Debugging Machine Learning Pipelines

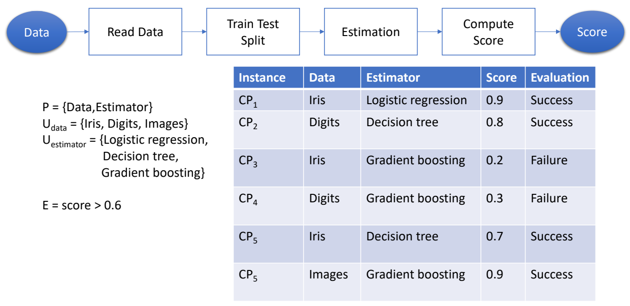
Raoni Lourenço
New York University
raoni@nyu.edu

Juliana Freire
New York University
juliana.freire@nyu.edu

Dennis Shasha
New York University
shasha@courant.nyu.edu

ABSTRACT

Machine learning tasks entail the use of complex computational pipelines to reach quantitative and qualitative conclusions. If some of the activities in a pipeline produce erroneous or uninformative outputs, the pipeline may fail or produce incorrect results. Inferring the root cause of failures and unexpected behavior is challenging, usually requiring much human thought, and is both time consuming and error prone. We propose a new approach that makes use of iteration and provenance to automatically infer the root causes and derive succinct explanations of failures. Through a detailed experimental evaluation, we assess the cost, precision, and recall of our approach compared to the state of the art. Our source code and experimental data will be available for reproducibility and enhancement.



P = {Data,Estimator}
U_{data} = {Iris, Digits, Images}
$U_{estimator}$ = {Logistic regression, Decision tree, Gradient boosting}

E = score > 0.6

Instance	Data	Estimator	Score	Evaluation
CP_1	Iris	Logistic regression	0.9	Success
CP_2	Digits	Decision tree	0.8	Success
CP_3	Iris	Gradient boosting	0.2	Failure
CP_4	Digits	Gradient boosting	0.3	Failure
CP_5	Iris	Decision tree	0.7	Success
CP_5	Images	Gradient boosting	0.9	Success

Figure 1: Machine learning pipeline and its provenance. Users can select different input datasets and classifier estimators to solve a classification problem.

CCS CONCEPTS

• **Information systems** → *Data provenance*;

ACM Reference Format:
Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2019. Debugging Machine Learning Pipelines. In *International Workshop on Data Management for End-to-End Machine Learning (DEEM'30), June 30, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3329486.3329489

1 INTRODUCTION

Machine learning pipelines, like many computational processes, are characterized by interdependent modules, associated parameters, data inputs, and outputs from which conclusions are derived. If one or more modules in a pipeline produce erroneous outputs, the conclusions may be incorrect.

Discovering the root cause of failures in a pipeline is challenging because problems can come from many different sources, including bugs in the code, input data, and improper parameter settings. This problem is compounded when multiple pipelines are composed. For example, when a machine learning pipeline feeds predictions to a second analytics pipeline, errors in the machine learning model can lead to erroneous actions taken based on the analytics results. Similar challenges arise in pipeline design, when developers need to understand trade-offs and test the effectiveness of their design decisions, such as the choice of a particular learning method or the selection of a training data set. To debug pipelines, users currently expend considerable effort reasoning about possibly incorrect or sub-optimal settings and then executing new pipeline instances to test hypotheses. This is tedious, time-consuming, and error-prone.

MLDebugger. We propose *MLDebugger*, a method that automatically identifies one or more minimal causes of failures or unsatisfactory performance in machine learning pipelines. It does so (i) by using the provenance of previous runs of a pipeline (i.e., information about the runs and their results), and (ii) by proposing and running carefully selected configurations consisting of so far untested new combinations of parameter values.

To see why we need to test new configurations, consider a setting in which several analytical algorithms can be used each with a set of hyperparameters. The results of using some hyperparameter settings can lead to useless outputs (e.g., low quality predictions) or even a crash. Sometimes, it is unclear which single hyperparameter-value setting or which combinations cause such results. Figure 1 shows a concrete example: a generic template for a machine learning pipeline and a log of different instances of the pipeline that were run with their associated results. The pipeline reads a data set, splits it into training and test subsets, creates and executes an estimator, and computes the F-measure score using 10-fold cross-validation. A data scientist uses this template to understand how different estimators perform for different types of input data, and ultimately, to derive a pipeline instance that leads to high scores. This entails exploring different parameter-values, training data sets, and learning classifiers.

Analyzing the provenance of the runs, we can see that *gradient boosting* leads to much lower scores than other methods for two of the data sets (i.e., Iris and Digits), but it has a high score for the *Images* data. This is counter-intuitive and may suggest that there is a problem with *gradient boosting* for some parameters. Because each of these runs used different parameters for each method depending on the data set, a definitive conclusion has to await a more systematic exploration through additional experiments with different parameter settings. Furthermore, since many parameter-value combinations can contribute to a poor outcome (or bug), it is also important to derive concise, ideally minimal explanations for the behavior.

Understanding why the values are lower can help identify bugs. In this case, as we discuss in Section 3, *MLDebugger* discovers that the low scores for Gradient Boosting happen only when a specific version the machine learning library containing the estimators is used, suggesting that a bug may have been introduced in this version.

We note that, in the above example, the problem was traced back to a library upgrade. Sometimes, however, the problem can be related to values assigned to variables in specific functions or the data set itself. For example, an industrial colleague cited an example in which an input to an analysis pipeline changed its resolution from monthly to weekly, causing the analysis to produce erroneous results. Our approach would identify that change in the data to be the root cause of the error. In fact, as Section 2 shows, our approach captures the most relevant aspects of a pipeline, including data, data type, library versions, and values assigned to input parameters.

