The VLDB Journal (2023) 32:75–101
https://doi.org/10.1007/s00778-022-00733-5

REGULAR PAPER

BugDoc

Iterative debugging and explanation of pipeline

Raoni Lourenço[1] · Juliana Freire[1] · Eric Simon[2] · Gabriel Weber[3] · Dennis Shasha[4]

Received: 17 May 2021 / Revised: 21 October 2021 / Accepted: 21 January 2022 / Published online: 23 February 2022
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

Abstract
Applications in domains ranging from large-scale simulations in astrophysics and biology to enterprise analytics rely on computational pipelines. A pipeline consists of modules and their associated parameters, data inputs, and outputs, which are orchestrated to produce a set of results. If some modules derive unexpected outputs, the pipeline can crash or lead to incorrect results. Debugging these pipelines is difficult since there are many potential sources of errors including: bugs in the code, input data, software updates, and improper parameter settings. We present *BugDoc*, a system that automatically infers the root causes and derive succinct explanations of failures for black-box pipelines. *BugDoc* does so by using provenance from previous runs of a given pipeline to derive hypotheses for the errors, and then iteratively runs new pipeline configurations to test these hypotheses. Besides identifying issues associated with computational modules in a pipeline, we also propose methods for: "opportunistic group testing" to identify portions of data inputs that might be responsible for failed executions (what we call), helping users narrow down the cause of failure; and "selective instrumentation" to determine nodes in pipelines that should be instrumented to improve efficiency and reduce the number of iterations to test. Through a case study of deployed workflows at a software company and an experimental evaluation using synthetic pipelines, we assess the effectiveness of *BugDoc* and show that it requires fewer iterations to derive root causes and/or achieves higher quality results than previous approaches.

1 Introduction

Computational pipelines are widely used in many domains, from science to enterprise analytics. These pipelines consist of modules—with associated parameters and data inputs—that are orchestrated to produce a set of results. In our data-driven world, such results often form the basis of conclusions that lead to 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 failures and obtain explanations for pipeline behavior.

Discovering the root causes of failures 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, forming cascading dependency chains. Consider the following real but sanitized examples.

Example 1 (Enterprise analytics) In an application deployed by a major software company, plots for sales forecasts derived by an analytics group 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 the incorrect forecasts reflected in the plots.

Amazon: Work done when author was at SAP.

✉ Raoni Lourenço
raoni@nyu.edu

Juliana Freire
juliana.freire@nyu.edu

Eric Simon
eric.simon@sap.com

Gabriel Weber
gabrwebe@amazon.com

Dennis Shasha
shasha@courant.nyu.edu

[1] Tandon School of Engineering, New York University, 370 Jay Street, Brooklyn, NY 11201, USA

[2] SAP, 35 rue d'Alsace, Levallois-Perret 92309, France

[3] Amazon, Av. Chedid Jafet, 200, São Paulo 04551065, Brazil

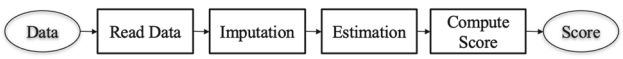
[4] Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1110, USA

Example 2 (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.

Example 3 (Designing a machine learning pipeline) Machine learning pipelines include a variety of components (e.g., training data, imputation methods for missing values, classification techniques). Failures as manifested by poor recall/precision scores may result from a poor choice of methods or hyperparameter values, as well as incorrectly labeled or unbalanced training examples.

To debug complex pipelines, during development or after deployment, 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 and test hypotheses manually, which is tedious, time-consuming, and error-prone.

The need for systematic iteration Figure 1 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, imputes missing values, creates and executes an estimator, and computes the F-measure score using 10-fold cross-validation. A data scientist uses this template during pipeline design to explore multiple estimators and imputation strategies for different types of input data, and ultimately, derive a pipeline instance that leads to high scores. This requires the creation and execution of multiple instances of the template that use different combinations of parameters values, training datasets, imputer strategies, and learning classifiers.



Instance	Dataset	Imputer Strategy	Estimator	Score	Evaluation
CP_1	Dataset 1	Mean	Logistic regression	0.9	Succeed
CP_2	Dataset 2	Frequency	Decision tree	0.8	Succeed
CP_3	Dataset 2	Mean	Gradient boosting	NAN	Fail
CP_4	Dataset 1	Frequency	Gradient boosting	0.2	Fail
CP_5	Dataset 3	Mean	Decision tree	0.7	Succeed
CP_6	Dataset 3	Frequency	Logistic regression	0.9	Succeed

Fig. 1 Machine learning pipeline and its execution provenance. A data scientist will in general experiment with various input datasets, imputation strategies, and classifier methods in a classification pipeline. After cross-validation, the pipeline outputs a quality score. An instance fails if the execution crashes (e.g., CP_3) or the score is low (e.g., CP_4)

By examining the provenance of the runs, we can observe that: *gradient boosting* leads to low scores for two of the datasets (*Dataset 1* and *Dataset 2*); *decision trees* work well for both the *Dataset 2* and *Dataset 3*; and *logistic regression* leads to a high score for *Dataset 1* and *Dataset 3*. This suggests a few hypotheses: there may be a problem with the *gradient boosting* module for the selected imputation methods; or it is not a good learner for the specific datasets; *decision trees* provide a suitable compromise for different data; and *logistic regression* achieves the best score. Because each run used different parameters for each method over the various datasets and strategies, a definitive conclusion has to await testing of additional combinations of parameters values.

Finding explanations of pipeline failure: challenges As the above examples illustrate, there are many potential causes for a pipeline to fail. Examining the provenance of a pipeline can help users derive hypotheses for the causes of the observed behavior and provide hints for debugging. Prior work used provenance to explain errors in computational processes that derive data [22,55]. 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. For example, to verify whether logistic regression is indeed the best approach, we would need to run it with additional combinations of datasets and imputer strategies. Doing so manually is time-consuming and error-prone, but automating this process is challenging.

Trying all possible combinations of parameter-values leads to a combinatorial explosion of instances to execute (exponential in the number of parameters) and therefore can be prohibitively expensive. Thus, a critical challenge lies in designing a provably efficient strategy for finding root causes in realistic computational pipelines.

Without loss of generality for any specific behavior, we focus this paper on fault-causing scenarios.

Algorithms for root cause identification To address these challenges, we designed *BugDoc*, an approach that makes use of provenance and algorithmically produced tests to (i) infer the root causes of failures and (ii) derive succinct explanations of failures in pipelines [42–44].

As noted above, a naive approach to this problem would explore all possible parameter-values, requiring a number of iterations that is exponential in the number of parameters. *BugDoc* makes uses of two algorithms that efficiently search the space of parameter values for a pipeline to identify root causes. The first one, called *Shortcut*, uses a minimal pair-style strategy to find root causes consisting of conjunctions of parameter-equality-value pairs (e.g., Estimator = *Logistic Regression* and Input Dataset Instance = *Dataset 1*) using a number of iterations that is linear in the number of parameters. The second one, *Debugging Decision Trees*, constructs an evolving decision tree whose leaves indicate success

or failure and whose nodes are tests on parameter values. *Debugging Decision Trees* can find root causes consisting of inequalities (e.g., of the form Param1 $> x$ and Param2 $\neq y$). While *Debugging Decision Trees* can require a number of iterations that is exponential in the number of parameters, in practice that rarely happens. Our algorithms derive succinct explanations involving only the parameter-comparator-value combinations that lead to failure, thus helping human debuggers focus on problem areas.

From algorithms to practice We carried out an evaluation in which we used *BugDoc* to identify the causes of problems that arose in workflows used in production by a large software company. should we also talk about deployment of bugdoc? or should we leave that for an industrial paper we can submit to sigmod? The evaluation confirmed the effectiveness of the algorithms we designed, and it also uncovered new, practical challenges in debugging pipelines and deploying the *BugDoc* system in an enterprise environment, as well as opportunities to further improve efficiency.

When *BugDoc* identifies a root cause consisting of a particular instance of a dataset, the immediate question is which portion of that dataset causes the problem. This is particularly challenging for pipelines that process large volumes of data—simply reporting that the problem lies in some instance of input data still requires users to identify the portions of the data that might that cause the behavior. We propose a new algorithm, called *Opportunistic Group Testing*, that heuristically determines the subset of a dataset instance that causes a pipeline to fail, thus making it easier for developers to identify and correct the problem.

We also observed that for many pipelines, we can prune the search process by selectively instrumenting one or more interior nodes in the pipeline with failure detectors. This instrumentation can make debugging more efficient, allowing the exploration to focus on the pipeline components that are directly connected with the behavior of interest. Intuitively, if the failure detector at an interior node $n1$ indicates no failure, but the final node $n2$ indicates a failure, then *BugDoc* can focus attention on the parameters that follow $n1$. The *Selective Instrumentation* heuristic analyzes the parameter dependencies induced by the pipeline topology and suggests nodes in the pipeline to instrument to reduce the search time. Finally, and perhaps most obviously, we extend *BugDoc* to exploit parallelism in order to reduce the search time.

Contributions and outline In this paper, we provide a comprehensive overview of our previous work [42–44], both the underlying methods and the open-source system that implements them. As new contributions, we describe the evaluation we carried out in collaboration with a major software company in which we used *BugDoc* to debug production pipelines. In addition, we introduce *Opportunistic Group Testing* for data debugging and *selective instrumentation* for search space reduction and discuss their design, implementation, and evaluation.

The remainder of this paper is organized as follows. We review related work in Sect. 2. Section 3 introduces the model we use for computational pipelines and formally defines the problem we address. In Sect. 4, we present algorithms to search for simple and complex causes of failures. We further isolate likely root causes to subsets of data by Opportunistic Group Testing in Sect. 5. Section 6 discusses Selective Instrumentation. Our methods are combined in a system specified in Sect. 7, and applied to enterprise data workflows in a case study presented in Sect. 8. We compare *BugDoc* with the state of the art in Sect. 9 and conclude in Sect. 10, 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 [5,16,22,47,55]. Some have focused on explaining where and how errors occur in the data generation process [55] and which data items are most likely to be causes of relational query outputs [47,56]. Others have attempted to use data to explain *salient* features in data (e.g., outliers) by discovering relationships among attribute values [5,16,22]. Dagger [50] takes a different approach and tries to adapt traditional software debugging strategies to investigate data—it provides white-box primitives for human debugging of data transformations. In contrast, *BugDoc* aims to diagnose abnormal behavior in computational pipelines that may be due to errors in data, programs, or sequencing of operations. When white-box approaches [50] are possible (requiring both source code and competent engineers), *BugDoc* could be used to identify the troublesome program code and then the white-box approach could be used to debug it.

Previous work on pipeline debugging has relied solely on execution histories to identify problematic parameter settings or inputs; they did not iteratively create and test new workflow instances to ascertain whether their explanations are accurate. Bala and Chana [6] 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. [14] developed a system that identifies problems by finding the differences between execution histories (encoded as trees) of good and bad runs. However, these differences do not necessarily identify root causes, though they often contain them. contain root

causes but may also include false positives—components that do not cause the problem.

Some systems have been developed to debug specific applications. Viska [28] 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. AID [23] uses causal inference and group testing to apply interventions on runtime conditions of programs to find root causes of software bugs. Its current implementation relies on a mechanism that extracts temporal predicates from .NET applications. Another counterfactual work is CADET [36] that aims to find root causes of non-functional faults (e.g. latency, energy dissipation) in highly configurable system architectures (hardware and software).

