BugDoc: A System for Debugging Computational Pipelines

Raoni Lourenço
New York University
raoni@nyu.edu

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

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

Abstract

Data analysis for scientific experiments and enterprises, large-scale simulations, and machine learning tasks all entail the use of complex computational pipelines to reach quantitative and qualitative conclusions. If some of the activities in a pipeline produce erroneous outputs, the pipeline may fail to execute or produce incorrect results. Inferring the root cause(s) of such failures is challenging, usually requiring time and much human thought, while still being error-prone. We recently proposed a new approach that makes provenance to automatically and iteratively infer root causes and derive succinct explanations of failures; such an approach was implemented in our prototype, *BugDoc*. In this demonstration, we will illustrate *BugDoc*'s capabilities to debug pipelines using few configuration instances.

CCS Concepts

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

ACM Reference Format:
Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. *BugDoc*: A System for Debugging Computational Pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384692

1 Introduction

Computational pipelines are widely used in many domains, from astrophysics and biology to enterprise analytics. They are characterized by interdependent modules, associated parameters, and data inputs. Results derived from these pipelines form the basis for conclusions and, often, 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, understand, and correct the causes of errors and unexpected behavior.

Discovering the root cause of failures in a pipeline is challenging because problems can come from many different sources, including bugs in the code, input data, software updates, and improper parameter settings. Connecting the erroneous result to its root cause is especially difficult for long pipelines or when multiple pipelines are composed since these entail cascading dependency chains.

In previous work [14], we proposed and implemented new methods to debug machine learning pipelines that automatically and iteratively identify one or more minimal causes of failures, thereby avoiding the tedious and error-prone task of manually tuning and executing new pipeline instances to test and derive new hypotheses for the failures. We have extended this initial work and built *BugDoc* [15], a system that identifies root causes for errors in general computational pipelines (or workflows).

In this demo, we showcase the capabilities of *BugDoc* by inviting participants to play a game of finding root causes: they will compose configuration instances and compete against our method. We will present comparisons with the instances crafted by human players and those derived by *BugDoc* to demonstrate its benefits. Additionally, participants will have the opportunity to work cooperatively with *BugDoc* on real-world problems.

2 *BugDoc*

Figure 1 shows the high-level architecture of *BugDoc*. Given (i) a computational pipeline description, collection of programs connected together that contains a set of manipulable parameters; (ii) a set of pipeline instances, i.e., provenance that stores 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. In what follows, we give a brief overview of our debugging methodology. For a more detailed discussion, see [14, 15].

Identifying Root Causes. Consider the example in Figure 2, which shows a generic template for a machine learning pipeline and a log of different instances that were run with their associated results. The pipeline reads a data set,

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

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384692

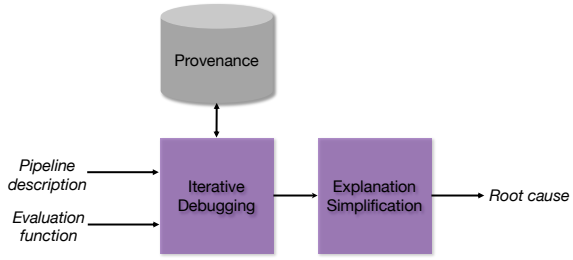

Figure 1: *BugDoc*: overview of the approach.

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

Figure 2: Machine learning pipeline and the provenance of multiple runs.

splits it into training and test subsets, creates and executes an estimator, and computes the F-measure score using 10-fold cross-validation. A data scientist uses this template to understand how different estimators perform for different types of input data, and ultimately, to derive a pipeline instance that leads to high scores.

Analyzing the provenance of the runs, we can see that *gradient boosting* leads to low scores for two of the data sets (*Iris* and *Digits*), but it has a high score for *Images*. By contrast, *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 for some parameters, that *decision trees* provide a good compromise for different data, and that *logistic regression* is good for the *Iris* data. Because each of these runs used different parameters for each method depending on the data set, a definitive conclusion has to await additional testing of these hyperparameters. But doing so manually is time-consuming and error-prone.

Provenance of pipeline execution can help users derive *hypotheses* for the causes of the observed behavior and provide hints for debugging. In fact, provenance has been used to explain errors in computational processes that derive data [8, 19]. to predict whether a pipeline will fail [3], and to identify the cause of problems by computing the differences between good and bad runs of a pipeline [6]. However, as this example illustrates, to test hypotheses and derive more complete (and accurate) explanations, new pipeline instances must be executed that vary the different components of the pipeline. But doing this manually is both time consuming and error prone.