Contributions. We present a bug-finding and explanation method that makes two novel contributions:

(1) Our algorithm (i) takes a given set of pipeline instances some of which give erroneous results, forms a hypothesis about possible root causes, then (ii) carefully selects new pipeline instances to test so-far untested properties-value combinations. It then iterates on (i) and (ii) until some time budget is exhausted or until it finds a definitive root cause.

(2) From a set of definitive root causes, our method finds and reports a minimal root cause, represented as a Boolean formula containing a minimal set of property-comparator-values that would cause the bug.

Because relying solely on stored provenance can lead to incomplete (or incorrect) explanations, Contribution 1 ensures fewer false positives compared to the state of the art. Contribution 2 helps with the precision of the diagnosis, which is necessary for the swift resolution of the bug.

We also carry out a detailed experimental evaluation which shows that our approach attains higher precision and recall compared with the state of the art, and that it also derives more concise explanations. The experimental data and source code for our system will be made available as open source for reproducibility and enhancement. We note that the algorithms of *MLDebugger* are general and can be applied to computational pipelines other than machine learning. Such applications are, however, out of the scope of this paper.

Outline. The remainder of this paper is organized as follows. Section 2 introduces the model we use for machine learning pipelines and formally defines the problem we address. In Section 3, we present algorithms to search for simple and complex causes of failures. We compare our approach with the state of the art in Section 4. We review related work in Section 5, and conclude in Section 6, where we outline directions for future work.

2 DEFINITIONS AND PROBLEM STATEMENT

Intuitively, given a set of pipeline instances, some of which have led to bad or questionable results, our goal is to find the root causes of these results possibly by creating and executing new pipeline instances.

Definition 1 (Pipeline, pipeline instance, property-value pairs, value universe, results)**.** A **machine learning pipeline** MP is associated with a set of properties P (i.e., including hyperparameters, input data, versions of programs, computational modules) each of which can take on various values. A **pipeline instance**, denoted as MP_i, of MP defines values for the properties. Thus, an instance MP_i is associated with a list of **property-value pairs** Pv_i containing some assignment (p, v) for all $p \in P$. For each property $p \in P$, the **property-value universe** U_p is the set of all values that have been assigned to p by any pipeline instance so far, i.e., $U_p = \{v | \exists i(p, v) \in P_i\}$.

Note that in a normal use case, our goal is to find the root causes for problematic instances, not to do software verification. Therefore our universe of property values for each property p is U_p, the set already seen. That is, we seek to understand a root cause from among the *existing* values.

An application-dependent evaluation procedure can be defined to decide whether the pipeline results are acceptable or not, and flag instances that should be investigated. In a machine learning context, an evaluation procedure may be to test whether the cross-validation accuracy is above a certain threshold.

Definition 2 (Evaluation). Let E be a procedure that **evaluates** the results of an instance such that $E(MP_i) = \texttt{succeed}$ if the results are acceptable, and $E(MP_i) = \texttt{fail}$ otherwise.

Definition 3 (Hypothetical root cause of failure). Given a set of instances $G = MP_1, \ldots, MP_k$ and associated evaluations $E(MP_1), \ldots., E(MP_k)$, a hypothetical root cause of failure is a set C_f consisting of a Boolean conjunction of property-comparator-value triples which obey the following properties among the instances G: (i) there is at least one MP_i such that Pv_i satisfies C_f and $E(MP_i) = \texttt{fail}$; and (ii) if $E(MP_i) = \texttt{succeed}$, then the property-values pairs Pv_i of MP_i do not satisfy the conjunction C_f.

To illustrate the converse of point (ii), if $C_f = A > 5$ and $B = 7$, and MP_i has the property values $A = 15$ and $B = 7$ and succeeds, then C_f is unacceptable as a hypothetical root cause of failure. Given a hypothetical root cause D, our framework derives new pipeline instances using different combinations of property values to confirm whether D is a definitive root cause.

Definition 4 (Definitive root cause of failure). A *hypothetical root cause of failure D* is a *definitive root cause of failure* if there is no instance MP_q from the universe of U_p for each property p such that $E(CP_q) = \texttt{succeed}$ and Pv_q satisfies D. That is, D does not lead to false positives.

Definition 5 (Minimal Definitive Root Cause of Failure). A definitive root cause D is minimal if no proper subset of D is a definitive root cause.

The example in Figure 1 illustrates these concepts using the simple machine learning pipeline. A possible evaluation procedure would test whether the resulting score is greater than 0.6. In this case, `Data` being different from *Images* and `Estimator` equal to *gradient boosting* is a hypothetical root cause of failure. Section 3 presents algorithms that determine if this root cause is definitive and minimal.

Problem Definition. Given a machine learning pipeline *MP* and a set of property-value pairs associated with previously run instances of *MP*, our goal is to derive minimal definitive root causes.

3 DEBUGGING STRATEGY OVERVIEW

Given a set of pipeline instances, *MLDebugger* derives minimal root causes of the problematic instances. Since trying every possible property-value pair combination of the property-value universe (an approach that is exponential in the number of properties) is not feasible in practice, *MLDebugger* uses heuristics that are effective at finding promising configurations. In addition, several causes may contribute to a problem, thus the derived explanations must be concise so that users can understand and act on them.

MLDebugger uses an iterative debugging algorithm called *Debugging Decision Tree*, presented in Section 3.1. It discovers simple and complex root causes that can involve a single or multiple properties and possibly inequalities. Because the results of the *Debugging Decision Tree* algorithm consist of disjunctions of conjunctions, they may contain redundancies which we simplify using a heuristic approximating the Quine-McCluskey algorithm described in Section 3.2.

