BugDoc: Algorithms to Debug Computational Processes

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

Data analysis for scientific experiments and enterprises, large-scale simulations, and machine learning tasks all entail the use of complex computational pipelines to reach quantitative and qualitative conclusions. If some of the activities in a pipeline produce erroneous outputs, the pipeline may fail to execute or produce incorrect results. Inferring the root cause(s) of such failures is challenging, usually requiring time and much human thought, while still being 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 experimental data and processing software is available for use, reproducibility, and enhancement.

CCS Concepts

• **Information systems** → *Data provenance.*

ACM Reference Format:
Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. BugDoc: Algorithms to Debug Computational Processes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389763

1 Introduction

Computational pipelines are widely used in many domains, from astrophysics and biology to enterprise analytics. They are characterized by interdependent modules, associated parameters, and data inputs. Results derived from these pipelines lead to conclusions and, potentially, actions. If one or more modules in a pipeline produce erroneous or unexpected outputs, these conclusions may be incorrect. Thus, it is critical to identify the causes of such failures.

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, software updates, and improper parameter settings. Connecting the erroneous result to its root cause is especially difficult for long pipelines or when multiple pipelines are composed. Consider the following real but sanitized examples.

Example: Enterprise Analytics. In an application deployed by a major software company, plots for sales forecasts showed a sharp decrease compared to historical values. After much investigation, the problem was tracked down to a data feed (coming from an external data provider), whose temporal resolution had changed from monthly to weekly. The change in resolution affected the predictions of a machine learning pipeline, leading to incorrect forecasts.

Example: Exploring Supernovas. In an astronomy experiment, some visualizations of supernovas presented unusual artifacts that could have indicated a discovery. The experimental analysis consisted of multiple pipelines run at different sites, including data collection at the telescope site, data processing at a high-performance computing facility, and data analysis run on the physicist's desktop. After spending substantial time trying to verify the results, the physicists found that a bug introduced in the new version of the data processing software had caused the artifacts.

To debug such problems, users currently expend considerable effort reasoning about the effects of the many possible different settings. This requires them to tune and execute new pipeline instances to test hypotheses manually, which is tedious, time-consuming, and error-prone.

We propose new methods and a system that automatically and iteratively identifies one or more minimal causes of failures in general computational pipelines (or workflows).

The Need for Systematic Iteration. Consider the example in Figure 1, which shows a generic template for a machine learning pipeline and a log of different instances that were run with their associated results.

The pipeline reads a dataset, splits it into training and test subsets, creates and executes an estimator, and computes the

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389763

Instance	Data	Estimator	Library Version	Score	Evaluation
CP_1	Iris	Logistic regression	1.0	0.9	Success
CP_2	Digits	Decision tree	1.0	0.8	Success
CP_3	Iris	Gradient boosting	2.0	0.2	Failure
CP_4	Digits	Gradient boosting	2.0	0.3	Failure
CP_5	Iris	Decision tree	1.0	0.7	Success
CP_6	Images	Gradient boosting	1.0	0.9	Success

Figure 1: Machine learning pipeline and its provenance. A data scientist can explore different input datasets and classifier estimators to identify a suitable solution for a classification problem.

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.

Analyzing the provenance of the runs, we can see that *gradient boosting* leads to low scores for two of the datasets (*Iris* and *Digits*), but it has a high score for *Images*. By contrast, *decision trees* work well for both the *Iris* and *Digits* datasets, and *logistic regression* leads to a high score for *Iris*.

This may suggest that there is a problem with the *gradient boosting* module for some parameters, that *decision trees* provide a suitable compromise for different data, and that *logistic regression* is good for the *Iris* data. Because each run used different parameters for each method depending on the dataset, a definitive conclusion has to await additional testing of these hyperparameters. Doing so manually is time-consuming and error-prone, while *BugDoc* automates this process.

Identifying Root Causes of Failures: Challenges. As the above examples illustrate, there are many potential causes for a given problem. Prior work used provenance to explain errors in computational processes that derive data [18, 46]. However, to test these hypotheses and obtain complete (and accurate) explanations, new pipeline instances must be executed that vary the different components of the pipeline.

Trying all possible combinations of parameter-values leads to a combinatorial explosion of instances to execute, and therefore can be prohibitively expensive. Thus, a critical challenge lies in the design of a strategy that is provably efficient (often requiring only a linear number of pipeline executions in the number of parameters) for finding root causes. Causes of errors can include multiple parameters, each of which may have large domains. So, it is important to have clear and concise explanations in terms of the parameter values already tried.

Contributions. In this paper, we introduce *BugDoc*, a new approach that makes use of iteration and provenance to infer the root causes automatically and derive succinct explanations of failures in pipelines. Our contributions can be summarized as follows:

(1) *BugDoc* finds root causes autonomously and iteratively, intelligently selecting so-far untested combinations.
(2) We propose debugging algorithms that find root causes using fewer pipeline instances than state-of-the-art methods, avoiding unnecessary costly computations. In fact, *BugDoc* often finds root causes using only a number of pipeline instances linear in the number of parameters.
(3) The *BugDoc* system further reduces time by exploiting parallelism, and
(4) Finally, *BugDoc* derives concise explanations, to facilitate the tasks of human debuggers.

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

2 Related Work

Debugging Data and Pipelines. Recently, the problem of explaining query results and interesting features in data has received substantial attention in the literature [4, 14, 18, 39, 46]. Some have focused on explaining where and how errors occur in the data generation process [46] and which data items are most likely to be causes of relational query outputs [39, 47]. Others have attempted to use data to explain *salient* features in data (e.g., outliers) by discovering relationships among attribute values [4, 14, 18]. In contrast, *BugDoc* aims to diagnose abnormal behavior in computational pipelines that may be due to errors in data, programs, or sequencing of operations.

Previous work on pipeline debugging has focused on analyzing execution histories to identify problematic parameter settings or inputs, but such work does not iteratively infer and test new workflow instances. Bala and Chana [5] applied several machine learning algorithms to predict whether a particular 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 attempt to find the root causes of such failures. Chen et al. [12] developed a system that identifies problems by finding the differences between provenance (encoded as

trees) of good and bad runs. However, in general, these differences do not necessarily identify root causes, though they often contain them.

Some systems have been developed to debug specific applications. Viska [24] helps users identify the underlying causes for performance differences for a set of configurations. Users infer hypotheses by exploring performance data and then test these hypotheses by asking questions about the causal relationships between a set of selected features and the resulting performance. Thus, Viska can be used to validate hypotheses but not identify root causes. Molly [1] combines the analysis of lineage with SAT solvers to find bugs in fault-tolerance protocols for distributed systems. It simulates failures, such as permanent crash failures, message loss, and temporary network partitions, in order to test fault-tolerance protocols over a specified period.

Although not designed for computational pipelines, Data X-Ray [46] provides a mechanism for explaining the systematic causes of errors in the data generation process. The system finds shared features among corrupt data elements and produces a diagnosis of the problems. Given the provenance of pipeline instances together with error annotations, Data X-Ray derives explanations consisting of features that describe the parameter-value pairs responsible for the errors. Explanation Tables [18] provides explanations for binary outcomes. Like Data X-Ray, it forms hypotheses based on a log of executions, but it does not propose new ones. Based on a table with a set of categorical columns (attributes) and one binary column (outcome), the algorithm produces interpretable explanations of the causes for the outcome in terms of the attribute-value pairs combinations. The explanations consist of a disjunction of patterns, and each pattern is a conjunction of attribute-value pairs. As discussed in Section 5, *BugDoc* produces explanations that are similar to those of Data X-Ray and Explanation Tables, but they are also minimal and able to express inequalities and negations. Furthermore, *BugDoc* employs a systematic method to intelligently generate new instances that enable it to derive concise explanations that are root causes for a problem.

Hyperparameter Tuning Our work is related algorithmically to approaches from hyperparameter tuning [6, 8, 17, 42, 43], since we can view the generation of new pipeline instances for debugging as an exploration of the space of its hyperparameters. Bayesian optimization methods are considered state of the art for the hyperparameter optimization problem [7, 8, 17, 42, 43]. 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 [6] 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 computational pipelines. Moreover, our goal is not to identify bad configurations (we usually have those, to begin with), but to identify the root cause(s), which are due to a subset of the parameters. Optimization, by contrast, seeks entire (in their case, good) configurations.

