e.g., those provided by TensorFlow). 2. Uncertainty estimation for the results of metric comparison by distributed hypothesis testing. 3. A search strategy that identifies most of the problematic slices without exhausting the search space.

2 Problem Definition

We assume a dataset $D = \{(x_F^{(1)}, y^{(1)}), (x_F^{(2)}, y^{(2)}), \ldots, (x_F^{(N)}, y^{(N)})\}$ with N examples, where $x_F^{(n)}$ is the n^{th} example and $y^{(n)}$ is the corresponding ground truth label. $F = \{F_1, F_2, \ldots, F_M\}$ is the set of features in each example. We also assume a model h to be tested, and a metric $\psi(S, h)$ measuring how well the model makes predictions about the ground truth labels on a subset $S \subseteq D$.

A slice S_P is a subset of D that satisfies a conjunctive predicate $P = \bigwedge_i F_{m_i} \in V_i$, where V_i is a set of values from the domain of feature F_{m_i}. For categorical features, each V_i contains one value from the feature domain. If the domain of a feature is too large (it contains more than J values), we put each of the J most frequent values into one singleton set, and all the rest into one set. For numerical values, each V_i corresponds to a bin. We call each $F_{m_i} \in V_i$ a singleton predicate, and use $|P|$ to denote the number of singleton predicates in P (cross size of P). We use O to denote the predicate that is always true, and thereby S_O is the overall slice that contains all examples.

Automated slicing aims to find slices with significantly higher or lower metric values, compared to the overall slice or the same slice under a baseline model. Formally, we seek S_P such that $\Delta\psi_{S_P}$ is significantly greater or less than 0, where $\Delta\psi_{S_P} = \psi(S_P, h) - \psi(S_O, h)$ in the former scenario, and $\Delta\psi_{S_P} = \psi(S_P, h) - \psi(S_P, h')$ (h' is a baseline model) in the latter. We quantify significance by p-values from hypothesis testing. Any slice with p-value less than α is considered significant where α is a user-specified threshold.

The search space forms a lattice structure, where the l^{th} layer contains all possible slices with cross size l. An example of the search space is shown in Figure 1. To limit the number of candidate slices, we introduce the maximum cross size L as a user-specified parameter. Another reason for setting the maximum cross size is that as the cross size increases, the results become less interpretable.

In some cases, the user may not want slices that are too small since they do not have a large impact on the overall performance of the model. Therefore, we also introduce the minimum slice size N_{min} as another user-specified parameter.

In summary, we seek S_P's such that the difference of a given metric $\Delta\psi_{S_P}$ is significantly greater (or less) than 0, with $|P| \leq L$ and $|S_P| \geq N_{\text{min}}$, where $|S_P|$ is the number of examples in the slice.

3 Distributed Computation Framework

AutoSlicer uses the Beam programming model [1] to express distributed programs that can be executed in a variety of environments.

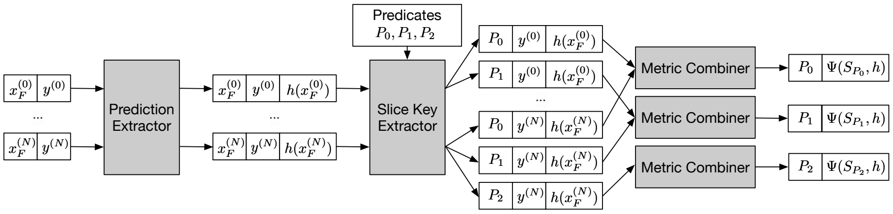
$x_F^{(0)}$ $y^{(0)}$
...
$x_F^{(N)}$ $y^{(N)}$
Prediction Extractor
$x_F^{(0)}$ $y^{(0)}$ $h(x_F^{(0)})$
...
$x_F^{(N)}$ $y^{(N)}$ $h(x_F^{(N)})$
Predicates P_0, P_1, P_2
Slice Key Extractor
P_0 $y^{(0)}$ $h(x_F^{(0)})$
P_1 $y^{(0)}$ $h(x_F^{(0)})$
...
P_0 $y^{(N)}$ $h(x_F^{(N)})$
P_1 $y^{(N)}$ $h(x_F^{(N)})$
P_2 $y^{(N)}$ $h(x_F^{(N)})$
Metric Combiner
Metric Combiner
Metric Combiner
P_0 $\Psi(S_{P_0}, h)$
P_1 $\Psi(S_{P_1}, h)$
P_2 $\Psi(S_{P_2}, h)$

Figure 2: A diagram showing the distributed computation of sliced evaluations.

