An Industrial Case Study of Automatically Identifying Performance Regression-Causes

Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan
Queen's University, Kingston, Ontario, Canada
{thanhnguyen,mei,ahmed}@cs.queensu.ca

Mohamed Nasser, Parminder Flora
Performance Engineering, Blackberry, Canada

ABSTRACT

Even the addition of a single extra field or control statement in the source code of a large-scale software system can lead to performance regressions. Such regressions can considerably degrade the user experience. Working closely with the members of a performance engineering team, we observe that they face a major challenge in identifying the cause of a performance regression given the large number of performance counters (e.g., memory and CPU usage) that must be analyzed. We propose the mining of a regression-causes repository (where the results of performance tests and causes of past regressions are stored) to assist the performance team in identifying the regression-cause of a newly-identified regression. We evaluate our approach on an open-source system, and a commercial system for which the team is responsible. The results show that our approach can accurately (up to 80% accuracy) identify performance regression-causes using a reasonably small number of historical test runs (sometimes as few as four test runs per regression-cause).

Categories and Subject Descriptors

D.2 [**Software/Program Verification**]: Statistical methods; C.4 [**Performance of Systems**]: Measurement techniques; H.3 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Performance, verification

Keywords

Control charts, load testing, performance testing, performance regression

1. INTRODUCTION

Performance is an important aspect of large software systems since a large number of field problems are performance related [1]. Performance problems have serious implications on the profitability of a business. For instance, web users are likely to abandon a transaction after a ten second wait [2].

A new version is said to have a performance regression when it offers a worse user experience (e.g., longer response time), or consumes additional resources (e.g., more CPU or memory usage) while possibly maintaining the same user experience. While in some cases, one can bring in additional hardware infrastructure to offset the regression, such solutions are not economical and are often not feasible (especially for very large scale deployments). Hence, performance engineers conduct performance tests prior to the deployment of every new version to detect performance regressions.

Once a performance regression is detected, the performance team must provide guidance to the development team as to what are the possible causes (e.g., added fields in a long-living object). Developers use such guidance to narrow down their investigation to the offending change(s). In an ideal setting, performance tests would be run after each checkin so the causes of regressions could easily be mapped to the very specific checkins (i.e., code change). However, given the complexity of performance tests (e.g., requiring large lab setups, complex manual configurations, and lengthy executions times), per-checkin performance tests are rarely feasible in a large scale industrial setting. Instead performance tests are conducted across versions which often contain a large number of changes. Hence determining the cause of an identified regression (which we call as a regression-cause in the rest of the paper) is often a very time consuming effort – requiring hours or even days, depending on the experience of the engineers, the complexity of the system, and the performance regression itself.

Working with a performance engineering team, we investigate the use of mining software repositories approaches to automate the process of identifying regression-causes. We mined a repository of performance regression-causes. The repository contains the results of prior large-scale performance tests (along with associated performance counter data), as well as the verified regression-cause. Our approach matches the behaviour of a new version (through the performance counters) to the behaviour of prior tests runs where a performance regression has occurred, in order to determine the regression cause in the new version.

Given the confidential nature of the commercial system, we evaluate our approach using the popular open-source Dell DVD store software and the aforementioned large commercial system. Using the two case studies, we answer the following research questions:

- **RQ1: How accurate is our approach?** We find that in 74%-80% of the cases, the regression-cause identified by our approach was indeed the actual cause.
- **RQ2: How much training data is required?** Since the number of historical performance regressions might be limited, we examine the amount of training data required by our approach. We find that, even though having more training data improves the accuracy of our approach, we can get high accuracy with as little as four runs per regression-cause.

The contributions of this study are:

- Through a preliminary field study on a large scale commercial system, we find that 83% of the regressions can be attributed to just four common regression-causes.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597092

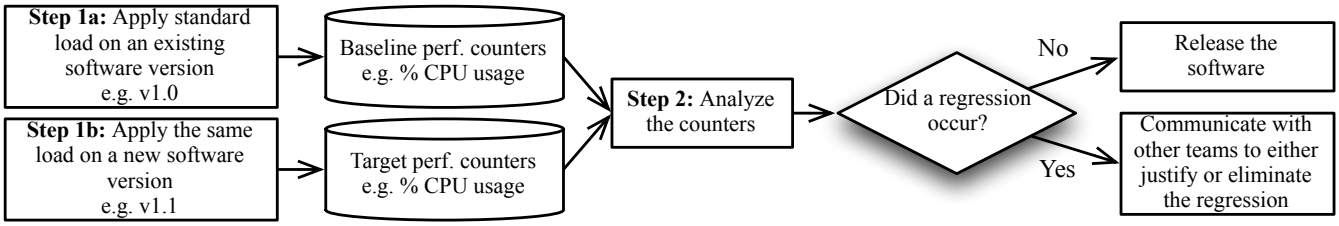


Figure 1: **The performance engineering team's workflow for regression testing**

- We propose and evaluate an approach, which can automatically determine if the regression-cause of a new performance test matches one of the known causes.