Although not designed for computational pipelines, Data X-Ray [55] 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 [22] 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 Sect. 9, *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 [9,10,20,51,52], 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 [8,10,20,51,52]. 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 [9] 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 [54] 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 [19] 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 [25,34], statistical debugging [40,60], and bug localization [3,4,29] 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 [40,60]. 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 [15]. 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., [24,26,31]). Exceptions are Causal Testing [34] and Delta Debugging [2]. 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. Delta Debugging also tests one root cause at a time,

looking for *minimal pairs* of conflicting programs configurations (one that fails and other that succeeds). In distinction, *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 [11,17], 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 will 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 will be inferred. No more minimal functional dependencies such as `A` $\rightarrow$ *Success Or Fail* will 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 cause(s) of failure(s) among the parameter values already tried, possibly by creating and executing new pipeline instances.

Definition 1 (PIPELINE, INSTANCE, PARAMETER- VALUE PAIRS, VALUE UNIVERSE) A **computational pipeline** (or workflow) CP is (i) a collection of programs each of which contains a set of manipulable parameters P (i.e., including hyperparameters, versions of programs, computational modules) and (ii) a further collection of data sources, each of which is considered a parameter whose values are instances of those data sources (e.g., in Fig. 1, *Dataset* is the parameter and *Dataset 1*, *Dataset 2* are values). Each parameter is associated with a node and is uniquely identifiable. Parameters corresponding to files can feed into different nodes in which case changing a file instance will change what is fed into all such nodes.

The programs and data sources are represented as nodes in a directed (possibly cyclic) graph where an edge (n, n') indicates that the output of node n is an input of n' (either because the output of n streams into n' or that n completes before n' begins). Parallel execution of a single instance is possible provided it has the same semantics as any serial execution consistent with the graph topology. In other words, parallelism should not interfere with determinism.

We denote by 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 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 Sect. 7, a user may expand the parameter-value universe U by explicitly defining the parameter domains. These domains can be discrete, e.g., a particular parameter can take integer values between 1 and 10. They can also be continuous, e.g., real numbers between 0 and 1000. However, even in the case of continuous parameter domains, *BugDoc* will analyze only the specific values already available in the history of correct and incorrect executions, i.e., the universe of values U. Untried values may also result in bugs, but *BugDoc* tries to find only the causes for bugs that have already arisen.

Note that *BugDoc* treats files as parameters and instances of files as values: the algorithms treat conventional parameters (e.g., options on imputation methods) and data parameters uniformly. As we discuss "Sect. 5", Opportunistic Group Testing performs data debugging (within a file) in a second phase.

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 some result(s) of a given pipeline instance.

Intuitively, a bug is some collection pipeline instances that 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. The goal of *BugDoc* is to find the root causes (and minimal root causes) of bugs.

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 $E(CP_i) = \texttt{fail}$; and (ii) if $E(CP_i) = \texttt{succeed}$, then the parameter-values pairs Pv_i of CP_i do not satisfy the conjunction C_f.

To illustrate the converse of point (ii), if a would-be $C_f = A > 5$ 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 (i.e., the history of pipeline instances in G), C_f leads to `fail`, but further evidence may refute that hypothesis.

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 U with the property that $E(CP_q) = \texttt{succeed}$ 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. For the sake of brevity, we sometimes drop the word "Definitive" and call D a minimal root cause.

The example in Fig. 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 *Dataset 3* and `Estimator` equal to *gradient boosting* is a hypothetical root cause of failure. To determine if this is a definitive root cause, we must check that no instance leading to `succeed` can be created with these parameter values. Section 4 presents algorithms that determine whether a root cause is definitive and minimal.

We note that the root causes defined here should not be interpreted as the *actual causes* of pipeline problems as characterized by causality theory [48]. 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 [39], i.e., the pipeline would not necessarily `succeed` had the root cause not been observed, because perhaps another root cause may come into play. *BugDoc* can discover disjunctive combinations of configurations that lead to failure.

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. In other words, we aim to achieve our debugging goals at a minimum cost in terms of number of pipeline instances tried.

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 two iterative pipeline 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-equality-value triples. The second, called *Debugging Decision Trees* and introduced in [42], discovers more complex definitive root causes involving inequalities (e.g., A takes a value between 5 and 13). When a single conjunction of parameter-values constitutes a definitive root cause, the *Shortcut* algorithm finds that root cause using fewer pipeline instances than *Debugging Decision Trees* or the state of the art.

4.1 Looking for simple 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.

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_current ← CP_f;
for p ∈ P do
    CP_current' ← CP_current;
    CP_current'[p] ← CP_g[p];
    if E(CP_current') = fail then
        CP_current ← CP_current';
    end
end
D ← CP_current ∩ CP_f;
/* check if proper subset from CPI                         */
for CP_i ∈ CPI do
    if D ⊆ CP_i and E(CP_i) = succeed then
        return ∅
    end
end
return D
```

Definition 6 (Disjoint instances) Two pipeline instances CP_x and CP_y are disjoint if $CP_x[p] \neq CP_y[p]$, for each $p \in P$ 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 intuition 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 (Sect. 9).

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

Example 4 Consider the machine learning pipeline in Fig. 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; `Imputer Strategy`, whether missing values will be imputed with the mean or the most frequent value of a column; and `Estimator`, the classification algorithm to be executed.

Table 1 shows the execution history (also called provenance) of the pipeline.

From the initial traces shown in Table 1, the *Shortcut* algorithm chooses two disjoint instances with different evaluations:

CP_g = {(Dataset, Dataset 3),
(Imputer Strategy, Frequency),
(Estimator, Logistic regression)}
CP_f ={(Dataset, Dataset 2),
(Imputer strategy, Mean),
(Estimator, Gradient boosting)}

Examining parameter `Dataset`, we replace its corresponding value in the current instance to be executed from Dataset 2 to Dataset 3. Because the execution evaluates to `succeed`, suggesting that the replaced parameter-value was important for `fail`, we cannot keep this replacement in the current instance, so we roll back to Dataset 2. Similarly, when we update the value of parameter `Imputer Strategy` to Frequency, the instance evaluation also `succeed`, so we discard that replacement as well.

However, when `Estimator` is changed to Logistic Regression, the resulting configuration still evaluates to `fail`. This suggests that `Estimator` is likely not a cause of the bug. The algorithm then yields the following root-cause: (Dataset,Dataset 2) **and** (Imputer Strategy,Mean).

Table 2 displays all pipeline instances evaluated, including the new instances generated by the *Shortcut* algorithm.

For Pipelines with singleton parameter-value root causes as in Example 4, the algorithm will find a minimal definitive root cause. It will always work for singleton root-causes as well. This is a common base case for software that has been working but has recently been modified.

Table 1 An initial (given) set of classification pipelines instances that return `succeed` if *score* ≥ 0.6 and `fail` otherwise

Dataset	Imputer strategy	Estimator	Score	Evaluation
Dataset 1	Mean	Logistic regression	0.9	`succeed`
Dataset 2	Frequency	Decision tree	0.8	`succeed`
Dataset 2	Mean	Gradient boosting	NAN	`fail`
Dataset 1	Frequency	Gradient boosting	0.2	`fail`
Dataset 3	Mean	Decision tree	0.7	`succeed`
Dataset 3	Frequency	Logistic regression	0.9	`succeed`

Table 2 Set of classification pipelines instances including the new instances created by *Shortcut* by substituting values one at a time of parameters in CP_f (Dataset 1, Frequency, Gradient Boosting) by corresponding values in CP_g (Dataset 3, Mean, Decision Tree)

Dataset	Imputer strategy	Estimator	Score	Evaluation	Origin
Dataset 1	Mean	Logistic regression	0.9	`succeed`	Provenance
Dataset 2	Frequency	Decision tree	0.8	`succeed`	Provenance
Dataset 2	Mean	Gradient boosting	NAN	`fail`	Provenance
Dataset 1	Frequency	Gradient boosting	0.2	`fail`	Provenance
Dataset 3	Mean	Decision tree	0.7	`succeed`	Provenance
Dataset 3	Frequency	Logistic regression	0.9	`succeed`	Provenance
Dataset 3	Mean	Gradient boosting	0.8	`succeed`	Replacing dataset 2 by dataset 3
Dataset 2	Frequency	Gradient boosting	0.9	`succeed`	Replacing mean by frequency
Dataset 2	*Mean*	*Logistic regression*	*NAN*	`fail`	*Replacing gradient boosting by logistic regression*

The rightmost column (Origin) indicates if the instance was in the provenance or it was created by any step of our algorithms

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 $E(CP_g) = \texttt{fail}$. 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 root cause would remain. □

Guarantees of the shortcut algorithm For bugs that may depend on disjunctions of conjunctions, 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.

Example 5 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 $CP_{\text{current}}[p_1] = v_1'$. But $E(CP_{\text{current}'}) = \texttt{fail}$ because $D_2 \subseteq CP_{\text{current}'}$. 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. Truncated assertions are still heuristically useful, because they uncover reasons for bugs, but may miss part of the reason for a bug. So we want to characterize when *Shortcut* behaves properly.

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 construction of D in the shortcut algorithm, if $(p, v) \in D$ then $CP_f[p] = v$ 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 6 Consider a slight modification of Example 5, 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 $CP_{\text{current}}[p_1]$ with v_1' since with that value $E(CP_{\text{current}'}) = \texttt{succeed}$, because $D_1 \nsubseteq CP_{\text{current}'}$ and $D_2 \nsubseteq CP_{\text{current}'}$. Similarly, the value of $CP_{\text{current}}[p_2]$ is not changed. Only $CP_{\text{current}}[p_3]$ 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 6, 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. By definition, there exists a minimal root cause D_x such that $D_x \subseteq CP_f$. Let's assume that the shortcut algorithm returns a root cause D that is a proper subset of D_x (i.e., a truncated assertion). Then, let p be the first parameter in $P_{D_x} - P_D$, such that $CP_{\text{current}'}[p] = CP_g[p]$ and $E(CP_{\text{current}'}) = \texttt{fail}$. Such a parameter must exist based on the shortcut procedure. By the Disjointness Condition, $CP_{\text{current}'}[p] \neq D_x[p]$. Thus, there exists another minimal root cause $D_y \neq D_x$ such that $D_y \subseteq CP_{\text{current}'}$. However, since D_y and D_x are sufficiently different (by assumption), by point (i) of Definition 7, there exists a parameter $p' \neq p$ such that $p' \in P_{D_x} \cap P_{D_y}$ and $D_y[p'] = CP_{\text{current}'}[p]$. But because p is the first parameter tested by shortcut by assumption, the value of p' has not been changed before p in shortcut. Thus, we also have $CP_{\text{current}'}[p] = D_x[p]$, which contradicts point (ii) of Definition 7. Hence, D cannot be a proper subset of D_x. □

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, for each p, $CP_1[p] \neq CP_2[p]$. A set of instances is mutually disjoint if every pair of instances are disjoint.

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

Theorem 5 *Suppose a set of iteration instances CP_i, for $i \in \{1, 2, ..., k\}$, have the properties that (i) $E(CP_i) = \texttt{succeed}$ (ii) is disjoint from CP_f, and (iii) for any i, j such that $i \neq j$, CP_i and CP_j are disjoint. If there are fewer*

than or equal to k distinct minimal definitive root causes, then the Stacked Shortcut *Algorithm will never make a truncated assertion.*

Proof By construction. For each other minimal definitive root cause $D \nsubseteq CP_f$, 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.

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, called *debugging decision trees*, that can characterize inequalities as well as equalities, thus potentially capturing an unbounded number of minimal definitive root causes. In the worst case, debugging decision trees may require a number of instances that is exponential in the number of parameters, but in practice they locate bugs even with a small budget [42].

A *debugging decision tree* uses 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**, if there are some failing and some succeeding instances. The algorithm (detailed in Algorithm 3) works as follows:

1. Given an initial set of instances CPI, construct a decision tree based on the evaluation results (`succeed` or `fail`) for those instances. 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, unequal to, less than, or greater than or equal to *Value*. See Fig. 2 for an example based on the initial history.
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. For example, consider the failing leaf

Algorithm 3: *Debugging Decision Trees* pseudo-code. Intuitively, find each pure failing path π and test instances that satisfy the conditions of π. If an instance consistent with π `succeed` then rebuild the tree. Otherwise assert that π is a minimal root cause.

