BugDoc: Algorithms and a System to Debug Computational Processes

Raoni Lourenço
New York University
raoni@nyu.edu

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

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

ABSTRACT

Data analysis for scientific experiments, large-scale simulations, and machine learning tasks all entail the use of complex computational pipelines to reach quantitative and qualitative conclusions. If some of these activities produce erroneous or uninformative outputs, the pipeline may fail or derive incorrect results. Inferring the root cause of failures is challenging, usually requiring much human thought, and is prone to error. We propose a new approach that makes use of iteration and provenance to automatically infer the root causes and derive succinct explanations of failures. Through a detailed experimental evaluation, we assess the cost, precision, and recall of our approach compared to the state of the art.

PVLDB Reference Format:
Raoni Lourenço, Juliana Freire, and Dennis Shasha. BugDoc: Algorithms and a System to Debug Computational Processes. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

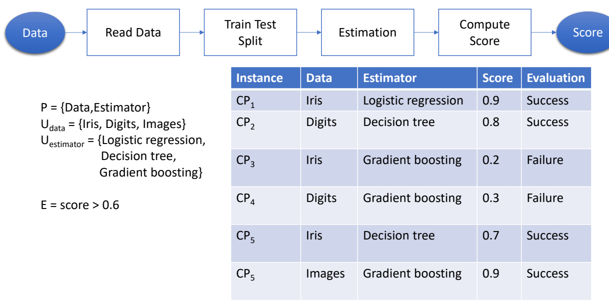
Computational pipelines in many domains, from astrophysics and biology to machine learning, are often complex, consisting of interdependent tasks and associated parameters, data inputs, and outputs. If one or more tasks of a pipeline produce erroneous outputs, the experimental and analytical conclusions may be compromised. Discovering the root cause of failures is challenging as errors can come from many different sources, including bugs in the code, input data, and improper parameter settings. In addition, some pipelines will not fail explicitly, but return abnormal results. The problem is compounded due to the fact that an error early in the pipeline may only surface at later steps. To understand these problems and track their causes, users must expend considerable effort reasoning about possible incorrect settings and executing new pipeline instances to test hypotheses. This is both tedious and time consuming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 45th International Conference on Very Large Data Bases, August 2019, Los Angeles, California.
Proceedings of the VLDB Endowment, Vol. 12, No. xxx
Copyright 2018 VLDB Endowment 2150-8097/18/10... $ 10.00.
DOI: https://doi.org/TBD

We propose an approach that automatically identifies one or more minimal causes of failures in computational processes. The problem of data diagnosis has received a lot of attention in the recent literature [21, 2, 9, 17]. Some works have focused on explaining where and how errors occur in the data generation process [21] and which data items are more likely to be causes of relational query outputs [17, 22]. Others have attempted to use data to explain *salient* features in data (e.g., outliers) by discovering relationships between attribute values [2, 9]. Unlike these approaches, we aim to explain abnormal behavior in computational pipelines that result from bad parameter settings.

To motivate our problem, consider a setting in which several analytical algorithms can be used each with a variety of hyperparameters. The results of using some hyperparameter settings can lead to useless outputs (e.g., low quality predictions) or even a crash. Sometimes, it is unclear which single hyperparameter-value setting or which combinations cause such results. Instead of having a person trying to guess and check, our goal is to derive a minimal explanation for what is causing these useless (for practical purposes, buggy) results. Consider the example in Figure 1, which shows a generic template for a simple machine learning pipeline and a log of different instances that were run with their associated results. The pipeline reads a data set, splits it into training and test subsets, creates and runs an estimator, and computes the F-measure score using 10-fold cross-validation. A data scientist uses this template to understand how different estimators perform for different types of input data, and ultimately, to derive a pipeline instance that leads to high scores. This entails exploring different parameters, data sets and learning classifiers. Note that, *gradient boosting* leads to low scores for two of the data sets, but it has a high score for another. At the same time, *decision trees* worked well for both the *Iris* and *Digits* data sets, and *logistic regression* leads to a high score for *Iris*. This may suggest that there is a problem with the *gradient boosting* module, that *decision trees* provide a good compromise for different data, and that *logistic regression* is good for the *Iris* data. However, this is hard to determine with confidence without running the pipeline for additional data sets and potentially, additional estimators and different settings for the estimators.

While prior work has relied solely on data or querying existing provenance to derive explanations, we *leverage the ability to execute computational pipelines to test new parameter settings and generate additional provenance that can be used to synthesize better explanations*. This, however, introduces new challenges. First, pipelines can consist of a



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

Figure 1: Simple machine learning pipeline that enables users to select different input datasets and classifier estimators

large number of tasks which in turn may contain many parameters. Running all possible combinations of these over a potentially large number of input data sets can be prohibitively expensive. Second, readable and concise explanations are needed to help users understand the causes of abnormal behavior.

Contributions. We introduce *BugDoc*, a new approach that analyzes the provenance of pipeline instances and derives new instances to identify sources of problems. Our approach takes the parameter-value pairs underlying already executed pipelines, as well as their success/fail status, and either proposes new executions or generates minimal root causes. This feedback-driven approach uses heuristics to swiftly discard false root causes and works well in parallel.

Our algorithms combine parameter exploration, similar to hyperparameter tuning works [5], with explainability [21]. We perform an extensive experimental evaluation to assess the effectiveness and efficiency of our approach. The results indicate that for settings in which there are even as few as seven or eight parameters, our explanations are better than those of the state-of-the-art algorithms when using the same number of pipeline instances.

Outline. The remainder of this paper is organized as follows. Section 2 introduces how we model computational processes and formally defines the problem we address. In Section 3, we present algorithms to search for simple and complex causes of failures. Our experimental evaluation and its results are discussed in Section 4. We review related work in Section 5, and conclude in Section 6, where we outline directions for future work.

2. DEFINITIONS AND PROBLEM STATEMENT

With the intuition that all data that vary with different pipeline instances is an input parameter in mind, we formally define our problem as follows.

Definition 1 (Computational process, process instance, parameter-value pairs, value universe, results)**.** Given an experiment modeled as a **computational process** (a pipeline) CP containing a set of parameters P (i.e., including hyperparameters, input data, versions of programs, computational modules, and so on), we denote as CP_i a **process instance** of CP that defines values for the parameters. Thus, an instance CP_i is associated with a list of **parameter-value pairs** Pv_i containing some assignment (p, v) for all $p \in P$. For each parameter $p \in P$, the **parameter-value universe** U_p is the set of all values assigned to p, i.e., $U_p = \{v | \exists i(p, v) \in P_i\}$.

Definition 2 (Evaluation)**.** Let E be a procedure that **evaluates** the results of an instance such that $E(CP_i) =$ `succeed` if the results are acceptable, and $E(CP_i) =$ `fail` otherwise. Acceptability might mean passing a set of tests that a user determines.