Examples of hyperparameter tuning techniques include OtterTune and BOAT. OtterTune [45] is a system that uses supervised learning techniques to find optimal settings of database system administrator knobs given a database workload and a set of metrics (optimization functions). BOAT [16] also optimizes database system configurations using Bayesian Optimization. However, instead of starting the optimization with a standard Gaussian process, it allows a user to input an initial probabilistic model that exploits previous knowledge of the problem.

Software Testing. State-of-the-art techniques for software testing [21, 30], statistical debugging [35, 51], and bug localization [2, 3, 25] are often application-specific and/or require a user-defined test suite. Some approaches require the instrumentation of binaries or source code in the form of predicates that can be observed during computational runs [35, 51]. Such information, if available, can be helpful to localize and explain bugs. *BugDoc*, however, does not assume any knowledge of the internal code of the computational processes: it was designed to debug black-box pipelines where we can observe only the inputs and outputs. Hence, our explanations are expressed in terms of input parameters. However, an interesting direction for future work would be to consider variables (or predicates) that can be observed but not manipulated in our formalism to generate potentially richer explanations. Approaches have also been proposed for bug localization in a black-box scenario; however these were designed for specific applications and environments, e.g., Pinpoint for J2EE [13]. By contrast, *BugDoc* was designed to support language-independent workflows.

Automated test generation techniques also derive new tests (or instances in our terminology). However, they do not aim to identify root causes (see, e.g., [19, 22, 27]). One exception is Causal Testing [30]. Similar to *BugDoc*, Causal Testing aims to help users identify root causes for problems. However, it requires the user to specify a (single) suspect variable to be investigated in a white-box scenario, while *BugDoc* searches for potential causes for failures in a black-box scenario. Further these causes may include multiple variables and value assignments.

BugDoc helps a user to trace back the potential cause of a given behavior to a component of a pipeline. Nevertheless,

since a pipeline can orchestrate a multitude of sophisticated tools, to identify and correct the bug, it may be necessary to drill down into an individual component. If source code is available for that, traditional debugging techniques can be used.

Identifying Denial Constraints. Our approach is also related to the discovery of denial constraints in relational tables [9, 15], particularly functional dependencies. The similarity can be illustrated as follows: imagine that there is a column indicating "successful instance" or "failed instance" for some set of parameter-values. Call it *Success Or Fail*. If a failure occurs exactly when parameter `A = 5` and `B = 6`, then that would manifest as a functional dependency `AB` $\rightarrow$ *Success Or Fail*, i.e., the result is a function of parameters `A` and `B`. However, if the failure happens when a disjunction holds, e.g., `A = 5` or `B = 6`, the same functional dependency would be inferred. No more minimal functional dependencies such as `A` $\rightarrow$ *Success Or Fail* would be inferred, because, for example, when `A = 4`, there can be success or failure depending on the `B` value. Thus, functional dependencies are not expressive enough to characterize root causes.

3 Definitions and Problem Statement

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

Definition 1. *(Pipeline, instance, parameter-value pairs, value universe, results) A* ***computational pipeline*** *(or workflow) CP is a collection of programs connected together that contains a set of manipulable parameters P (i.e., including hyperparameters, input data, versions of programs, computational modules). We denote as CP_i a* ***pipeline instance*** *of CP that defines values for the parameters for a particular run of CP. Thus, an instance CP_i is associated with a list of* ***parameter-value pairs*** *Pv_i containing an assignment (p, v) for each $p \in P$. We denote by $CP_i[p] = v$ the assignment of value v for parameter p in the instance CP_i. For each parameter $p \in P$, the* ***parameter-value universe*** *U_p is the set of all property-values assigned to p by any pipeline instance thus far, i.e., $U_p = \{v | \exists i (p, v) \in CP_i\}$. The* ***Universe*** *$U = \{(p, U_p) | p \in P\}$.*

As we discuss in Section 4, the initial parameter-value universe U can be expanded by explicitly defining the parameter domains (e.g., parameter satisfaction can take integer values between 1 and 10).

Definition 2. *(Evaluation) Let E be a procedure that* ***evaluates*** *the result of an instance such that $E(CP_i)$ = `succeed` if the results are acceptable, and $E(CP_i)$ = `fail` otherwise. Normally, the evaluation procedure will be code that looks at some property of the result of a given pipeline instance.*

Thus a bug, for the purposes of this paper, is a collection of pipelines that, when executed, evaluate to `fail`. Note that this is a deterministic definition that doesn't capture intermittent failures, e.g., timing bugs or non-deterministic failures. Even in such cases, however, if the bugs occur often enough, then *BugDoc* may help, though without guarantee.

Definition 3. *(Hypothetical root cause of failure) Given a set of instances $G = CP_1, ..., CP_k$ and associated evaluations $E(CP_1), ..., E(CP_k)$, a* ***hypothetical root cause of failure*** *is a set C_f consisting of a Boolean conjunction of parameter-comparator-value triples (e.g., a triple may be of the form $A > 5$) which obey the following conditions among the instances G: (i) there is at least one CP_i such that Pv_i satisfies C_f and $E(CP_i)$ = `fail`; and (ii) if $E(CP_i)$ = `succeed`, then the parameter-values pairs Pv_i of CP_i do not satisfy the conjunction C_f.*

Example. To illustrate the converse of point (ii), if $C_f = A > 5$ and $B = 7$, and CP_i has the parameter values $A = 15$ and $B = 7$ and succeeds, then C_f does not obey condition (ii) of a hypothetical root cause of failure.

C_f is called *hypothetical* because, based on the evidence so far, C_f leads to `fail`, but further evidence may refute that hypothesis.

We should note that the root causes defined here should not be interpreted as the *actual causes* of pipeline problems as characterized by causality theory [40]. The goal of *BugDoc* is to help the user identify sets of parameter-value pairs for which a black-box pipeline will always `fail`. However, the root causes we output are not counterfactuals [34], i.e., the pipeline would not necessarily `succeed` had the root cause not been observed, because perhaps another root cause may come into play. We simply want to determine the following implication definitively: *root-cause* $\Longrightarrow$ `fail` for a single root cause. *BugDoc* can, however, also discover disjunctive combinations of configurations that lead to failure.

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 CP_q from the universe of U with the property that $E(CP_q)$ = `succeed` and Pv_q satisfies D. Informally, no pipeline instance that includes D as a subset of its parameter-value settings leads to* `succeed`.

Definition 5. *(Minimal Definitive Root Cause) 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 from the introduction. 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 4 presents algorithms that determine whether this root cause is definitive and minimal.

Problem Definition. Given a computational pipeline CP (e.g., a query, script, simulation) and a set of parameter-value pairs associated with previously-run instances $G = CP_1, ..., CP_k$, we consider two goals: (i) to find at least one minimal definitive root cause or (ii) to find all minimal definitive root causes. Our cost measure for both goals is the number of executed pipeline instances beyond any given, previously run, instances.

4 Debugging Algorithms

Given a set of pipeline instances, *BugDoc* identifies minimal definitive root causes for failures. As noted above, a naive strategy would be to try every possible parameter-value pair combination of the parameter-value universe, requiring the testing of a number of pipeline instances that is exponential in the number of parameters. Instead, *BugDoc* uses heuristics that turn out to be quite effective at finding promising configurations.

BugDoc uses two iterative debugging algorithms in turn. The first, called *Shortcut*, discovers definitive root causes (which we sometimes abbreviate to, simply, bugs) consisting of a single conjunction of parameter-value (formally, parameter-equality-value) pairs. The second, called *Debugging Decision Trees* and introduced in [36], discovers more complex definitive root causes involving inequalities (e.g., A takes a value between 5 and 13).

Because the results of the *Debugging Decision Trees* algorithm consist of disjunctions of conjunctions, they may contain redundancies, which we simplify using the Quine-McCluskey algorithm [28]. The goal is to create concise explanations, making it easy for users to understand and act on them.

4.1 Looking for Single Root Causes: The Shortcut Algorithm

The *Shortcut* algorithm, shown in Algorithm 1, starts from a pipeline instance CP_f that evaluates to `fail`. It then uses pipeline instances that succeeded and are *disjoint* , i.e., they share no parameter-values, from CP_f to construct new tests.

Definition 6 (Disjoint Instances). *Two pipeline instances CP_x and CP_y are disjoint if $CP_x[p] \neq CP_y[p], \forall p \in P$ associated to CP.*

Intuitively, the *Shortcut* algorithm starts with the failing pipeline instance CP_f and a disjoint successful instance CP_g. The existence of such a disjoint succeeding pipeline instance is a requirement for the theoretical results that follow and is called the *Disjointness Condition*. If the Disjointness Condition does not hold, then this method may still be useful as a heuristic.