Intuitively, our method works as follows. Given an initial set of instances, some of which lead to bad outcomes, the algorithms generate new property-value configurations (from the same universe) for the suspect instances and combine them first with property-values that led to good outcomes. That approach has the benefit of swiftly eliminating hypothetical minimal root causes that are not confirmed by the newly generated instances. While instances can be manually derived by users running instances of the workflow, an initial set of experiments can also be generated by random combinations of property values, or combinatorial design [11].

3.1 Debugging Decision Trees

An *instance* consists of a conjunction of property-values and an evaluation (success or failure). A *Debugging Decision Tree* is derived by applying a standard decision tree learning algorithm to all such instances. Leaves of the decision tree are either (i) purely *true*, if all pipeline instances leading to a leaf evaluate to `succeed`, (ii) purely *false*, if all pipeline instances leading to a leaf evaluate to `fail`, or (iii) *mixed*.

The *Debugging Decision Tree* algorithm works as follows:

(1) Given an initial set of instances *MPI*, construct a decision tree based on the evaluation results for those instances (`succeed` or `fail`). An inner node of the decision tree is a triple (*Property,Comparator,Value*), where the *Comparator* indicates whether a given *Property* has a value equal to, greater than (or equal to), less than (or equal to), or unequal to *Value*.

(2) If a conjunction (a path in the tree) involving a set of properties, say, P_1 P_2, and P_3 leads to a consistently failing execution (a pure leaf in decision tree terms), then that combination becomes a "suspect".

Table 1: Initial set of classification pipelines instances

Dataset	Estimator	Library Version	Score	Evaluation ($score \geq 0.6$)
Iris	Logistic Regression	1.0	0.9	`succeed`
Digits	Decision Tree	1.0	0.8	`succeed`
Iris	Gradient Boosting	2.0	0.2	`fail`

(3) Each suspect combination is used as a filter in a Cartesian product of the property values from which new instances will be sampled. For example, suppose $P_1 = v_1$, $P_2 = v_2$, and $P_3 = v_3$ is a suspect. To test this suspect, all other properties will be varied. If every instance having the property-values $P_1 = v_1$, $P_2 = v_2$, and $P_3 = v_3$ leads to failure, then that conjunction constitutes a *definitive root cause of failure.* If the suspect conjunction includes non-equality comparators (e.g., $P_1 = v_1$, $P_2 = v_2$, and $P_3 > 6$), then we can choose any value for properties that satisfy the inequalities as an example, (e.g., $P_3 = 7$ or $P_3 = 8$) and choose pipeline instances having those values. Conversely, if any of the newly-generated instances presents a good (`succeed`) pipeline instance, the decision tree is rebuilt taking into account the whole set of executed pipeline instances *MPI* and a new suspect path (one leading to a pure fail outcome) is tried.

Note that if the values associated with a property are continuous, *MLDebugger* starts by choosing the values already attempted. Further analysis can sample other values to uncover additional bugs, but, as mentioned above, our purpose here is to understand the problems already uncovered rather than to verify the software which is of course undecidable in general [4].

Below we present a simple example that illustrates how the *Debugging Decision Tree* algorithm works.

Example 3.1. Consider again the machine learning pipeline in Figure 1. Here, the user is interested in investigating pipelines that lead to low F-measure scores and defines an evaluation function that returns `succeed` if $score \geq 0.6$.

For this pipeline, the user is interested in investigating three properties: `Dataset`, the input data to be classified; `Estimator`, the classification algorithm to be executed; and `Library Version`, property that indicates the version of the machine learning library used. Table 1 shows examples of three executions of the pipeline.

A decision tree is created from the instances shown in Table 1 that contains a single node: (*Estimator,Equals to,Gradient Boosting*). After assembling new configurations where this triple is true (Table 2), the *Debugging Decision Tree* algorithm observes that new instances present mixed results. Hence, we eliminate the hypothetical cause for `Estimator` with value "Gradient Boosting" and the decision tree is rebuilt. After rebuilding, the algorithm finds a new single node tree with the triple (`Library Version`, *Equals to*,2.0), indicating a potential problem in that version of the library. Additional instances are then created to inspect the new root cause candidate, they all `fail` as can be seen in Table 3, confirming the hypothesis, which is output as a definitive root cause.

Table 2: Set of classification pipelines instances including the new instances created by *Debugging Decision Tree* based on triple (*Estimator,Equals to,Gradient Boosting*)

Dataset	Estimator	Library Version	Score	Evaluation ($score \geq 0.6$)
Iris	Logistic Regression	1.0	0.9	`succeed`
Digits	Decision Tree	1.0	0.8	`succeed`
Iris	Gradient Boosting	2.0	0.2	`fail`
Digits	Gradient Boosting	2.0	0.2	`fail`
Digits	Gradient Boosting	1.0	0.7	`succeed`

Table 3: Set of classification pipelines instances including the new instances created by *Debugging Decision Tree* based on triple (*Library Version,Equals to,2.0*)

Dataset	Estimator	Library Version	Score	Evaluation ($score \geq 0.6$)
Iris	Logistic Regression	1.0	0.9	`succeed`
Digits	Decision Tree	1.0	0.8	`succeed`
Iris	Gradient Boosting	2.0	0.2	`fail`
Digits	Gradient Boosting	2.0	0.2	`fail`
Digits	Gradient Boosting	1.0	0.7	`succeed`
Digits	Logistic Regression	2.0	0.3	`fail`
Iris	Decision Tree	2.0	0.1	`fail`