Software testing automatically generates test suites for specific purposes, such as for generating new test cases for existing test suites [12]. The goal is to determine whether bugs are present, not what causes them. Techniques for statistical debugging [13, 20] and bug localization [1, 2, 10] aim to identify the bugs, but they are often application-specific or require a user-defined test suite.

Efficient techniques have also been proposed to optimize hyperparameters in machine learning methods [4, 5, 7, 17, 18]. However, while these identify a *good* set of parameter values, they do not derive explanations for why the values are selected. Also, existing approaches make assumptions (e.g., smoothness) about the distribution of parameter values that do not hold in our setting, since we are dealing with arbitrary computations and not learning a classifier.

BugDoc automates the process of deriving root causes for problems in pipelines. It does so without any knowledge of the internal code of the computational processes: it views pipelines as black boxes and observes only their inputs and outputs. Trying all possible combinations of parameters and values leads to a combinatorial explosion of pipeline executions and is thus not a practical solution. In addition, causes for errors can include multiple parameters, each of which may have large domains. *BugDoc* uses an iterative strategy that is provably efficient (possibly linear in the number of parameters) and provides concise explanations.

Iterative Debugging. *BugDoc* combines two iterative debugging algorithms. The first, called *Shortcut*, discovers definitive root causes (or bugs) consisting of a single parameter-value (formally, parameter-equality-value) or a single conjunction of parameter-values. The second, called *Debugging Decision Trees*, discovers more complex definitive root causes involving multiple parameters and possibly inequalities.

In operation, *BugDoc* first runs the *Shortcut* algorithm. *Shortcut* applies heuristics to select and test combinations of parameter-value pairs. Under reasonable assumptions, it 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 *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, that lead to `fail`. Those will be our suspects. For that reason, we build a complete decision tree, i.e., no pruning. The main advantage of *Debugging Decision Trees* is that it can identify root causes that depend on inequalities.

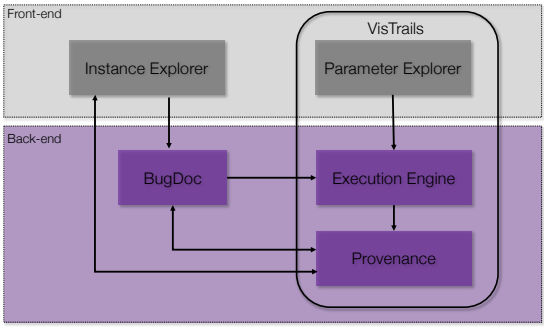

Figure 3: Demo Prototype: BugDoc combined with the open-source VisTrails system.

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 [11].

3 Demonstration Description

The goal of our demo is to allow users to compare a manual approach to debugging with *BugDoc*'s iterative approach. To do so, we will first present a training example similar to the pipelines described by Figure 2, where we will explain the operations the participants can perform: Parameter exploration (Section 3.1), and instance exploration (Section 3.2). Then we will invite the participants to, initially, compete against (Section 3.3), and later cooperate with (Section 3.4) *BugDoc* to debug pipelines, displaying all instance configurations tried and their result.

For this demo, *BugDoc* was integrated with the VisTrails system [9] in order to take advantage of its provenance capabilities and its interface for parameter exploration, as shown in Figure 3.

3.1 Parameter Exploration

VisTrails provides a parameter exploration interface that allows the user to explore the parameter space by binding parameters to a range of values. This bulk operation makes it possible to generate and run several different instances automatically. Our interface re-uses two of the main components of Parameter Explorer: *canvas* and *parameter list*.

Input parameters that may be set during a pipeline execution appear in the parameter list (Figure 4 top right) and they can be dragged and dropped into the canvas (Figure 4 left). The pipeline view (Figure 4 right-bottom) indicates which modules each parameter belongs to.

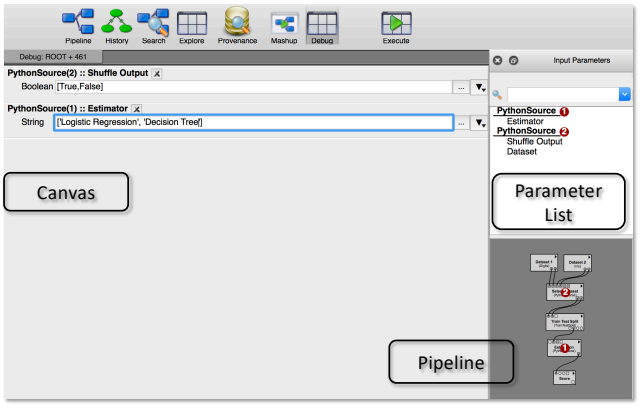

Figure 4: Parameter Exploration in VisTrails.