The *current* instance $CP_{current}$ is initialized to CP_f. Then, using some order among parameters, for each parameter p, an instance

$$CP_{current'}$$

is executed that consists of a copy of $CP_{current}$ except that $CP_{current'}[p] = CP_g[p]$. If the instance $CP_{current'}$ fails then $CP_{current}$ is changed to $CP_{current'}$ and the next parameter is considered. The intution is that the value of p in CP_f did not cause the failure. In the end, the definitive minimal root cause asserted by the *Shortcut* will be a subset of the pipeline instance CP_f that is still present in the final instance of $CP_{current}$. We denote that subset as D.

The algorithm then performs a sanity check to see whether any superset of the hypothetical minimal root cause D is in an already executed successful execution. If so, then the *Shortcut* algorithm has found a proper subset of the definitive minimal root cause, but not an actual definitive minimal root cause.

As noted above, if the Disjointness Condition does not hold, then the *Shortcut* algorithm can still be used as a heuristic: take an instance that differs in as many parameter-values as possible. While the theoretical results that follow will not hold, this will often be good enough, as the experimental results show (Section 5).

Here is an example that illustrates how the *Shortcut* algorithm works.

Example 1. *Consider the machine learning pipeline in Figure 1 again. 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* ≥ 0.6 *and* `fail` *otherwise.*

For this pipeline, the user investigates three parameters: `Dataset`*, the input data to be classified;* `Estimator`*, the classification algorithm to be executed; and* `Library Version` *indicates the version of the machine learning library used. Table 1 shows examples of three executions of the pipeline.*

Table 1: An initial (given) set of classification pipelines instances

Dataset	Estimator	Library Version	Score	Evaluation (*score* ≥ 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

In the initial traces shown in Table 1, there are only two disjoint instances with different evaluations:

Table 2: Set of classification pipelines instances including the new instances (shown in blue) created by *Shortcut* by substituting values of parameters in CP_f by corresponding values in CP_g.

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	Decision Tree	2.0	0.3	fail
Digits	*Decision Tree*	*1.0*	*0.8*	succeed

CP_g = {(Dataset,Digits),
(Estimator,Decision Tree),
(LibraryVersion,1.0)}
CP_f ={(Dataset,Iris),
(Estimator,Gradient Boosting),
(LibraryVersion,2.0) }

Examining parameter `Dataset`, we replace its corresponding value in the current instance to be executed from Iris to Digits. Because the execution evaluates to `fail`, we keep this replacement in the current instance. Similarly, when we update the value of parameter `Estimator` to Decision Tree, the instance evaluation is still `fail`, so we keep that replacement as well.

However, when `Library Version` is changed to 1.0, the resulting configuration evaluates to `succeed`. This suggests that `Library Version` 2.0 may be the source of the problem. Table 2, displays all pipeline instances evaluated, including the new instances generated by the *Shortcut* algorithm.

For Pipelines with root causes similar to the ones in Example 1, the algorithm will find a minimal definitive root cause.

Theorem 1. *If all definitive root causes are singleton parameter-values and the disjointness condition holds, then the shortcut algorithm will always assert exactly a minimal definitive root cause.*

Proof. By construction. If all definitive root causes are singletons, then CP_g cannot contain any element of a root cause, otherwise $E(CP_g)$ = `fail`. By contrast, CP_f must contain at least one root cause. When iterating over parameter p, the *Shortcut* algorithm will replace $CP_f[p]$ by $CP_g[p]$ (because the values must be different on all parameters p by the Disjointness Condition) while there is still one root cause in CP_{current}. Therefore, by the end of the algorithm, only the the root cause would remain. □

Guarantees of the Shortcut Algorithm. The *Shortcut* algorithm may be too aggressive in the sense that it can return a root cause D that is a proper subset of an actual minimal definitive root cause of failure.

Algorithm 1: *Shortcut* Algorithm

Input: *CPI*, the set of pipeline instances in the execution history characterized by their parameter-values
Input: E, the evaluation function
Input: P, list of parameters
Input: CP_f, pipeline instance evaluated as `fail`
Input: CP_g, pipeline instance evaluated as `succeed` disjoint to CP_f
Output: D, asserted minimal definitive root cause

```
/* Initialization                                  */
```
$CP_{\text{current}} \leftarrow CP_f$;
for $p \in P$ **do**
  $CP_{\text{current}'} \leftarrow CP_{\text{current}}$;
  $CP_{\text{current}'}[p] \leftarrow CP_g[p]$;
  if $E(CP_{current'})$ = `fail` **then**
    $CP_{\text{current}} \leftarrow CP_{\text{current}'}$;
  end
end
$D \leftarrow CP_{\text{current}} \cap CP_f$;
for $CP_i \in CPI$ **do**
  if $D \subseteq CP_i$ *and* $E(CP_i)$ = `succeed` **then**
    return $\emptyset$
  end
end
return D

Example 2. *Suppose that we have two minimal definitive root causes:*

(1) $D_1 = \{(p_1, v_1), (p_2, v_2)\}$
(2) $D_2 = \{(p_1, v_1'), (p_3, v_3)\}$

Consider also a computational pipeline consisting of three parameters $P = \{p_1, p_2, p_3\}$*, and* CP_f *and* CP_g *as follows:*

- $CP_f = \{(p_1, v_1), (p_2, v_2), (p_3, v_3)\}$
- $CP_g = \{(p_1, v_1'), (p_2, v_2'), (p_3, v_3')\}$

Clearly $D_1 \subseteq CP_f$*, therefore it is the root cause of the failure of* CP_f*. However, when iterating over parameter* p_1*, the* Shortcut *algorithm updates* $CP_{current}[p_1] = v_1'$*. But* $E(CP_{current'})$ = `fail` *because* $D_2 \subseteq CP_{current'}$*. The same is observed when the algorithm iterates over parameter* p_2*. Consequently, the algorithm outputs* $D = \{(p_3, v_3)\}$ *as the root cause, but that is a proper subset of the minimal definitive root cause* D_2*.*

In this case, we say that D is a **truncated assertion**, i.e., it is too short. Note, however, D will never be too long.

Theorem 2. *The* Shortcut *algorithm never asserts a superset of a minimal definitive root cause, provided the Disjointness Condition holds.*

Proof. By contradiction. We assume that $\exists(p, v) \in D$, such that (p, v) is not a necessary condition for an instance to `fail`. By the construction at the beginning of the shortcut algorithm, if $(p, v) \in D$, $CP_f[p] = v$ and $CP_g[p] \neq v$ by the Disjointness Condition.

When the *Shortcut* algorithm iterates over parameter p, we observe $CP_{\text{current}}[p] = CP_f[p]$ and $CP_{\text{current'}}[p] = CP_g[p]$. Hence, since (p, v) is not needed for an instance to `fail`, at this iteration, $E(CP_{\text{current'}}) = \texttt{fail}$, so (p, v) would be removed from current and therefore would never be asserted to be part of the root cause. Contradiction. □

To address the problem of truncated assertions, let us first observe another case when they do not arise, beyond the singleton case of Theorem 1.

Example 3. *Consider a slight modification of Example 2, where we add another parameter-value pair to D_2, defining the following scenario:*

- $D_1 = \{(p_1, v_1), (p_2, v_2)\}$
- $D_2 = \{(p_1, v_1'), (p_2, v_2''), (p_3, v_3)\}$
- $CP_f = \{(p_1, v_1), (p_2, v_2), (p_3, v_3)\}$
- $CP_g = \{(p_1, v_1'), (p_2, v_2'), (p_3, v_3')\}$

When iterating over parameter p_1, the Shortcut *algorithm does not update $CP_{current}[p_1] = v_1$, since $E(CP_{current'}) =$ `succeed` because $D_1 \nsubseteq CP_{current'}$ and $D_2 \nsubseteq CP_{current'}$. Similarly, the value of $CP_{current}[p_2]$ is not changed. Only $CP_{current}[p_3]$ is updated to v_3'. Thereafter, the algorithm would assert $D = \{(p_1, v_1), (p_2, v_2)\} = D_1$ as minimal definitive root cause, which is correct.*

In Example 3, both D_1 and D_2 contain values for p_1 and p_2 that are distinct from their counterpart in the other definitive root cause, i.e., $D_1[p_1] \neq D_2[p_1]$ and $D_1[p_2] \neq D_2[p_2]$. We say that D_1 and D_2 are *sufficiently different*. This characteristic directly influences when the *Shortcut* algorithm will yield truncated assertions and is formally defined as follows.