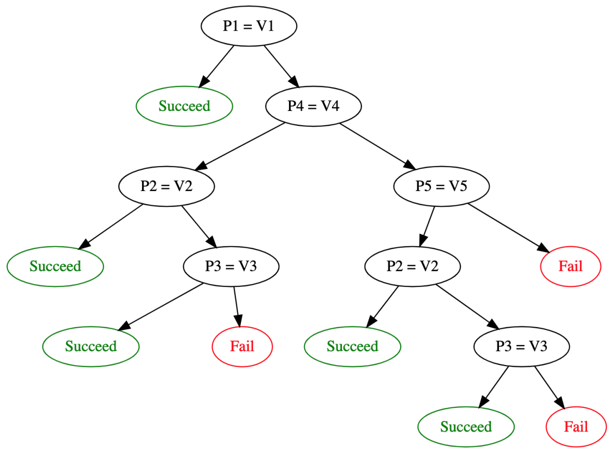


Figure 2: Decision tree fitted over instances. Right branches correspond to equality. Left branches to inequality. Evaluations to `fail` occur if ($P1 = V1, P4 \neq V4, P2 = V2$, and $P3 = V3$) or ($P1 = V_1, P4 = V4$, and $P5 = V5$) or ($P1 = V1, P4 = V4, P5 \neq V5, P2 = V2$, and $P3 = V3$).

3.2 Simplifying Explanations

Decision trees are easy to read, but they do not always provide minimal explanations. For example, we may have two paths leading to pure *false* leaves that differ only in the values of the first property which takes just two values. Such paths can be reduced to a single conjunction consisting of the property-values they share. To generate concise explanations from the decision tree, we apply the Quine-McCluskey algorithm [19], which provides a method to minimize Boolean functions. Because the algorithm is exponential and encodes the Set Cover problem which is NP-complete, we use heuristics that do not achieve complete minimality but still reduce the size of the explanation. We illustrate this process below in Example 3.2 and evaluate the effectiveness of the simplification in Section 4. Because the use of Quine-McCluskey is not our research contribution, our explanation is brief.

Example 3.2. Consider an experiment whose instances lead to the decision tree shown in Figure 2. There are three paths through the tree that evaluate to pure fail outcomes. The Quine-McCluskey algorithm attempts to shorten these paths to a simpler expression or expressions. The output of the algorithm contains the following disjunction of conjunctions (either one of them will constitute a minimal definitive root cause, but both of which should receive debugging attention):

$$((P1, V1) \wedge (P2, V2) \wedge (P3, V3)) \vee ((P_1, V1) \wedge (P4, V4) \wedge (P5, V5))$$

4 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of *MLDebugger*, we compare it against state-of-the-art methods for deriving explanations using a benchmark of machine learning pipeline templates for different tasks. We examine different scenarios, including when a single minimal definitive root cause is sought (which may be one of several) and when a budget for the number of instances that can be run is set.

4.1 Experimental Setup

Baseline Methods. We use two methods for deriving explanations as baselines: Data X-Ray [24] and Explanation Tables [13]. Both analyze the provenance of the pipelines, i.e., the instances previously run and their results, but do not suggest new ones. For that reason, to generate pipeline instances for explanation methods, we gave to each explanation method the instances generated by *MLDebugger* and by the Sequential Model-Based Algorithm Configuration (SMAC) [20]. SMAC is an iterative method for hyperparameter optimization that has been shown to be effective compared to previous methods [6]. Normally, SMAC looks for good instances, but for debugging purposes, we change its goal to look for bad pipeline instances. Note that SMAC proposes new pipeline instances in an iterative fashion, but it always outputs a complete pipeline instance (containing value assignments for all properties): the best it can find given a budget of instances to run and a criterion. This makes sense for SMAC's primary use case, which is to find a set of parameters that performs well, but it is less helpful for debugging, because a complete pipeline instance is rarely a minimal root cause. In summary, we combine the explanations with the generative methods: applying Data X-Ray and Explanation Tables to suggest root causes for the pipeline instances generated by SMAC and *MLDebugger*.

We also ran experiments using random search as an alternative, i.e., randomly generating instances and then analyzing them. However, the results were always worse than those obtained using SMAC or *MLDebugger*. Therefore, for simplicity of presentation and to avoid cluttering the plots, we omit these results.

Machine Learning Pipeline Benchmark. We generated a benchmark using the pipeline in Figure 3, whose structure is similar to that of Example 3.1. It solves the tasks of classification and regression, and it can be used as a template for Kaggle competitions [17]. As we describe below, we experimented with three different Kaggle competitions.

We define a threshold for acceptable performance. A pipeline instance that achieves or exceeds that threshold is good. Those that do not are considered to be bad. A Kaggle contestant may use these results to avoid bad parameter values,

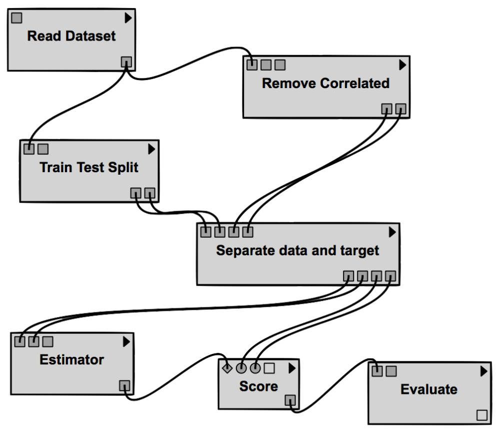


Figure 3: Pipeline template for classification and regression tasks

thus reducing the search space to explore. For instance, consider the dataset for a Kaggle competition regarding life insurance assessment[14], and an accuracy threshold of 0.4. *MLDebugger* identifies two root causes for bad instances (i.e., instances yielding a threshold less than 0.4):

I.1 *Estimator* = Gaussian NB
I.2 *Estimator* = K-Neighbors Classifier

If we increase the value of the accuracy threshold, then *MLDebugger* finds the following root cause:

II.1 *Estimator* ≠ Random Forest

In addition to the life insurance dataset, we selected: a Kaggle classification competition [16] whose goal it is to diagnose breast cancer based on a dataset from the University of Wisconsin, and a regression competition [15] whose goal is to predict annual sales revenue of restaurants.