Definition 3 (Hypothetical root cause of failure)**.** Given a set of instances $CP_1, ..., CP_k$ and their 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 which obey the following properties: (i) if $E(CP_i) =$ `succeed`, then the parameter-values pairs Pv_i of CP_i do not satisfy the conjunction C_f; and (ii) there is at least one CP_i such that Pv_i satisfies C_f and $E(CP_i) =$ `fail`. For example, if $C_f = A > 5$ and $B = 7$, and CP_i has the parameter values $A = 15$ and $B = 7$ and succeeds, then C_f is unacceptable as a hypothetical root cause of failure.

Definition 4 (Definitive root cause of failure)**.** A *hypothetical root cause of failure* D is a *definitive root cause of failure* if there is no instance CP_q such that Pv_q takes values from the parameter-value universe with the property where $E(CP_q) =$ `succeed` and Pv_q satisfies D. That is, D does not lead to false positives.

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

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

Problem Definition. Given a computational process and a set of parameter-value pairs, our goal is to derive minimal definitive root causes. Note that this model is general and can capture any computational process, including queries, scripts, simulations.

3. DEBUGGING STRATEGIES

Approach Overview. BugDoc consists of two iterative debugging algorithms, the first, called minimal pairs, discovers single parameter-value (formally, parameter-equal-value) definitive root causes (or, more simply, bugs) and the second, called debugging decision tree, discovers conjunctive bugs. When a single parameter-value constitutes a definitive root cause, the heuristics of the minimal pairs algorithm find such root cause using fewer pipeline instances.

In operation, we first run the minimal pairs algorithm, described in Section 3.1, and, if the minimal pairs algorithm does not find any definitive root causes, we run the debugging decision tree algorithm (Section 3.2) – building

from the results of the pipeline instances run by the minimal pair algorithm. The debugging decision tree approach may yield root causes having parameter-comparator-value triples in which the comparator includes inequalities as well as equalities. Because the results of the debugging decision tree algorithm are disjunctions of conjunctions, they may contain redundancies which we simplify using the heuristic described in Section 3.3.

3.1 Looking for Single Root Causes: The Minimal Pairs Algorithm

The *minimal pairs* algorithm is inspired by methods in phonology to determine which changes in sounds convey meaning in a language [16]. For example, in English the minimal pair "bad" and "pad" shows that "voice" (the difference caused by the vocal chords between the phones "b" and "p") conveys meaning. By analogy, the minimal pairs method starts with a set of initial pipeline instances CPI, some of which have failed and some have succeeded. Our goal is to find at least one minimal definitive root cause for the failures. Thus, the initial instances define the parameter-value universe that we care about. The algorithm proceeds as follows:

1. If every pipeline in CPI that contains some parameter-value pair (P_i, v) evaluates to `fail`, then (P_i, v) is considered a "suspect". Here, (P_i, v) is shorthand for the triple $P_i = v$, because the minimal pair algorithm works only for equality comparisons.
2. Next, for each suspect parameter-value pair (P_i, v), the minimal pairs algorithm creates pipeline instances containing (P_i, v) and for every other attribute P_j, for $i \neq j$, it preferentially chooses a value y such that (P_j, y) has led only to `succeed` instances (which we sometimes refer to as good instances) among the pipeline instances tried so far. If any instance with (P_i, v) turns out to be good, then remove (P_i, v) from the suspect list. Otherwise, try pipeline instances with other values for P_j.
3. If all possible combinations of other parameter-values produce failure, then (P_i, v) by itself is a sufficient cause of a bad execution, so it is a minimal definitive root cause. Note that this is the only safe way of being sure that (P_i, v) is definitive.

To improve the efficiency of the debugging process, the minimal pairs algorithm applies two heuristics: (1) it discards simple potential root causes quickly by prioritizing parameter values among other attributes that have led to successful executions in the past, and (2) if there are multiple parameter-value pairs that are always associated with failure in the initial set of experiments, then the minimal pair method will start with the parameter that has the most values, thus reducing the worst case number of instances to explore. Below we present a simple example that illustrates how the Minimal Pairs Algorithm works.

Example 1. Consider the machine learning pipeline in Figure 1. This pipeline selects a data set, splits it into training and test sets, fits a classification estimator to the training set and outputs the accuracy score on the test set. In this scenario, we assume that the evaluation function returns `succeed` if $score \geq 0.6$.

In this pipeline, three parameters can be changed as follows:

- `Dataset`, the input data to be classified.
- `Estimator`, the classification algorithm to be executed.
- `Library Version`, parameter that indicates the version of the machine learning library used.

Note that the parameters of our model can be used to capture different properties of an experiment. In addition to actual parameters in pipeline modules (or steps), they can encode for example, specific algorithms or versions of libraries and operating system. Table 1 shows examples of three executions of the pipeline.

For each parameter, as detailed in Algorithm 1, this strategy creates three lists of values:

1. Pure good list *Good*, containing parameter-value pairs that are present only in instances that evaluate to `succeed`.
2. Pure bad list *Bad*, containing parameter-value pairs that are present only in instances that evaluate to `fail`.
3. Mixed list *Mixed*, containing parameter-value pairs that are present in some instances that evaluate to `succeed` and in others to `fail`.

Algorithm 1: Create pure good and bad lists for use in the heuristics

Input: CPI, the set of pipeline instances in the execution history characterized by their parameter-values
Input: *evaluation*, the evaluation function
Output: The lists *Good*, *Bad* and *Mixed*
`/* Initialization */`
$good_set \leftarrow \emptyset$;
$bad_set \leftarrow \emptyset$;
for $cp_i \in CPI$ **do**
  if $evaluation(cp_i) =$ `succeed` **then**
    $good_set \leftarrow good_set \cup \{cp_i\}$;
  else
    $bad_set \leftarrow bad_set \cup \{cp_i\}$;
  end
end
$puregood \leftarrow \{good_set - bad_set\}$;
$purebad \leftarrow \{bad_set - good_set\}$;
$mixed \leftarrow \{good_set \cap bad_set\}$;
return *puregood, purebad and mixed*

From the initial traces shown in Table 1 for Example 1, the following lists are generated:

Good = {Dataset={Iris,Digits},
Estimator={"Logistic Regression","Decision Tree"},
LibraryVersion={1.0} }

Bad = {Dataset={Iris},
Estimator={"Gradient Boosting"},
LibraryVersion={2.0} }

PureGood = {Dataset={Digits},
Estimator={"Logistic Regression","Decision Tree},
LibraryVersion={1.0} }

Table 1: An initial set of classification pipelines instances

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

PureBad = {Estimator={"Gradient Boosting"}, LibraryVersion={2.0} }

Mixed = {Dataset={Iris}}

The elements in *puregood* and *purebad* are parameter-value candidates that lead to good (i.e., evaluate to `succeed`) or bad (i.e., evaluate to `fail`) results, respectively. For example, we may suspect that the experiment will fail if *Gradient Boosting* is the selected estimator.

