The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Differential Performance Debugging with Discriminant Regression Trees*

Saeid Tizpaz-Niari, Pavol Černý,
Bor-Yuh Evan Chang, Ashutosh Trivedi
University of Colorado Boulder

Abstract

Differential performance debugging is a technique to find performance problems. It applies in situations where the performance of a program is (unexpectedly) different for varying classes of inputs. The task is to explain the differences in asymptotic performance among various input classes in terms of program internals. We propose a data-driven technique based on *discriminant regression tree* (DRT) learning problem where the goal is to discriminate among different classes of inputs. We propose a new algorithm for DRT learning that first clusters the data into functional clusters, capturing different asymptotic performance classes, and then invokes off-the-shelf decision tree learning algorithms to explain these clusters. We focus on linear functional clusters and adapt classical clustering algorithms (K-means and spectral) to produce them. For the K-means algorithm, we generalize the notion of the cluster centroid from a point to a linear function. We adapt spectral clustering by defining a novel kernel function to capture the notion of "linear" similarity between two data points. We evaluate our approach on benchmarks consisting of Java programs where we are interested in debugging performance. We show that our algorithm outperforms other well-known regression tree learning algorithms in terms of running time and accuracy of classification.

1 Introduction

Developers often face the problem of finding and fixing performance problem in their programs. Performance bugs manifest themselves only on certain pathological inputs. For instance, there can be two inputs of the same size on which the performance is unexpectedly different in an otherwise functionally correct program.

We study the *differential performance problem*, where the goal is to explain the difference in performance between two classes of inputs in terms of program internals, such as which functions were called and how many times were they called. This information is useful, as it allows a programmer or an analyst to better assess whether the performance difference is inherent to the problem, or is a result of a coding inadequacy. The problem is hard for both traditional static as well as dynamic analysis techniques. Static analysis commonly target logical correctness properties (and not the performance), and are not as scalable as techniques based on machine learning. On the other hand, dynamic analysis techniques such as profiling, focus on individual traces, whereas for the differential performance problem, we need to compare the performance on different traces.

We propose a technique called *differential performance debugging*, based on inference of *discriminant regression trees* (DRTs). DRTs are regression trees where the goal is to classify input data. In contrast, the objective of standard regression tree learning is to predict the output for a previously unseen input. The input to the differential performance problem is a set of program traces. Each trace is represented as follows. We have input variables (such as the size of the user input), auxiliary variables (such as the functions called), and the output variable (such as the running time). The output to the differential performance problem is the DRT. The internal nodes of the tree has predicates on auxiliary variables. The leaf nodes model the output variable as a function of input variables. The leaf nodes represent the performance for different classes of inputs capturing asymptotically different performance behaviors.

In accordance with Occam's razor, we are interested in finding a DRT with a small number of clusters, while minimizing the modeling error. Furthermore, the DRT should be a human readable explanation, which also suggests that smaller number of clusters is preferable. There are two major steps in our algorithm. First, we project the data into the input and output variables and cluster the data in this domain. Second, we consider the auxiliary variables only and identify what separates the clusters in terms of these variables. We use an off-the-shelf decision tree learning algorithm for the second step. The first, clustering, step thus reduces the regression tree inference problem to the decision tree inference problem.

For our approach, we need a clustering algorithm that produces *functional* clusters, that is, clusters that represent functions from input variables to the output variable. We adapt two classical clustering algorithms. First, we extend the K-means algorithm to produce linear functional clusters. This is done by generalizing the notion of the cluster centroid from a point to a linear function. Second, we adapt the spectral clustering algorithm by defining a new notion

*This research was supported in part by DARPA under agreement FA8750-15-2-0096.
Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

of similarity between two data points that we dub *alignment* kernel. Here, two data points are more similar when the line defined by them captures more data points.

The key contributions of this paper are:

- We propose *discriminant regression trees* which are regression trees where the goal is to classify input data into a small number of clusters.
- We give a new algorithm for learning discriminant regression trees. It finds (functional) clusters first, which enables learning the tree using an efficient algorithm for learning decision trees.
- We present extensions to two classical clustering algorithms: k-means and spectral clustering. These extensions allow us to obtain functional clusters.
- We implement our approach in the tool DPDEBUGGER and evaluate it on benchmarks consisting of a suite of Java programs. Our experiments that the approach is scalable and is able to explain the differences in performance between different classes of inputs.

2 Overview

We show how our prototype tool DPDEBUGGER can be used for diagnosing performance problems on a real-life example. We also use the example to explain how the tool works and compare it to existing approaches.

Performance problem with Apache FOP. Apache FOP (Formatting Objects Processor) is a Java application that reads a formatting object such as an XML file and renders the resulting pages to a specified output format such as PDF and PS. The formatting document can specify that an external image in, for example, a PNG or JPEG format should be included. A user had a suspicion that there is a performance bug in handling PNG images. They reported in a forum post in 2011 that they have two PNG images, which have the same size, but one of them takes seven times as much to render as the other one[1].

Performance debugging with DPDEBUGGER. Our tool, DPDEBUGGER, can be used exactly in this situation, to help an analyst to explain the differences in performance. The analyst can then decide whether the differences are inherent to the problem or they are a manifestation of a coding error. Song and Lu (Song and Lu 2014) reported that in 60% of bugs analyzed by them, users notice huge performance differences among similar inputs.

The analyst has to collect a number of inputs which in this case are PNG and JPEG images of various sizes. We remark that in many cases, the inputs can be collected from log files of a system or generated by existing software fuzzers (Cadar et al. 2008).