For each competition, we applied the machine learning pipeline template of Figure 3, varying the range of acceptable accuracy score, for classification tasks, and of acceptable R-squared score, for the regression task.

We performed an exhaustive parameter exploration and manually investigated the root causes for failures to create a ground truth for our experiments, so that quality metrics can be computed.

Evaluation Criteria. We considered two goals: (i) *FindOne* find at least one minimal root cause; (ii) *FindAll* find all minimal root causes. The use case for (i) is a debugging setting where it might be useful to work on one bug at a time in the hopes that resolving one may resolve others. The use case for (ii) is when a team of debuggers can work on many bugs in parallel. *FindAll* may also be useful to provide an overview of the set of issues encountered.

We used the following criteria to measure quality: precision, which measures the fraction of causes identified by any given method that are in fact minimal definitive root causes; and recall, which measures (i) in the FindOne case, the fraction of pipelines for which at least one minimal definitive root cause is found and (ii) in the FindAll case, the fraction of all minimal root causes that are found. We also report the F-measure, i.e., the harmonic mean of precision and recall:

Formally, let UMP be a set of machine learning pipelines, where each pipeline $MP \in UMP$ (for example the pipeline of Figure 3) is associated with a set of minimal definitive root causes $R(MP)$. Given a set of minimal root causes asserted by an algorithm A, precision is the number of minimal root causes predicted by A that are truly minimal definitive root causes (true positives) divided by the size of the set of all root causes asserted by A over all MP in UMP. Precision is thus defined as:

$$Precision = \frac{\sum_{MP \in UMP} |A(MP) \cap R(MP)|}{\sum_{MP \in UMP} |A(MP)|}$$

where $A(MP) \cap R(MP) \neq \emptyset$ evaluates to 1 if $A(MP)$ corresponds to at least one of the conjuncts in $R(MP)$.

For the FindOne scenario, recall is the fraction of the $|UMP|$ pipelines when a true minimal definitive root cause is found by A:

$$Recall\ for\ FindOne = \frac{\sum_{MP \in UMP} A(MP) \cap R(MP) \neq \emptyset}{|UMP|}$$

where $A(MP) \cap R(MP) \neq \emptyset$ evaluates to 1 if $A(MP)$ corresponds to at least one of the conjuncts in $R(MP)$.

In the FindAll scenario, recall is the fraction of all the $R(MP)$ minimal root causes, for all $MP \in UMP$, that are found by the algorithms:

$$Recall\ for\ FindAll = \frac{\sum_{MP \in UMP} |A(MP) \cap R(MP)|}{\sum_{MP \in UMP} |R(MP)|}$$

$$F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Our first set of tests allow *MLDebugger* to find at least one minimal definitive root cause and then uses the same number of instances for the Data X-Ray and Explanation Tables. Thus, it gives the same budget to each algorithm and checks its precision and recall for the FindOne case. A second set of tests tries different budgets of pipeline instances and evaluates how each algorithm performs in terms of these same quality metrics. In these tests, Data X-Ray and Explanation Tables are given (i) the instances generated by *MLDebugger* and, in a separate test, (ii) the instances generated by SMAC. A similar pair of tests is performed for the FindAll case.

Implementation The current prototype of *MLDebugger* is implemented in Python 2.7. It contains a dispatching component which runs in a single thread and spawns multiple pipeline instances in parallel. In our experiments, we used five execution engine workers to execute the pipeline instances.

We used the SMAC version for Python 3.6. We also used the code, implemented by the respective authors, for both the Data X-Ray algorithm (implemented in Java 7) [24] and Explanation Tables [13] (written in python 2.7). As described above, we used the pipeline instances generated by both *MLDebugger* and SMAC as inputs to Data X-Ray and Explanation Tables.

The machine learning pipelines were constructed as VisTrails workflows[1], which allow us to capture the execution provenance of all instances.

All experiments were run on a Linux Desktop (Ubuntu 14.04, 32GB RAM, 3.5GHz × 8 processor). For purposes of reproducibility and community use, we will make our code and experiments available.[2]

4.2 Results

For our first set of experiments for FindOne, we set *MLDebugger* to stop iterating as soon as it found one minimal definitive root cause for failure. Figure 4 shows that *MLDebugger* and Explanation Tables both achieve perfect precision for the FindOne problem using the instances generated by *MLDebugger* when that is allowed run until it finds at least one minimal definitive root cause. Data X-Ray finds not only definitive root causes, but also configurations that do not always yield bad instances, resulting in its lower precision. *MLDebugger* enjoys higher recall, due to its ability to capture also root causes whose comparators are negations or inequalities.

Running pipelines can be expensive, so executing a very large number of instances to find a definitive root cause may not be feasible in practice. Hence, we also evaluated the effectiveness of the different methods when a budget is defined for the maximum number of instances to be executed for the FindOne problem. Figure 5 shows the results for different budgets for FindOne. Not surprisingly, *MLDebugger* makes mistakes when there is insufficient data to characterize the minimal definitive root causes, being as precise as Explanation Tables on high budget and attaining higher recall throughout.