```
   Input: CPI, the set of pipeline instances
   Input: E, the evaluation function
   Input: P, list of parameters
   Output: D, asserted minimal definitive root cause
 1 D ← ∅;
 2 rebuild = true;
 3 while rebuild do
 4 |   rebuild = false;
 5 |   T ← decision_tree(CPI);
 6 |   Let Π be the set of paths in T that fail; /* Each node
   |      in a path is triple (p,c,v) triples,
   |      for Parameter p, Comparator c, Value
   |      v                                              */
 7 |   for π ∈ Π do
 8 |   |   Let each V_p be the set of all values for p in CPI, ∀p ∈ P;
 9 |   |   for (p, c, v) ∈ π do
10 |   |   |   case c is = do
11 |   |   |   |   V_p ← {v};
12 |   |   |   end
13 |   |   |   case c is ≠ do
14 |   |   |   |   V_p ← V_p − {v};
15 |   |   |   end
16 |   |   |   case c is < do
17 |   |   |   |   V_p ← {x ∈ V_p | x < v};
18 |   |   |   end
19 |   |   |   case c is ≥ do
20 |   |   |   |   V_p ← {x ∈ V_p | x ≥ v};
21 |   |   |   end
22 |   |   end
   |   |   /* Try instances that satisfy π              */
23 |   |   for CP_i ∈ ∏_{p∈P} V_p do
24 |   |   |   if E(CP_i) = succeed then
25 |   |   |   |   rebuild = true;
   |   |   |   |   /* π is not a failing path          */
26 |   |   |   end
27 |   |   |   CPI ← CPI ∪ {CP_i};
28 |   |   |   if not rebuild then
29 |   |   |   |   D ← D ∪ π;
   |   |   |   |   /* π is a minimal root cause        */
30 |   |   |   end
31 |   |   end
32 |   end
33 end
34 return D
```

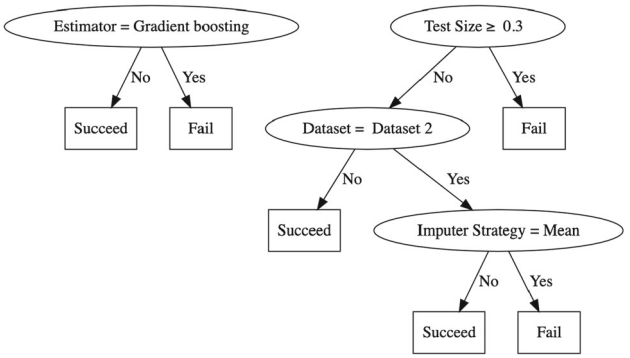


Fig. 2 The decision tree on the left is fitted on the instance history of Example 4 presented in Table 3. The decision tree on the right includes the new instances created by the *Debugging Decision Trees* algorithm in Table 4

corresponding to the conjunction `Test Size` < 0.3 and `Dataset` = Dataset 2 and `Imputer Strategy` = Mean. We would want to test different values of the Estimator to see whether that path is always leading to failure.

Generalizing from the example, step 3 works as follows. Suppose a failing path consists of general comparators (e.g., $P_1 = v_1$, $P_2 = v_2$, and $P_3 > 6$), then the algorithm chooses satisfying values for each of those parameters as a prototype, and chooses pipeline instances having those values (e.g., pipelines in which $P_1 = v_1$, $P_2 = v_2$, and $P_3 \in \{7, 8, 9, ...\}$) by varying values of other parameters, e.g. P_4, P_5, ... Note that new instances can be created by a cartesian product of the values seen so far. The pseudo-code shows this in Line 23 of Algorithm 3. Heuristically, *BugDoc* uses combinatorial design [18]. Alternatively, one could apply interpolation for continuous parameters and define parameter domains for non-bounded inequalities.

If any of the newly generated instances results in `succeed` then the suspected path does not always lead to `fail`. Reflecting that, *BugDoc* rebuilds the decision tree taking into account the whole set of executed pipeline instances CPI, which now includes the new instances.

When the values associated with a parameter are continuous, *BugDoc* 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 [7].

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 paths, possibly including inequalities, that lead to `fail`. Those will be our suspects. For that reason, we build a complete decision tree, i.e., with no pruning.

Below we revisit Example 4 and the initial set of pipeline execution in Table 1 to illustrate how the *Debugging Decision Trees* algorithm works. Additionally, we introduce a new numerical parameter, the size of the test data, to compute the pipeline score, as depicted by Table 3.

A decision tree is created from the instances shown in Table 3, containing a single node, as shown in the left decision tree of Fig. 2, with the parameter-value only seen in failing instances: (*Estimator*,*Gradient Boosting*). After assembling new configurations with this parameter-value, the *Debugging Decision Trees* algorithm observes that some of the generated pipeline instances `succeed`, as depicted in Table 4. Hence, the hypothesis that `Estimator` with value "Gradient Boosting" causes failure is invalid. So, the decision tree is rebuilt, fitted on the instances of Table 4. After rebuilding, the algorithm finds a new tree with two paths leading to `fail`, as shown in the right decision tree of Fig. 2. This presents two new hypothetical root causes:

(i) (*Test Size,* $\geq$ *,0.3*).
(ii) (*Test Size,* $<$ *,0.3*) **and** *(Dataset,=,Dataset 2)* **and** *(Imputer Strategy,=,Mean)*.

Debugging Decision Trees creates new pipeline instances based on the root cause candidates, which all `fail` as can be seen in Table 5. This confirms the candidate hypotheses, which are output as definitive root causes.

Table 3 Set of classification pipelines instances from Example 4 with an additional numerical parameter, `Test Size`

Dataset	Imputer strategy	Estimator	Test Size	Score	Evaluation	Origin
Dataset 1	Mean	Logistic regression	0.1	0.9	`succeed`	Provenance
Dataset 2	Frequency	Decision tree	0.1	0.8	`succeed`	Provenance
Dataset 2	Mean	Gradient boosting	0.4	NAN	`fail`	Provenance
Dataset 1	Frequency	Gradient boosting	0.3	0.2	`fail`	Provenance
Dataset 3	Mean	Decision tree	0.2	0.7	`succeed`	Provenance
Dataset 3	Frequency	Logistic regression	0.2	0.9	`succeed`	Provenance

Table 4 Set of classification pipelines instances including the new instances created by *Debugging Decision Trees* based on parameter-value (*Estimator*, =, *Gradient Boosting*)

Dataset	Imputer strategy	Estimator	Test Size	Score	Evaluation	Origin
Dataset 1	Mean	Logistic regression	0.1	0.9	`succeed`	Provenance
Dataset 2	Frequency	Decision tree	0.1	0.8	`succeed`	Provenance
Dataset 2	Mean	Gradient boosting	0.4	NAN	`fail`	Provenance
Dataset 1	Frequency	Gradient boosting	0.3	0.2	`fail`	Provenance
Dataset 3	Mean	Decision tree	0.2	0.7	`succeed`	Provenance
Dataset 3	Frequency	Logistic regression	0.2	0.9	`succeed`	Provenance
Dataset 3	Mean	Gradient boosting	0.2	0.9	`succeed`	Testing gradient boosting
Dataset 3	Mean	Gradient boosting	0.1	0.8	`succeed`	Testing gradient boosting
Dataset 1	Frequency	Gradient boosting	0.2	0.9	`succeed`	Testing gradient boosting
Dataset 1	Frequency	Gradient boosting	0.1	0.8	`succeed`	Testing gradient boosting
Dataset 2	Mean	Gradient boosting	0.2	NAN	`fail`	Testing gradient boosting
Dataset 2	Mean	Gradient boosting	0.1	NAN	`fail`	Testing gradient boosting
Dataset 3	Frequency	Gradient boosting	0.4	0.5	`fail`	Testing gradient boosting
Dataset 3	Mean	Gradient boosting	0.3	0.4	`fail`	Testing gradient boosting
Dataset 3	Mean	Gradient boosting	0.4	0.4	`fail`	Testing gradient boosting
Dataset 1	Frequency	Gradient boosting	0.4	0.5	`fail`	Testing gradient boosting
Dataset 1	Mean	Gradient boosting	0.3	0.3	`fail`	Testing gradient boosting
Dataset 2	Frequency	Gradient boosting	0.4	0.5	`fail`	Testing gradient boosting
Dataset 2	Mean	Gradient boosting	0.3	NAN	`fail`	Testing gradient boosting
Dataset 2	Frequency	Gradient boosting	0.3	0.3	`fail`	Testing gradient boosting

Table 5 Set of new classification pipelines instances created by *Debugging Decision Trees* based on paths [*Dataset, = ,Dataset 2)*, *(Imputer Strategy, = ,Mean)*] , and [*(Test Size, ≥, 0.3)*]

Dataset	Imputer Strategy	Estimator	Test Size	Score	Evaluation	Origin
Dataset 3	Frequency	Decision Tree	0.3	0.3	`fail`	Testing `Test Size` ≥ 0.3
Dataset 1	Frequency	Decision Tree	0.3	0.4	`fail`	Testing `Test Size` ≥ 0.3
Dataset 3	Mean	Logistic Regression	0.4	0.5	`fail`	Testing `Test Size` ≥ 0.3
Dataset 1	Mean	Logistic Regression	0.3	0.4	`fail`	Testing `Test Size` ≥ 0.3
Dataset 2	Frequency	Gradient Boosting	0.3	0.4	`fail`	Testing `Test Size` ≥ 0.3
Dataset 2	Mean	Decision Tree	0.4	NAN	`fail`	Testing `Test Size` ≥ 0.3
Dataset 2	Mean	Decision Tree	0.1	NAN	`fail`	Testing Dataset 2 and Mean
Dataset 2	Mean	Decision Tree	0.2	NAN	`fail`	Testing Dataset 2 and Mean
Dataset 2	Mean	Logistic Regression	0.1	NAN	`fail`	Testing Dataset 2 and Mean
Dataset 2	Mean	Logistic Regression	0.2	NAN	`fail`	Testing Dataset 2 and Mean

5 Finding causal data errors: opportunistic group testing

The algorithms in Sect. 4 identify computational steps that cause a particular behavior, and thus make it easier for users to find potential bug. When a definitive root cause contains a parameter-value that corresponds to a tabular dataset, we would also like to identify the part of the dataset that may be responsible for the failure.

In this section, we introduce a new heuristic approach to this problem. The idea behind our heuristic is to look for values that are very different in the failed dataset instance (a "value" of this dataset parameter) from the succeeding dataset values.

Definition 8 (Dataset Parameter) A parameter P_d of a computational pipeline is a dataset parameter if it semantically represents a data table. Any value (an instance of that table) of P_d must have the same schema and must have a key field or fields.

Definition 9 (Key Score) Let B be a value (an instance of a table) of a dataset parameter P_d such that (P_d, B) is part of a minimal definitive root cause of failure. Let $G_1, G_2, ...$ and G_n be values of P_d that are not in any root cause. That is, (P_d, G_i) corresponds to values of data table P_d in successful pipeline instances and (P_d, B) represents a data table in a failing pipeline instance.

Let $(P_d, B)(k).c$ be the value of a field in column c in table (P_d, B) corresponding to a row identified by key k; similarly for $(P_d, G_i)(k).c$ for $i \in \{1, 2, .., n\}$. We then measure a *deviation score* per key value. Large deviation scores correspond to extreme value differences between the good and bad files, e.g., in the good file a person has an age of 53 and in the bad file that person has an age of 353.

Different methods can be used to derive a deviation score. As a concrete example, the deviation score we use for numerical columns in the case study we discuss in Sect. 8 is based on contrasting the value of the bad file $P_d.B$ at some key value k and column c (the $(P_d, B)k.c$ value for short) with all the good files. Specifically, the method compares the bad file's $(P_d, B)k.c$ value with the minimum and maximum of the $(k.c)$ values of the good files.

For example, if the values for $(k.c)$ in the good tables range from 40 to 60 and the bad table's $(P_d, B)k.c$ value is 75, then the deviation score will be 15 (15 above 60). If the $(P_d, B)k.c$ value is 50, then the score will be 0 (because it's in the range of 40 to 60). If the $(P_d, B)k.c$ value is 23, then the score will be 17 (because 23 is 17 below 40).[1]

Thus, for our case study, the deviation score of the $(P_d, B)k.c$ value is how much greater than the maximum of the good table values or less than the minimum of the good table values that value deviates from the values of the c columns for key k in the good tables. This simple idea is expressed in the following expression. Please check it against the example of the previous paragraph.