Given the collection of inputs, the tool produces the two diagrams in Figure 1. The analyst can diagnose the performance problem using these two figures. The left diagram is a plot of the image size (input variable) and the running time (output variable). From the plot, the analyst can see that there are two performance clusters. However, the analyst does not know what separates these two clusters. It is instructive to emphasize that looking at the two groups of

[1] https://bz.apache.org/bugzilla/show_bug.cgi?id=51465

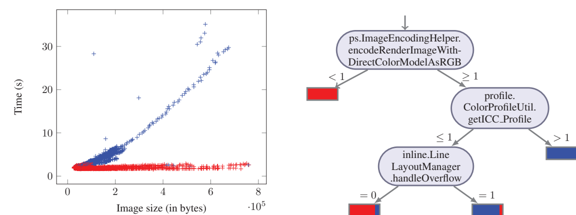


Figure 1: Performance clusters (left) in the FOP data set and a DRT (right) explaining the the clusters in terms of function calls.

inputs does not explain the difference. All the JPEG images are in the lower (red) cluster, but there are PNG images of similar size in both clusters.

We thus turn to the DRT on the right side of Figure 1 for the explanation. It says that if for an input, the function `encodeRender...RGB` is not called, then the input will be in the red cluster. The user can analyze the reverse call graph to see how the function is called, and realize that it is called for PNG files, but not for JPEG files. Further, the node to the right of the root has the function `getICCprofile`. This function is what distinguishes the red (fast) cluster from the blue (slow) one among PNG files. It is called once for every PNG file, but it is called more than once only for PNG files with a color scheme that needs to be deflated. After code analysis, we see that one source of the performance problem is that some PNG files have a compressed color scheme that needs to be deflated. Another way how a PNG image can be in the blue cluster is that the dimension of the input image overflows the allowed size (see the lowest internal node).

The user thus learns from the discriminant regression tree (DRT) that what separates the two clusters is the fact that the images in the blue clusters either need to be deflated first, or overflow. So in this case, the diagnosis is that the difference in performance is not a coding mistake but it is inherent to the problem. The user can conclude this without needing to sift through almost 40,000 methods that Apache FOP has.

Inside the tool DPDEBUGGER. We now describe how DPDEBUGGER obtains the diagrams in Figure 1. The diagram on the left side is obtained as follows. The program is run on all the inputs, and the graph of input size versus running time is plotted. Then we need to cluster the data. As a preliminary step, we need to get the values of the auxiliary variables. In this example, they indicate how many times a function was called. We have a variable for each (non-standard library) function. Then, we find functional clusters in the data from the left diagram. That is we consider input and output variables only (not auxiliary variables) and identify a given number K of clusters. These clusters are intended to capture the classes of inputs with different (asymptotic) performance. For our example, spectral clustering identifies the clusters marked red and blue in Figure 1.

To obtain the right diagram, the next step is decision tree learning. We want to learn what separates the clusters in terms of predicates on the auxiliary variables. Note that here

each data point is labeled with one of the K labels (indicating to which cluster the data point belongs), therefore efficient decision tree learning (such as CART) can now be used to construct the regression tree. The final discriminant regression tree is on the right part of Figure 1.

Comparison with existing regression-tree algorithms. We applied the state-of-the-art algorithms for learning regression trees (M5Prime (Witten et al. 2016) and GUIDE (Loh 2002)) to our problem. Our goal is different from the goal of these algorithms: we aim to classify data, whereas both Guide and M5Prime aim to predict the output for previously unseen input. We believe that this accounts for the following differences.

M5Prime finds a linear regression tree with 23 different linear models in leafs. Guide finds 4 linear models – two of them same as our algorithm, but two of the in-between, perhaps to account for noise. Note that DPDEBUGGER identified 2 clusters. Furthermore, the running time of the algorithms are as follows: M5Prime 97 seconds, Guide 1233.6 seconds, and our DPDEBUGGER 14.4 seconds.

3 Discriminant Regression Tree Learning

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \in \mathbb{R}^n$ be the set of input variables, $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_m\} \in \mathbb{R}^m$ be the set of auxiliary variables, and $\mathbf{y} \in \mathbb{R}$ be the performance (observable output) variable of our target program. To keep the presentation simple, we assume that there is a single performance variable $\mathbf{y}$, although techniques presented in the paper can easily be extended to include a set of performance variables representing time-series data on various performance measures such as time and memory.

An *execution trace* T of the program is a tuple $\langle X, Z, y\rangle$ wherein $X = \langle x_1, x_2, \ldots, x_n\rangle$, $Z = \{z_1, \ldots, z_m\}$, and y represent the valuations to the input, auxiliary, and output variables, respectively. We further assume that the valuations of the auxiliary variables deterministically depend only on valuations of the input variables. However, we allow the process of measuring performance to be noisy. Due to this we can potentially have multiple traces of the program with the same values for the input and the auxiliary variables but different values for the performance variable.