Similar results hold for the FindAll case as shown in Figure 6 and Figure 7.

Discussion. Sometimes, when Data X-Ray uses the instances generated by *MLDebugger*, it does better, at least in recall.

[1]www.vistrails.org

[2]https://github.com/raonilourenco/MLDebugger

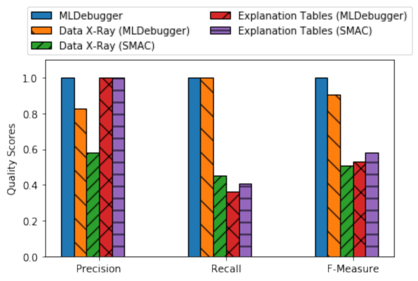


Figure 4: Machine Learning Pipeline Template case study for FindOne. *MLDebugger* and Explanation Tables have perfect precision, while Data X-Ray returns some false positives. *MLDebugger* and Data X-ray using the instances generated by *MLDebugger* achieve perfect recall, which means that they always find at least one minimal root cause.

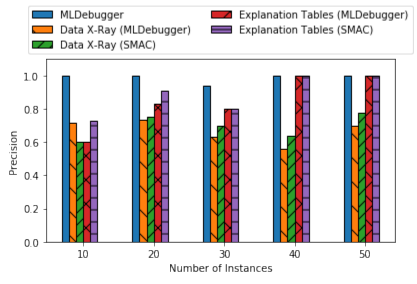
This is expected for the case where the root causes are conjunctions of property-comparator-value triples since Data X-Ray was designed to find relevant conjunctions. That is, Data X-Ray produces a conjunction of property-value combinations that lead to bad scenarios. By contrast, *MLDebugger* finds minimal decision trees for the data seen so far, prioritizing disjunction. When there is little data, jumping to generalizations can be a bad strategy (a lesson we have all learned from real life). This underscores the importance of doing a systematic and iterative search to obtain more data.

The answers provided by Explanation Tables represent a prediction of the pipeline instance evaluation result expressed as a real number. Here, we consider explanations whose prediction is 1.0, which means a certain, bad result. Therefore, the precision of Explanation Tables is always high, but the recall is usually low.

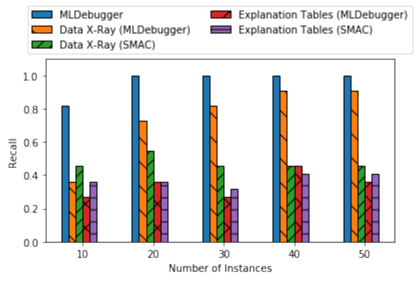
MLDebugger dominates the other methods and the performance difference increases as the budget grows. This can be explained by the fact that Data X-Ray provides explanations that are not minimal definitive root causes and Explanation Tables do not handle negations and inequalities.

5 RELATED WORK

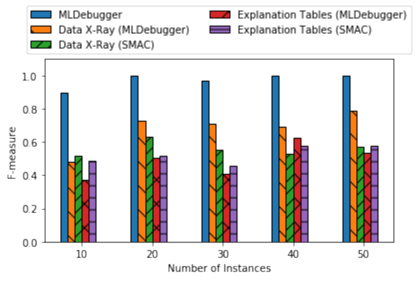
Recently, the problem of explaining query results and interesting features in data has received substantial attention in the literature [2, 10, 13, 21, 24]. Some works have focused on explaining where and how errors occur in the data generation process [24] and which data items are more likely to be causes of relational query outputs [21, 25]. Others have attempted to use data to explain *salient* features in data (e.g., outliers) by discovering relationships between attribute values [2, 10, 13]. These approaches have either focused on



(a) Precision



(b) Recall



(c) F-measure

Figure 5: FindOne problem. Machine Learning Pipeline Template case study on a budget. When we debug the Machine Learning Pipeline on a budget, *MLDebugger* provides answers with perfect precision, which is also achieved by Explanation Tables with more instances. *MLDebugger* shows perfect recall, hence f-measure as well, except when running on a very low budget

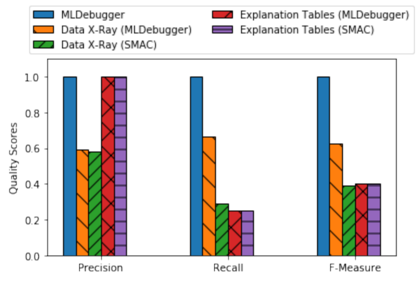


Figure 6: Machine Learning Pipeline Template case study for FindAll. As in the FindOne problem *MLDebugger* and Explanation Tables have perfect precision, while Data X-Ray returns some false positives. *MLDebugger* attains higher recall due to its ability to handle inequalities and negations.

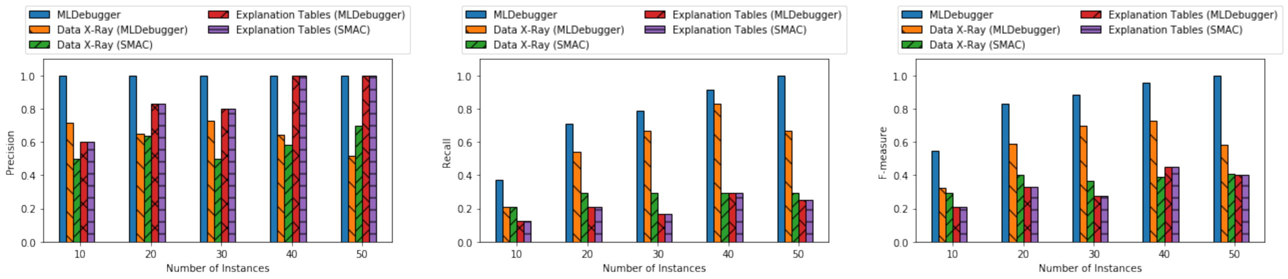
using data, including provenance, to explain data or considered pipelines consisting of relational algebra operations. In contrast to the goals of these approaches, *MLDebugger* aims to diagnose abnormal behavior in machine learning pipelines that may result from any source of error: data, programs, or sequencing of operations.