Given a parameter-value pair *pv* in *purebad* we assemble new tests containing *pv* and the values for the other parameters. We sample first from *puregood* and *mixed*, because pipeline instances containing those parameter-value pairs are heuristically the most likely to evaluate to true (i.e., pass all the tests). If any do evaluate to true then *pv* by itself is not a definitive root cause. By contrast, we declare *pv* to be a definitive root cause only if every pipeline instance containing *pv* leads to failure regardless of the values of the other parameters.

After assembling new configurations for the Example 1, as presented in Table 2, the Minimal Pairs algorithm returns the following lists:

PureGood = {LibraryVersion={1.0} }

PureBad = {LibraryVersion={2.0} }

Mixed = {Dataset={Iris,Digits}, Estimator={"Logistic Regression", "Decision Tree", "Gradient Boosting"}}

Hence, we would find our bug at parameter `Library Version` with value 2.0.

The most time consuming aspect of the Minimal Pairs approach is the execution of the pipeline instances. Fortunately, since each pipeline instance is independent, 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). We experimentally evaluate the parallel execution in Section 4.

3.2 Looking for Complex Explanations: Debugging Decision Trees

Often, a conjunction of parameter-values must all be present for a pipeline to fail. To give an emotive example, the large number of deaths on the Titanic resulted from several failures: the ship was going too fast, there was no lookout, there were not enough lifeboats, there was no training in water safety.... If all those factors had not been true at the same time, the loss of life would have been far less. In a scenario where a conjunction or a disjunction of conjunctions of parameter-value pairs may lead to failing pipeline instances, Minimal Pairs would not be able to establish candidate lists because no single parameter-value pair would be a definitive root cause.

We propose the construction of a *debugging decision tree* using the parameters of the pipeline as features and the evaluation of the instances as the target. Thus the leaves are either purely *true* – if all pipeline instances leading to a leaf evaluate to `succeed`, *false* – if all pipeline instances leading to a leaf evaluate to `fail`, or *mixed*. The decision tree is constructed as follows:

1. Given some initial set of instances CPI (which may be generated at random or by some combinatorial design technique [10]), construct a decision tree based on the evaluation results for those instances (`succeed` or `fail`). Each interior node of the decision tree represents a triple (`Parameter`, `Comparator`, `Value`), where the `Comparator` indicates whether a given `Parameter` has a value equal to, greater than (or equal to), less than (or equal to), or unequal to `Value`.
2. If a conjunction involving a set of parameters, say, P_1 P_2, and P_3 leads to a consistently failing execution (a pure leaf in decision tree terms), then that combination becomes a suspect by analogy to the way that a single parameter-value pair became a suspect in the Minimal Pairs algorithm.
3. Each conjunction leading to a pure fail outcome (i.e., each suspect) is used as a filter in a Cartesian product of the parameter values from which new experiments will be sampled. For simplicity, consider an example where all comparators denote equality. Suppose a path in the decision tree consists of $P_1 = v_1$, $P_2 = v_2$, and $P_3 = v_3$. To test that path, all other parameters will be varied. If every instance having the parameter-values $P_1 = v_1$, $P_2 = v_2$, and $P_3 = v_3$ leads to failure, then that conjunction constitutes a *definitive root cause of failure*. If the path consists of other comparators (e.g., $P_1 = v_1$, $P_2 = v_2$, and $P_3 > 6$), then choose a value for each of those parameters as an example, (e.g., $P_3 = 7$) and choose pipeline instances having those values (e.g., all pipelines $P_1 = v_1$, $P_2 = v_2$, and $P_3 = 7$). Conversely, if any of the newly generated experiments presents a good (succeed) pipeline instance, then the decision tree is rebuilt taking into account the whole set of executed pipeline instances CPI and a new suspect path is tried.

Note that if 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 our purpose here is to understand the bugs already uncovered rather than to give any form of complete verification as that problem is in general undecidable [4].

3.3 Simplifying Explanations

Decision trees are easy to read, but they do not always provide minimal explanations. For example, we may have two paths leading to pure *false* leaves that differ only in

Table 2: Set of classification pipelines instances after assembling new instances based on pure bad, pure good, and mixed lists

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

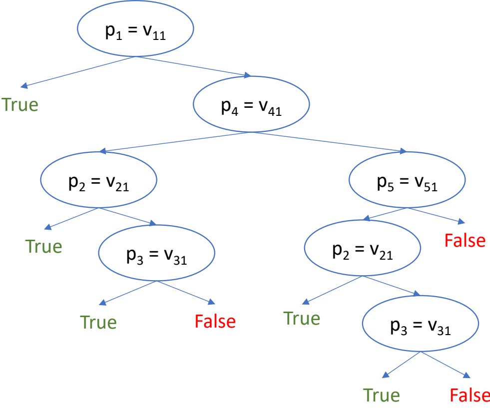


Figure 2: Decision tree fitted over instances where evaluations to **`fail`** occur if $P_1 = v_{1_1}$, $P_2 = v_{2_1}$, and $P_3 = v_{3_1}$ or $P_1 = v_{1_1}$, $P_4 = v_{4_1}$, and $P_5 = v_{5_1}$. Tree paths through the tree lead to pure bad (**`fail`**) results.

their value of the first parameter which takes just two values. Such paths can be reduced to a single conjunction consisting of the parameter-values they share. To generate concise explanations from the decision tree, we apply the Quine-McCluskey algorithm [14], which provides a method to minimize Boolean functions. Because the algorithm is exponential and encodes the Set Cover problem which is NP-complete, we use heuristics that do not achieve complete minimality but still reduce the size of the explanation. We illustrate this process below in Example 2. Because the use of Quine-McCluskey is not our research contribution, our explanation is brief.

Example 2. Consider an experiment whose instances lead the decision tree shown in Figure 2. There are three paths through the tree that evaluate to pure fail outcomes. The Quine-McCluskey algorithm attempts to shorten these paths to a simpler expression or expressions. The output of the algorithm contains the following conjunctions:

- (P_1, v_{1_1}),(P_2, v_{2_1}), and (P_3, v_{3_1})
- (P_1, v_{1_1}),(P_4, v_{4_1}), and (P_5, v_{5_1})

4. EXPERIMENTAL EVALUATION

This section presents the results of our experimental evaluation of *BugDoc* compared against state-of-the-art methods for deriving explanations as well as for hyperparameter optimization, using both synthetic and real pipelines. We examined different scenarios, including when a single solution is sought and when a budget for the number of instances that can be run is set. We also explore the scalability of our approach when multiple cores are available to execute pipeline instances in parallel.