A *trace discriminant* is defined as a disjoint hyper-rectangular partitioning of the space of auxiliary variables along with an affine function for each partition modeling the performance variable as a function only of input variables. Formally, a trace discriminant $\Psi = (\mathcal{F}, P)$ is a set of affine functions $\mathcal{F} = \langle f_1, f_2, \ldots, f_K\rangle$—where each $f_j : \mathbb{R}^n \to \mathbb{R}$ models the performance variable y as a function of the input variables—and a hyper-rectangular partition $P = \langle(\phi_1, d_1), (\phi_2, d_2), \ldots, (\phi_l, d_l)\rangle$ where each $\phi_i : \mathbb{R}^m \to \{\mathtt{true}, \mathtt{false}\}$ is a hyper-rectangular predicate over auxiliary variables $\mathbf{Z}$, and each $d_i : \mathcal{F} \to [0, 1]$ is discrete probability distribution over $\mathcal{F}$ giving a probabilistically weighted modeling of the affine functions from $\mathcal{F}$. The size $\mathtt{size}(\Psi)$ of a discriminant Ψ is defined as total number of affine functions (i.e., $\mathtt{size}(\Psi) = |\mathcal{F}|$). Given a trace $T = \langle X, Z, y\rangle$ and a discriminant $\Psi = (\mathcal{F}, P)$, we define the prediction error as $\epsilon(T, \Psi) \stackrel{\text{def}}{=} \left(y - \sum_{j=1}^{K} d_i(f_j) \cdot f_j(x_1, x_2, \ldots, x_n)\right)^2$, where $1 \leq i \leq l$ is the index of the unique partition in P such that $Z \models \phi_i$ (i.e., the predicate ϕ_i evaluates to true for the valuation Z). Given a set of traces $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$, and a discriminant Ψ, we define the fitness of the discriminant as *m*ean-squared-error $\mu(\mathcal{T}, \Psi) = \frac{1}{N}\sum_{i=1}^{N} \epsilon(T_i, \Psi)$.

Given a set of traces $\mathcal{T}$, a bound on the size of the discriminant $B \in \mathbb{N}$, and a bound on the error $B_\varepsilon \in \mathbb{R}$, the *discriminant learning problem* is to find a discriminant Ψ with $\mathtt{size}(\Psi) \leq B$ and $\mu(\mathcal{T}, \Psi) \leq B_\varepsilon$. It follows from Theorem 1 in (Alur and Singhania 2014) that the discriminant learning problem is NP-HARD. For this reason, we use heuristics to construct discriminant using classification and regression trees.

A *discriminant regression tree* is a trace discriminant represented as a binary tree structure whose nodes contain predicates over auxiliary variables and leaves contain a discrete probability distribution over affine functions in input variables. An example of a distribution regression tree is shown in Figure 1 where each leaf represents a partition, and the probability distribution over functions is pictorially depicted using relative sizes of different colors.

Classical regression tree algorithms can be used to learn the discriminant regression trees. The most straightforward way to generalize the decision tree algorithm to learn regression trees is computationally expensive (Loh 2011) as it requires solving two linear regression problems for each split candidate. Popular regression tree algorithms algorithms CART (Breiman et al. 1984), M5Prime (Witten et al. 2016), GUIDE (Loh 2002) propose various ways to avoid this problem. CART is a piecewise constant regression tree model that uses the standard regression-tree algorithm (with piecewise constant clusters) and then applies cross-validation to prune the tree. M5Prime (Witten et al. 2016) algorithm first constructs a piecewise constant model, and then fits linear regression models to leaves during pruning step. GUIDE re-

Algorithm 1: LEARNDISCRIMINANTREGRESSIONTREE($\mathcal{T}, B, B_\varepsilon$)

Input: A set of traces $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$, an upper bound B on discriminant size, and a bound on mean-squared error B_ε.

Output: Return a discriminant regression tree of size B and error bound B_ε, if possible. Otherwise return NULL.

1. Extract points $\mathcal{E} = \{\langle X_1, y_1\rangle, \langle X_2, y_2\rangle, \ldots, \langle X_N, y_N\rangle\}$ from the trace set $\mathcal{T} = \{T_1, \ldots, T_N\}$ where $T_i = \langle X_i, Z_i, y_i\rangle \in \mathcal{T}$.
2. Using linear clustering algorithms presented in the next section, find the smallest number B' of linear clusters $\mathcal{F} = \left\{f_1, \ldots, f_{|B'|}\right\}$ that can fit the data with mean-squared error smaller than B_ε.
3. **if** $B' > B$ **then return** NULL
4. **else**
5. Extract points $\mathcal{E}' = \{\langle Z_1, \ell_1\rangle, \ldots, \langle Z_N, \ell_N\rangle\}$ where $\ell_i \in \mathcal{F}$ is the label assigned by the clustering algorithm to the dataset $\langle X_i, y_i\rangle$.
6. Use a standard decision tree algorithm to learn a decision tree (along with its accuracy based on k-fold cross-validation) from $\mathcal{E}'$.
7. Return the discriminant regression tree (along with its accuracy) by replacing labels at the leaves with corresponding linear functions.

gression tree algorithm (Loh 2002), at each node, fits the best regression model that predicts the response variable and computes the residual. Then, it adds different class labels for traces with negative and positive residuals and solves classification problem to find the auxiliary variable to split over.

In our setting (where the goal is classification and a tight upper bound on the number of linear clusters is known), we propose a simple but rather effective method to overcome the complexity of repeatedly fitting the piecewise linear model. Our approach is summarized as Algorithm 1. Our approach is to first cluster traces along the lines based only on input and output variables, and then assign different labels to various traces based on the linear clusters into which they fall. The next step is to learn a classification decision tree in auxiliary variables with the leaves as clusters labels (classes) learned in the first step. Using a set of microbenchmarks related to performance debugging, in Section 5 we show that our algorithm performs better than other regression tree algorithms specially when the relationship between auxiliary variables and linear clusters is complex.

Algorithm 2: K-LINEARCLUSTERINGALGORITHM

Input: Data $\mathcal{E} = \{\langle X_1, y_1\rangle, \ldots, \langle X_N, y_N\rangle\}$ and number of clusters K.

Output: A partition of the set of traces $\mathcal{E}$ in K sets $\langle S_1, S_2, \ldots, S_K\rangle$.