Our work is related algorithmically to approaches from hyper-parameter tuning, workflow debugging, and denial constraint identification. Hyper-parameter tuning methods explore the parameter space of pipelines to optimize their outcome – they automatically derive instances with improved performance. While their goal is to find good combinations for parameter values, they do not provide any insights into which combinations are most responsible for that performance, which would be analogous to what a high precision debugger finds.

Prior work on workflow debugging aims to identify and explain problems based on existing provenance, but they do not iteratively derive and test new workflow instances. As we demonstrated in Section 4, *MLDebugger* derives good explanations starting the debugging process from scratch and generating fewer pipeline instances than hyper-parameter optimization frameworks. Overall, *MLDebugger* also gives better recall and precision than non-iterative workflow-debugging tools.

Our approach is also related to the discovery of denial constraints in relational tables (denial constraints are generalizations of common constraints like functional dependencies) [8]. The main similarity is that both Denial constraints and the diagnoses of *MLDebugger* identify conjunctions of parameter-value inequalities. The Hydra algorithm in particular performs what it calls "focussed sampling" to find tuples that satisfy a predicate. *MLDebugger* does something similar in spirit when looking for pipeline instances that may disprove a hypothesis.

Hyperparameter Optimization. Methods based on Bayesian optimization are considered the state of the art for the hyperparameter optimization problem [5, 7, 12, 22, 23]. They can outperform manual setting of parameters, grid search or random search [6]. These methods approximate a probability model of the performance outcome given a parameter configuration that is updated from a history of executions. Gaussian Processes and Tree-structured Parzen Estimator are examples of probability models [5] used to optimize an unknown loss function using the 'expected improvement' criterion as acquisition function. To do this, they assume the search space is smooth and differentiable. This assumption, however, does not hold in general for arbitrary pipelines. Moreover, we are not interested in identifying a bad configurations (we have those to begin with if there have been some bugs already), but in finding minimal root causes.



(a) Precision **(b) Recall** **(c) F-measure**

Figure 7: FindAll problem. Machine Learning Pipeline Template case study on a budget. When we debug the Machine Learning Pipeline on a budget, *MLDebugger* provides answers with perfect precision, which is also achieved by Explanation Tables with more instances. *MLDebugger* continues to show higher recall, which increases as the budget increases, and f-measure than Data X-Ray and Explanation Tables.

Debugging and Predicting Pipelines. Previous work on pipeline debugging (not limited to machine learning) has focused on analyzing execution history with the goal of identifying problematic parameter settings or inputs. Because they do not use an iterative approach to derive new instances (and associated provenance), they can miss important explanations and also derive incorrect one. That said, the analytical portion of *MLDebugger* uses many similar ideas to pipeline debugging.

Bala and Chana [3] applied several machine learning algorithms (Naïve Bayes, Logistic Regression, Artificial Neural Networks and Random Forests) to predict whether a particular computational pipeline instance will fail to execute in a cloud environment. The goal is to reduce the consumption of expensive resources by recommending against executing the instance if it has a high probability of failure. The system does not try to find the root causes of failure.

The system developed by Chen et al. [9] identifies problems by differentiating between provenance (encoded as trees) of good runs and bad ones. They then find differences in the trees that may be the reason for the problems. However, the trees do not necessarily provide a succinct explanation for the problems, and there is no assurance that the differences found correspond to root causes.

Viska [18] allows users to define a causal relationship between workflow performance and system properties or software versions. It provides big data analytics and data visualization tools to help users to explore their assumptions. Each causality relation defines a treatment (causal variable) and an outcome (performance measurement), the approach is limited to analyze one binary treatment at a time with user in the loop.

The Molly system [1] combines the analysis of lineage with SAT solvers to find bugs in fault tolerance protocols for distributed systems. Molly simulates failures, such as permanent crash failures, message loss and temporary network partitions, specifically to test fault tolerance protocols over a certain period of (logical clock) time. The process considers all possible combinations of admissible failures up to a user-specified level (e.g., no more than two crash failures and no more message losses after five minutes). While the goal of that system is very specific to fault tolerance protocols, its attempt to provide completeness has influenced our work. In the spirit of Molly, *MLDebugger* tries to find *minimal definitive root causes*.

Explaining Pipeline Results. Although not designed for pipelines, Data X-Ray [24] provides a mechanism for explaining systematic causes of errors in the data generation process. The system finds common features among corrupt data elements and produce a diagnosis of the problems. If we have provenance of the pipeline instances together with error annotations, Data X-Ray's diagnosis would derive explanations consisting of features that describe the parameter-value pairs responsible for the errors.

Explanation Tables [13] is another data summary that provides explanations for binary outcomes. Like Data X-Ray, it forms its hypotheses based on given executions, but does not propose new ones. Based on a table with some categorical columns (attributes) and one binary column (outcome), the algorithm produces interpretable explanations of the causes of the outcome in terms of the attribute-value pairs combinations. Explanation Tables express their answers as a disjunction of patterns and each pattern is a conjunction of attribute-equality-value pairs.