Sliced Evaluation AutoSlicer is able to evaluate a wide variety of model performance metrics on slices of large datasets. In Figure 2, we show a diagram of the metric computing process. First, the prediction extractor computes model predictions for each input example. Next, the slice key extractor checks a specified set of slicing predicates against the per-example features. The predicates are generated by the candidate slice generator described in Section 4. For each matched example, a triplet containing the predicate, prediction and label is created. Note that the prediction and the label for an example that matches more than one predicates will appear more than once in the output. Conversely, predicates that match no example will not be present in the output and therefore will not contribute to the computational costs of downstream processing. Then the metric combiner aggregates the predictions and labels per predicate, which yields a collection of per-slice performance metrics. For example, to compute the precision of a binary classification model on a slice S_P, the combiner will count the number of true positives and false positives based on the predictions and labels whose keys match P. Finally, AutoSlicer computes metric differences $\Delta\psi$ to yield another collection of per-slice differences. The differences are either between the metric on each slice and the entire dataset (S_O), or between the metrics on the same slice evaluated on the two models that need to be compared.

Statistical Significance Testing In order to decide whether each metric difference $\Delta\psi$ is significant, AutoSlicer performs distributed hypothesis testing. AutoSlicer uses a parametric Poisson bootstrap approximation [9] to estimate the sampling distribution of $\Delta\psi$. Each of the B bootstrap replicates starts from a Poisson re-sampling of the entire input dataset, which is then fed to the sliced evaluation algorithm described above. This results in per-slice collections of B metric differences, $\{\Delta\psi_b\}_{b=0}^{B}$, over which we compute the t-statistics for statistical significance testing where the null hypothesis is $\Delta\psi = 0$. Per-slice p-values are then computed for significance-based output filtering.

4 Search Strategies

AutoSlicer relies on a candidate slice generator that automatically proposes candidate slices from the search space. The candidate slice generator supports several search strategies. We describe them as follows. The complete algorithms for the last two are provided in Appendix A.1.

Batch Strategy The candidate slice generator enumerates all the candidates at once, and then sends them to the slice key extractor for metric computation.

Iterative Strategy The candidate slice generator enumerates slices with cross size l in the l^{th} iteration. Metric computation is performed for the candidate slices in the current iteration before the beginning of the next. With this strategy, the search space is pruned based on the following criterion: 1. If a slice is significant, we do not consider any of its subsets as candidate slices in the subsequent iterations. This pruning rule follows SliceFinder [11]. 2. If the size of a slice is less than the minimum slice size, we do not consider any of its subsets since the size can only be even smaller.

Priority Strategy Since the search space might be too large to exhaust, we want a strategy that finds most of the significant slices by inspecting only part of it. In the priority strategy, we assign a priority score to each nonsignificant slice to prioritize those that are more likely to contain significant sub-slices for further exploration. We choose p-value as the priority score assuming that slices with lower p-values are more likely to contain significant sub-slices due to their more extreme metric values. This choice is validated in Appendix A.2.3. In addition, we limit the number of candidate slices per iteration so that the computation cost will not be excessively high for any iteration.

In the first iteration, all the candidates with cross size one are proposed. Afterwards, the nonsignificant slices are pushed into the priority queue. In each subsequent iteration, we pop base slices from the priority queue, and subdivide them by adding one more singleton predicate until the estimated number of nonempty candidates slices reaches the limit K. Then we compute the metrics and perform hypothesis testing for the candidates and push the new nonsignificant slices into the priority queue. In this strategy, we follow the same criterion as in the iterative strategy to prune the search space.

Note that we use an estimated number of nonempty candidate slices to decide if enough base slices have been popped from the queue. This is because some candidates are empty, introducing no computation cost. With this estimation, we have a better control over the computation cost per iteration. We keep track of the nonempty rate for each cross size l, i.e., the ratio between the actual number of nonempty slices and the number of slice candidates with cross size l that we have seen so far. When we count the number of nonempty slices, each candidate is discounted based on the nonempty rate corresponding to its cross size. For example, a candidate with cross size 3 will be counted as 0.5 slices if the nonempty rate of cross size 3 is 0.5. If we have not seen any slice with cross size l, the empty rate of cross size l will be predicted as that of cross size $l - 1$.