The paper is organized as follows. Section 2 introduces the background which motivates our approach for identifying the regression-causes. Section 3 explains the details of our approach. Section 4 introduces the two case studies used to evaluate our approach. Section 5 presents the results for our two research questions. Section 6 discusses the applications of our approach in an industrial setting and the feedback we got from the performance engineers. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2. CASE STUDY AND BACKGROUND

In this section, we introduce our industrial case study and the background information on performance testing.

2.1 Industrial Setting

We study a performance testing team of a commercial company. The software system, for which the team is responsible, is a high performance and high availability system that is deployed on several thousand nodes each, across several data centres around the globe. The software services millions of users worldwide.

2.2 Performance Regression Testing

The goal of performance regression testing is to ensure that performance of the system remains at an acceptable level under field-like load. During the software's life cycle, every change to the source code might degrade the software performance by consuming more resources. This situation is called a performance regression. To ensure that the user experience is not compromised, performance engineers conduct performance tests for each new software version.

Figure 1 shows the typical steps of performance regression testing – a black-box testing process. Step 1 applies field-like load to both the old version and new version of the software on the same field-like hardware. This step is usually done by load generation tools such as HP's Load Runner or Apache's JMeter. During the test runs, performance counters such as CPU or memory utilization are collected. Then, in Step 2, the two sets of counters, which are called baseline for the old version and target for the new version, are analyzed.

The performance of a software system depends on two factors. The software itself and the hardware on which it runs. If there is a regression, e.g., the software uses more CPU than before, the performance team works closely with developers to identify the cause and understand the rationale. If the rationale is not justified, the developers will need to make changes to the software to eliminate the regression. If the rationale is justified, then additional hardware would be provisioned to offset the observed regression.

2.3 Analyzing a Performance Regression Test

In Step 2 of Figure 1, the performance engineers need to analyze the performance counters to:

- determine if there is a performance regression; and,
- identify the regression-cause

We study the performance regressions, which the team reported in a period of one year to understand the typical outputs of Step 2 in Figure 1. With the help of the engineers, we analyze the content of bug reports using an approach similar to Jin et. al. [3], where bug reports were manually analyzed and grouped into different types of causes.

Table 1 shows a breakdown of the various causes of regressions over a year (only the percentage is shown for confidentiality reason). We name the regression-causes according to the wording in the bug reports. However, these regression-causes are widely used in practice under similar names. Altman et. al. [4] also report similar regression-causes, e.g., Memory Leak, Database Bottleneck, or Disk I/O.

As we can see, four of the regression-causes can explain 82.94% of regression problems over a year. Only for 17% of the identified performance regressions, no regression-cause was identified. Such cases are common since not all performance metrics are exposed. These cases usually require extensive debugging.

We interviewed the performance engineers responsible for the industrial software system, to understand these regression-causes. The engineers noted that the same regression-cause in different parts of the code will result in similar values for the performance counters (i.e. signature) as long as the regression causing code is inserted anywhere along the same execution path. However, if the regression-cause is inserted along other execution paths, it likely won't result in the same signature. For example, take a typical e-commerce application. Adding a regression-cause in different parts of the execution path that deals with the checkout process will lead to a common counter signature. However, adding the same regression-cause in the execution path responsible for the registration process will lead to a different counter signature. Given the large size of the code base and the team, they found that the insertion of the same regression-cause along the same path tends to occur often in their system.

Table 1: **Typical output of Step 2 in Figure 1**

Regression-cause	**%**
Added frequently executed DB query or miss matched DB indices	30.54
Added frequently executed logic	16.67
Added frequently accessed fields and objects	30.18
Added blocking I/O access	5.55
Symptom of regression is detected (e.g., response time increased) but no regression-cause can be determined	16.67

2.4 The Challenge of Analyzing the Results of Performance Tests

Unfortunately, identifying the regression-cause is not a simple task because:

- There are a large amount of counters. Apart from the standard counters in the OS's monitoring system, specific counters are also added to common trouble spots. Sometimes, profilers are also used as a fast way to create specific performance counters. Both result in thousands of counters to be analyzed for each test run.
- Most performance regression problems involve a combination of counters. For example, if a database index is missing, both the CPU utilization and memory are likely to increase. Hence looking at just one counter would not be sufficient.
- A regression in a component is very likely to impact the performance of other components. For example, if the database slows down, the front end will have to wait more for queries to return. Hence, the front end will use less CPU.