Baselines. Two state-of-the-art methods for pipeline debugging and hyperparameter optimization are, respectively, Data X-Ray [21] and Sequential Model-Based Algorithm Configuration (SMAC) [15]. Note that the algorithms are not strictly comparable with one another nor with BugDoc. Data X-Ray analyzes existing pipeline instances, but does not suggest new ones. SMAC does propose new pipeline instances in an interactive fashion, but it stops once it finds a pipeline of interest. This makes sense for SMAC's primary use case, which is to find a well-performing set of parameters, but it is less helpful for debugging, because it makes no attempt to find a minimal root cause. For example, if a minimal definitive root cause of a pipeline is that parameter P_i must have a value of 5, SMAC will return a pipeline that fails which will have P_i set to 5. But since the pipeline may have many other parameters, the user has no way of knowing that $P_i = 5$ is the minimal definitive root cause and thus no way of knowing how to rectify the bug. To give SMAC a reasonable chance to find minimal root causes, we apply Data X-Ray to suggest root causes for the pipeline instances generated by SMAC.

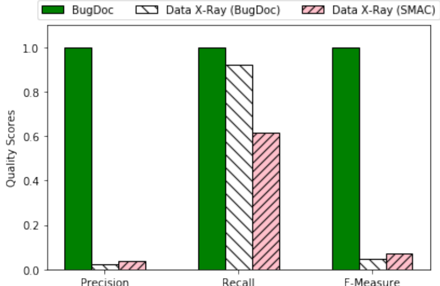
Evaluation Criteria. We used two criteria to measure quality: precision, which measures the fraction of causes identified by any given method are in fact minimal definitive root causes; and recall, which measures the fraction of test cases for which at least one minimal definitive root cause is found.

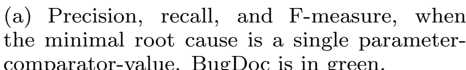
Our first set of tests allow *BugDoc* to find at least one minimal definitive root cause and then uses the same number of instances for the other algorithms. Thus, it gives the same budget to each algorithm and checks its precision and recall. A second set of tests tries different budgets of pipeline instances and sees how each algorithm performs in terms of these same quality metrics.

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

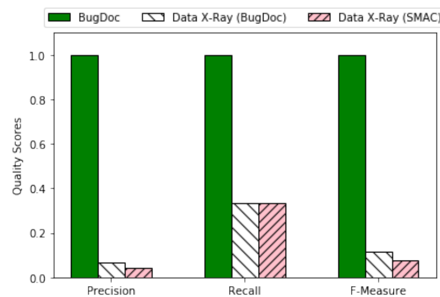
Let UCP be a set of computational pipelines, where each $CP \in UCP$ is associated with a set of minimal definitive root causes $R(CP)$.

Precision and recall are evaluated for all pipelines. Formalizing these definitions, given a set of minimal root causes asserted by an algorithm A for all CP in UCP, precision is the number of minimal root causes predicted by A that are truly minimal definitive root causes (true positives) divided by the size of the set of all root causes asserted by A, which includes true and false positives. That is,

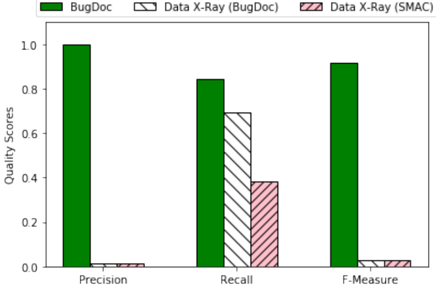




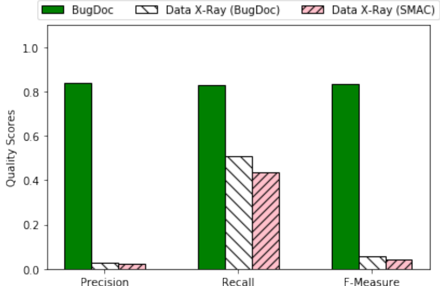
(a) Precision, recall, and F-measure, when the minimal root cause is a single parameter-comparator-value. BugDoc is in green.



(b) Precision, recall, and F-measure, when the minimal root cause is a disjunction of parameter-comparator-values.



(c) Precision, recall, and F-measure, when the minimal root cause is a conjunction of parameter-comparator-values.



(d) Precision, recall, and F-measure, when the root cause is a disjunction of conjunctions of parameter-comparator-values.

Figure 3: Synthetic Pipelines. Precision, Recall and F-measure for the first solution scenario in which BugDoc runs until it reports a minimal definitive root cause. The same instances are given to the Bayesian technique SMAC. Data X-Ray uses the results (succeed or fail) of executing the pipeline instances chosen by BugDoc and, separately, the results of the pipeline instances chosen by SMAC.

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

Recall is the fraction of times when a true minimal definitive root cause is found by A.

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

We also report the F-measure, i.e., the harmonic mean of precision and recall:

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall}$$

Pipelines. We evaluate our approach using both synthetic and real pipelines. The synthetic test set spans the following four scenarios, each consisting of multiple pipelines: (i) a single parameter-comparator-value triple constitutes a minimal definitive root cause, (ii) a disjunction of parameter-comparator-value triples, each constituting a minimal definitive root cause, (iii) a single conjunction of some parameter-comparator-value triples constitutes a minimal root cause, and (iv) a disjunction of conjunctions of parameter-comparator-value triples constitute a minimal root cause. We also present two case studies of real pipelines (see Sections 4.4 and 4.5). Each pipeline is associated with one or more known definitive root causes based on these scenarios.

We tried to generate synthetic data that reflect typical pipelines in data science and computational science which often involve multiple components and associated parameters. The pipelines have between three and fifteen parameters and each parameter has between five and forty values. The parameter values are chosen to be ordinal (e.g., temperature) or categorical (e.g., color) each with probability 1/2. The example below illustrates the parameter space and definitive root cause for a pipeline.

Example 3. A pipeline having three parameters with four possible values each could have the following configuration:

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

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

We used SMAC version for Python 3.6. The Data X-Ray algorithm is implemented in Java 7 and the code was kindly shared by its authors [21]. Since Data X-Ray does not generate new tests, we use the pipeline instances created

by BugDoc as input to the feature model input of Data X-Ray. Separately, we convert the pipeline instances created by SMAC as input to the feature model of Data X-Ray.

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 at our GitHub repository[1] and it includes a binding to the VisTrails workflow system.

4.1 Finding One Solution

For our first set of experiments, we set BugDoc to stop iterating as soon as it finds one minimal definitive root cause for failure. Figure 3 presents the precision, recall and F-measure for the four scenarios of root causes for a pipeline to explore the relative generality of all the algorithms:

1. pipelines with a single root cause.
2. pipelines with a disjunction of root causes.
3. pipelines whose failure root cause is a conjunction of parameter-comparator-value triples.
4. pipelines whose root cause is a disjunction of conjunctions.