5 Experimental Evaluation

We compare the search strategies on a variety of real-world datasets. In the Appendix, we provide micro-benchmarks over several design choices and a scalability evaluation (Appendix A.2.3, A.2.4).

We describe the datasets, ML models, and parameters of AutoSlicer in Appendix A.2.1. We run all the experiments on an internal cluster. Due to the resource allocation mechanism that we do not have control over, the wall time of the same run may vary a lot. Therefore, we report the total CPU time across workers for fair comparison of computation cost.

The comparison between the strategies is shown in Table 1. We run the priority strategy for 5 iterations, and set the target number of candidates per iteration (K) to be 12% of the size of batch strategy search space. When the batch strategy cannot finish, we set K to 2500. Results under other configurations can be found in Appendix A.2.2. The significant slices in this experiment are those the model performs poorly on compared to the overall dataset. We report the number of significant slices found by each strategy (significant slices that are subsets of other significant ones are pruned), the actual number of candidate slices computed, the total CPU usage across workers and the amount of shuffle bytes. Note that due to the randomness of bootstrapping, the number of significant slices is slightly different across runs, which is the reason why the number of significant slices found by the priority strategy is larger than that by the exhaustive batch strategy in some cases.

The results show the effectiveness of our search space pruning and the priority strategy. On OpenML Electricity and UCI Census, the iterative strategy takes at least 45% less CPU seconds and 90% less shuffle bytes than the batch strategy due to the pruning. The priority strategy further reduces those costs. Compared to the iterative strategy, the priority strategy takes 40% less CPU seconds on OpenML Electricity and 76% less on UCI Census to find a comparable number of significant slices. The shuffle bytes are also reduced by at least 38% on both datasets. For Safe Driving and Traffic Stop, the priority strategy is able to finish while the other two strategies run out of time.

Table 1: Comparison of different search strategies.

Dataset	Strategy	# Significant Slices Found*	# Candidate Slices	CPU Usage (CPU-sec)	Shuffle Bytes (GB)
OpenML Electricity	Batch	317	30000	168K	233
	Iterative	327	23499	90.6K	19.3
	Priority	302	15114	54.9K	9.68
UCI Census	Batch	303	63861	412K	971
	Iterative	316	29079	213K	70.1
	Priority	374	30308	50.8K	43.4
Traffic Stop**	Priority	146	10157	1.34M	326
Safe Driving**	Priority	369	10532	1.19M	442

* The number of significant slices may vary across runs due to the randomness of bootstrapping.

** The batch and the iterative strategy cannot finish in 8 hours.

References

[1] An advanced unified programming model. https://beam.apache.org/.

[2] Safe driving dataset. https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/data.

[3] AGARWAL, A., BEYGELZIMER, A., DUDÍK, M., LANGFORD, J., AND WALLACH, H. A reductions approach to fair classification. In *International Conference on Machine Learning* (2018), PMLR, pp. 60–69.

[4] AGRAWAL, R., IMIELIŃSKI, T., AND SWAMI, A. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (1993), pp. 207–216.

[5] AGRAWAL, R., SRIKANT, R., ET AL. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (1994), vol. 1215, Citeseer, pp. 487–499.

[6] BAU, D., ZHOU, B., KHOSLA, A., OLIVA, A., AND TORRALBA, A. Network dissection: Quantifying interpretability of deep visual representations. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 6541–6549.

[7] CALMON, F. P., WEI, D., VINZAMURI, B., RAMAMURTHY, K. N., AND VARSHNEY, K. R. Optimized pre-processing for discrimination prevention. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), pp. 3995–4004.

[8] CELIS, L. E., HUANG, L., KESWANI, V., AND VISHNOI, N. K. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the conference on fairness, accountability, and transparency* (2019), pp. 319–328.

[9] CHAMANDY, N., MURALIDHARAN, O., NAJMI, A., AND NAIDU, S. Estimating uncertainty for massive data streams. Tech. rep., Google, 2012.

[10] CHEN, V. S., WU, S., WENG, Z., RATNER, A., AND RÉ, C. Slice-based learning: A programming model for residual learning in critical data slices. *Advances in neural information processing systems 32* (2019), 9392.