$$
\begin{aligned}
d = (\max(& \\
& 0, \\
& ((P_d, B)(k).c - \max_{i \in \{1,2,..,n\}} ((P_d, G_i)(k).c)), \\
& ((\min_{i \in \{1,2,..,n\}} ((P_d, G_i)(k).c)) - (P_d, B)(k).c))) \\
)
\end{aligned}
$$

For the purpose of the measure of our case study, null values are given extreme negative values. That way, if there are nulls in both the good and bad key k column c field, there the $(k.c)$ value will yield a small deviation score. If only the bad file's $(k.c)$ value is null, there will be a large deviation score. We apply this also to categorical columns, so a null in just the bad file will look to be an extremely deviant value.

Let S be a list of keys in the bad file instance (P_d, B) in descending order of deviation scores for a given column c. To find the values of $(P_d, B).c$ that might have caused the failure, the Opportunistic Group Testing protocol uses binary search. It first replaces all of them by $(P_d, G_i).c$, for some i, maintaining the key association with each c value. If that results in a successful execution, then the protocol replaces only the c values associated with the first half of S (those with the highest deviation scores). If the pipeline instance again succeeds, then the protocol takes the top quarter of S and so on. The goal is to find a relatively small set of keys in (P_d, B) associated with the highest deviation scores for column c whose replacement would create a successful pipeline instance. Whatever produced those bad values might be the reason for the observed failure.

Deviant values We call the key-values associated with failing instances, *deviant values*. The failing data tables in the root causes found in Sect. 4.2 contain deviant values in certain columns that *Opportunistic Group Testing* has identified. *Dataset 1* has missing or null values in a numeric column with continuous values. By contrast, *Dataset 2* has missing values in a categorical column.

We discuss the application of *Opportunistic Group Testing* for our case study in Sect. 8.3.1.

6 Leveraging pipeline topology: selective instrumentation

In Sect. 3, we introduced the notion of an evaluation function that determines whether a pipeline instance "succeeds" or "fails". In a practical implementation, which we discuss in Sect. 8, this evaluation function is executed after each pipeline instance is executed, as a side effect.

Evaluation functions can also be added to different stages of a pipelines to reduce the search space of pipeline instances to consider when debugging. In this section, we propose *selective instrumentation*, a new strategy to automatically recommend nodes in a pipeline graph to be instrumented, and show how this instrumentation improves performance by reducing the number of parameter-value combinations that need to be explored.

Recall that a computational pipeline consists of a directed graph $G = (N, E)$ of nodes and edges, with a set of end nodes, denoted END. Each node may be associated with one or more parameters, each of which can have many possible values. The nodes in paths leading to a node n, denoted nodes_to(n), are the set of nodes in G starting from any root

[1] Though we don't use it in the case study, a related score is relative deviation, e.g., expressed as the score divided by the mean of the good $k.c$ values.

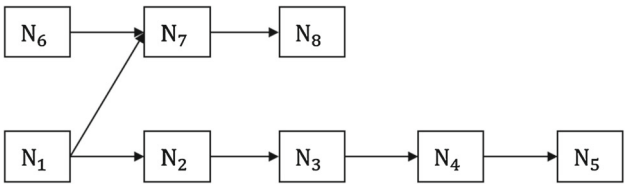


Fig. 3 Example of data pipeline. The END nodes are N_8 and N_5

of G that belong to a path that ends at n.

$$nodes\_to(n) = \{m | m \in N \text{ and } m \in leads\_to(n)\}$$

Let the parameters associated with a node m be denoted $params(m)$. The parameters that could influence the output of n are:

$$params\_to(n) = \{params(m) | m \in nodes\_to(n)\}$$

Some of the parameters in $params\_to(n)$ may influence END independently of n, i.e., there may be paths to one or more nodes of END that do not go through n. For that reason, we form a new set called the *exclusive parameters to n*. These are the parameters p in nodes m for which every path from m to any node in END goes through n: $exclusive\_to(n) = \{p | p \in params\_to(n)$ and for each node m such that $p \in params(m)$, every path from m to any node in END goes through $n\}$

Suppose an evaluation function $eval(n)$ is inserted at node n. We want $eval(n)$ to have the following *isolation* property: if a pipeline instance fails at END but $eval(n)$ returns success, then any root cause of the failure at END cannot be due to any parameters in $exclusive\_to(n)$. This will reduce the search space for debugging.

Example 7 Consider the pipeline example in Fig. 3 and suppose that the evaluation function is inserted at the output of node N_4. Then the isolation property suggests that any failure of that function cannot be caused by parameters exclusive to N_6, N_7, N_8 and N_5, because values in those parameters cannot affect the output of N_4.

To help choose the best single node to instrument, we define the Cartesian or cross-product of a sequence of parameters $S = (p_1, ..., p_{||S||})$ as:

$$cross(S) = values(p_1)X...Xvalues(p_{||S||})$$

where $values(p)$ for some parameter p are the values for p in the pipeline instances tried so far.

Take any ordering of $exclusive\_to(n)$ and call that $seq(exclusive\_to(n))$. An *instrumentation* at n would be a test on the output of n with the isolation property. Now consider the nodes where selective instrumentation is possible and call those $POSS$. Two heuristic considerations can help decide which node in $POSS$ to choose:

- Cover troublesome parameters: If some failures have already been discovered and they involve some parameter p, it may be of benefit to choose a node n such that $p \in exclusive\_to(n)$.
- Search space reduction: If no failures have been discovered, we want to do a selective instrumentation at n in $POSS$ if the size of the set of parameter combinations can be minimized. That is, it would be good to minimize

$$||cross(seq(exclusive\_to(n)))|| + ||cross(seq(ALLPARAMS - exclusive\_to(n)))||$$

where $ALLPARAMS$ are all the parameters of the workflow. Thus, choosing the best n to instrument can be determined by:

$$\arg\min_{n \in POSS} ||cross(seq(exclusive\_to(n)))|| + ||cross(seq(ALLPARAMS - params\_to(n)))||$$

Given a node or set of nodes for selective instrumentation, the next question is how to use it (respectively, them) to achieve more efficient testing. For purposes of illustration, assume there is just one end node e and a node n in a pipeline associated with an instrumentation function. In that case, we derive one set of tests for the parameters exclusive to n and then another set for all other parameters.

Suppose that $P1$, $P2$, $P3$ are exclusive to n and the other parameters are $P4$ to $P10$. *BugDoc* can use *Debugging Decision Trees* for $P1$, $P2$, $P3$ considering the evaluation function on n alone. (*BugDoc* will even cut off the computation of each instance after the evaluation at n.) *Debugging Decision Trees* may identify combinations of bad values among these three parameters. Human debuggers can then resolve those.

By the isolation property, when the evaluation of n indicates success for some pipeline instance, then any failure of that instance will be due to some combination of values of the remaining parameters $P4$ to $P10$. So, after finishing with $P1$, $P2$, and $P3$, *BugDoc* fixes some set of values on those parameters that leads to success at node n. *BugDoc* uses *Debugging Decision Trees* on $P4$ to $P10$ and the evaluation function on end node e.

7 System design and implementation

Figure 4 shows the high-level architecture of *BugDoc*. Given (i) a computational pipeline description, consisting

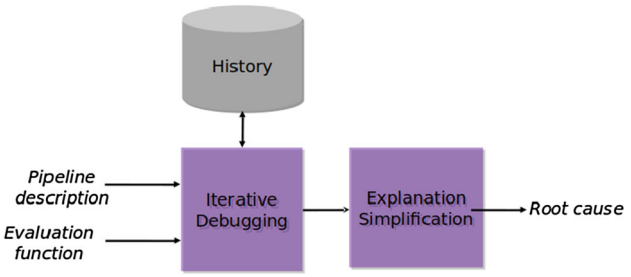


Fig. 4 *BugDoc*: overview of the approach

collection of programs connected together and an associated set of manipulable parameters; (ii) a set of pipeline instances, i.e., the history of the pipeline that contains values for the parameters of all pipeline runs and their outcome; and (iii) an arbitrary evaluation function that determines whether the pipeline results are acceptable or not, our goal is to find the minimal root causes of these results by iteratively executing new pipeline 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 parameter values, or combinatorial design [18].

Intuitively, all debugging algorithms presented in Sect. 4 work as follows. Given an initial set of instances, some of which lead to bad outcomes, the algorithms generate new parameter-value configurations (from the same universe) for the suspect instances and combine them with parameter-values that led to good outcomes. This approach has the benefit of swiftly eliminating hypothetical minimal root causes that are not confirmed by the newly generated instances.

Iterative Debugging In operation, *BugDoc* first runs the *Stacked Shortcut* algorithm. *Stacked Shortcut* applies heuristics to select and test combinations of parameter-value pairs. As discussed in Sect. 4, under reasonable assumptions, *Stacked Shortcut* finds minimal definitive root causes, using a number of pipeline instances proportional to the number of parameters.

When there are few parameters, *BugDoc* runs the *Debugging Decision Trees* algorithm—starting from the results of the pipeline instances run by the *Stacked Shortcut* algorithm and using the parameters of the pipeline as features and the evaluation of the instances as the target. *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 use the tree to discover short paths, possibly characterized by inequalities in parameter-values, that lead to `fail`. Those will be our "suspects". Unlike *Stacked Shortcut*, *Debugging Decision Trees* can identify root causes that depend on inequalities.

Parallelism The most time-consuming aspect of automated 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). When the search space is large, this extra overhead turns out to be small, as we show in Sect. 9.2.

Distributed pipeline execution The implementation of *BugDoc* is decoupled from the pipeline execution engine (which is application specific). If several instances of the pipeline execution engine are deployed in a distributed environment, then a single *BugDoc* instance can manage them. *BugDoc* can send pipeline instance specifications to the pipeline execution engines that execute them and return the results.

Explanation simplification Causes for errors can include multiple parameters, each of which may have large domains. It is thus essential to give concise explanations so that the user can both understand and act on them. Because the results of the *Debugging Decision Trees* algorithm consist of disjunctions of conjunctions, they may contain redundancies, which *BugDoc* simplifies using the Quine-McCluskey algorithm [32], 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.