1 Let $S^{(1)} = \langle S_1^{(1)}, S_2^{(1)}, \ldots, S_K^{(1)}\rangle$ be an arbitrary partition of the points $\mathcal{E}$.
2 Set i to 0
3 **repeat**
4 Set i to $i+1$
5 For each set $S_j^{(i)} = \{\langle X_P, y_P\rangle\}$ where $\langle X_P, y_P\rangle$ is a set of points assigned to partition S_j at i-th iteration, learn a linear function $f_j = \mathbf{A}_j\mathbf{x} + c_j$ minimizing $\sum_{p\in P}(y_p - \mathbf{A}_j X_p - c_j)^2$.
6 Compute $S^{(i+1)} = \langle S_1^{(i+1)}, S_2^{(i+1)}, \ldots, S_K^{(i+1)}\rangle$ such that for each $1 \leq j \leq k$ we have $\langle X_P, y_P\rangle \in S_j^{(i+1)}$ if $(y_p - \mathbf{A}_j X_p - c_j)^2 = \min_{1\leq h\leq k}(y_p - \mathbf{A}_h X_p - c_h)^2$, with condition that $\langle X_p, y_p\rangle \in S_j^{(i)}$ and $\langle X_p, y_p\rangle \in S_h^{(i+1)}$ for $j \neq h$ implies

$$(y_p - \mathbf{A}_j X_p - c_j)^2 > (y_p - \mathbf{A}_h X_p - c_h)^2 \quad (1)$$

7 **until** $S^{(i)} = S^{(i+1)}$
8 **return** clusters $S^{(i)}$ and linear "centroids" $\langle f_1, f_2, \ldots, f_K\rangle$.

4 Linear Clustering Algorithms

In this section, we study K-linear clustering problem required at the clustering step of Algorithm 1. Consider the set $\mathcal{E} = \{\langle X_1, y_1\rangle, \ldots, \langle X_N, y_N\rangle\}$ of data points where $X_i \in \mathbb{R}^n$ is an n-dimensional vector of valuations to the input variables and $y_i \in \mathbb{R}$ is the value of the output (response) variable. Given the number of desired clusters K, the *K-linear clustering problem* asks to compute a partition of set $\mathcal{E}$ into K clusters $S = \langle S_1, S_2, \ldots, S_K\rangle$ minimizing the residual-sum-of-squares (RSS) defined as $\sum_{i=1}^{K}\sum_{\langle X,y\rangle\in S_i}\min_{f\in\mathcal{F}}(y - f(X))^2$, where f is a linear function $f : \mathbb{R}^n \to \mathbb{R}$ over input variables in the form of $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{c}$ with $\mathbf{x}$ as an n-dim. vector and $\mathbf{c}$ as a scalar.

Regarding the computational complexity of the *K-linear clustering problem*, observe that for a given cluster S, the RSS can be computed using (least squares) linear regression in polynomial time (linear in the number of points and quadratic in input dimension). Since, there are only finitely many (K^N) distinct clusters possible, the K-linear clustering problem is decidable. The NP-hardness of K-linear clustering problem follows from NP-hardness of K-means clustering problem which is known to be NP-hard both for general dimensions and 2 clusters (Aloise et al. 2009), as well as 2 dimension and K clusters (Mahajan, Nimbhorkar, and Varadarajan 2009). For this reason, we present two heuristics to solve K-linear clustering problem. The first algorithm, which we call "K-linear" clustering, extends K-means algorithm by using line centroids instead of point centroids, while the second algorithm is based on spectral clustering with a new notion of measuring similarity between points in order to detect functional relationships.

4.1 K-Linear Clustering

We propose a modification of the standard K-means clustering algorithm to give a heuristic to solve K-linear clustering problem as shown in Algorithm 2. The termination of our algorithm is guaranteed as the number of distinct clusters possible are finite, and in each step we get a strict improvement in residual-sum-of-squares due to the restriction (1) in Algorithm 2 on changing the set only in the case of a strict improvement. However, similar to the K-means algorithm, there is no guaranteed convergence to a global optimum. The choice of initial partition to fit linear "centroids" is crucial in converging towards the global optimal solution. One way to choose a good partition is to pick lines defined by pairs of points such that ϵ-size tubes around the lines pass through a large number of points. Another possible heuristic to achieve better partition is similar to that often seen with K-means algorithm—we execute the K-linear algorithm a couple of times with randomly selected initial partitions, and then we choose the result that gives minimum RSS.

4.2 Spectral Clustering with ALIGNMENT Kernel

Spectral clustering is a popular clustering algorithm that views the clustering data as a weighted graph of points and the clustering problem as a graph partitioning problem. Spectral clustering algorithms are parameterized by the notion of adjacency between two data points defined using *kernel functions*. Spectral clustering is useful in clustering problems where the measurement of the center and the spread of cluster are not a suitable description of clusters (Von Luxburg 2007).