[11] CHUNG, Y., KRASKA, T., POLYZOTIS, N., TAE, K. H., AND WHANG, S. E. Automated data slicing for model validation: A big data-ai integration approach. *IEEE Transactions on Knowledge and Data Engineering 32*, 12 (2019), 2284–2296.

[12] GRALINSKI, F., WRÓBLEWSKA, A., STANISŁAWEK, T., GRABOWSKI, K., AND GÓRECKI, T. Geval: Tool for debugging nlp datasets and models. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP* (2019), pp. 254–262.

[13] HAJIAN, S., AND DOMINGO-FERRER, J. A methodology for direct and indirect discrimination prevention in data mining. *IEEE transactions on knowledge and data engineering 25*, 7 (2012), 1445–1459.

[14] HAN, J., PEI, J., AND YIN, Y. Mining frequent patterns without candidate generation. *ACM sigmod record 29*, 2 (2000), 1–12.

[15] KAHNG, M., FANG, D., AND CHAU, D. H. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (2016), pp. 1–6.

[16] KAMIRAN, F., AND CALDERS, T. Data preprocessing techniques for classification without discrimination. *Knowledge and Information Systems 33*, 1 (2012), 1–33.

[17] KOHAVI, R., ET AL. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Kdd* (1996), vol. 96, pp. 202–207.

[18] KRISHNAN, S., AND WU, E. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics* (2017), pp. 1–6.

[19] MA, S., LIU, Y., LEE, W.-C., ZHANG, X., AND GRAMA, A. Mode: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 175–186.

[20] PIERSON, E., SIMOIU, C., OVERGOOR, J., CORBETT-DAVIES, S., JENSON, D., SHOEMAKER, A., RAMACHANDRAN, V., BARGHOUTY, P., PHILLIPS, C., SHROFF, R., ET AL. A large-scale analysis of racial disparities in police stops across the united states. *Nature human behaviour 4*, 7 (2020), 736–745.

[21] SAGADEEVA, S., AND BOEHM, M. Sliceline: Fast, linear-algebra-based slice finding for ml model debugging. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2290–2299.

[22] SELVARAJU, R. R., COGSWELL, M., DAS, A., VEDANTAM, R., PARIKH, D., AND BATRA, D. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision* (2017), pp. 618–626.

[23] SUNDARARAJAN, M., TALY, A., AND YAN, Q. Axiomatic attribution for deep networks. In *International Conference on Machine Learning* (2017), PMLR, pp. 3319–3328.

[24] VANSCHOREN, J., VAN RIJN, J. N., BISCHL, B., AND TORGO, L. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter 15*, 2 (2014), 49–60.

[25] ZAFAR, M. B., VALERA, I., GOMEZ RODRIGUEZ, M., AND GUMMADI, K. P. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th international conference on world wide web* (2017), pp. 1171–1180.

[26] ZHANG, H., CHU, X., ASUDEH, A., AND NAVATHE, S. B. Omnifair: A declarative system for model-agnostic group fairness in machine learning. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2076–2088.

[27] ZHANG, J., WANG, Y., MOLINO, P., LI, L., AND EBERT, D. S. Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models. *IEEE transactions on visualization and computer graphics 25*, 1 (2018), 364–373.

[28] ZHOU, B., KHOSLA, A., LAPEDRIZA, A., OLIVA, A., AND TORRALBA, A. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 2921–2929.

A Appendix

A.1 Complete Algorithms for the Search Strategies

We show the complete algorithms for the iterative and priority strategies in Algorithm 1 and Algorithm 2.

A.2 Additional Experimental Details

A.2.1 Experimental Setup

Datasets and Models We consider four real-world datasets that contain a mixture of numerical and categorical features. The properties of these datasets are shown in Table 2. The prediction task is binary classification and the metric is binary accuracy. We consider different types of models that are widely used in practice, including random forest (RF), deep neural network (DNN) and gradient-boosted decision tree (GBDT). The models for OpenML Electricity and UCI Census are RF and GBDT respectively; the model for the others is DNN.

Parameters In all the experiments, we set the maximum cross size L to 3, the minimum slice size N_{min} to 1, and the p-value threshold α to 0.01 by default. We re-sample 20 times for the Poisson bootstrap approximation. For categorical features, we set J to 100 so that the categories that are not among the top-100 frequent ones are combined in to one category. For numerical features, we bucketize the value domain so that each bin contains 10% of the examples.