Definition 7 (Sufficiently different instances). *Two definitive root causes D_x and D_y are* **sufficiently different** *if (i) they share at least two properties and (ii) for all properties they have in common they differ in their values. Formally,*
(i) $|P_{D_x} \cap P_{D_y}| \geq 2$;
(ii) and $D_1[p] \neq D_2[p], \forall p \in P_{D_x} \cap P_{D_y}$.

Theorem 3. *If the Disjointness Condition holds and all minimal definitive root causes are pairwise sufficiently different, then the shortcut algorithm will never produce a truncated assertion.*

Proof. By contradiction. Suppose there are two sufficiently different minimal definitive root causes D_x and D_y, such that $D_x \subseteq CP_f$, CP_{current} is initialized to CP_f, and at some point the *Shortcut* algorithm creates an instance CP_{current} such that $D_y \subseteq CP_{\text{current}}$. We will show that this cannot happen.

Consider the first parameter $p \in P_{D_x}$, such that

$$CP_{\text{current'}}[p] = CP_g[p] \text{ and } E(CP_{\text{current'}}) = \texttt{fail}$$

Now, $D_x \nsubseteq CP_{\text{current'}}$ because D_x and D_y differ on at least two properties. In addition, $D_y \nsubseteq CP_{\text{current'}}$, since $CP_{\text{current'}}[p]$ is taken from P_{D_x}. Therefore, $E(CP_{\text{current'}}) = \texttt{succeed}$ because of the pairwise sufficient difference condition. Therefore, $CP_{\text{current}}[p]$ will not change its value. Thus, $D_y \subseteq CP_{\text{current}}$ will never occur. □

Stacked Shortcut Algorithm. Clearly, we cannot be sure *a priori* that all definitive root causes are single parameter-value pairs or that the minimal definitive root causes are sufficiently different, either of which would ensure that the *Shortcut* makes no truncated assertions. However, even if neither holds, we may be able to avoid truncated assertions by a specific reapplication of *Shortcut*.

To see how, we first observe that *Shortcut* makes truncated assertions only if all elements of a minimal root cause are contained in the union of CP_f and CP_g. This *union property* is formally described in Theorem 4.

Theorem 4. *The shortcut algorithm will yield a truncated assertion for a given CP_f and CP_g only if there is a minimal definitive root cause D, such that $D \subseteq CP_f \cup CP_g$ and $D \not\subset CP_f$.*

Proof. In the course of the *Shortcut* algorithm, all property values in CP_{current} come from CP_f or CP_g. By construction, the asserted root cause is the intersection of CP_f and CP_{current}. So if the asserted root cause is truncated, CP_{current} must have elements from CP_g that cause CP_{current} to evaluate to `fail`. Therefore there is a minimal definitive root cause in the union of CP_f and CP_g. □

Based on the previous theorems, we extended the shortcut algorithm to the *Stacked Shortcut* algorithm which basically runs a given failed configuration CP_f individually against multiple disjoint good configurations and then takes the union of the inferred root causes. Algorithm 2 shows the algorithm's pseudo-code. *Stacked Shortcut* is guaranteed to produce a correct solution if *BugDoc* can find k **mutually disjoint** successful instances, and there are at most k distinct minimal root causes.

Recall that two instances CP_1 and CP_2 are disjoint if they have different values for all properties. That is, $\forall p CP_1[p] \neq CP_2[p]$. A set of instances is mutually disjoint if every pair of instances are disjoint.

Theorem 5. *If all CP_i, such that $E(CP_i) =$ `succeed`, for $i \in \{1, 2, ..., k\}$, are mutually disjoint and disjoint from CP_f, and there are fewer than or equal to k distinct minimal definitive root causes, then the* Stacked Shortcut *Algorithm will never make a truncated assertion.*

Algorithm 2: *Stacked Shortcut* Algorithm

Input: CPI, the set of pipeline instances in the execution history characterized by their parameter-values
Input: E, the evaluation function
Input: P, list of parameters
Output: D, asserted minimal definitive root cause

```
/* Initialization                                  */
```
$D \leftarrow \emptyset$;
```
/* Find an instance that evaluates to fail
   */
```
Let CP_f be such that $CP_f \in CPI$, and $E(CP_f) =$ `fail`;
```
/* Find k successful instances disjoint
   with respect to CP_f and mutually
   disjoint if possible                            */
```
$CPG \leftarrow \{CP_1, CP_2, ..., CP_k\}$, such that CP_i, for $i \in \{1, 2, ...k\}$, are mutually disjoint and $E(CP_i) =$ `succeed`;
for $CP_g \in CPG$ **do**
  $D \leftarrow D \cup shortcut(CPI, E, P, CP_f, CP_g)$;
end
return D

Proof. By construction. For each other minimal definitive root cause $D \not\subseteq CP_f$, there can be at most one CP_i with the property that $D \subseteq CP_i \cap CP_f$, since all instances are disjoint. Because there are fewer than k distinct minimal definitive root causes by assumption, there exists at least one CP_i, which does not have the union property with respect to CP_f. So, by the construction of D, the *Stacked Shortcut* algorithm will yield an assertion (candidate root cause) that is not truncated. □

Note that even if all successful instances are not mutually disjoint (perhaps because some parameters have very few values), each additional call to *shortcut* (i.e., each call to *Shortcut* with a different disjoint good instance) reduces the likelihood of yielding a truncated assertion. The reason is that the second-to-last line of the *Stacked Shortcut* algorithm can only grow the hypothetical root causes.

Finally, note that both *Shortcut* and *Stacked Shortcut* are linear in the number of parameters, a very useful property when there are hundreds of parameters having at least two values each.

4.2 Finding Bugs with Inequalities: Debugging Decision Trees

While the *Shortcut* and *Stacked Shortcut* algorithms can find a single minimal definitive root cause very efficiently, usually without truncation (as we will see in the experimental section), characterizing all minimal definitive root causes is challenging. For this purpose, we use an algorithm that is exponential (in the number of parameters) in the worst case, but can characterize inequalities as well as equalities and does well heuristically even with a small budget [36].

The algorithm constructs a *debugging decision tree* using the parameters of the pipeline as features and the evaluation of the instances as the target. Thus a leaf is either purely `succeed`, if all pipeline instances so far tested that lead to that leaf evaluate to `succeed`; or `fail`, if all pipeline instances leading to that leaf evaluate to `fail`, or *mixed*. The algorithm works as follows:

(1) Given an initial set of instances CPI, 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 (*Parameter*,*Comparator*,*Value*), where the *Comparator* indicates whether a given *Parameter* has a value equal to, greater than (or equal to), less than (or equal to), or unequal to *Value*.
(2) If a conjunction involving a set of parameters, 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.
(3) Each suspect is used as a filter in a Cartesian product of the parameter values from which new experiments will be sampled.

Step 3 requires some explanation. Consider an example where all comparators denote equality. Suppose a path in the decision tree consists of $P_1 = v_1$, $P_2 = v_2$, and $P_3 = v_3$. To test that path, all other parameters will be varied. If every instance having the parameter-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 path consists of non-equality comparators (e.g., $P_1 = v_1$, $P_2 = v_2$, and $P_3 > 6$), then the algorithm chooses a satisfying value for each of those parameters as a prototype, (e.g., $P_3 = 7$) and choose pipeline instances having those values (e.g., all pipelines $P_1 = v_1$, $P_2 = v_2$, and $P_3 = 7$). Conversely, if any of the newly generated instances presents a (`succeed`) pipeline instance, the decision tree is rebuilt, taking into account the whole set of executed pipeline instances CPI, and a new suspect path is tried.

Note that *BugDoc* uses decision trees in an unusual way. We are not trying to predict whether an untested configuration will lead to `succeed` or `fail`, but simply use the tree to discover short paths, possibly characterized by inequalities, that lead to `fail`. Those will be our suspects. For that reason, we build a complete decision tree, i.e., with no pruning.

4.3 Parallelism

The most time-consuming aspect of debugging is the execution of pipeline instances. Fortunately, each pipeline instance

is independent. Hence different instances can be run in parallel. However, such an approach may lead to the execution of pipelines that are ultimately unnecessary (e.g., if one pipeline instance shows that $A.v$ is not a definitive root cause, then further tests on $A.v$ may not be useful). If the search space is large, this extra overhead turns out to be small, as we show in Section 5.2.