In order to define the notion of adjacency in terms of being close to a given linear cluster, we characterize a novel kernel function—called *alignment kernel*—that puts two points closer to each other if the line passing through those points have multiple other points in the line's neighborhood. The concept of alignment kernel is shown in the figure 2 where points A and B are closer to each-other in linear sense than points A and C, although the latter points are closer than former points in terms of Euclidean distance. Given a data set $\mathcal{E}$, we define an *alignment kernel function* $\alpha_{\mathcal{E}}^{\Delta}$:

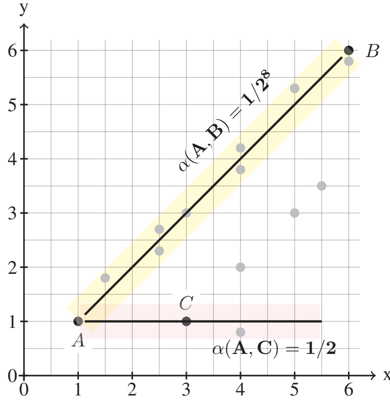


Figure 2:

$(\mathbb{R}^n \times \mathbb{R}) \times (\mathbb{R}^n \times \mathbb{R}) \to \mathbb{R}$ to be a real-valued symmetric and non-negative function defined as the following for every pair of neighboring points $\langle X_i, y_i \rangle$ and $\langle X_j, y_j \rangle$:

$$\alpha_{\mathcal{E}}^{\Delta}(\langle X_i, y_i\rangle, \langle X_j, y_j\rangle) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i = j \\ 2^{-|R_{ij}|} & \text{if } |R_{ij}| >= 1 \\ \infty & \text{otherwise,} \end{cases}$$

where R_{ij} is the set of points $\langle X_r, y_r \rangle$ in $\mathcal{E}$ such that $r \neq i \neq j$, and it has Δ distance from the line passing through $\langle X_i, y_i \rangle$ and $\langle X_j, y_j \rangle$. Finally, we construct similarity matrix by calculating $\mathrm{e}^{-\alpha_{\mathcal{E}}^{\Delta}(A,B)}$ for every pair of points A and B. Observe that the exact computation of the alignment matrix is cubic in number of data points. However, we have implemented a quadratic procedure (see supplemental material in (Tizpaz-Niari et al.)) computing an approximation of the the alignment kernel. In our experiments, we have found that for linear clusters the quality of the alignment kernel is better than the RBF and the nearest-neighborhood kernels. In comparison with the K-linear clustering, spectral clustering with alignment kernel can often detect non-linear clusters.

5 Microbenchmark Results

Empirical evaluation questions. We compare our approach to existing algorithms for learning regression trees and focus on the following questions. 1. *How deep are the regression trees and how many leaves do they have?* As we expect our regression trees to serve as human readable explanations, smaller height and number of leaves are better. 2. *How scalable is our approach compared to the existing approaches?* 3. *What is the prediction ability (as measured by coefficient of determination) compared to state-of-the-art approaches?* The metric we are interested in is accuracy of classification based on 10-fold cross-validation. However, the standard approaches are built with the goal of prediction measured by coefficient of determination R^2. We therefore compare performances with R^2.

Synthetic benchmarks. We compared the performance of regression tree learning approaches on a set of microbenchmarks. The benchmarks were constructed in such a way that the clusters have increasingly complex characterizations. We consider micro-benchmarks named R_n#v where n is the number of functions in the benchmark, and v is the version number. For R_2 to R_7, in each case there is a cluster which consists of inputs where all the functions were called. R_200 to R_6400 are versions of the same benchmarks with many other functions that do not influence performance. Each function call executes a for loop statement where the number of iterations depends on the inputs. Each trace of a benchmark invokes a set of functions that lead to the different time of execution. We want to detect clusters and explain them based on function calls.

Results. Table 1 summarizes the results of applying CART, M5Prime, and GUIDE (the standard algorithms) as well as our algorithms with K-linear and spectral clustering.

Our first question is about the simplicity of the explanation produced by our tool. Table 1 shows that CART and M5Prime produce significantly deeper trees with more models than GUIDE and DPDebugger. For instance, for benchmark R_7, CART produces a tree with more than 5000 nodes, M5Prime with more than one node, whereas the benchmark has only 4 clusters.

Our second question is about scalability. As the first question established that CART and M5Prime are unsuitable for our purpose, we compare scalability only with the GUIDE algorithm. Table 2 shows the performance of GUIDE regression tree and DPDebugger with K-linear clustering when there are many features (function calls). We see that DPDebugger is more scalable on this set of benchmarks. For instance, for R_6400, GUIDE takes more than 9 hours, whereas DPDebugger takes less than 5 minutes.

Our third question asks to compare the coefficient of determination R^2. From Table 1, we have the following: CART and M5Prime generally perform well (but there are some outliers where the coefficient of determination drops). The main problem with these algorithms for our purpose is the large size of the regression trees. For GUIDE, the coefficient of determination is lower for more complex examples. DPDebugger performs uniformly well in this metric.

Finally, we compare the two versions of our algorithm: one with K-linear and one with spectral clustering. Table 1 shows that for these benchmarks with linear clusters, they are similar in all metrics except running time, where the K-linear clustering is slightly better.

6 Case Study

Table 4 summarizes the results over eight case studies involving large Java applications, while Table 3 shows clustering and classification steps in DPDebugger for seven of them (the eighth is in Figure 1). We next explain how DPDebugger explained differences in performance for different classes of inputs for these seven applications (Apache FOP has been treated already in Section 2) [2].

1. **Charts4j**. Charts4j is a Java chart library that enables developers to generate different charts available in the Google Chart API. The input data set for our experiments consists of asking for different plots of the same data. Our

[2]see supplemental material for detailed information in (Tizpaz-Niari et al.)