Dataset debugging When there are root causes pointing to input files representing datasets, we apply a two-step debugging approach, first running BugDoc to explain the parameter-values and then displaying visualizations with meta-features and statistics about the data that could uncover meaningful information about the pipeline behavior. This step requires user interaction because *BugDoc* is agnostic about the parameter semantics; therefore, dataset parameters should be identified manually. With dataset annotation, in addition to displaying data profiles, we can apply Opportunistic Group Testing and present data root causes as described in Sect. 5.

Discussion The different algorithms of *BugDoc* can be used separately or together, depending on the purpose of the application. Either the *Shortcut* or the *Stacked Shortcut* algorithm discovers definitive root causes (or bugs) consisting of a single parameter-value (formally, parameter-equality-value) or a single conjunction of parameter-values. Since these algorithms find a definitive root cause within a single pipeline instance, they provide a **local explanation** of the failure. *Debugging Decision Trees* discover more complex definitive root causes which may involve multiple parameters and inequalities. If allowed to run to completion, they will provide a **global explanation**.

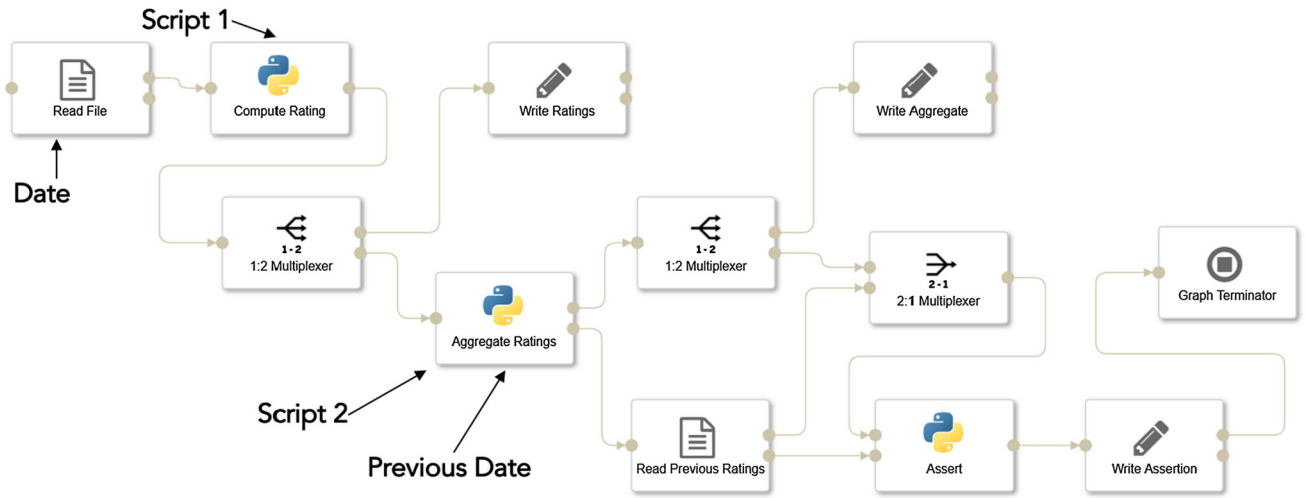


Fig. 5 Case Study Pipeline. A data pipeline reads a file containing product reviews written in given time period, computes ratings for each product, and compares them with ratings computed in previous periods. An evaluation function was added to the pipeline (Assert node) to make it debuggable by *BugDoc*. The arrows represent the properties of the pipeline that change among different instances, which translates to parameters for *BugDoc*

8 Case study: enterprise big data analytics

In a cooperative project with a large software company, we applied *BugDoc* to a computational pipeline deployed within the company. In this section, we present the pipeline, described as a data workflow, and how it evolved in the course of our experiments. We also discuss how *BugDoc* was integrated with this specific environment, and how this process can be used in general for different workflow systems. Last, but not least, we discuss how *BugDoc* was used to debug the workflow and find the root causes for the errors we encountered.

8.1 Workflow and experiment description

The workflow used for our case study is depicted in Fig. 5. Node *Read File* reads a data file containing the values of 10 features for a product (identified by an unique *id*) measured in a given *period* of the day. Each feature takes an integer value between 0 and 10, or has a null value. Node *Compute rating* uses the feature values to compute a rating value for each product id and period, and the result is stored in a file by node *Write Ratings*. Node *Aggregate Ratings* computes the count of rating values aggregated by id and the result is stored in a file by node *Write Aggregate*. Node *Read Previous Ratings* accesses the aggregate ratings of the previous day which are then compared with the aggregate ratings of the current day (when they are available) by node *Assert* which returns either a value "succeed" or "fail." The result value is stored in a file by node *Write Assertion*.

The evaluation function for pipeline instances is computed by node *Assert*. Nodes *Read Previous Ratings* and *Multiplexer* are used to integrate the evaluation function with the rest of the pipeline. Node *Write Assertion* is distinct from *Assert* for the sake of modularity.

We tracked the execution of this pipeline, which ran daily, for three months. During this period, some changes were made to the *Compute Ratings* script. Initially, the program computed a weighted sum of the 10 features and divided that value by the maximum possible value, which is 10×10. In the weighted sum, a null value is counted as 0. Then the resulting value (a number between 0 and 1) was categorized into two values: B (for Bad) and G (for Good).

A first change was to ignore features with null values in the weighted sum and to normalize the weighted sum by dividing it by $10 \times N$, where N is the number of non-null feature values.

The second modification of the pipeline consisted in categorizing the normalized weighted sum into 4 values: VB (for Very Bad), B (for Bad), G (for Good) and VG (for Very Good). However, the program for node *Assert* was unchanged: it only compared the aggregated count values for G and B ratings regardless of the changes made to the *Compute Rating* program. If the ratio of the number of G and B values per product id was beyond a given threshold, the pipeline instance was considered to have "failed". This lack of coordination is typical of a real-world situation where the version of a software component c evolves, but another component c' that depends on c does not evolve in tandem.

8.2 Integrating *BugDoc* with a third-party workflow

The enterprise workflow ran in production every day, suffered from bugs that manifested in different ways, and made use of input data files that sometimes were the same as the previous day's output data files. To find root causes, *BugDoc* must be able to create and run configurations that might involve both programmatic elements and data files from the same or different days, have a clear indicator of success or failure, and to do all this without disturbing production runs. To enable debugging for this workflow, we followed the following steps:

- We created a sandbox environment in which (i) configurations can be run without disturbing the production flow, and (ii) a function that indicates whether a run fails or succeeds.
- For each day's processing, we captured (i) the data inputs, (ii) all program versions, parameter, and hyperparameter settings, (iii) all data outputs, and (iv) an indication of success or failure.
- We represented the workflow as a directed graph in which each node represents either a data input or output or a process. An edge from node n to n' represents the flow from the output of n to the input of n'. Further each node is associated with a group of user-settable parameters.

Clearly, setting up a sandbox environment can be time consuming, but systematic manual debugging would require the same setup. Enterprises commonly deploy development and test sandbox environments separately from production environments. While keeping all data inputs and data outputs as well as program versions and parameter settings does incur additional space overhead, the benefit is that it allows the discovery of minimal root causes.

Concretely, here is how we achieved this setup for the case study. Pipelines/workflows at the company are specified in a JSON format and are deployed and executed on the cloud. Each pipeline instance is a directed graph G, and each node of G is associated with a set of parameters. In addition, there are global parameters that can be accessed by any node. Some parameters are the names of files that can be shared among pipeline instances and can be edited by other pipelines or external actors, and thus the same file name can take on different values. That is why we consider a file name to be a parameter and an instance of that file name to be a value.

We parsed the JSON specification of every pipeline instance that was run to extract both the topology of the pipeline as well as all the parameters and their values. The parsing procedure identified dozens of parameters, but only four of them would change among instances: `Date`, that represented the input file to be read by node *Read File*; `Script 1`, the program code to compute the ratings; `Script 2`, the code to aggregate the ratings; and Previous Date, that indicates which previous aggregate ratings to read.

The requirement to keep the output files comes from the fact that an output file of one day can be the input file of the next. Specifically, the node *Read Previous ratings* retrieves the output of the previous day. To enable that, we introduced a global parameter, denoted "graph_handle", into the pipeline to retrieve the address of the appropriate files to read.

The sandbox environment could run any pipeline instance (consisting of various program versions and parameter settings) specified by *BugDoc* and would report success or failure.

8.3 Debugging the workflow

We applied *BugDoc* in chronological order, never using values from runs executed after a failing day f, to analyze the root causes of the failure at f. The reasoning is that any root cause at day f must involve the parameter-value universe up to and including f, but no future days.

During the three-month period in which we observed the workflow, the pipeline instances failed on seven days: March 22 and 23, April 1 and 10, and May 14, 15, and 16. For each failing date, we ran *BugDoc* following the approach described in Sect. 7, combining the *Stacked Shortcut* and *Debugging Decision Trees* algorithms. In the seven applications of *BugDoc* to the failed pipeline instances, root causes included settings of computational scripts and errors in data files.

For example, regarding the failure that occurred on March 22, *BugDoc* determined that the input dataset *product measurements* that day was responsible for the failure. Consequently, the *aggregated ratings* generated on the same day was corrupted. *BugDoc* also determined that future executions of the pipeline would fail when computing their ratings based on March 22 data (i.e., when *Read Previous Rating* module used the March 22 output as a parameter-value). By contrast, the ratings computed on March 23 would not cause errors.

Regarding the failure on April 1, a new version of the *Compute Rating* script caused an error for previous ratings from March, even when using uncorrupted data. However, this script did not propagate the error to future pipeline executions.

The failure observed on April 10 was traced to another spurious data input.

8.3.1 Use of opportunistic group testing

Because some of the errors involved data files, we invoked Opportunistic Group Testing to find the offending records or fields.

For each bad (i.e., failure-inducing) data file F version, we identified a minimal pair in which the only difference between a successful computational instance and a failing one was the different version of F. For the March 22 error, the file in question was *aggregate ratings*. Opportunistic Group Testing identified a single product id whose rating caused an error.