5 Experimental Evaluation

To evaluate the effectiveness of *BugDoc*, we compare it against state-of-the-art methods for deriving explanations as well as for hyperparameter optimization, using both real and synthetic pipelines. We examine different scenarios, including when a single minimal definitive root cause is sought and when a budget for the number of instances that can be run is set. We also evaluate the scalability of *BugDoc* when multiple cores are available to execute pipeline instances in parallel, and when the number of parameters and values increase.

Baselines. Because no previous approach both creates new instances and derives explanations, we compare our approach against combinations of state-of-the-art methods. We use Data X-Ray [46] and Explanation Tables [18] to derive explanations. To generate instances for all explanation algorithms, we use both the instances from *BugDoc* and Sequential Model-Based Algorithm Configuration (SMAC) [29].

SMAC is a method for hyperparameter optimization that is often more effective at searching configuration spaces than grid search [7]. 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 *BugDoc*. Therefore, for simplicity of presentation and to avoid cluttering the plots, we omit the random search results.

The explanation approaches analyze the provenance of the pipelines, i.e., the instances previously run and their results, but do not suggest new ones. By contrast, SMAC iteratively proposes new pipeline instances, but it always outputs a complete pipeline instance: the best it can find given a budget of instances to run and a criterion. This procedure makes sense for SMAC's primary use case, which is to find a set of parameter-values that performs well, but it is less helpful for debugging because it does not attempt to find a minimal root cause. For example, if a minimal definitive root cause of a pipeline is that parameter P_i must have a value of 5, SMAC will return a pipeline that fails, which has P_i set to 5. But since the pipeline may have many other parameter-values, the user has no way of knowing that $P_i = 5$ is the minimal definitive root cause and thus gains no insight into how to rectify the bug.

To give the explanation methods a reasonable chance to find minimal root causes, we combine the explanations with the generative techniques. We apply Data X-Ray and Explanation Tables to suggest root causes for the pipeline instances generated by SMAC, and also feed both methods with the instances created by *BugDoc*. Since SMAC looks for good instances, mostly for machine learning pipelines, we change its goal to look for bad pipeline instances.

Evaluation Criteria. We consider two goals: (i) *FindOne* – find at least one minimal definitive root cause in each pipeline; (ii) *FindAll* – find all minimal definitive root causes. The use case for *FindOne* is a debugging setting where it might be useful to work on one bug at a time, in the hope that resolving one may resolve or at least mitigate others. The use case for *FindAll* 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 use precision and recall to measure quality. These are defined differently for the *FindOne* case than for the *FindAll* case.

Formally, let UCP be a set of computational pipelines, where each pipeline $CP \in UCP$ (e.g., the pipeline of Figure 1) is associated with a set of minimal definitive root causes $R(CP)$. Given a set of root causes $A(CP)$ asserted by an algorithm A on pipeline CP for the *FindOne* case, we check if $A(CP)$ has at least one actual root cause. Precision is then the number of computational pipelines for which at least one minimal definitive root cause is found divided by the sum of the total number of pipelines where at least one minimal definitive root cause is found and the number of false positives (predicted root causes that are not, in fact, minimal definitive root causes). Formally, the *precision for FindOne* is:

$$\frac{\sum_{CP \in UCP} |A(CP) \cap R(CP) \neq \emptyset|}{\sum_{CP \in UCP} |A(CP) \cap R(CP) \neq \emptyset| + |A(CP) - R(CP)|}$$

where $A(CP) \cap R(CP) \neq \emptyset$ evaluates to 1 if $A(CP)$ corresponds to at least one of the conjuncts in $R(CP)$. Recall for *FindOne* is the fraction of the $|UCP|$ pipelines for which a minimal definitive root cause is found by A. The *recall for FindOne* is thus:

$$\frac{\sum_{CP \in UCP} |A(CP) \cap R(CP) \neq \emptyset|}{|UCP|}$$

For *FindAll*, precision is the fraction of root causes that A identifies that are, in fact, minimal definitive root causes. The *precision for FindAll* is defined as:

$$\frac{\sum_{CP \in UCP} |A(CP) \cap R(CP)|}{\sum_{CP \in UCP} |A(CP)|}$$

Recall for FindAll is the fraction of all the $R(CP)$ minimal definitive root causes, for all $CP \in UCP$, that are found by the algorithms:

$$\frac{\sum_{CP \in UCP} |A(CP) \cap R(CP)|}{\sum_{CP \in UCP} |R(CP)|}$$

For both *FindOne* and *FindAll*, we also report the F-measure, i.e., the harmonic mean of their respective measures of precision and recall.

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

Our first set of tests allows *BugDoc* to find at least one minimal definitive root cause using each of its algorithms (*Shortcut*, *Stacked Shortcut*, and *Debugging Decision Trees*). The experiment then grants the same number of instances to all other methods. Thus, the precision and recall for each algorithm is based on the same instance budget.

In these tests, Data X-Ray and Explanation Tables are given (i) the instances generated by *BugDoc* and, in a separate test, (ii) the instances generated by SMAC.

Pipeline Benchmark. We evaluate our approach using both synthetic and real pipelines. We have created synthetic data that reflect typical pipelines in data science and computational science, which often involve multiple components and associated parameters. The pipelines have between three and fifteen parameters, and each parameter has between five and thirty values. The parameter values are either ordinal (e.g., temperature) or categorical (e.g., color), each with probability 1/2. Each synthetic pipeline consists of a parameter space and a definitive root cause of failure automatically generated as follows:

(1) We uniformly sample a non-empty subset of parameters to be part of a conjunction.
(2) For each parameter in the subset, we uniformly sample from its values.
(3) For each parameter-value pair, we uniformly sample from the set of comparators $C = \{=, \leq, >, \neq\}$.
(4) After adding a conjunctive root cause, we add another conjunctive root cause with a certain probability.

The example below illustrates the parameter space and the definitive root cause for one of the synthetic pipelines.

Example 4. *A pipeline having three parameters with four possible values each could be characterized as follows:*

- *Parameter Space:* $p_1 \in [1.0, 2.0, 3.0, 4.0]$, $p_2 \in [1, 2, 3, 4]$, *and* $p_3 \in$ ["$p31$", "$p32$", "$p33$", "$p34$"].
- *Minimal definitive Root Cause :* $(p_1 = 4)$ *or* $(p_2 < 3.0$ *and* $p_3 \neq$ "$p34$").

We also evaluate the debugging strategies on real-world computational pipelines (see Section 5.3).

Implementation and Experimental Setup. The current prototype of *BugDoc* contains a dispatching component that runs in a single thread and spawns multiple pipeline instances in parallel. In our experiments, we used five execution engine workers to run the 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) [46] and Explanation Tables [18] (written in python 2.7). Since Data X-Ray does not generate new tests, we use the pipeline instances created by *BugDoc* as input to the feature model input of Data X-Ray. Separately, we converted the pipeline instances created by SMAC as input to the feature model of Data X-Ray. Similarly, we used the pipeline instances generated by both *BugDoc* and SMAC to populate the database schema required by Explanation Tables.

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 made our code and experiments available (https://github.com/ViDA-NYU/BugDoc).