Table 1: Micro-benchmark results for comparison different affine cost model learning algorithms. Legend: **#M**: number of functions, **#N:** number of traces, **T**: computation time in seconds, R^2: coefficient of determination **H**: decision-tree height, **L**: Number of detected models, **A**: accuracy of classification model, $\epsilon < 0.1$ **sec.**

Bench	#M	#N	CART				M5prime				GUIDE				DPDEBUGGER (K-linear)					DPDEBUGGER (spectral)				
			T	R^2	**H**	**L**	**T**	R^2	**H**	**L**	**T**	R^2	**H**	**L**	**T**	**A**	R^2	**H**	**L**	**T**	**A**	R^2	**H**	**L**
R_2	2	400	ϵ	0.99	14	237	3.5	0.99	6	17	ϵ	0.99	2	4	0.7	99%	0.99	2	3	0.2	96%	0.98	2	3
R_3#1	3	800	0.15	0.77	14	486	4.5	0.7	1	1	ϵ	0.99	3	8	1.3	100%	0.99	3	2	0.7	100%	0.99	3	2
R_3#2	3	800	0.14	0.99	15	470	6.9	0.9	8	41	ϵ	0.86	3	6	1.7	100%	0.99	3	3	0.8	99%	0.99	3	3
R_4#2	4	1200	0.2	0.99	14	652	8.8	0.99	7	23	0.2	0.99	3	5	2.9	100%	0.99	4	4	1.6	98%	0.99	4	4
R_4#1	4	1600	0.28	0.99	20	893	9.7	0.99	7	25	0.2	0.99	4	7	3.5	99%	0.99	4	3	3.0	98%	0.99	4	3
R_4#3	4	1600	0.27	0.97	16	955	9.9	0.9	8	87	0.2	0.93	4	11	3.5	99%	0.99	4	3	2.8	99%	0.99	4	3
R_5	5	3200	0.54	0.94	16	1810	17.6	0.71	11	147	0.4	0.73	5	15	6.7	99%	0.99	5	3	11.3	99%	0.99	5	3
R_6	6	6400	1.1	0.99	22	3695	24.1	0.98	12	173	1.1	0.6	4	11	16.3	99%	0.99	6	4	42.5	98%	0.99	6	4
R_7	7	12800	2.4	0.99	32	5126	49.6	0.99	12	142	1.4	0.63	4	14	31.8	97.9%	0.98	7	4	210.1	95.5%	0.97	7	4

Table 2: Micro-benchmark results comparing GUIDE and DPDEBUGGER with dummy function calls.

Benchmark	# M	#N	GUIDE				DPDEBUGGER (K-linear)				
			T	R^2	**H**	**L**	**T**	**A**	R^2	**H**	**L**
R_200	200	400	3.0	0.99	2	4	0.7	99.6%	0.99	2	3
R_400#1	400	800	12.0	0.99	4	12	1.3	95.0%	0.99	3	2
R_400#2	400	800	12.2	0.99	3	8	1.8	100%	0.99	3	3
R_600	600	1200	38.2	0.99	3	5	3.4	100%	0.99	4	4
R_800#1	800	1600	85.0	0.99	4	7	3.9	98.9%	0.99	4	3
R_800#2	800	1600	84.6	0.96	4	15	4.0	98.3%	0.99	4	3
R_1600	1600	3200	624.9	0.73	4	15	9.1	98.3%	0.99	5	3
R_3200	3200	6400	4706.2	0.59	4	12	59.5	99.1%	0.99	6	4
R_6400	6400	12800	34897.0	0.63	4	14	261.2	98.0%	0.99	7	4

tool applied spectral clustering with the number of clusters set to 3. It finds three functional clusters between execution time and the number of data items. The DRT for this experiment (shown in Table 3) gives two key insights: First, we notice that green cluster consists of plots that call *scale* function. Upon further investigation, the scale function is indeed expensive since it computes minimum and maximum for the input array and normalized values for each element of the input array. Next, the class initialization for `PlotImpl` distinguishes blue and red clusters. This happens for plots with one dimension where it needs to generate another dimension. It also needs to convert the new data set to a double array. This part is a performance bug because it can build a double array structure for the new dimension instead of double list.

2. **Snapbuddy**. SnapBuddy is a mock social network application where users can make their public profiles (Borges et al. 2017; Tizpaz-Niari et al. 2017). As inputs for our experiments, we passively monitored users' interaction with public profile pages. We applied K-linear clustering algorithm where we set the number of clusters to 5. As a result, our tool finds five linear relationships between the size of public profile image and download time. The DRT learned by our tool is shown in Table 3. It reports that filter combinations applied in profile pictures are the key discriminants.

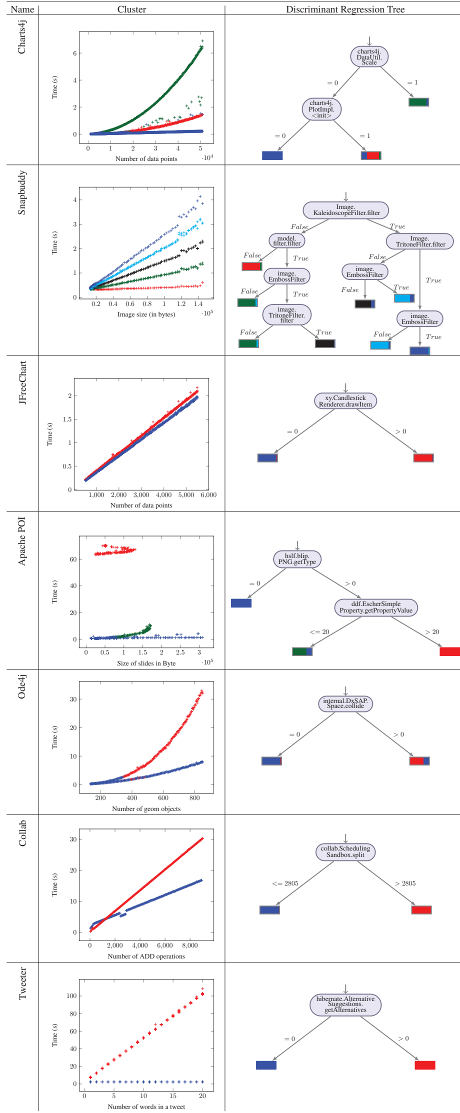
3. **JFreeChart**. JFreeChart is a free Java chart library that helps developers to plot different charts in their applications. The input data set is the set of open-high-low-close (OHLC) items. We use this library to plot OHLC items with different renders. Our tool applied K-linear clustering algorithm with $K = 2$. The clustering step finds two linear relationships between time of execution and the number of OHLC items. The DRT produced by our algorithm is shown in Table 3. We observe that whether the plot applies `CandlestickRenderer` and calls `drawItem` distinguishes between red and blue clusters. In particular, the number of calls to `drawItem` is equal to the number of data points in OHLC dataset, and each call executes the loop statement when a candlestick renderer is applied. This performance bug was also reported in (Olivo, Dillig, and Lin 2015) using static analysis.