Yet, the analysis is usually done manually – a very time consuming and error-prone process [5]. To analyze this massive amount of counter data, performance engineers usually have to rely on experience to select a subset of counters to manually compare. It can take several hours or even days to analyze each performance test.

3. APPROACH

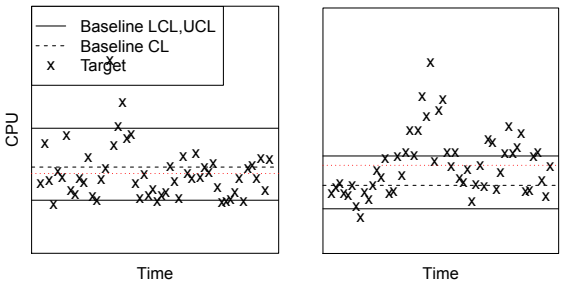
Leveraging our prior experience with mining software repositories, we believe that if the regression test results are kept in a repository, we can leverage information from prior tests when analyzing a new test run. Figure 3 shows an overview of our proposed approach which replaces the manual analysis in Step 2 (Figure 1). However, we first need to explain the two conceptual steps of our approach:

- **Step A - Data reduction:** We reduce the counter data into a smaller set of data using a statistical control technique called control charts. Thus a simpler and easier to understand performance counter dataset is created to be used in the next step of our approach.
- **Step B - Detect regression and identify regression-cause:** Using machine learning techniques, we match the regression-cause of the current test run with the regression-cause of historical runs. One of the regression-cause is "Normal", which means there is no regression. So we can say a) whether regression happened and b) what is the probable cause of the regression.

Step A: Data Reduction Using Control Charts

Goal: Data reduction is essential when analyzing a large number of counters (usually 2 to 3 thousand counters). Prior studies employ techniques such as principal component analysis [6, 7], projection pursuit [8], or normalized mutual information [9] for data reduction. However, these techniques primarily fuse the counters together creating new index counters that are often hard to map back to intuitive concepts.

Control charts: Control charts are a common tool in statistical quality control [10]. They are used to detect problems in manufacturing processes where raw materials are inputs and the completed products are outputs. A control chart outputs a measurement index, called violation ratio, which indicates the amount of deviance of the current state of the process compared to the norm.



(a) Normal operation ⇔ **Low violation ratio** (b) Out-of-control ⇔ **High violation ratio**

Figure 2: **Examples of control chart which detect deviation in process output (performance counters in this case).**

Figures 2(a) and 2(b) show two example control charts. The x-axis is time, e.g., minutes. The y-axis is the counter values. In this example, we are monitoring the CPU usage.

A control chart is typically built using two datasets: a baseline dataset and a target dataset. The baseline dataset is used to create the control limits. The Upper Control Limit and the Lower Control Limit are the two solid lines which represent the 90th and the 10th percentile. The dashed line in the middle is the Centre Line which represents the median, or the 50th percentile, of the baseline dataset. The target dataset, which are the crosses, is used to score the violation ratios on the chart. The violation ratio is the percentage of target dataset values that are outside the control limits. The violation ratio represents the degree to which the current operation is out-of-control. The thresholds for violation ratios are determined by the performance engineer based on their experience.

Figure 2(a) is an example where the CPU utilization is within its control limits. This should be the normal operation of the server. Figure 2(b), on the other hand, is an example where the CPU utilization (process output data) is out-of-control. In this case, operators should be alerted for further investigation.

Reducing performance counter data using control charts: In a performance test run, we have two sets of data, which are the old version's test data (the baseline) and the new version's test data (the target) (see Figure 1). Control charts reduce, as explained above, the counters of the baseline and target tests into violation ratios for each counter. To be more precise, control charts can output two violation ratios for each counter: upper violation ratio and lower violation ratio. The upper violation ratio is the ratio of dataset values that are higher than the upper control limit. The lower violation ratio is the ratio of dataset values that are lower than the lower control limit. The collection of violation ratios of all the counters for each test run is the performance counter dataset of that test run.