BugDoc outperforms Data X-Ray in all four scenarios, both when Data X-Ray uses instances generated by BugDoc and SMAC. If the root cause is a disjunction (Figures 3b and 3d) or a single parameter-comparator-value (Figure 3a), BugDoc also attains high recall, since it usually identifies the minimal definitive root cause. The likelihood that BugDoc does not find a definitive answer is higher in the scenario where a root cause is a conjunction of factors, as can be seen in the relatively lower recall in Figure 3c, though the result is still better than that of Data X-Ray.

4.2 Finding Solutions on a Budget

Running pipelines can be expensive, so executing instances until we find a definitive root cause may not be feasible in practice. Hence, for this set of experiments we give the algorithm a budget of pipeline instances to be executed.

We compared BugDoc against Data X-Ray using the instances generated by BugDoc and SMAC, but setting a budget on the number of instances used. Figures 4–7 show the quality metrics for the four root cause scenarios described above as the number of instances increases (on the x axis).

Sometimes, when Data X-Ray uses the instances generated by BugDoc, it does better, at least in recall, for the case where the root causes are conjunctions of parameter-comparator-value triples. This is expected since Data X-Ray is meant to find relevant conjunctions. That is, Data X-Ray lists the parameter-value combinations that lead to bad scenarios. By contrast, BugDoc finds minimal decision trees for the data seen so far. When there is little data, jumping to generalizations can be a bad strategy (a lesson we have all learned from real life). This reinforces the importance of doing a systematic and iterative search to obtain more data. When the budget is high, BugDoc dominates the other methods, because Data X-Ray provides explanations that are not minimal definitive root causes.

[1] `https://github.com/ViDA-NYU/debugging-science`

4.3 Parallel execution

The major computational cost in any of these algorithms is the cost of running the pipeline instances. Fortunately, they can be run in parallel. The question is how good the scale up and speed-up are. Figure 8 shows that the scale-up is essentially linear with the number of cores for the first solution setup (Figure 3), thus given sufficient computing power, the approach is able to explore potentially large parameter spaces. Note that scale-up here leads to a speed up because the pipelines executions are the most costly operations, and we can discard the executions in the queue if our hypothesis proves to be false. By testing instances in parallel, we can reduce the time to detect such stop criterion.

4.4 Case Study: Linguistic Dataset

The authors of Data X-Ray [21] tested their algorithm with a linguistic database consisting of sentences consisting of subjects, verbs, and objects. The sentences were tagged with part-of-speech (POS) tags. Human labelers graded the sentences as good or bad. We used the same data to evaluate our framework. To model these data, we considered three parameters: subject, verb, object, and various tags for each. We applied BugDoc to generate instances (in this case sentences) and then tried to determine the minimal definitive root causes for the problems.

Because the sentence database was evaluated by people, there was a certain subjectivity in their evaluations. For that reason, there were cases where given two sentences $s1$ and $s2$, they could be identical with respect to their subject-value, verb-value, and object-value, yet $s1$ could be evaluated as good and $s2$ as bad. In such cases, if one evaluation was overwhelmingly more frequent (e.g., some subject-value, verb-value, object-value combination was evaluated 90% of the time as bad), then we set up the data to eliminate the minority evaluations.

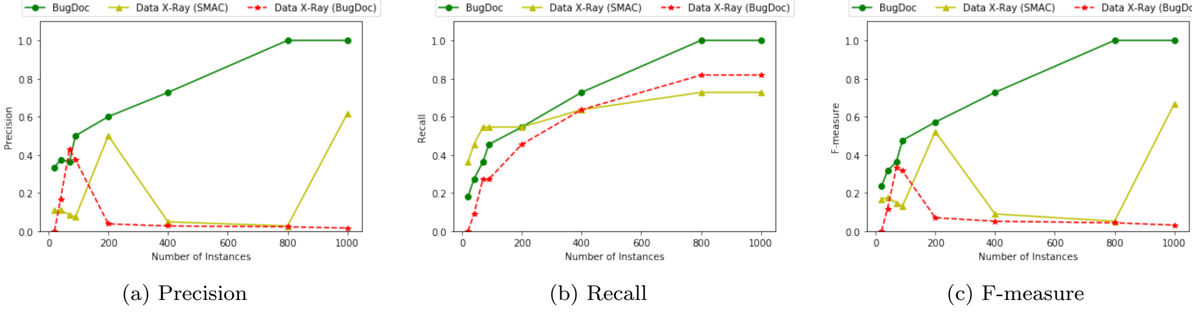
Figure 9 compares the results generated by BugDoc and Data X-Ray for these data. In these experiments all three algorithms (BugDoc, Data X-Ray using BugDoc instances, and Data X-Ray using SMAC instances) work well. When there are very few parameters and (in this case just subject, verb, object) and failure requires the combination of two or three parameter-values, BugDoc is neither better nor worse than the state of the art. Here, the root causes involve either two or three of the possible parameters, and all methods are able to find minimal explanations. By contrast, BugDoc's comparative advantage, as shown in Sections 4.1 and 4.2, occurs in pipelines that have many more parameters but where only few parameters constitute minimal definitive root causes.

4.5 Case Study: Data Polygamy Framework

To demonstrate our debugging strategies on a second real computational pipeline, case, we created a VisTrails workflow[2] for the Data Polygamy (DP) Framework [9], shown in Figure 10. Data Polygamy aims to discover statistically significant relationships among a large number of spatio-temporal data sets. The computational pipeline evaluates different methods for determining statistical significance.

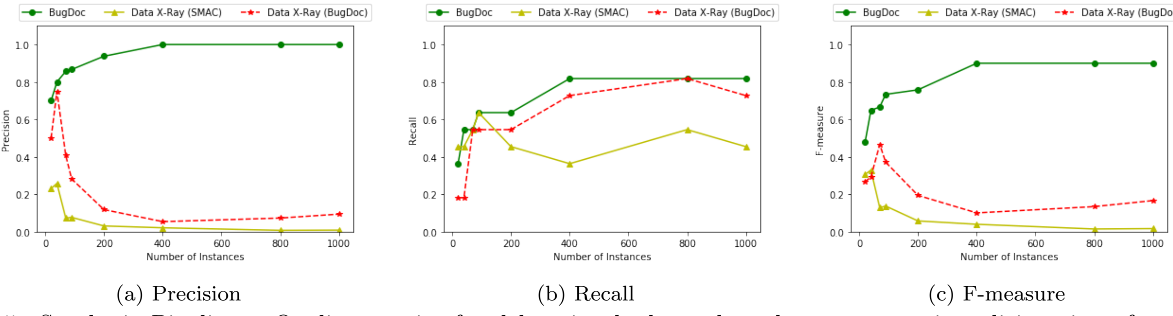
The pipeline reproduces an experiment designed by the Data Polygamy authors to evaluate the p-value and false discovery rate for Data Polygamy under different scenarios.