4. **Apache POI**. The Apache POI Project's mission is to read and write MS Excel, MS Word, and PowerPoint files using Java. As inputs to our experiments, we used different slides that randomly include text, table, images, shapes, and smartArts, We applied the spectral clustering algorithm with number of clusters set to 3. The DRT produced by our algorithm is shown in Table 3. The DRT shows that all program traces labeled with the blue cluster do not have any image data source (like PNG, smartArts, and so on). Green and red clusters are distinguished only based on their item sizes.

5. **Ode4j**. Ode4j is an open source, high performance library for simulating rigid body dynamics. The data set includes the different number of geom objects that interact with different APIs. We applied the spectral clustering algorithm with the number of clusters set to 2. The clustering finds two functional relationships between the time of execution and number of geom objects. The DRT (Table 3) reports `collide` function in `DxSAPSpace` class to distinguish between blue and red clusters. Looking into the source code, we can see the quadratic behavior in `collide` function. As there is a linear cost function named `collide2` in `DxSAPSpace` class, we suspect that quadratic behavior can be mitigated.

6. **Collab**. Collab is a scheduling application that allows users to create new events and modify existing events. The data set consists of different operations to add and modify events using `ADD`, `UNDO`, and `DONE` commands. We applied K-linear clustering algorithm with K equals to 2. It finds two linear relationships between the time of execution and the number of add operations. The DRT produced by our algorithm is shown in Table 3. The discriminant model shows that the traces in red cluster call `split` function more times than traces in blue cluster.

Table 3: Java applications studied using DPDEBUGGER.



The average number of calls to `split` function for blue cluster is $1,678$ and for red cluster is $88,507$. The temporary store data structure of Collab is a B-tree. The number of times that the B-tree needs to split a parent node is the main explanation to distinguish red and blue clusters.

7. **Tweeter**. Tweeter application is a mock of Twitter application. Users can post tweets and see tweets posted by other users. The data set consists of a maximum 20 words for a tweet. We applied K-linear clustering algorithm where we set K to 2. The clustering step finds two relationships between time of execution and number of words in a tweet. The DRT produced by our algorithm is shown in Table 3. The discriminant model produces `getAlternatives` function as a feature to distinguish blue and red clusters. This function is called more times when the input tweet includes mistakes, and it will never be called if the tweet is correct.

Table 4: Java applications studied using DPDEBUGGER. Legend: **#**M_T: total number of functions in application, **#**M_R: total number of observed functions, **#N:** number of collected traces, C_T**:** Clustering algorithm (K for K-linear and S for spectral), **T**: computation time of DPDEBUGGER in seconds, **A**: accuracy of classification model, **H**: decision-tree height, **#C**: Number of clusters, $\epsilon < 0.1$ **sec.**

Application	# M_T	# M_R	#N	C_T	T	A	H	# C
Apache FOP	39,694	2,765	1,988	S	14.4	96.4%	3	2
Charts4j	715	71	2,000	S	11.7	87.1%	2	3
SnapBuddy	3,071	150	616	K	1.6	88.6%	5	4
JFreeChart	9,162	527	1,000	K	1.6	99.4%	1	2
Apache POI	10,396	199	661	S	0.44	86.4%	2	3
Ode4j	4,564	114	577	S	7.8	85.0%	1	2
Collab	185	53	530	K	0.7	96.6%	1	2
Tweeter	947	31	320	K	0.6	100%	1	2

7 Related Works

Performance debugging The work by Tizpaz-Niari et al. (Tizpaz-Niari et al. 2017) is the closest to ours. They use decision tree learning for finding security vulnerabilities, whereas we focus on performance bugs. They heavily rely on the assumption that it is enough to consider constant clusters, whereas we consider a much more realistic and the general setting of linear functional clusters. On the algorithmic side, for Tizpaz-Niari et al. it is sufficient to use the standard K-means algorithm (since they have only constant clusters), whereas we needed to adapt the K-means and clustering algorithms for the linear functional case.

Spectrum-based fault localization is often used for explaining bugs(Wong et al. 2016; Song and Lu 2014; Jin et al. 2010; Jones, Harrold, and Stasko 2002; Liblit et al. 2003). In particular, Song and Lu (Song and Lu 2014) use a statistical based technique for performance debugging problem. They indicate that the problems are manifested by the performance not being uniform, but rather there being two set of inputs: bad and good inputs. They refer to bug report databases to generate the bad inputs. They use statistical models to find predicates that distinguish good and bad inputs. They assume that two sets of bad and good inputs are given. In contrast, we are not given any labeled data set. We obtain different classes by clustering techniques. In addition, we are not limited to two sets of inputs. Finally, Song and Lu find the ranking of predicates and choose the top one that is responsible for bad runs. On the other hand, we use decision tree learning that can produce conjunctions of predicates. This can be seen in the SnapBuddy example.