Once the parameters are on the canvas, the participants may adjust their range or specify a list of values to be explored. This operation creates a space consisting of the Cartesian product of the values for each parameter. This space will be used in the scenarios in Sections 3.3 and 3.4.

3.2 Instance Exploration

By exploring pipeline instances, a user can obtain information that helps in the formulation of hypotheses about the root causes of failure. *BugDoc* displays the instances in a table, as shown in Figure 5. Using this table, participants can inspect the different executed instances and their outcome, i.e., the results of the evaluation function. Configurations can be sorted and filtered by parameter-values, results (`success` or `failure`), and also by origin, a field that identifies the cohort of the instances (i.e., if they were derived by the same parameter exploration/user or generated by *BugDoc*).

3.3 Competitive Scenario

We create artificial problems based on the crime mystery game of Clue (or *Cluedo*) [16]. For each problem, there is a list of categorical variables, representing the parameters of a pipeline, and a conjunctive subset of variable-value pairs that are necessary and sufficient for a crime to happen, representing the root cause of a failure in the pipeline.

Participants will be asked to try to solve a problem (sampled at random) using fewer instances than *BugDoc*. They

origin	Estimator	Library Version	Dataset	result
demo_				
demo_user_2	Logistic regression	2.0	Iris	Failure
demo_user_2	Decision tree	1.0	Iris	Success
demo_user_2	Decision tree	2.0	Iris	Failure
demo_user_3	Logistic regression	1.0	Iris	Success
demo_user_3	Logistic regression	1.0	Digits	Success

Previous Page 5 of 12 5 rows Next

Figure 5: Instance Exploration.

will create sets of instances through the VisTrails parameter exploration interface and run them by clicking on the Execute button in VisTrails toolbar (Figure 4 top). All executed combinations are shown in the instances exploration table indicating if they led to a crime or not, so the participant can hypothesize. At any point, one is able to guess a root cause by inputting a textual conjunctive clause of parameter-value pairs.

When a participant guesses, we run *BugDoc* allowing it to test at most the same number of instances tried so far, the configurations generated by *BugDoc* are added to the table, and its answer is revealed. *BugDoc* will have no knowledge of the instances created during the manual exploration. If neither the debuggers have found the minimal answer, the game can continue, or we just let *BugDoc* run until the end.

3.4 Cooperative Scenario

After competing against *BugDoc*, participants will be invited to work cooperatively with it in order to debug real-world pipelines consisted of larger parameter spaces, containing categorical values and real numbers, and possibly presenting inequalities in their root causes of failure.

The parameter explorer will be used to prune the search space that *BugDoc* is allowed to test, distinctly from the previous scenario, when the Execute button is hit, our algorithms will choose which instances to try next (observing the value constraints) instead of a Cartesian product. Participants may decide the maximum number of instances *BugDoc* can create in each iteration; the selected instances are then executed and added to the instances exploration table.

BugDoc will output the best hypothetical root causes it is able to formulate with the given budget if the minimal definitive root cause of the real-world problem is not found the participant will be able to redefine the parameter space and continue the debugging process.

4 Conclusions

In this demonstration, we showcase the ability of *BugDoc* to find minimal definitive root causes in computational pipelines or workflows, either autonomously or cooperatively. *BugDoc* analyzes previously executed computational pipeline instances, selectively executes new pipeline instances, and finds minimal explanations. This proposal has presented an overview of the design principles behind the *Shortcut* and *Debugging Decision Trees* algorithms that have been shown, in our prior work [14], to outperform the state of the art in both number of instances required and F-measure. Our demo allows participants to compete or cooperate with *BugDoc*.

Acknowledgments. This work has been supported in part by NSF grants MCB-1158273, IOS-1339362, and MCB-1412232, CNPq (Brazil) grant 209623/2014-4, the DARPA D3M program, and NYU WIRELESS. 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] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of USENIX OSDI*. 307–320.

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

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

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

[5] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of ICML*. 115–123.

[6] Ang Chen, Yang Wu, Andreas Haeberlen, Boon T. Loo, and Wenchao Zhou. 2017. Data Provenance at Internet Scale: Architecture , Experiences, and the Road Ahead. In *Proceedings of CIDR*. 1–7.

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

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

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

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

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

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

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

[14] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2019. Debugging Machine Learning Pipelines. In *Proceedings of DEEM*.

[15] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. BugDoc: Algorithms and a System to Debug Computational Processes. In *Proceedings of ACM SIGMOD*.

[16] Leon Petrosjan and Vladimir V Mazalov. 2007. Description of Game Actions in Cluedo. In *Game theory and applications*. Vol. 11. 1–28.

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

[18] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. 2015. Scalable Bayesian Optimization Using Deep Neural Networks. In *Proceedings of the ICML*. 2171–2180.

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

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