This reduction does not compromise the explanatory power of the reduced data for a specific counter. Working with the performance engineers, we found that they are more comfortable to adopt approaches, like control charts, since they are easy to understand and explain [11, 12]. If the violation ratio of a counter is high, it means that the new target test behaves differently from the baseline test for that counter. If the violation ratio for a counter is low, it means that the new target test behaves the same as the baseline test for that counter. So even if we are not using the violation ra-

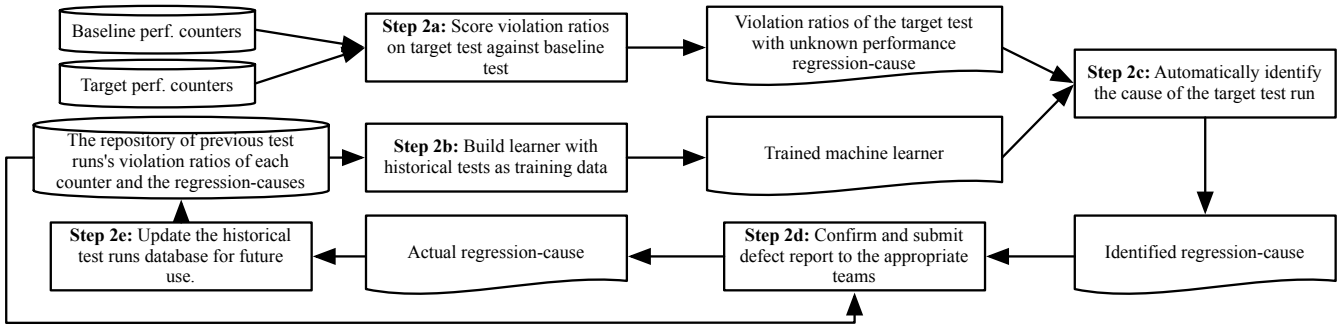


Figure 3: **Our proposed approach for determining the regression-cause, which replaces the manual analysis of performance counters (Step 2 of Figure 1).**

tios in the analysis step of performance regressions, which we will explain in the next subsection, the engineer can still make sense of the reduced data. If we use other data reduction technique such as principal component analysis [6, 7], we would still achieve reduction. However, it would be very difficult to make sense of the reduced data since the counters are combined into principal components.

Step B: Detecting the Regression Types Using Machine Learners

Goal: After Step A, the data for analysis is similar to the example in Table 2. Each row represents a test run (with relation to its baseline test run). Each column is the violation ratio of the corresponding counter when comparing the target run against its baseline run. We have one column for each counter of every component of the software. For example, if we have 15 components and each component has 36 counters (the standard counters that Microsoft Windows collects for a process), we will have $15 * 36 = 540$ columns.

The top part of Table 2 are the historical tests which are stored in a repository. The last column shows the identified cause of the performance regression in that run. If there is no regression, the regression-cause would be G ('Normal'). If there is a performance regression, the verified regression-cause is recorded. For example, in the first row, there is no performance regression, so the regression-cause is 'Normal'. In the second row, there is a performance regression due to adding code to a frequently accessed logic, so the regression-cause is recorded as 'Added hot code' (See Section 4 for a detailed explanation of the various regression-causes).

The last row of Table 2 is the new test run, which corresponds to the new version, and hence has an "unidentified" regression-cause (denoted as a '?'). To identify the regression-cause (if a regression has occurred) of this new test, an engineer would have to match the characteristics of the performance counters of this new test with the performance counters of all prior tests. In simple cases, only a few counters of a component will have high violation ratios. This means that the regression is isolated to that component. It would be easy to identify the regression-cause. As we mentioned in the previous section, because the components are dependent on each other, a regression in one component will also affect other connected components. In such a case, many of the counters would have high violation ratios therefore the engineer must spend considerable time to isolate the regression-cause using his/her experience. Our motivation is to introduce machine learners to mimic the experience of the engineers.

Table 2: **Example of data used in the machine learning of Step B**

Data	**Co.1 CPU VR**	**Co.2 Mem VR**	**Co.3 Net VR**	...	**Co.n VR**	**Regression-Cause (Sec. 4.1)**
In repository	3%	5%	6%	...	...	G
	14%	0%	3%	...	...	B
	13%	2%	5%	...	...	B
	3%	23%	4%	...	...	A
	3%	4%	2%	...	...	G
	...	...	...	...	...	...
New	X_1	X_2	X_3	...	X_n	**?**

Co.X = Component X, and VR = Violation Ratio

Identifying the causes of performance regressions: The matching of the current test to previous tests can be done using machine leaners such as Naive Bayes Classifier [13], J48 Decision Trees [14], or Multi-Layer Perceptron [15]. We train the learners using the data (violation ratios of performance counters and the identified regression-causes) from prior performance test runs. Then, we use the trained learners to identify the regression-cause of the new test run (for which we have the violation ratios of the performance counters).