Time series data has been used as well for profiling and failure detection (Hauswirth et al. 2004; Hauswirth et al. 2005; Sweeney et al. 2004; Adamoli and Hausiwrth 2010; Abreu et. al 2007). In particular, Hauswirth et al. (Hauswirth et al. 2005) consider different performance metrics of a system such as instruction per cycle (IPC) and monitor these factors over time. Then, they apply dynamic time warping (DTW) (Berndt and Clifford 1994) to combine the traces of the same input. When they observe a pattern like sudden IPC changes, they align all traces using DTW and apply sta-

tistical correlation measurements to find predicates that are highly correlated with the changes in the target metric as the cause of performance anomaly. In our work, however, we do not collect metrics over time as a time series, although an extension is possible. As we collect only the total execution time, we do not need trace alignment, but we efficiently cluster traces based on relationships between running time and user inputs.

Functional data clustering. Functional data clustering is a technique for clustering given *functions* (Jacques and Preda 2014a) (Jacques and Preda 2014), (Abraham et al. 2003). Even though there is a similarity in names, we are not given functions. Instead we are given individual data points, and we are discovering (linear) functions by clustering. Investigating the use of functional data clustering for debugging of software is an interesting area left for future work.

References

Abreu, R.; Zoeteweij, P; and Van Gemund, A. 2007 On the accuracy of spectrum-based fault localization. In TAICPART-MUTATION 2007, 89–98. IEEE.

Abraham, C.; Cornillon, P.-A.; Matzner-Løber, E.; and Molinari, N. 2003. Unsupervised curve clustering using b-splines. *Scandinavian journal of statistics* 30(3):581–595.

Adamoli, A., and Hauswirth, M. 2010. Trevis: A context tree visualization and analysis framework and its use for classifying performance failure reports. In *Software visualization*, 73–82. ACM.

Aloise, D.; Deshpande, A.; Hansen, P.; and Popat, P. 2009. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning* 75(2):245–248.

Alur, R., and Singhania, N. 2014. Precise piecewise affine models from input-output data. EMSOFT, 3:1–3:10. New York, NY, USA: ACM.

Berndt, D. J., and Clifford, J. 1994. Using dynamic time warping to find patterns in time series. In *KDD*, volume 10, 359–370. Seattle, WA.

Borges, M.; Phan, Q.-S.; Filieri, A.; and Păsăreanu, C. S. 2017. Model-counting approaches for nonlinear numerical constraints. In *NASA Formal Methods Symposium*, 131–138. Springer.

Breiman, L.; Friedman, J.; Stone, C. J.; and Olshen, R. A. 1984. *Classification and regression trees*. CRC press.

Cadar, C.; Dunbar, D.; Engler, D. R.; et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, 209–224.

Hauswirth, M.; Sweeney, P. F.; Diwan, A.; and Hind, M. 2004. Vertical profiling: Understanding the behavior of object-priented applications. In *OOPSLA*, 251–269. New York, NY, USA: ACM.

Hauswirth, M.; Diwan, A.; Sweeney, P. F.; and Mozer, M. C. 2005. Automating vertical profiling. In *OOPSLA*, 281–296. New York, NY, USA: ACM.

Jacques, J., and Preda, C. 2014a Functional data clustering: a survey. *Advances in Data Analysis and Classification* 8(3):231–255.

Jacques, J., and Preda, C. 2014b Model-based clustering for multivariate functional data. Computational Statistics & Data Analysis, 71: 92106

Jin, G.; Thakur, A.; Liblit, B.; and Lu, S. 2010 Instrumentation and sampling strategies for coopera tive concurrency bug isolation. In OOPSLA, 241255 New York, NY, USA ACM.

Liblit, B.; Aiken, A.; Zheng, A. X.; and Jordan, M. I. 2003. Bug isolation via remote program sampling. In *PLDI*, 141–154. New York, NY, USA: ACM.

Loh, W.-Y. 2002. Regression tress with unbiased variable selection and interaction detection. *Statistica Sinica* 361–386.

Loh, W.-Y. 2011. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1(1):14–23.

Mahajan, M.; Nimbhorkar, P.; and Varadarajan, K. 2009. The planar k-means problem is np-hard. In *WALCOM*, 274–285. Springer-Verlag.

Olivo, O.; Dillig, I.; and Lin, C. 2015. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, 369–378. New York, NY, USA: ACM.

Song, L., and Lu, S. 2014. Statistical debugging for real-world performance problems. In *OOPSLA*, 561–578.

Sweeney, P. F.; Hauswirth, M.; Cahoon, B.; Cheng, P.; Diwan, A.; Grove, D.; and Hind, M. 2004. Using hardware performance monitors to understand the behavior of java applications. In *Virtual Machine Research and Technology Symposium*, 57–72.

Tizpaz-Niari, S.; Černý, P.; Chang, B.-Y. E.; and Trivedi, A. Differential performance debugging with discriminant regression trees. arXiv:1711.04076.

Tizpaz-Niari, S.; Černỳ, P.; Chang, B.-Y. E.; Sankaranarayanan, S.; and Trivedi, A. 2017. Discriminating traces with time. In *TACAS*, 21–37. Springer.

Von Luxburg, U. 2007. A tutorial on spectral clustering. *Statistics and computing* 17(4):395–416.

Witten, I. H.; Frank, E.; Hall, M. A.; and Pal, C. J. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wong, W. E.; Gao, R.; Li, Y.; Abreu, R.; and Wotawa, F. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42(8):707–740.