In this study, we observed different errors in a data generation process: the rating values changed from binary (good or bad) to multiclass (very good, good, bad, very bad). This represents a single column error. However, there could be errors that straddle multiple columns. For example, the difference in values between two fields of a row could exceed a certain threshold.

Outlier detection methods such as Isolation Forests [41] can identify extreme values in columns or rows. However, it does this on a per file instance basis, independently of the history of the file in question. By contrast, *Opportunistic Group Testing* compares good file instances with bad file instances on a key value-by-key value basis to identify rows whose non-key fields have diverged significantly. The opportunistic module performs a binary search, taking different divergence scores thresholds in each iteration to filter columns and rows.

To assess the effectiveness of *Opportunistic Group Testing*, we mixed two error-generating models inspired by the industrial case study: one causes failures when there are extreme values in a column, and the other causes failures when there is an extreme subtractive difference between the values in two columns. We controlled the number of rows (keys in the case of single-column errors) affected by the injected errors to compute the precision and recall of the algorithm. Regardless of the number of affected rows, *Opportunistic Group Testing* obtained perfect recall and roughly 70% precision. That is, all problematic data were identified, but roughly 30% of the data that were identified as problematic were not in fact problematic. In Fig. 6, we compare the *Opportunistic Group Testing* method with Isolation Forests and observed that the precision and recall of the outlier detection decrease as the number of affected rows increases, i.e., as the corrupted values cease to be outliers.

8.3.2 Opportunity for selective instrumentation

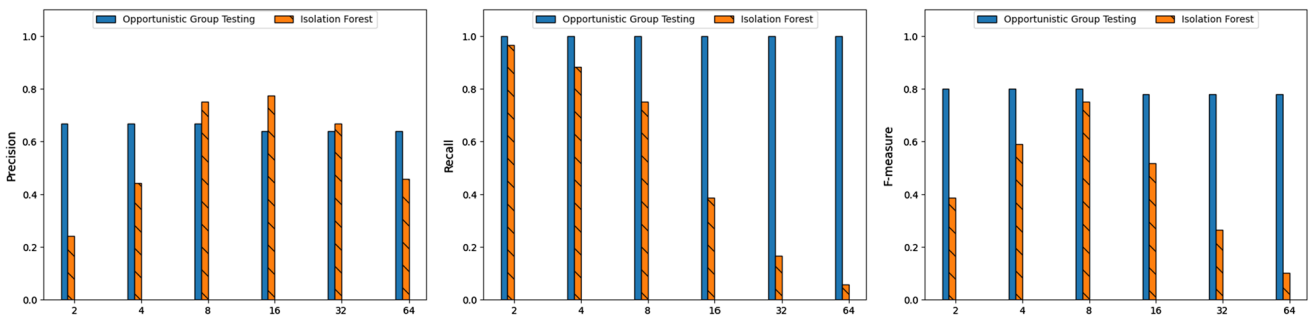
Because the JSON description of the workflow provided the graph relationship of the processes in the pipeline (Fig. 5), selective instrumentation as described in Sect. 6 could have been used to reduce the combinatorial explosion of configurations to test when using debugging decision trees.

Consider again the pipeline in Fig. 5. *Read File* node is the pipeline entry point, whereas *Write Ratings*, *Write Aggregate*, and *Graph Terminator* are output or *E N D* nodes. Thus, the set nodes_to(Read Previous Ratings) contains the nodes: *Read File*, *Compute Rating*, *Aggregate Ratings*, and *Read Previous Ratings* itself.

Table 6 shows that the node *Read Previous Ratings* has an exclusive parameter and hence could be a useful node to instrument with a success/failure function to reduce the search space.

9 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 debugging scenarios, including when looking for a single minimal definitive root cause and when there is a budget for the number of instances that can be run. We also evaluate the scalability of *BugDoc* when multiple cores are available to execute pipeline instances in parallel, and with respect to the number of parameters and values increase.



(a) Precision per # of affected rows **(b)** Recall per # of affected rows **(c)** F-Measure per # of affected rows

Fig. 6 *Opportunistic Group Testing*. The challenge here is to identify the rows that are responsible for failure. *Opportunistic Group Testing* always achieves 100% recall and precision of at least 69%, meaning *Opportunistic Group Testing* sometimes accuses rows of responsibility of failure unnecessarily. Using the default value of the contamination parameter, Isolation Forest achieves different levels of recall and precision depending on the number of bad rows. *Opportunistic Group Testing* always has a better F-measure than Isolation Forest

Table 6 Cardinality of the cartesian products of nodes of pipeline in Fig. 5

Node n	params_to(n)	exclusive_to(n)	$\|\|cross(seq(exclusive\_to(n)))\|\|$	$\|\|cross(seq(ALLPARAMS - exclusive\_to(n)))\|\|$
Read file	{Date}	{}	0	4
Compute rating	{Date, Script 1}	{}	0	4
Aggregate ratings	{Date, Script 1, Script 2}	{}	0	4
Read previous ratings	{Date, Script 1, Script 2, Previous date}	{Previous date}	1	3

Baselines. Because no previous approach both creates new instances and derives explanations, we compare *BugDoc* against combinations of state-of-the-art methods. We use Data X-Ray [55] and Explanation Tables [22] to derive explanations, and to generate instances used by all explanation algorithms, we use both *BugDoc* and Sequential Model-Based Algorithm Configuration (SMAC) [33]. *SMAC* is a method for hyperparameter optimization that is often more effective to search configuration spaces than grid search [8]. 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 the best complete pipeline instance 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 definitive 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 might return a pipeline instance that fails, which includes P_i set to 5. But since the pipeline may have other parameter-values, the user has no way of knowing that $P_i = 5$ is the minimal definitive root cause and thus gains little 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 a setting in which 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 CP be a pipeline (e.g., the pipeline represented at the top of Fig. 1) and CP_i be some instance of the pipeline (e.g., the CP_4 of Fig. 1). The set of minimal definitive root causes of CP is denoted $R(CP)$. The set of root causes asserted by an algorithm A on pipeline CP is denoted $A(CP)$. Our experiments are run over a large set of pipelines UCP, each of which is associated with multiple pipeline instances.

For the *FindOne* case, we check if $A(CP)$ has at least one actual root cause. Precision is the fraction of root cause assertions where at least one minimal root cause is found. 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. (Note that all UCP pipelines have at least one root cause.) The *recall for FindOne* is thus:

$$\frac{\sum_{CP \in UCP} |A(CP) \cap R(CP) \neq \emptyset|}{|UCP|} \quad (1)$$

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 run until it finds at least one minimal definitive root cause using each of its algorithms (*Shortcut*, *Stacked Shortcut*, and *Debugging Decision Trees*).

The experiment then allocates the same number of instances created in the previous step 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 have created synthetic data that reflect typical pipelines in data science and computational science, which often involve multiple components and associated parameters. The pipelines vary in the number of parameters and values from small to large, in order to reflect the space of typical data science benchmarks. That is, the synthetic pipelines are generalizations of the real pipelines we have observed (see Sects. 8 and 9.3) along the following dimensions: more parameters, more values, and more complex bugs.

The goal of the benchmark is to see whether *BugDoc* becomes more or less advantageous as the problem becomes more complex. 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 disjunctive definitive root cause consisting of conjunctions generated as follows:

1. For each conjunction, given $|P|$ parameters, we select a subset of parameters of size i, chosen uniformly from 1 to $|P|$.
2. For each parameter in the subset, we choose one value v chosen uniformly from its values.
3. For each numerical parameter, we choose from the comparators $C = \{=, \leq, >, \neq\}$ with probability 1/4 (except that we exclude $>$ when value v is maximum).
4. Additional conjunctive root causes are included in the set of minimal root causes with a certain probability that varies with the experiment.

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

Example 8 A pipeline having three parameters with four possible values each could be represented as follows:

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

Besides using the synthetic benchmark, we also evaluate the debugging strategies on real-world computational pipelines (see Sect. 9.3).

Implementation and experimental setup The current prototype of *BugDoc* contains a dispatching component that runs in a single thread and spawns multiple worker processes to run the new 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, provided by the respective authors, for both the Data X-Ray algorithm (implemented in Java 7) [55] and Explanation Tables [22] (written in Python 2.7). Since the state-of-the-art baselines do 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 $\times$ 8 processor). For purposes of reproducibility and community use, we have made our code and experiments available (https://github.com/ViDA-NYU/BugDoc).

9.1 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.

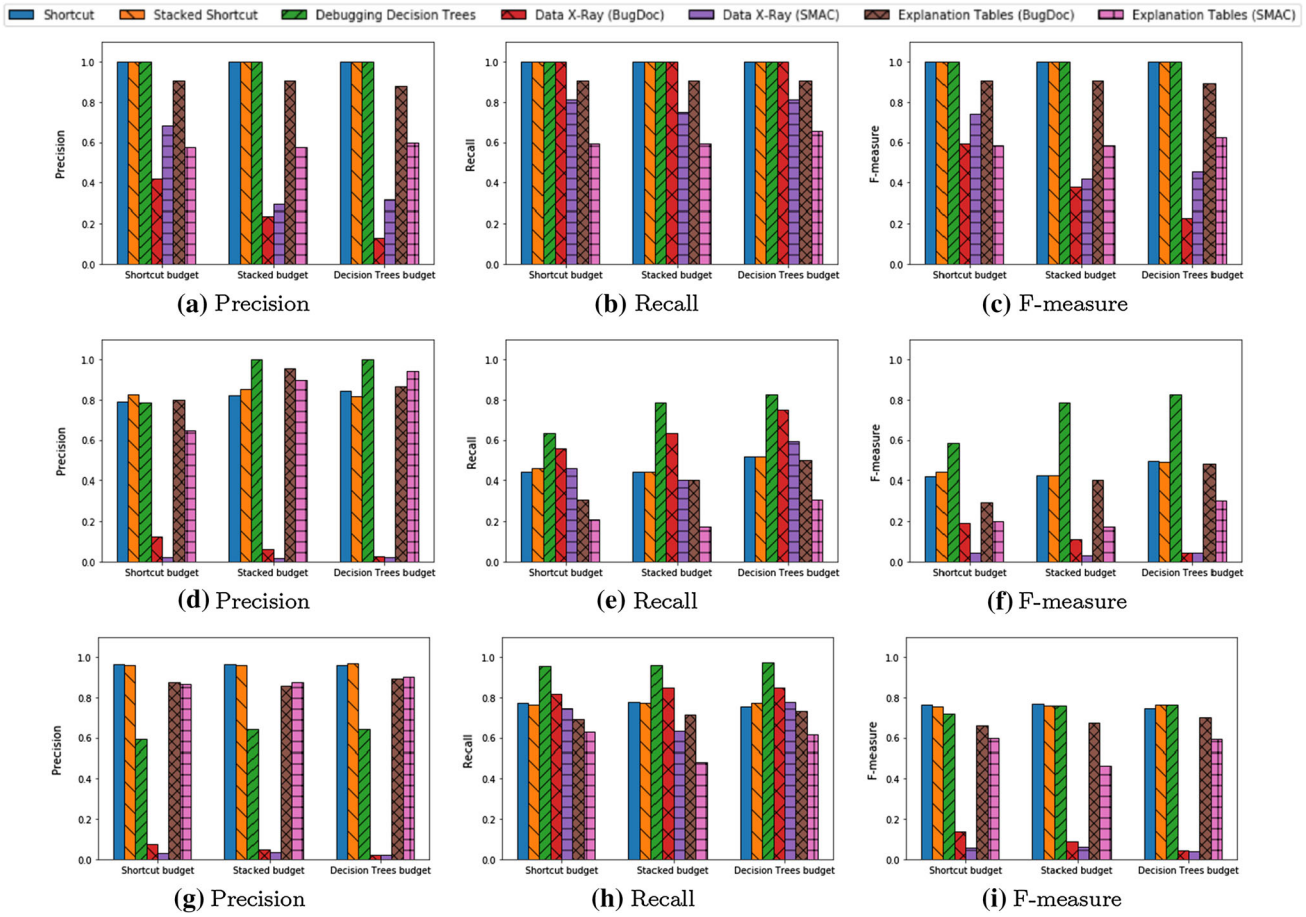


Fig. 7 Synthetic Pipelines. Metrics for the *FindOne* problem when the root cause is a single parameter-value-comparator (top row, Fig. 7a, b, and c), a single conjunction (middle row, Fig. 7d, e, and f), or a disjunction of conjunctions (bottom row, Fig. 7g, h, and i). In each figure, the leftmost group uses as many instances as does *Shortcut*, the middle uses as many as *Stacked Shortcut*, the rightmost as many as *Debugging Decision Trees*

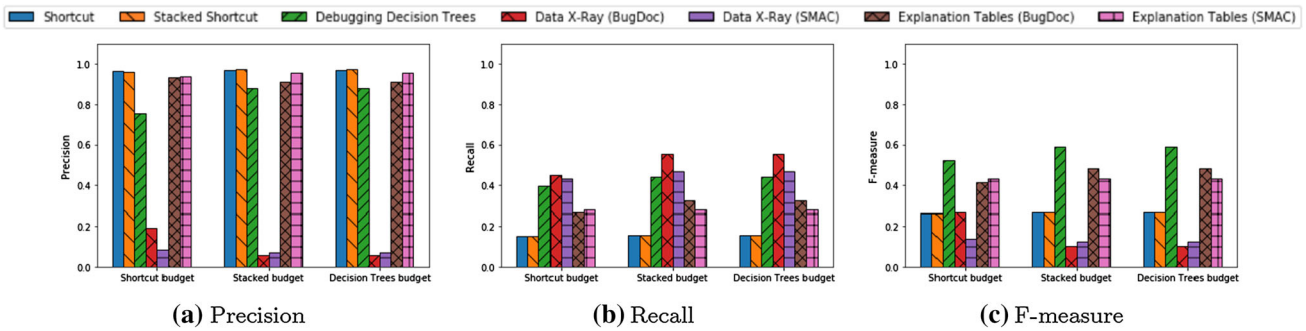


Fig. 8 Synthetic Pipelines. Metrics for the *FindAll* problem when the root cause is a disjunction of conjunctions (Fig. 8a, b, and c). In each sub-figure, the leftmost group uses as many instances as does *Shortcut*, the middle group as many *Stacked Shortcut*, the rightmost as many as *Debugging Decision Trees*

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

Precision, recall, and F-measure Figure 7 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 *Short-*

cut, *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 by SMAC. If the minimal root cause is a single parameter-comparator-value (Fig. 7a, b, and c), *Shortcut* and *Stacked Shortcut* achieve similar precision and recall to *Debugging Decision Trees*. By contrast, *Debugging Decision Trees* dominates when the minimal root causes are more complex. Recall that *Debugging Decision Trees* may require a number of instances that is exponential in the number of parameters in the worst case. In these experiments, they perform well using combinatorial design even with relatively small budgets.

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 Fig. 7e. 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 more truncated assertions, and their precision score is lower in Fig. 7d as compared to Fig. 7a and g. 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 as Fig. 8 shows. The non-minimal approach of Data X-Ray pays off in this scenario when there are multiple reasons for a pipeline to `fail`. However, *Debugging Decision Trees* presents a better trade-off between precision and recall (Fig. 8c).