A.2.2 More Configurations for the priority strategy

For the priority strategy, we fix the number of iterations to 5, and set four different configurations (Priority-Config 1-4) by varying the target number of candidate slices per iteration (K). When the

Algorithm 1: Slice finding algorithm with the iterative strategy

```
Inputs: dataset $D$, model $h$, metric $\psi$, maximum cross size $L$, minimum slice size $N_{\text{min}}$, $p$-value
  threshold $\alpha$;
Outputs: set of significant slices $\mathcal{S}$;
$i \leftarrow 1$ ;                                            /* Iteration number.  */
$\mathcal{S} \leftarrow \emptyset$ ;                                      /* Set of significant slices.  */
$\mathcal{E} \leftarrow \emptyset$ ;                                   /* Set of slices with size $< N_{\text{min}}$.  */
$\mathcal{N}_{\text{prev}} \leftarrow \{S_O\}$ ; /* Set of nonsignificant slices from the previous iteration.
  Only contain the overall slice in the beginning.  */
$\mathcal{P}_{\text{singleton}} \leftarrow$ GetSetOfSingletonPredicates($D$);
while $i \leq L$ do
    $\mathcal{C} \leftarrow \emptyset$ ;                                     /* Set of candidate slices.  */
    for $S_P \in \mathcal{N}_{prev}$ do
        for $P_{singleton} \in \mathcal{P}_{singleton}$ do
            $P' \leftarrow P \wedge P_{\text{singleton}}$;
            if $S_{P'}$ is not a subset of any slices in $\mathcal{S}$ or $\mathcal{E}$ then
                $\mathcal{C} \leftarrow \mathcal{C} \cup \{S_{P'}\}$ ;
    $\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}} \leftarrow$ MetricComputingAndHypothesisTest($\mathcal{C}, D, h, \psi, \alpha$);
    $\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}}, \mathcal{E}_{\text{new}} \leftarrow$ FilterBySize($\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}}, N_{\text{min}}$);
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{\text{new}}$;
    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}_{\text{new}}$;
    $\mathcal{N}_{\text{prev}} \leftarrow \mathcal{N}_{\text{new}}$;
    $i \leftarrow i + 1$;
```

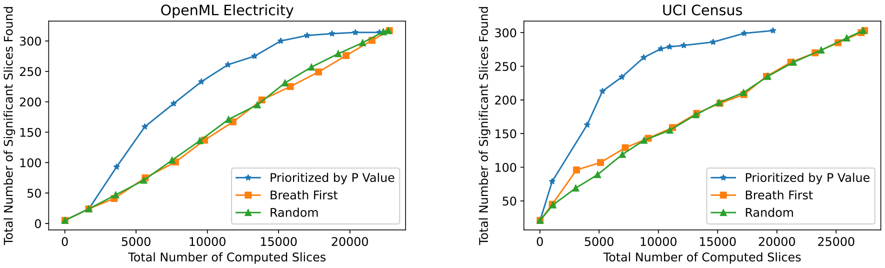

Figure 3: Number of significant slices found as the total number of computed slices varies.

batch strategy can finish, we choose proper values for K so that the sizes of the target search space are 10%, 20%, 40%, 60% of that of the batch strategy search space for Config 1-4. When the batch strategy cannot finish, we set K to 2500, 5000, 10000, 20000 for Config 1-4 respectively. The results are shown in Table 3.

A.2.3 Micro-Benchmarks

We provide several micro-benchmarks that validate the effectiveness of the design choices in AutoSlicer.

Effectiveness of p-Value as Priority Score We compare different types of priority scores in the priority-queue-based iterative strategy. The results are shown in Figure 3. For the breath-first search, we use the depth (cross size) as the priority score. For the random search, we generate one random number for each slice as the priority score. We set the target number of candidate slices per iteration to 2000, and report the total number of significant slices found as the actual number of computed candidates increases. The results show that AutoSlicer can find more slices in its early stage with p-value as the priority score. For example, on OpenML Electricity, the p-value-prioritized strategy finds 90% of the significant slices in its first 14000 candidates, while the others take around 20000 candidates. Similarly, on UCI Census, the p-value-prioritized strategy takes less than 12000 candidates while the others take around 25000 candidates.