[2] `www.vistrails.org`



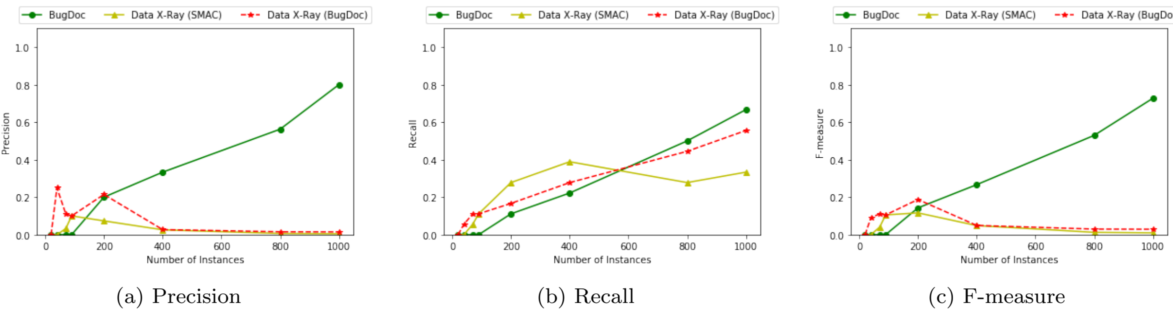
(a) Precision (b) Recall (c) F-measure

Figure 4: Synthetic Pipelines. Quality metrics for debugging on a budget (a fixed number of pipeline instances) when the root cause is a single parameter-comparator-value triple. In this case, BugDoc dominates Data X-Ray in precision when Data X-Ray uses the instances generated by SMAC, but not always when Data X-Ray uses the instances generated by BugDoc. BugDoc outperforms the other methods as the number of pipeline instances increases.



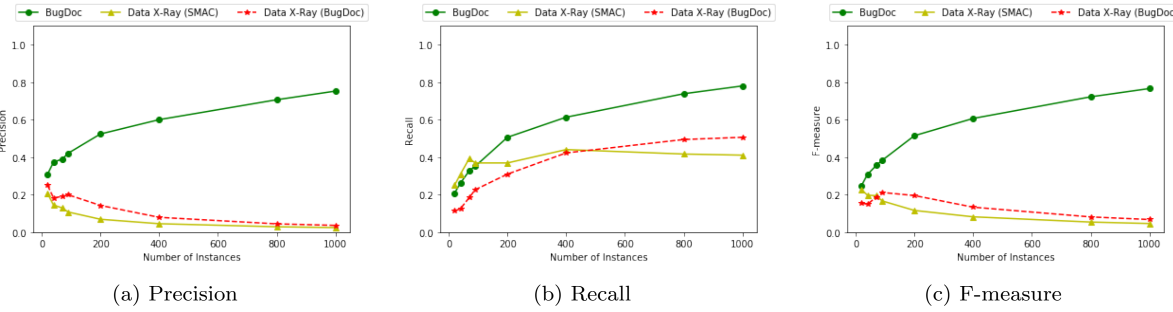
(a) Precision (b) Recall (c) F-measure

Figure 5: Synthetic Pipelines. Quality metrics for debugging budget when the root cause is a disjunction of parameter-comparator-value triples. In this case, again, BugDoc outperforms Data X-Ray when Data X-Ray uses the instances generated by SMAC, and when Data X-Ray uses the instances generated by BugDoc.



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

Figure 6: Synthetic Pipelines. Quality metrics for debugging on a budget when the minimal root cause is a conjunction of parameter-comparator-value triples. In this case, the results are more mixed. BugDoc only outperforms Data X-Ray when Data X-Ray uses the instances generated by SMAC, and when Data X-Ray uses the instances generated by BugDoc as the number of instances increases, specially for recall.



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

Figure 7: Synthetic Pipelines. Quality metrics for debugging on a budget when the minimal root cause is a disjunction of conjunctions of parameter-comparator-value triples. BugDoc attains higher precision and recall than Data X-Ray no matter whether Data X-Ray uses the same instances as those generated by BugDoc or those used by SMAC.

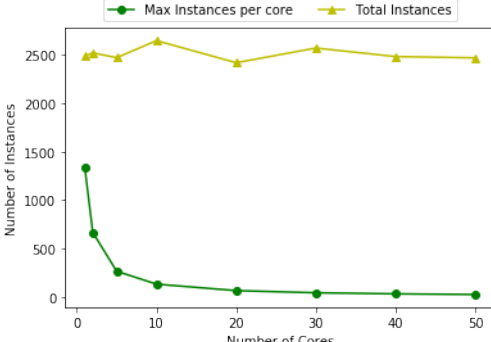


Figure 8: Simulated data. Scalability of BugDoc running on multiple cores.

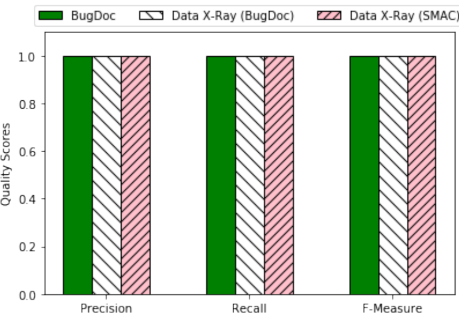


Figure 9: Linguistic Data Set. BugDoc and Data X-Ray using the instances generated by BugDoc and SMAC do equally well on this data set from the Data X-Ray paper [21]. That data set has three parameters (subject, verb, object) and where failures require a combination of at least two parameter-values. BugDoc has no comparative advantage over the other methods when there are few parameters and all of them are involved conjunctively in any failures

The data sets are synthetically generated and their features are given as input parameters for the experiment. This process is a good use case for our approach because it has the following properties:

- Data Polygamy constitutes a complex pipeline, including steps for data cleaning, data transformation, feature identification, multiple hypothesis testing, and other activities.
- The input data is heterogeneous – over 300 data sets at different spatio-temporal resolutions.
- The parameter space is large, making hand-debugging impractical.

For this experiment, we selected, together with the Data Polygamy authors, seven parameters to be debugged. To achieve the goal of finding the best multiple hypothesis testing method for a set of attributes in the input data sets that may be more or less correlated with one another, the pipeline generates ground truth data consisting of attribute pairs that are *truly* related and other attribute pairs whose

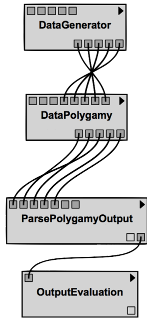


Figure 10: VisTrails pipeline for Data Polygamy Framework

correlations may be spurious. Table 3 lists these parameters and their data types.