5.1 Synthetic Pipelines

The results for the synthetically generated pipelines are reported according to the characteristics of their definitive root causes. The characteristics span three scenarios, consisting of multiple pipelines and covering different lengths of definitive root causes:

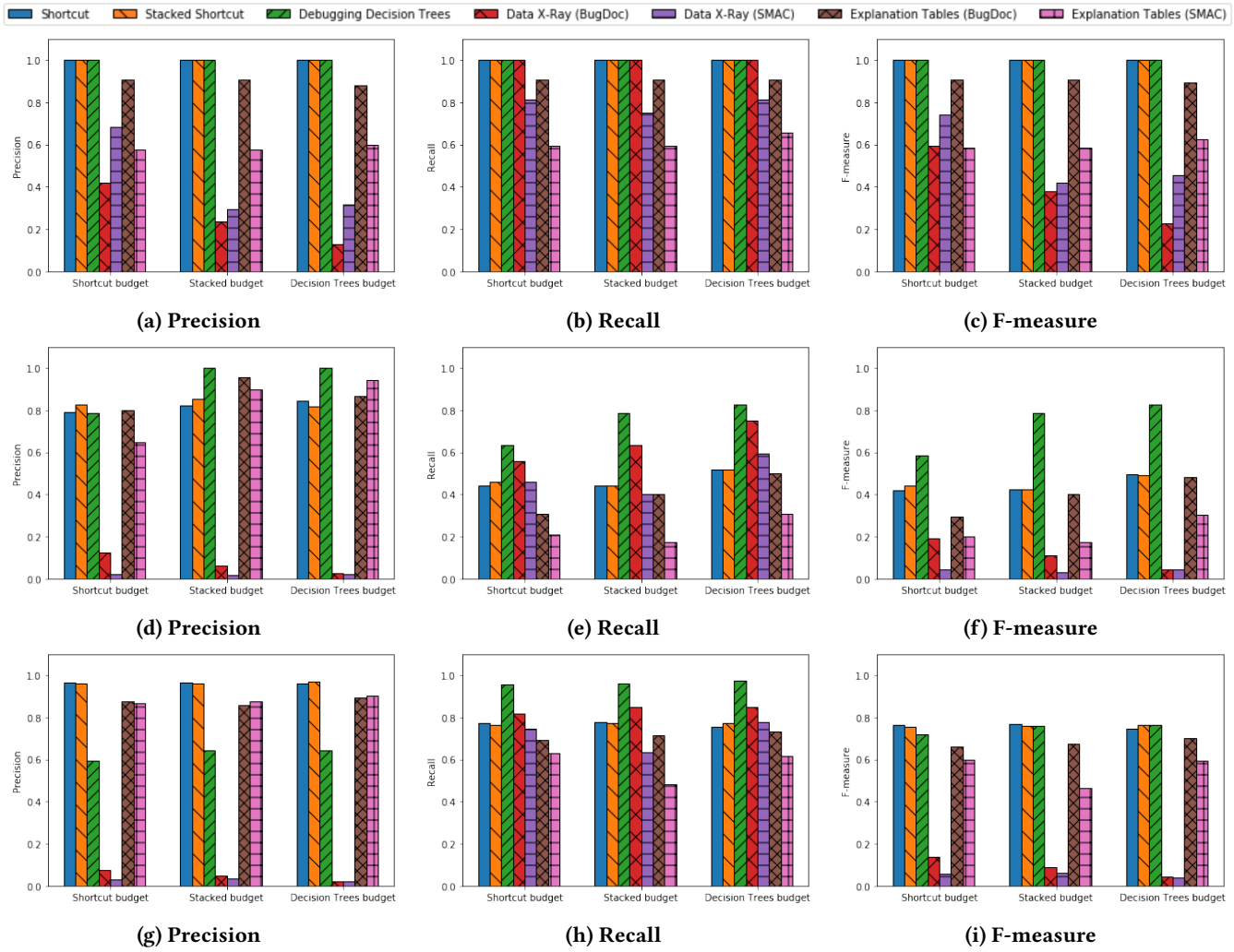
(1) a single parameter-comparator-value triple;
(2) a single conjunction of triples containing parameter-comparator-value; and
(3) a disjunction of conjunctions of parameter-comparator-value triples.

These scenarios are useful to assess the generality and expressiveness of the different approaches to explanation.

Precision, Recall, and F-measure. Figure 2 shows the precision, recall, and F-measure for the *FindOne* problem for the three types of definitive root causes. In the horizontal axis of each plot, we group all debugging methods by the maximum number of instances they were allowed to use to derive explanations, i.e., the number of new instances it took *Shortcut*, *Stacked Shortcut* with four shortcuts, and *Debugging Decision Trees* to solve the problem.

BugDoc's algorithms outperform Data X-Ray and Explanation Tables in all three scenarios, both when the baselines use instances generated by *BugDoc* and SMAC. If the definitive root cause is a single parameter-comparator-value (Figures 2a, 2b, and 2c), *Shortcut* and *Stacked Shortcut* achieve similar precision and recall to *Debugging Decision Trees*, which dominates the other scenarios.

Since we look for individual parameter-comparator-value triples with *Shortcut* and disjoint patterns in the data with decision trees, the likelihood that *Shortcut* does not find a definitive answer is higher in the scenario where a definitive root cause is a conjunction of factors, as can be seen in the relatively lower recall in Figure 2e. Conjunctions that are composed of equalities and inequalities have a high probability of presenting configurations with the union property. Hence the *Shortcut* and *Stacked Shortcut* algorithms generate



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

(d) Precision **(e) Recall** **(f) F-measure**

(g) Precision **(h) Recall** **(i) F-measure**

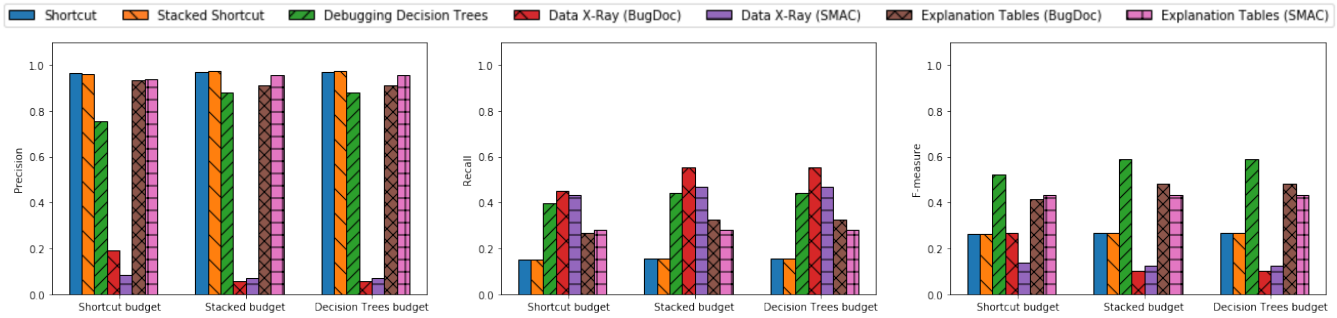
Figure 2: Synthetic Pipelines. Metrics for the *FindOne* problem when the root cause is a single parameter-value-comparator (top row, Figures 2a, 2b, and 2c), a single conjunction (middle row, Figures 2d, 2e, and 2f), or a disjunction of conjunctions (bottom row, Figures 2g, 2h, and 2i). In each figure, the leftmost group uses as many instances as does *Shortcut*, the middle uses as many as *Stacked Shortcut*, the rightmost as many as *Debugging Decision Trees*.

more truncated assertions, and their precision score is lower in Figure 2d as compared to Figures 2a and 2g. However, the shortcut algorithms still give better performance than the state-of-the-art algorithms.

Also note that in most cases, the state-of-the-art methods using instances generated by *BugDoc* outperform those methods using the SMAC instances. This suggests that our approach effectively proposes more useful test cases.

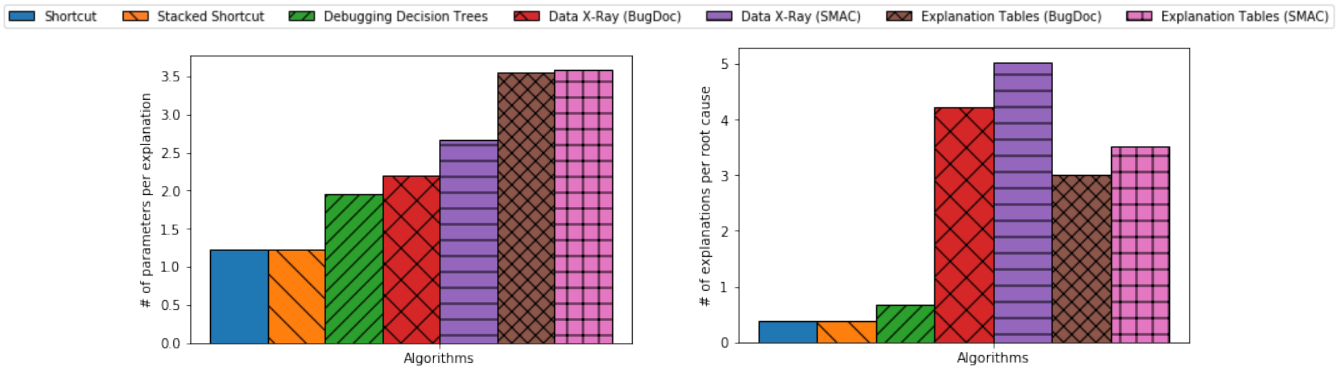
Similar relative results hold for the *FindAll* problem Figure 3 shows, although we observe the expected decrease in recall in Figure 3b, as a single root cause is no longer sufficient. The non-minimal approach of Data X-Ray pays off in this scenario with multiple reasons for a pipeline to `fail`. However, *Debugging Decision Trees* presents a better trade-off between precision and recall (Figure 3c).