Algorithm 2: Slice finding algorithm with the priority strategy

```
Inputs: dataset $D$, model $h$, metric $\psi$, maximum cross size $L$, minimum slice size $N_{\text{min}}$, $p$-value
  threshold $\alpha$, number of iterations $I$, target number of candidate slices per iteration $K$;
Outputs: set of significant slices $\mathcal{S}$;
$i \leftarrow 1$ ;                                              /* Iteration number. */
$\mathcal{S} \leftarrow \emptyset$ ;                            /* Set of significant slices. */
$\mathcal{E} \leftarrow \emptyset$ ;                            /* Set of slices with size $< N_{\texttt{min}}$. */
$\mathcal{Q} \leftarrow []$ ;                                   /* Priority queue sorted by $p$-values. */
$\mathcal{P}_{\text{singleton}} \leftarrow$ GetSetOfSingletonPredicates($D$);
while $i \leq I$ do
    $\mathcal{C} \leftarrow \emptyset$ ;                        /* Set of candidate slices. */
    if $l = 1$ then
        for $P \in \mathcal{P}_{\textit{singleton}}$ do
            $\mathcal{C} \leftarrow \mathcal{C} \cup \{S_P\}$
    else
        $k \leftarrow 0$ ;                  /* Estimated number of nonempty candidate slices. */
        while $k < K$ do
            $S_P \leftarrow$ PopFromQueue($\mathcal{Q}$) for $P_{\textit{singleton}} \in \mathcal{P}_{\textit{singleton}}$ do
                $P' \leftarrow P \wedge P_{\text{singleton}}$;
                if $S_{P'}$ is not a subset of any slices in $\mathcal{S}$ or $\mathcal{E}$ and $|P'| \leq L$ then
                    $\mathcal{C} \leftarrow \mathcal{C} \cup \{S_{P'}\}$ ;
                    $k \leftarrow k +$ NonEmptyRate($|P'|$);
    $\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}} \leftarrow$ MetricComputingAndHypothesisTest($\mathcal{C}, D, h, \psi, \alpha$);
    UpdateNonEmptyRate($\mathcal{C}, \mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}}$);
    $\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}}, \mathcal{E}_{\text{new}} \leftarrow$ FilterBySize($\mathcal{S}_{\text{new}}, \mathcal{N}_{\text{new}}, N_{\text{min}}$);
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{\text{new}}$;
    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}_{\text{new}}$;
    for $S_P \in \mathcal{N}_{\textit{new}}$ do
        PushToQueue($\mathcal{Q}, S_P$)
    $i \leftarrow i + 1$;
```

Table 2: Properties of the datasets in the experiments.

Dataset	# Records	# Numerical Features	# Categorical Features
OpenML Electricity [24]	3,621	8	1
UCI Census [17]	3,918	5	9
Traffic Stop [20]	72,398	9	7
Safe Driving [2]	47,655	58	7

Effectiveness of Nonempty Rate In the priority-queue-based iterative strategy , we keep track of the nonempty rate for each cross size to estimate the number of nonempty slices to be computed. We compare the estimated and the actual number of nonempty slices, and show the results in Figure 4. The data points are collected from several runs across different datasets. We can see that the estimated number of nonempty slices gets very close to the actual number by keeping track of the nonempty rate.

Effectiveness of Minimum Slice Size AutoSlicer uses the minimum slice size, N_{min}, to prune the search space for the iterative strategies. N_{min} is set to 1 by default. Increasing N_{min} significantly reduces the number of candidate slices to be explored. For example, when we run the simple iterative strategy on OpenML Electricity, the number of candidate slices explored reduces by 58.3% when increasing N_{min} to 50, and further reduces by 33.4% when increasing N_{min} to 100.

A.2.4 Scalability

We examine the scalability of AutoSlicer by increasing the dataset size. Figure 5 shows the CPU usage as we increase the dataset size. Here a value 4x on the x-axis means that we replicate the

Table 3: Comparison of different search strategies.