Discussion The answers provided by Explanation Tables represent a prediction of the pipeline instance evaluation result expressed as a real number, where 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.

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.

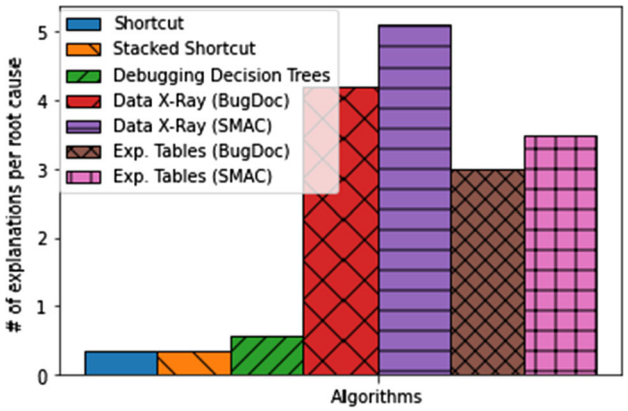


Fig. 9 Synthetic Pipelines. Average logarithmic number of asserted root causes per actual definitive root cause for each method. 0.0 on this log scale means that a method asserts exactly one root cause per definitive root cause. Both *Stacked Shortcut* and *Debugging Decision Trees* are close to 0.0

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 or the minimal root cause(s) are parameter-equality-value pairs.

Conciseness of explanation Figure 9 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 but also that it does not assert more root causes than there are.

9.2 Scalability of debugging algorithms

The primary computational cost for automated debugging is the cost of running the pipeline instances. Figure 10 shows the number of instances created by each of *BugDoc*'s algorithms as a function of the number of parameters of the 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.

Selective instrumentation Another strategy to reduce the number of iterations is to use the Selective Instrumentation strategy of Sect. 6. This entails inserting evaluation functions within the pipeline. To evaluate the potential benefit of Selective Instrumentation on the number of generated instances, we run *BugDoc* with and without instrumentation until both approaches find all the root causes for synthetic pipelines with a varying number of parameters. In these experiments,

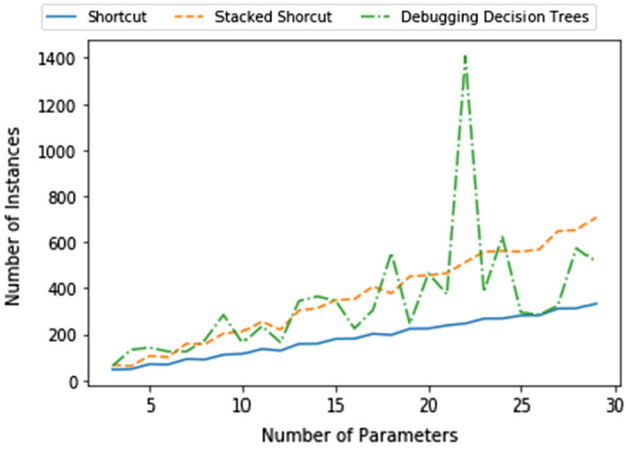


Fig. 10 Instances required to execute each algorithm as a function of the number of parameters

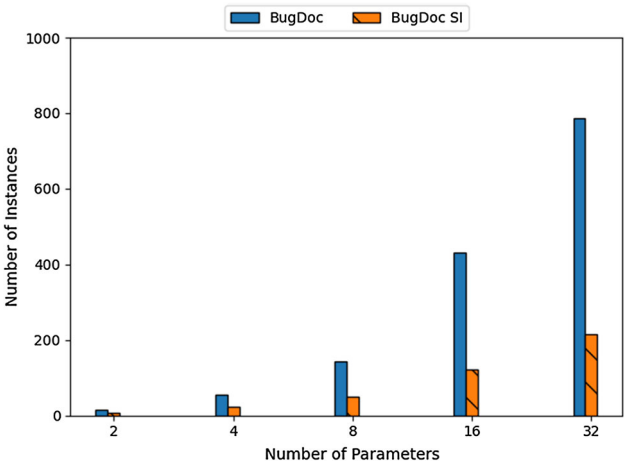


Fig. 11 Instances required to execute *BugDoc* with and without Selective Instrumentation as a function of the number of parameters. As explained in the text, the number of four-value parameters exclusive to the node having the added evaluation function is half the total number of parameters in each case. In every case, Selective Instrumentation reduces the number of instances required for the *Debugging Decision Trees* algorithm

each parameter has four possible values. The number of parameters exclusive to the node having the added evaluation function is half the total number of parameters in each case. For example, for 32 parameters, 16 parameters are exclusive to the node to which an evaluation function has been added as illustrated in Fig. 12. Figure 11 shows that Selective Instrumentation reduces the number of instances needed by *Debugging Decision Trees* to find all root cause in linear pipelines by approximately 70% across all numbers of parameters.

Parallelism As noted above, the pipeline instances to test can be run in parallel, but at some risk of unnecessary computation. To evaluate scalability, we re-execute the experiment with synthetic data, described in Sect. 9.1, with different

Fig. 12 Illustration of a linear pipeline with instrumentation in the center node (N3)

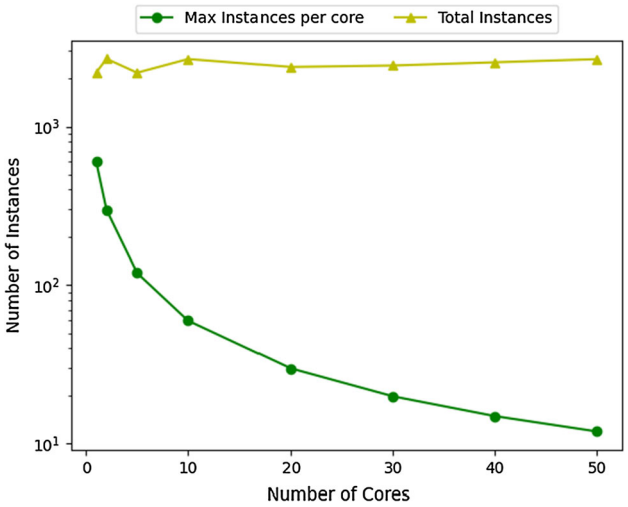


Fig. 13 Scalability of *BugDoc* when running the *Debugging Decision Trees* algorithm on multiple cores. As we increase the number of workers responsible for running instances (cores), the green line shows the maximum number of instances sent to each worker. That number decreases nearly linearly with the number of cores. The red line shows the total number of instances tested as the number of cores increases. The red line shows that there is a very slight inefficiency, because in the case of multiple cores, some instances are tested even after a hypothetical root cause has been shown to be incorrect

numbers of parallel computational cores and checked how many instances each core processed. As Fig. 13 shows, the performance improvement is essentially linear with the number of cores for the *Debugging Decision Trees* algorithm solving the *FindAll* problem in which large numbers of slightly different configurations must be explored. Thus given sufficient computing power, even *Debugging Decision Trees* can explore relatively large parameter spaces.

9.3 Real-world pipelines

In addition to synthetic pipelines, we also evaluated the effectiveness of *BugDoc* for pipelines used in real experiments. Calculating recall for the discovery of minimal root causes for real-world datasets is problematic because the set of all minimal root causes (the ground truth) is unknown. Instead, we compare *BugDoc* with the state-of-the-art baselines Data X-Ray and Explanation Tables based on *accuracy*. Accuracy is defined as the fraction of correct predictions on workflow instances. A prediction about an instance is correct if the prediction is that the instance will `succeed` (respectively, `fail`) and the instance in fact does `succeed` (respectively, `fail`).

Table 7 Accuracy results comparing *BugDoc* with the state-of-the-art black box algorithms on three real world datasets. Accuracy is percentage of instances predicted to succeed (respectively, fail) that actually succeed (respectively, fail)

Pipeline	Accuracy *BugDoc*	Data X-ray	Explanation tables
Data polygamy	100%	92%	28%
GANs	100%	91%	95%
TPC-C	98%	70%	70%

We run *BugDoc* with the combination of *Stacked Shortcut* and *Debugging Decision Trees* algorithms, all instances created in this procedure are given as input to Data X-Ray and Explanation Tables. We then generate random instances to simulate a test set, and compute the accuracy of *BugDoc* and the state-of-the-art baselines. Each method predicts that an instance will `fail` if the instance is a superset of at least one of minimal cause that the method finds. The results are summarized in Table 7.

Evaluating the data polygamy framework Data Polygamy aims to discover statistically significant relationships among a large number of spatio-temporal datasets [16]. One of the experiments described in [16] uses synthetically generated datasets whose features are given as input parameters for the experiment. This experiment is a good use case for *BugDoc* 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 processed data is heterogeneous, consisting of over 300 synthetic datasets at different spatio-temporal resolutions that are created, filtered out, and aggregated in a manner that depends on parameter settings.
- The parameter space is relatively 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 extremely time-consuming.

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 sought minimal definitive root causes consisting of combinations of parameter-values which cause the execution to crash (i.e., `fail` in the language of *BugDoc*). We used those to establish the accuracy in determining whether so far unseen instances will `fail` or not.

BugDoc identified four root causes:

R.1 *Percentage* < 0
R.2 *Percentage* > 50
R.3 *Diff* < 0 and *Percentage* ≥ 0
R.4 *Diff* > 100 and *Percentage* ≥ 0

Note that while the Data Polygamy authors knew about root causes **R.1** and **R.2**, they were surprised to find out that the parameter *Diff* out of the interval [0, 100] only affects the pipeline execution negatively if *Percentage* is positive (**R.3** and **R.4**).

Using these root causes, discovered based on 600 instances, *BugDoc* predicted correctly (with 100% accuracy) which of 2000 randomly generated instances would lead to `succeed` or `fail`. We used the same training and test set of instances for the state-of-the-art algorithms. Data X-Ray had an accuracy 92%, and Explanation Tables 28%, which is consistent with the lower recall this method displayed for inequalities with synthetic data.

Training adversarial networks Generative adversarial networks (GAN) [27] are widely used for image generation and semi-supervised learning [49,58]. Training these generative models involves an expensive computational process with several configuration parameters, such as the architecture topology and a large number of hyperparameters. 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 [12] entailed modifying baseline architectures and setting hyperparameters manually over three months, using hundreds of cores of a Google TPUv3 Pod [35]. Lucic et al. [45] 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 [59] on CIFAR-10 [37] and applied *BugDoc* to find root causes of one of the most common problems of GAN training: *mode collapse* [13]. Our evaluation function sets a threshold on the Frechet Inception Distance (FID) [30] metric, which is a proxy for mode collapse. This pipeline specified only 6 parameters, each limited to 5 possible values. The bottleneck was the execution time because each configuration requires approximately 10 hours to train, depending on the discriminator and generator learning rates and the number of steps. *BugDoc* found that mismatches between the number of steps and the loss function have the most significant impact on FID.

We used the same training (created by *BugDoc*) and test (randomly generated) sets of instances for the state-of-the-art algorithms. *BugDoc* had an accuracy of 100%, Data X-Ray 91%, and Explanation Tables 95%.

Transactional database performance DBSherlock [57] 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 [53], 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. [5] 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 entailed overcoming the challenge that data and results were given, but not the actual computational pipeline. So it was not possible to run additional instances. We simulated the creation of new instances by reading the results from the authors' datasets when possible. Unfortunately, changing the value of one parameter $P1$ while keeping another $P2$ fixed often created a configuration that was not in the authors' datasets. So we focused on the 15 parameters for which there were the most combinations in the dataset. We also bucketized the values of each such parameter into 8 buckets using a decision tree classifier to identify splitting points to each parameter [21]

We split the dataset into three parts: 50% of the data were used for training; 25% was the budget for pipeline instances that any sub-method of *BugDoc* requested. *BugDoc* found 24 root causes

We tested those root causes on the 25% holdout to assess accuracy. *BugDoc* was accurate 98% of the time, results that are comparable to those reported in [5]. The baseline state-of-the-art algorithms Data X-Ray and Explanation Tables were not able to find minimal definitive root causes that could generalize for the holdout data because that required the ability to handle inequalities. So they predicted that every instance in the holdout set would result in `succeed`. In fact that was true for 70% of the holdout set.