Discussion. The answers provided by Explanation Tables represent a prediction of the pipeline instance evaluation result expressed as a real number, were 1.0 corresponds to a root cause. The precision of Explanation Tables is always high, but the recall is usually low. The converse happens with Data X-Ray, whose precision is low, but the recall is high. The reason for this is that Data X-Ray provides explanations that are not minimal definitive root causes. Further, neither Data X-Ray nor Explanation Tables support negation and inequality.



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

Figure 3: Synthetic Pipelines. Metrics for the *FindAll* problem when the root cause is a disjunction of conjunctions (Figures 3a, 3b, and 3c). In each sub-figure, the leftmost group uses as many instances as does *Shortcut*, the middle group as many *Stacked Shortcut*, the rightmost as many as *Debugging Decision Trees*.



(a) Parameters per asserted root cause

(b) Log of the number of asserted root causes per actual definitive root cause

Figure 4: Synthetic Pipelines. (a) Average number of parameters per asserted root causes for each algorithm and (b) average logarithmic number of asserted root causes per actual definitive root cause for each method.

Because both Data X-Ray and Explanation Tables achieved higher performance when using the instances generated by *BugDoc* than when using the instances generated by SMAC, we omit the SMAC configurations from the case studies with real-world pipelines presented later in this section.

The takeaway message from the experiments is that *BugDoc* dominates the other methods based on F-measure in every case, with *Debugging Decision Trees* dominating the shortcut methods unless the budget is small.

Conciseness of Explanation. Figure 4 shows that *BugDoc*'s algorithms not only provide explanations that are more concise in the number of parameters than Data X-Ray and Explanation Tables (Figure 4a) but also that it does not assert more root causes than there are (Figure 4b).

5.2 Scalability

The primary computational cost for all algorithms we consider is the cost of running the pipeline instances. Figure 5 shows the number of instances created by each of *BugDoc*'s algorithms as a function of the number of parameters of the

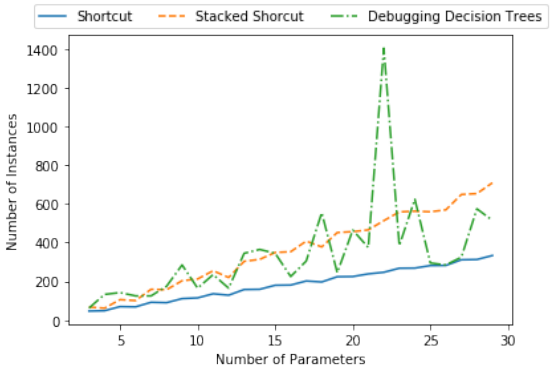


Figure 5: Instances required to execute each algorithm as a function of the number of parameters.

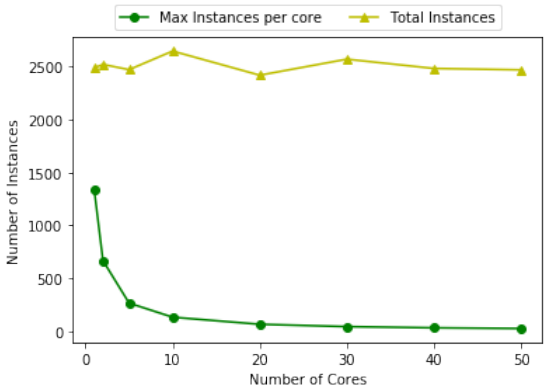


Figure 6: Scalability of *BugDoc* when running the *Debugging Decision Trees* algorithm on multiple cores.

pipeline. *Shortcut* and *Stacked Shortcut* increase linearly as expected. Because the time performance of *Debugging Decision Trees* has no simple relationship with root causes and could be exponential with the number of parameters, the user should choose *Shortcut* or *Stacked Shortcut* if there are many parameters and instances are expensive to run.

As noted above, the pipeline instances to test can be run in parallel, but at some risk to unnecessary computation. To evaluate scalability, we re-execute the experiment with synthetic data, described in Section 5.1, with different numbers of parallel computational cores and checked how many instances each core processed. As Figure 6 shows, the scale-up is essentially linear with the number of cores for the *Debugging Decision Trees* algorithm solving the *FindAll* problem. Thus given sufficient computing power, even *Debugging Decision Trees* can explore relatively large parameter spaces.

5.3 Real-World Pipelines

Data Polygamy Framework.

Data Polygamy aims to discover statistically significant relationships in a large number of spatio-temporal datasets [14]. We created a VisTrails [20] pipeline that reproduces an experiment designed by the Data Polygamy authors to evaluate the p-value and false discovery rate for their approach under different scenarios. Specifically, the pipeline evaluates different methods for determining statistical significance. The datasets used are synthetically generated, and their features are given as input parameters for the experiment. This process is a good use case for our approach because it has the following properties:

- The experiment requires a complex pipeline, including steps for data cleaning, data transformation, feature identification, multiple hypotheses testing, and other activities.
- The input data is heterogeneous – over 300 datasets at different spatio-temporal resolutions.
- The parameter space is large, consisting of 2 boolean, 3 categorical (3 to 10 possible values), and 7 numerical parameters. Each instance takes 20 minutes to run, making manual debugging impractical.

For this experiment, we selected different data types and steps of the computational pipeline. Each parameter can conceivably take on any value belonging to its type (e.g., Integer or Boolean). Given a set of pipeline instances, some of which crash and some of which execute to completion, we want to find at least one minimal set of parameter-values or combinations of parameter-values among those in the given pipeline instances, which cause the execution to crash.

GAN Training. Generative adversarial networks (GAN) [23] are widely applied to image generation and semi-supervised learning [41, 50]. Training these generative models involves an expensive computational process with several configuration parameters, such as the architecture choices and a high-dimensional hyperparameter space to tune. Sequence model-based approaches like Bayesian Optimization are prohibitively expensive in practice, since a single configuration could take more than a week to train. The most extensive study on the pathology of GAN training [10] entailed modifying baseline architectures and setting hyperparameters manually over three months, using hundreds of cores of a Google TPUv3 Pod [31]. Lucic et al. [37] evaluated seven different GAN architectures and their hyperparameter configurations, performing a random search in an experimental setting that would take approximately 6.85 years using a single NVIDIA P100.

We created a computational pipeline that trains a modified SAGAN [49] on CIFAR-10 [32] and applied *BugDoc* to find root causes of one of the most common problems of GAN training: *mode collapse* [11]. Our evaluation function sets a threshold on the Frechet Inception Distance (FID) [26] metric, which is a proxy for mode collapse. This pipeline specified only 6 parameters limited to 5 possible values. The bottleneck was the execution time because each configuration is trained in approximately 10 hours, depending on the discriminator and generator learning rates and the number of steps.

Transactional Database Performance. DBSherlock [48] is a tool designed to help database administrators diagnose online transaction processing (OLTP) performance problems. DBSherlock analyzes hundreds of statistics and configurations from OLTP logs and tries to identify which subsets of that data are potential root causes of the problems. In their experiments, the authors ran different settings of the TPC-C benchmark [44], introducing 10 distinct classes of performance anomalies varying the duration of the abnormal behavior. For each type of anomaly, they collected the workload logs, creating a dataset of logs, each labeled as normal or anomalous.

This dataset was used by Bailis et al. [4] to demonstrate Macrobase's ability to distinguish abnormal behavior in OLTP servers, where a classifier was trained to identify servers presenting degradation in performance.

We ran *BugDoc* on this data to identify the root causes of each class of performance anomaly. This experiment poses two additional challenges. The first challenge comes from the fact that, for this example, it is not possible to derive and run additional instances. We simulated the creation of new instances by reading only part of provenance and testing the algorithms on unread data, with an early stop when the pipeline instance to be tested was not present.

The second challenge was the number of properties – a total of 202 numerical statistics. We applied feature selection and aggregated the values in buckets in order to increase the probability of configurations that share parameter-value combinations. This reduced the configuration space to 15 parameters with 8 possible values (buckets) each. Since we were dealing with historical data, the instance execution time here is negligible.