Dataset	Strategy	# Significant Slices Found*	# Candidate Slices	CPU Usage (CPU-sec)	Shuffle Bytes (GB)
OpenML Electricity	Batch	317	30000	168K	233
	Iterative	327	23499	90.6K	19.3
	Priority-Config 1	101	2859	14.1K	0.967
	Priority-Config 2	163	5638	17.8K	2.72
	Priority-Config 3	261	10724	40.3K	8.36
	Priority-Config 4	302	15114	54.9K	9.68
UCI Census	Batch	303	63861	412K	971
	Iterative	316	29079	213K	70.1
	Priority-Config 1	195	5690	14.1K	2.8
	Priority-Config 2	258	10784	15.2K	5.88
	Priority-Config 3	321	24484	46.4K	29.6
	Priority-Config 4	374	30308	50.8K	43.4
Traffic Stop**	Priority-Config 1	146	10157	1.34M	326
	Priority-Config 2	204	19317	1.79M	981
	Priority-Config 3	202	34442	1.44M	1720
	Priority-Config 4	245	65899	4.01M	7350
Safe Driving**	Priority-Config 1	369	10532	1.19M	442

* The number of significant slices may vary across runs due to the randomness of bootstrapping.

** The strategies or configurations that are not reported cannot finish in 8 hours.

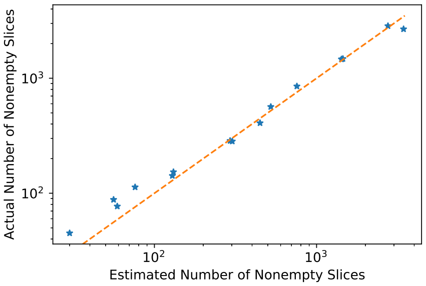

Figure 4: Estimated and actual number of nonempty slices collected from different runs. The dashed line represents the ideal case where the actual number is the same as estimated.

dataset 4 times. From the figure, we can see a linear relationship between the dataset size and the computation cost.

A.3 Related Work

Data slicing for model debugging MLCube [15] and Manifold [27] provide visualizing tools where the user can interactively specify slices and get their statistics. Slice-based Learning [10] allows the user to specify slices to which extra model capacity will be allocated. SliceFinder [11] searches for problematic slices where a model performs poorly in an automatic manner by statistical tests. It enumerates and computes one slice each time sequentially. SliceLine [21] formulates slice finding as a linear-algebra problem, which can be solved efficiently by matrix computing tools.

Other model debugging tools PALM [18] approximates how much each training example contributes to the prediction so that the user can take actions for those that are responsible for the failure modes. MODE [19] performs model state differential analysis to identify problematic internal features in the model and fix them by input selection and retraining. GEval [12] finds problematic test data, errors in prepossessing and issues in models that contribute to performance degeneration for natural language processing tasks. Feature attribution [28, 22, 6, 23] techniques provide scores to quantify how each input feature contributes to the final prediction.

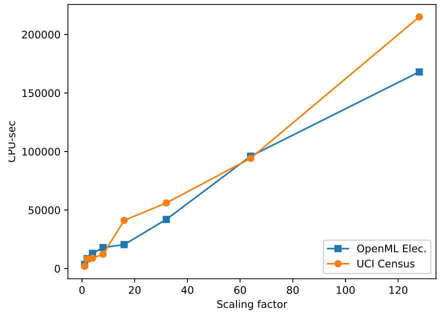

Figure 5: CPU usage on increasing dataset size.

Fairness in machine learning Machine learning fairness deals with the problem where a model treats certain groups of data in a biased way. To solve this problem, some works [16, 13, 7] rely on the preprocessing steps to reduce bias in the training data. Another line of works [25, 3, 8, 26] improve model fairness by incorporating fairness constraints in the training process. For example, OmniFair [26] allows the user to specify fairness constraints under which it maximizes the model accuracy. For the fairness problem, AutoSlicer is a useful tool to identify groups that are treated unfairly by the model.

Frequent Itemset Mining The problem of frequent itemset mining [4] aims to find item sets that are frequently shown up together from a set of transactions. The lattice search space of automated slicing is similar to that of the frequent itemset mining if we treat each singleton predicate as a single item and the conjunction of them as itemsets. Algorithms solving the frequent itemset mining problem [5, 14] typically prune the search space based on the downward closure lemma which states that subsets of a frequent itemset must also be frequent. We also rely on this property to prune the candidates whose parent contains less than N_{min} examples. However, such monotonicity property does not hold for the significance of slices in the automated slicing problem.