10 Conclusion

To the best of our knowledge, *BugDoc* is the first method that autonomously finds minimal definitive root causes in black box computational pipelines. *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-equal-value triples, 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. The shortcut techniques are particularly useful when there are many, e.g. 15 or more, parameters.

When there are few parameters or sufficient computation time, the *Debugging Decision Trees* method of *BugDoc* performs best.

In contrast 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. To do this, we need to incorporate several features into such a provenance system:

1. In order to create new configurations out of the ones already run, *BugDoc* must have access to every value of every user-settable parameter in the history of execution instances.
2. In order to evaluate an execution instance, every execution instance, both historical and generated, should yield a success or failure indication.
3. To capture historical dependency, each data instance should have a timestamp so that if a failure occurs at datetime d, *BugDoc* can limit the search among parameter-values that appeared at or before d.
4. For opportunistic group testing to work, whenever a parameter is a file, any distinct instance of that file should be stored as a different value of that file.
5. For selective instrumentation to work, the provenance system must be able to export a graphical description of the workflow where each node is either an input or a process and each edge indicates that the output of one node feeds the input of another. In addition, the export must associate parameters and values with each node.

Second, we would like to explore general (as opposed to opportunistic group testing) group testing [38,46] to identify

problematic data elements when a dataset has been identified as a root cause.

Acknowledgements We thank the 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 IIS-1916505, IIS-2106888, IOS-1339362, MCB-1158273, MCB-1412232, and OAC-1934464; CNPq (Brazil) grant 209623/2014-4; the DARPA D3M program; 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. Alvaro, P., Rosen, J., Hellerstein, JM.: Lineage-driven fault injection. In: Proceedings of ACM SIGMOD, pp 331–346 (2015)
2. Artho, C.: Iterative delta debugging. Int. J. Softw. Tools Technol. Transfer **13**, 223–246 (2010)
3. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with confaid. login **36**(1), 1–14 (2011)
4. Attariyan, M., Chow, M., Flinn, J.: X-ray: Automating root-cause diagnosis of performance anomalies in production software. In: Proceedings of USENIX OSDI, pp 307–320 (2012)
5. Bailis, P., Gan, E., Madden, S., Narayanan, D., Rong, K., Suri, S.: Macrobase: Prioritizing attention in fast data. In: Proceedings of ACM SIGMOD, pp 541–556 (2017)
6. Bala, A., Chana, I.: Intelligent failure prediction models for scientific workflows. Expert Syst. Appl. **3**, 980–989 (2015)
7. Berger, B., Rompel, J., Shor, P.W.: Efficient NC algorithms for set cover with applications to learning and geometry. J. Computer Syst. Sci. **49**, 454–477 (1994)
8. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. JMLR **13**, 281–305 (2012)
9. Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Proceedings of NIPS, pp 2546–2554 (2011)
10. Bergstra, J., Yamins, D., Cox, DD.: Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of ICML, pp 115–123 (2013)
11. Bleifuß T, Kruse S., Naumann, F.: Efficient denial constraint discovery with hydra. Proceedings of VLDB Endowment **3**, 311–323 (2017)
12. Brock, A., Donahue, J., Simonyan, K.: Large scale gan training for high fidelity natural image synthesis. arxiv:1809.11096 (2018)
13. Che, T., Li, Y., Jacob, AP., Bengio, Y., Li, W.: Mode regularized generative adversarial networks. arxiv:1612.02136 (2016)
14. Chen, A., Wu, Y., Haeberlen, A., Loo, BT., Zhou, W.: Data provenance at internet scale: Architecture , experiences, and the road ahead. In: Proceedings of CIDR, pp 1–7 (2017)
15. Chen, MY., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of IEEE DSN, pp 595–604 (2002)
16. Chirigati, F., Doraiswamy, H., Damoulas, T., Freire, J.: Data polygamy: The many-many relationships among urban spatio-temporal data sets. In: Proceedings of ACM SIGMOD, pp 1011–1025 (2016)
17. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. Proceeding of VLDB Endowment **13**, 1498–1509 (2013)
18. Colbourn, C.J., Martirosyan, S.S., Mullen, G.L., Shasha, D., Sherwood, G.B., Yucas, J.L.: Products of mixed covering arrays of strength two. J. Comb. Des. **2**, 124–138 (2006)
19. Dalibard, V., Schaarschmidt, M., Yoneki, E.: Boat: Building autotuners with structured bayesian optimization. In: Proceedings of WWW, pp 479–488 (2017)
20. Dolatnia, N., Fern, A., Fern, X.: Bayesian Optimization with Resource Constraints and Production. In: Proceedings of ICAPS, pp 115–123 (2016)
21. Dubey, A.: https://towardsdatascience.com/discretisation-using-decision-trees-21910483fa4b. http://www.tpc.org/tpcc/, accessed: 2021-11-18 (2018)
22. El Gebaly, K., Agrawal, P., Golab, L., Korn, F., Srivastava, D.: Interpretable and informative explanations of outcomes. Proceedings of VLDB Endowment **1**, 61–72 (2014)
23. Fariha, A., Nath, S., Meliou, A.: Causality-guided adaptive interventional debugging. SIGMOD, https://doi.org/10.1145/3318464.3389694 (2020)
24. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions Softw. Eng. **2**, 276–291 (2013)
25. Galhotra, S., Brun, Y., Meliou, A.: Fairness testing: Testing software for discrimination. CoRR pp 1–13, arxiv:1709.03221 (2017)
26. Godefroid, P., Levin, MY., Molnar, DA.: Automated whitebox fuzz testing. In: Proceedings of NDSS, p 151-166 (2008)
27. Goodfellow, IJ., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. arxiv:1406.2661 (2014)
28. Gudmundsdottir, H., Salimi, B., Balazinska, M., Ports, DR., Suciu, D.: A demonstration of interactive analysis of performance measurements with viska. In: Proceedings of ACM SIGMOD, pp 1707–1710 (2017)
29. Gulzar, MA., Wang, S., Kim, M.: Bigsift: Automated debugging of big data analytics in data-intensive scalable computing. In: Proceedings of ESEC/FSE, pp 863–866 (2018)
30. Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S.: Gans trained by a two time-scale update rule converge to a local nash equilibrium. arxiv:1706.08500 (2017)
31. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proceedings of USENIX Security Symposium, pp 445–458 (2012)
32. Huang, J.: Programing implementation of the quine-mccluskey method for minimization of boolean expression. CoRR pp 1–22, arxiv:1410.1059 (2014)
33. Hutter, F., Hoos, HH., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proceedings of LION-5, pp 507–523 (2011)
34. Johnson, B., Brun, Y., Meliou, A.: Causal testing: finding defects' root causes. CoRR pp 1–12, arxiv:1809.06991 (2018)
35. Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Pl, Cantin, Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T.V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C.R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., Yoon, D.H.: In-datacenter performance analysis of a tensor processing unit. SIGARCH Computer Arch. News **2**, 1–12 (2017)
36. Krishna, R., Iqbal, MS., Javidian, MA., Ray, B., Jamshidi, P.: Cadet: A systematic method for debugging misconfigurations using counterfactual reasoning. arxiv:2010.06061 (2020)
37. Krizhevsky, A., et al.: Learning multiple layers of features from tiny images. Tech. rep, Citeseer (2009)

38. Lee, KW., Pedarsani, R., Ramchandran, K.: Saffron: A fast, efficient, and robust framework for group testing based on sparse-graph codes. IEEE Transactions Signal Process. pp 1–10 (2015)
39. Lewis, D.: Counterfactuals. Wiley (2013)
40. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In Proceedings of ACM SIGPLAN **6**, 15–26 (2005)
41. Liu, FT., Ting, KM., Zhou, ZH.: Isolation forest. In: 2008 eighth ieee international conference on data mining, IEEE, pp 413–422 (2008)
42. Lourenço, R., Freire, J., Shasha, D.: Debugging machine learning pipelines. In: Proceedings of DEEM (2019)
43. Lourenço, R., Freire, J., Shasha, D.: Bugdoc: A system for debugging computational pipelines. In: Proceedings of ACM SIGMOD (Demo) (2020a)
44. Lourenço, R., Freire, J., Shasha, D.: Bugdoc: Algorithms to debug computational processes. In: Proceedings of ACM SIGMOD (2020b)
45. Lucic, M., Kurach, K., Michalski, M., Gelly, S., Bousquet, O.: Are gans created equal? a large-scale study. arxiv:1711.10337 (2017)
46. Macula, A.J., Popyack, L.J.: A group testing method for finding patterns in data. Discrete Appl. Math. **1–2**, 149–157 (2004)
47. Meliou, A., Roy, S., Suciu, D.: Causality and explanations in databases. PVLDB **13**, 1715–1716 (2014)
48. Pearl, J.: Causality: Models, Reasoning and Inference, 2nd edn. Cambridge University Press (2009)
49. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. arxiv:1511.06434 (2015)
50. Rezig, EK., Cao, L.,, Simonini, G., Schoemans, M., Madden, S., Tang, N., Ouzzani, M., Stonebraker, M.: Dagger: A data (not code) debugger. In: CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings, www.cidrdb.org, http://cidrdb.org/cidr2020/papers/p35-rezig-cidr20.pdf (2020)
51. Snoek, J., Larochelle, H., Adams, RP.: Practical bayesian optimization of machine learning algorithms. In: Proceedings of NIPS, pp 2951–2959 (2012)
52. Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, MMA., Prabhat, P., Adams, RP.: Scalable bayesian optimization using deep neural networks. In: Proceedings of the ICML, pp 2171–2180 (2015)
53. TPC (2019) Tpc-c benchmark. http://www.tpc.org/tpcc/, accessed: 2020-02-10
54. Van Aken, D., Pavlo, A., Gordon, GJ., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of ACM SIGMOD, pp 1009–1024 (2017)
55. Wang, X., Dong, XL., Meliou, A.: Data x-ray: A diagnostic tool for data errors. In: Proceedings of ACM SIGMOD, pp 1231–1245 (2015)
56. Wang, X., Meliou, A., Wu, E.: Qfix: Diagnosing errors through query histories. In: Proceedings of ACM SIGMOD, pp 1369–1384 (2017)
57. Yoon, DY., Niu, N., Mozafari, B.: Dbsherlock: A performance diagnostic tool for transactional databases. In: Proceedings of ACM SIGMOD, pp 1599–1614 (2016)
58. Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., Metaxas, D.: Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In: Proceeding of ICCV, pp 5908–5916 (2017)
59. Zhang, H., Goodfellow, I., Metaxas, D., Odena, A.: Self-attention generative adversarial networks. arxiv:1805.08318 (2018)
60. Zheng, AX., Jordan, MI., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: Simultaneous identification of multiple bugs. In: Proceedings of ICML, pp 1105–1112 (2006)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.