We split the dataset into three parts: 50% of the data was used for training; 25% was the budget for pipeline instances that any sub-method of *BugDoc* requested; and we create a 25% holdout to assess the accuracy of *BugDoc*'s minimal root causes as a classifier to predict when a pipeline instance will fail. Precisely, if the pipeline instance is a superset of a minimal root cause, we predict failure. This method is accurate 98% of the time, results that are comparable to those reported in [4]. Thus, *BugDoc* achieves concise explanations of the bugs and high classification accuracy.

Quality Measures. The root causes identified for all aforementioned pipelines were manually investigated to assess their soundness and to create ground truth for the real-world data. The ground truth allowed us to compute precision and recall and to compare with Data X-Ray and Explanation Tables.

Empirical Results. The recall metric in Figure 7 shows that *BugDoc* methods found all the parameter-comparator-value triples that would cause the execution of the pipelines to fail. As in Section 5.1, Data X-Ray sometimes produces spurious root causes, yielding lower precision. By contrast, Explanation Tables shows high precision, but low recall.

6 Conclusion

To the best of our knowledge, *BugDoc* is the first method that autonomously finds minimal definitive root causes in computational pipelines or workflows. *BugDoc* achieves this by analyzing previously executed computational pipeline instances, selectively executing new pipeline instances, and finding minimal explanations.

When each root cause is due to a single parameter-value setting or a single conjunction of parameter-equality-values,

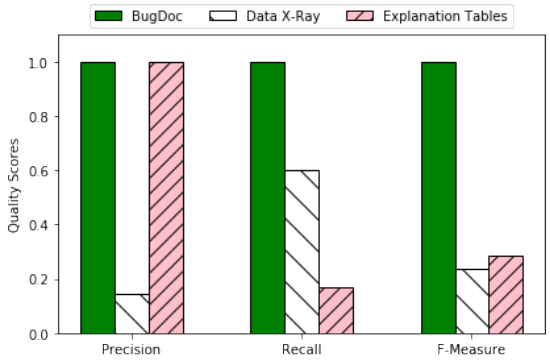


Figure 7: Real-World pipelines. *BugDoc* (using *Stacked Shortcut* and *Debugging Decision Trees* combined), outperforms Data X-Ray and Explanation tables.

the shortcut methods of *BugDoc* can provably guarantee to find at least one root cause in time proportional to the number of parameters (rather than exponential in the number of parameters as required by exhaustive search). Further, the shortcut approaches are guaranteed to find at least a subset of the parameter-values constituting a root cause in time linear in the number of parameters. When there are few parameters or sufficient computation time, the *Debugging Decision Trees* method of *BugDoc* performs best.

Compared to the state of the art, *BugDoc* makes no statistical assumptions (as do Bayesian optimization approaches like SMAC), but generally achieves better precision and recall given the same number of pipeline instances. In all cases, *BugDoc* dominates the other methods based on the F-measure, though it may sometimes lose based on precision or recall individually. *BugDoc* parallelizes well: pipeline instances can be executed in parallel, thus opening up the possibility of exploring large parameter spaces.

There are two main avenues we plan to pursue in future work. First, we would like to make *BugDoc* available on a wide variety of provenance systems that support pipeline execution to broaden its applicability. Second, we would like to explore group testing [33, 38] to identify problematic data elements when a dataset has been identified as a root cause. Another potential direction is the inclusion of observed variables (or predicates), properties that cannot be manipulated. While these cannot be used for deriving new instances, they can help enrich the explanations.

Acknowledgments. We thank Data X-Ray and Explanation Tables authors for sharing their code with us. We are also grateful to Fernando Chirigati, Neel Dey, and Peter Bailis for providing the real-world pipelines. This work has been supported in part by NSF grants MCB-1158273, IOS-1339362, and MCB-1412232, CNPq (Brazil) grant 209623/2014-4, the DARPA D3M program, and NYU WIRELESS.

References

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

[2] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of USENIX OSDI*. 307–320.

[3] Mona Attariyan and Jason Flinn. 2011. Automating Configuration Troubleshooting with ConfAid. *;login:* 1 (2011), 1–14.

[4] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *Proceedings of ACM SIGMOD*. 541–556.

[5] Anju Bala and Inderveer Chana. 2015. Intelligent Failure Prediction Models for Scientific Workflows. *Expert System Applications* 3 (Feb. 2015), 980–989.

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Proceedings of NIPS*. 2546–2554.

[7] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *JMLR* (Feb. 2012), 281–305.

[8] 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*. 115–123.

[9] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *Proceedings of VLDB Endowment* 3 (2017), 311–323.

[10] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2018. Large Scale GAN Training for High Fidelity Natural Image Synthesis. , 35 pages. arXiv:1809.11096

[11] Tong Che, Yanran Li, Athul Paul Jacob, Yoshua Bengio, and Wenjie Li. 2016. Mode Regularized Generative Adversarial Networks. , 13 pages. arXiv:1612.02136

[12] 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*. 1–7.

[13] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of IEEE DSN*. 595–604.

[14] 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.

[15] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *Proceeding of VLDB Endowment* 13 (Aug. 2013), 1498–1509.

[16] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building Auto-Tuners with Structured Bayesian Optimization. In *Proceedings of WWW*. 479–488.

[17] Nima Dolatnia, Alan Fern, and Xiaoli Fern. 2016. Bayesian Optimization with Resource Constraints and Production. In *Proceedings of ICAPS*. 115–123.

[18] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *Proceedings of VLDB Endowment* 1 (Sept. 2014), 61–72.

[19] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 2 (2013), 276–291.

[20] Juliana Freire, David Koop, Emanuele Santos, Carlos Scheidegger, Cláudio T. Silva, and H. T. Vo. 2011. The Architecture of Open Source Applications - Chapter 23. VisTrails. *Computer* (2011), 367–386.

[21] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. *CoRR* (2017), 1–13. arXiv:1709.03221

[22] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *Proceedings of NDSS*. 151–166.

[23] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. , 9 pages. arXiv:1406.2661

[24] 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.

[25] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing. In *Proceedings of ESEC/FSE*. 863–866.

[26] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. , 38 pages. arXiv:1706.08500

[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of USENIX Security Symposium*. 445–458.

[28] Jiangbo Huang. 2014. Programing implementation of the Quine-McCluskey method for minimization of Boolean expression. *CoRR* (2014), 1–22. arXiv:1410.1059

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

[30] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2018. Causal Testing: Finding Defects' Root Causes. *CoRR* (2018), 1–12. arXiv:1809.06991

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Computer Architecture News* 2 (June 2017), 1–12.

[32] Alex Krizhevsky et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[33] Kang Wook Lee, Ramtin Pedarsani, and Kannan Ramchandran. 2015. SAFFRON: A Fast, Efficient, and Robust Framework for Group Testing based on Sparse-Graph Codes. *IEEE Transactions on Signal Processing* (Aug. 2015), 1–10.

[34] David Lewis. 2013. *Counterfactuals*.

[35] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *In Proceedings of ACM SIGPLAN*. 15–26.

[36] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2019. Debugging Machine Learning Pipelines. In *Proceedings of DEEM*. Article 3, 10 pages.

[37] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. 2017. Are GANs Created Equal? A Large-Scale Study. , 21 pages. arXiv:1711.10337

[38] Anthony J. Macula and Leonard J. Popyack. 2004. A Group Testing Method for Finding Patterns in Data. *Discrete Applied Mathematics* 1-2 (Nov. 2004), 149–157.

[39] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *PVLDB* 13 (2014), 1715–1716.

[40] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference* (2nd ed.).

[41] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. , 16 pages. arXiv:1511.06434

[42] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of NIPS*. 2951–2959.

[43] 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.

[44] TPC. 2019. TPC-C benchmark. http://www.tpc.org/tpcc/. Accessed: 2020-02-10.

[45] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of ACM SIGMOD*. 1009–1024.

[46] 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.

[47] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. QFix: Diagnosing Errors Through Query Histories. In *Proceedings of ACM SIGMOD*. 1369–1384.

[48] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of ACM SIGMOD*. 1599–1614.

[49] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. 2018. Self-Attention Generative Adversarial Networks. , 10 pages. arXiv:1805.08318

[50] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. Metaxas. 2017. StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks. In *Proceeding of ICCV*. 5908–5916.

[51] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical Debugging: Simultaneous Identification of Multiple Bugs. In *Proceedings of ICML*. 1105–1112.