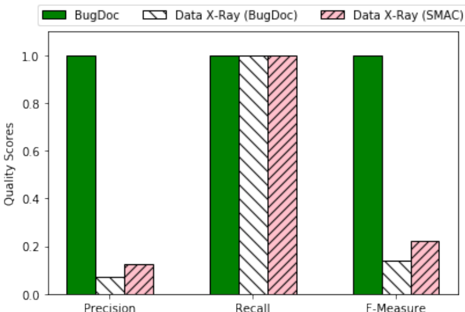


Figure 11: Data Polygamy case study. BugDoc and Data X-Ray using the instances generated by BugDoc and SMAC all find minimal definitive root causes as it is shown by the perfect recall. However BugDoc enjoys far higher precision.

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 which execute to completion, we want to find at least one minimal set of parameter-values or combinations of parameter-values which cause the execution to crash. Our experiments in fact found that certain parameter-values are root causes individually or in combination. These were manually investigated to assess their soundness, and four root causes were identified as follows:

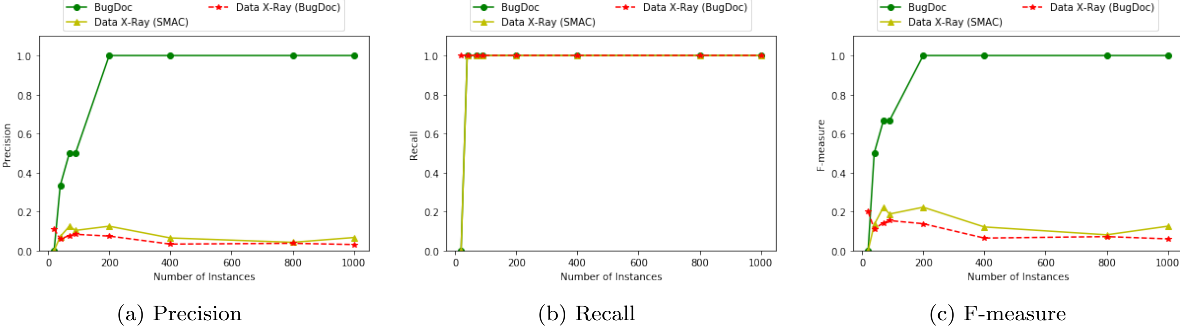
R.1 $\mathit{Percentage} < 0$

R.2 $\mathit{Percentage} > 50$

R.3 $\mathit{Diff} < 0$ and $\mathit{Percentage} \geq 0$

R.4 $\mathit{Diff} > 100$ and $\mathit{Percentage} \geq 0$

Root causes **R.1** and **R.2** were known by the Data Polygamy authors, but it was surprising to them 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 definitive root causes **R.1**–**R.4**, we created ground truth for our case study, which allowed us to compute the previous quality metrics and compare again with Data X-Ray.



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

Figure 12: Data Polygamy case study on a budget. Data X-Ray using the instances generated by BugDoc and SMAC finds all minimal definitive root causes, BugDoc also finds all minimal root causes and increases its precision with more instances.

Figure 11 shows that both BugDoc methods found all the parameter-comparator-value triples that would cause the execution of the pipeline to fail, as can be seen by the metric of recall. As in Section 4.1, Data X-Ray provides not only definitive root causes, but also configurations that do not always yield bad instances – this explain its lower precision.

The same behavior discussed in Section 4.2 was observed this case study: as shown in Figure 12, which shows the results of BugDoc, Data X-Ray using SMAC's instances, and Data X-Ray using BugDoc's instances on different budgets. BugDoc only makes mistakes when there is not sufficient data to characterize the minimal definitive root causes.

5. RELATED WORK

As discussed in Section 1, the problem of generating diagnoses for issues encountered in the results of computations and queries has received a lot of attention in the recent literature [21, 2, 9, 17]. Unlike these approaches, we aim to explain abnormal behavior in computational pipelines that result from bad parameter settings.

Our work combines aspects from hyper-parameter tuning and workflow debugging. Hyper-parameter tuning methods explore the parameter space of computational pipelines to optimize their outcome – they automatically derive instances with improved performance. While they may find good combinations for parameter values, they do not provide any insights into which other combinations may lead to problems or bad performance. In contrast, prior work on workflow debugging aims to identify and explain problems based on existing provenance, but they do not iteratively derive and test new workflow instances. As we demonstrated in Section 4, BugDoc derives good explanations starting the debugging process from scratch and generating fewer workflow instances than hyper-parameter optimization frameworks.

Hyperparameter Optimization. Scientific pipelines often entail the use of hyper-parameters such as cutoff thresholds for p-values and for variances, to name a few examples. Bayesian optimization methods are considered the state of the art for the hyperparameter optimization problem [5, 7, 18, 19, 12]. They can outperform manual setting of parameters, grid search or random search [6]. These methods approximate a probability model of the performance outcome given a parameter configuration that is updated from a history of executions. Gaussian Processes and Tree-structured Parzen Estimator are examples of probability models [5] used to optimize an unknown loss function using the 'expected improvement' criterion as acquisition function. To do this, they assume the search space is smooth and differentiable. This assumption, however, does not hold in general for arbitrary pipelines.

Recent projects modeled the database parameter tuning problem as a Bayesian optimization problem. OtterTune [20] is a system that uses supervised learning techniques to find optimal settings of DBMS knobs given a database workload and a set of metrics (optimization functions). OtterTune uses lasso regularization to select the most important knobs. Dalibar et al. [11] proposed BOAT, another auto tuning system that optimizes DBMS parameters configurations using Bayesian Optimization. However, instead of starting the optimization with a blank Gaussian process, it allows the user to input an initial probabilistic model that exploits previous knowledge of the problem, what is called Structured Bayesian Optimization. BOAT also combines parametric and non-parametric approaches to build probabilistic models, the latter are used to learn the difference between the former and the observed data.

At first glance, the workflow debugging problem could be viewed as the task of finding parameter configurations that minimize the occurrence of poor results. In that way, the above work could be applied. Unfortunately, in the debugging setting, certain data sets or parameter settings simply stop a workflow from working properly. There is no justification for any smoothness or other statistical assumptions.

Debugging and Predicting Pipelines. Previous work on pipeline debugging has focused on analyzing execution history with the goal of identifying problematic parameter settings or inputs. Because they do not use an iterative approach to derive new instances (and associated provenance), they can miss important explanations and also derive incorrect one. Nonetheless, to a large extent our work can be combined with these approaches.