As discussed in Section 4, *MLDebugger* produces explanations that are similar to those of Data X-Ray and Explanation Tables, but they are also minimal and are able to handle inequalities and negations. As mentioned above, *MLDebugger* employs a systematic method to automatically generate new instances that enable it to derive concise explanations that are root causes for a problem.

6 CONCLUSION

MLDebugger uses techniques from explanation systems and hyperparameter optimization approaches to address one of the most cumbersome tasks for data scientists and engineers: debugging machine learning pipelines. As far as we know, *MLDebugger* is the first method that iteratively finds minimal definitive root causes.

Compared to the state of the art, *MLDebugger* makes no statistical assumptions (as do Bayesian optimization approaches), but nevertheless achieves higher precision and recall given the same number of pipeline instances.

There are two main avenues we plan to pursue in future work. First, we would like to make *MLDebugger* available on a wide variety of systems that support pipeline execution to broaden its applicability. Second, we will use group testing to identify problematic subsets of datasets when a dataset has been identified as a root cause.

Acknowledgments. We thank the Data X-Ray authors and the Explanation Tables authors for sharing their code with us. This work has been supported in part by the U.S. National Science Foundation under grants MCB-1158273, IOS-1339362, and MCB-1412232, the Brazilian National Council for Scientific and Technological Development (CNPq) under grant 209623/2014-4, the DARPA D3M program the Moore-Sloan Foundation, and NYU WIRELESS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

REFERENCES

[1] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of ACM SIGMOD*. 331–346. https://doi.org/10.1145/2723372.2723711

[2] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *Proceedings of the ACM SIGMOD*. 541–556. https://doi.org/10.1145/3035918.3035928

[3] Anju Bala and Inderveer Chana. 2015. Intelligent Failure Prediction Models for Scientific Workflows. *Expert Syst. Appl.* 42, 3 (Feb. 2015), 980–989. https://doi.org/10.1016/j.eswa.2014.09.014

[4] Bonnie Berger, John Rompel, and Peter W. Shor. 1994. Efficient NC algorithms for set cover with applications to learning and geometry. *J. Comput. System Sci.* (1994). https://doi.org/10.1016/S0022-0000(05)80068-6

[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. *Advances in Neural Information Processing Systems (NIPS)* (2011), 2546–2554. https://doi.org/2012arXiv1206.2944S arXiv:1206.2944

[6] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2188385.2188395

[7] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of ICML*. I–115–I–123. http://dl.acm.org/citation.cfm?id=3042817.3042832

[8] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 311–323. https://doi.org/10.14778/3157794.3157800

[9] Ang Chen, Yang Wu, Andreas Haeberlen, Boon T. Loo, and Wenchao Zhou. 2017. Data Provenance at Internet Scale: Architecture , Experiences, and the Road Ahead. In *Proceedings of CIDR*.

[10] Fernando Chirigati, Harish Doraiswamy, Theodoros Damoulas, and Juliana Freire. 2016. Data Polygamy: The Many-Many Relationships Among Urban Spatio-Temporal Data Sets. In *Proceedings of ACM SIGMOD*. 1011–1025. https://doi.org/10.1145/2882903.2915245

[11] Charles J. Colbourn, Sosina S. Martirosyan, Gary L. Mullen, Dennis Shasha, George B. Sherwood, and Joseph L. Yucas. 2006. Products of mixed covering arrays of strength two. *Journal of Combinatorial Designs* 14, 2 (2006), 124–138. https://doi.org/10.1002/jcd.20065

[12] Nima Dolatnia, Alan Fern, and Xiaoli Fern. 2016. Bayesian Optimization with Resource Constraints and Production. In *International Conference on Automated Planning and Scheduling*. 115–123.

[13] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 61–72. https://doi.org/10.14778/2735461.2735467

[14] Google. 2015. Prudential Life Insurance Assessment. https://www.kaggle.com/c/prudential-life-insurance-assessment. (2015). Accessed: 2019-03-02.

[15] Google. 2015. Restaurant Revenue Prediction. https://www.kaggle.com/c/restaurant-revenue-prediction. (2015). Accessed: 2019-03-02.

[16] Google. 2016. Breast Cancer Wisconsin (Diagnostic) Data Set. https://www.kaggle.com/uciml/breast-cancer-wisconsin-data. (2016). Accessed: 2019-03-02.

[17] Google. 2018. Kaggle. http://www.kaggle.com. (2018). Accessed: 2019-03-02.

[18] Helga Gudmundsdottir, Babak Salimi, Magdalena Balazinska, Dan R.K. Ports, and Dan Suciu. 2017. A Demonstration of Interactive Analysis of Performance Measurements with Viska. In *Proceedings of ACM SIGMOD*. 1707–1710. https://doi.org/10.1145/3035918.3056448

[19] Jiangbo Huang. 2014. Programing implementation of the Quine-McCluskey method for minimization of Boolean expression. *CoRR* abs/1410.1059 (2014). arXiv:1410.1059 http://arxiv.org/abs/1410.1059

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. of LION-5*. 507–523.

[21] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *PVLDB* 7, 13 (2014), 1715–1716. http://www.vldb.org/pvldb/vol7/p1715-meliou.pdf

[22] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of NIPS*. 2951–2959. http://dl.acm.org/citation.cfm?id=2999325.2999464

[23] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. 2015. Scalable Bayesian Optimization Using Deep Neural Networks. In *Proceedings of the ICML*. 2171–2180. http://dl.acm.org/citation.cfm?id=3045118.3045349

[24] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of ACM SIGMOD*. 1231–1245. https://doi.org/10.1145/2723372.2750549

[25] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. QFix: Diagnosing Errors Through Query Histories. In *Proceedings of ACM SIGMOD*. 1369–1384. https://doi.org/10.1145/3035918.3035925