Bala and Chana [3] applied several machine learning algorithms (Naïve Bayes, Logistic Regression, Artificial Neural Networks and Random Forests) 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 try to find the root causes of failure. The system developed by Chen et al. [8] identifies problems by differentiating between provenance (encoded as trees) of good runs and bad

Table 3: Data Polygamy pipeline parameters

Parameter	Description	Type
Diff	This parameter governs the maximum difference allowed between a set of time series from the original time series from which they are generated. The larger this value, the more different they can be. This is used to compare different multiple hypothesize testing methods.	Integer
Percentage	Percentage of true relationships among pairs of time series attributes in the ground truth data.	Integer
Varying Size	If True, then the time series may have different lengths. If False, then they all have same lengths.	Boolean
Time Resolution	The temporal resolution of data: daily, weekly, monthly or random (in which case one of daily, weekly, or monthly is chosen with uniform probability).	String
Permutations	The number of permutations used for a shuffle test to determine the p-value of each correlation relationship.	Integer
Complete	If True, then each shuffle is a random permutation on all the data. If False, then shuffles are constrained to respect certain kinds of temporal locality (e.g. days within spring may only be swapped with other days in the spring)	Boolean
Sync	If True, the same permutations of an attribute are shared among all correlation relationships involving that attribute. If False, permutations are independent among different relationships.	Boolean

ones. They then find differences in the trees that may be the reason for the problems. However, the trees do not provide a succinct explanation for the problems, and it cannot state whether the differences encountered are indeed the root causes. Viska [13] allows users to define a causal relationship between workflow performance and system properties or software versions. It provides big data analytics and data visualization tools to help users to explore their assumptions. Each causality relation defines a treatment (causal variable) and an outcome (performance measurement), the approach is limited to analyze one binary treatment at a time with user in the loop. The Molly system [1] combines the analysis of lineage with SAT solvers to find bugs in fault tolerance protocols for distributed systems. Molly simulates failures, such as permanent crash failures, message loss and temporary network partitions, specifically to test fault tolerance protocols over a certain period of (logical clock) time. The process considers all possible combinations of admissible failures up to a user-specified level (e.g., no more than two crash failures and no more message losses after five minutes). While the goal of that system is very specific to fault tolerance protocols, its attempt to provide completeness has influenced our work. In the spirit of Molly, BugDoc tries to find *minimal definitive root causes.*

Although not designed for computational pipelines, Data X-Ray [21] provides a mechanism for explaining systematic causes of errors in the data generation process. The system finds common features among corrupt data elements and produce a diagnosis of the problems. If we have provenance of the pipeline instances together with error annotations, Data X-Ray's diagnosis would derive explanations consisting of features that describe the parameter-value pairs responsible for the errors. As discussed in Section 4, BugDoc produces explanations that are similar to those of Data X-Ray, but that are also minimal. In addition, BugDoc also employs a systematic method to automatically generate new instances that enable it to derive concise explanations that are root causes for a problem.

6. CONCLUSION

BugDoc combines features from explanation systems with hyperparameter optimization approaches to address one of the most cumbersome tasks for scientists: debugging the computational pipelines they use to encode experiments. It uses two algorithms, the first looks for root causes coming from single parameter-comparator-value triples and the second looks for more complex explanations (disjunctions of conjunctions of parameter-comparator-value triples).

Compared to the state of the art, BugDoc makes no statistical assumptions (as do Bayesian approaches like SMAC), but generally achieves better precision and recall given the same number of pipeline instances. The exceptions occur when there are very few parameters or very pipeline instances in which case Data X-Ray does better given the instances generated by BugDoc. This suggests that a useful strategy would be to have BugDoc generate instances and, if it cannot guarantee a minimal definitive root cause, then a user can use the hypotheses generated by both BugDoc and Data X-Ray. However, when our goal is to find minimal definitive root causes and there are at least several parameters, BugDoc dominates the other methods. Our experiments indicate that our approach is scalable and parallelizes well: pipeline instances can be executed in parallel, thus opening up the possibility of exploring even large parameter spaces. There are several avenues we plan to pursue in future work. First, we would like to make BugDoc available on a wide variety of systems that support pipeline execution and improve its usability. We will also explore the use of cost for different instances in our choice of which pipeline instances to try next.

Acknowledgments. We thank the Data X-Ray authors for sharing their code with us. This work has been supported in part by the U.S. National Science Foundation under grants MCB-1158273, IOS-1339362, and MCB-1412232, the Brazilian National Council for Scientific and Technological Development (CNPq) under grant 209623/2014-4, the Moore-Sloan Foundation, and NYU Wireless.

7. REFERENCES

[1] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *Proceedings of ACM SIGMOD*, pages 331–346, 2015.

[2] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the ACM SIGMOD*, pages 541–556, 2017.

[3] A. Bala and I. Chana. Intelligent failure prediction models for scientific workflows. *Expert Syst. Appl.*, 42(3):980–989, Feb. 2015.

[4] B. Berger, J. Rompel, and P. W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *Journal of Computer and System Sciences*, 1994.

[5] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[6] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

[7] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of ICML*, pages I–115–I–123, 2013.

[8] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou. Data provenance at internet scale: Architecture , experiences, and the road ahead. In *Proceedings of CIDR*, 2017.

[9] F. Chirigati, H. Doraiswamy, T. Damoulas, and J. Freire. Data polygamy: The many-many relationships among urban spatio-temporal data sets. In *Proceedings of ACM SIGMOD*, pages 1011–1025, 2016.

[10] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas. Products of mixed covering arrays of strength two. *Journal of Combinatorial Designs*, 14(2):124–138, 2006.

[11] V. Dalibard, M. Schaarschmidt, and E. Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of WWW*, pages 479–488, 2017.

[12] N. Dolatnia, A. Fern, and X. Fern. Bayesian Optimization with Resource Constraints and Production. In *International Conference on Automated Planning and Scheduling*, pages 115–123, 2016.

[13] H. Gudmundsdottir, B. Salimi, M. Balazinska, D. R. Ports, and D. Suciu. A demonstration of interactive analysis of performance measurements with viska. In *Proceedings of ACM SIGMOD*, pages 1707–1710, 2017.

[14] J. Huang. Programing implementation of the quine-mccluskey method for minimization of boolean expression. *CoRR*, abs/1410.1059, 2014.

[15] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507523, 2011.

[16] D. Jones. Chronemes and tonemes. *Acta Linguistica*, 4:1:11–10, 1944.

[17] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *PVLDB*, 7(13):1715–1716, 2014.

[18] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of NIPS*, pages 2951–2959, 2012.

[19] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, P. Prabhat, and R. P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the ICML*, pages 2171–2180, 2015.

[20] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of ACM SIGMOD*, pages 1009–1024, 2017.

[21] X. Wang, X. L. Dong, and A. Meliou. Data x-ray: A diagnostic tool for data errors. In *Proceedings of ACM SIGMOD*, pages 1231–1245, 2015.

[22] X. Wang, A. Meliou, and E. Wu. Qfix: Diagnosing errors through query histories. In *Proceedings of ACM SIGMOD*, pages 1369–1384, 2017.