Figure 3 shows our proposed approach which replaces the manual analysis in Step 2 of Figure 1. (**Step 2a**) We score the violation ratios using the new test data and the previous test data (the baseline) as explained in Step A. (**Step 2b**) Then we use the prior tests in the repository with known regression-causes to train a machine learner. (**Step 2c**) We use the machine learner to suggest the regression-cause of the new test run. (**Step 2d**) At this point, the engineer can confirm the identified regression-cause. If identified regression-cause is correct, the engineer can then file the defect report and communicate with the right development team for further investigation. (**Step 2e**) Once the correct regression-cause is identified, we add the test data as well as the regression-cause into the repository so that we can use it for future tests.

The regression-causes can be as simple as "Added hot code". Or they can be more complex such as "Added hot code into X component" or even "Added hot code into X component in thread pool A". The only requirement is that there must be enough prior tests with the same regression-causes. The number of tests required for each regression-cause is explored in Section 5.2. We can start with simpler/more generic regression-causes. Then as the number of tests increases, we can create a more detailed coding system for more specific regression-causes.

4. CASE STUDY SETUP

4.1 Performance Regression Injection

To evaluate our approach (Figure 3), we apply the approach on the commercial software (Commercial) whose performance team we are studying. We also apply the approach on an open source software, the Dell DVD store (DS), because we have limited disclosure on the Commercial case study. DS was used in many prior studies on software performance [5, 16].

In both case studies, we introduce six types of changes to the source code, each of which causes performance regressions (i.e., regression-causes). These regression-causes include those that are identified in Section 2.3 (in the future additional regression-causes can be added and explored – we limit our regression-causes to those that the engineers noted to be frequently occurring based on their experience):

(A) Added fields in long living objects: Causes memory usage increase. If a field is added to an object, and that object is created many times by the application, then even though the additional memory foot print of a field is small, the multiple instantiations can cause a large increase in memory usage.

(B) Adding frequently executed logic: Causes a CPU usage increase. Even a small set of additional calculations added to a part of the source code which is executed frequently can cause an increase in CPU usage.

(C) Added frequently executed DB query: Causes increased DB requests. In large software systems, each database can connect to hundreds of nodes. Each node processes millions of requests per minute. If a request performs one more database query than before, then the number of querying requests on the database would increase sharply.

(D) Mismatched DB indices: Missing column indices in the DB – causing longer DB requests. Necessary indices are required for frequently-executed queries. However, the necessity is dependant on the actual load. Hence this kind of regression is only discovered in a performance test.

(E) Mismatched text indices: Missing text indices. This is similar to (**D**) but with respect to text indices, which are required if text searches are performed on a column of the DB.

(F) Added blocking I/O access: Causes more and/or longer I/O access time. Accessing I/O storage devices, such as hard drives, are usually the slowest part of a transaction. Changes that add more I/O operation into a transaction usually causes performance regression if the transaction is executed frequently. For example, adding unnecessary log statements is a common mistake [17]. Log statements are usually required when implementing a new feature. There is a tendency to leave the logging statements behind in the source code when the change is finished. Unfortunately, if the log statements are part of a frequently accessed source code area, there will be a lot more log lines added in the log file, which can cause an increase in I/O.

(G) Normal: No regression. This is the case where there is no performance regression. The new test run performs as good as the previous test run.

For the Commercial case study, we inject issues corresponding to the first four regression-causes (A, B, C and D). Regression-cause E is not applicable since there is no text search in all transactions. We could not implement regression-cause F without altering the functioning of the software, which we want to avoid. For each cause, we inject the actual problem (change in source code) in six different parts of the source code. Every one of these six different parts of the code lies in the same execution path for an end user action, like say the checkout process in an e-commerce application (refer to the discussion at the end of Section 2.3). We conduct a test run for each code location of each regression-cause. For each of these test runs that will cause a performance regression, we calculate the violation ratios using the counters of that run and the counters of a normal run. We also calculate the violation ratio of the normal runs (G) by scoring their counters against another normal run. Thus, we will have the performance counter data (violation ratios) of all 30 test runs: six each, for the normal case and the four regression-causes.

For the DS case study, we use the runs with performance regression of the five regression-causes (A, B, D, E, and F), by injecting the issues into six different areas of the JSP code. For regression-causes D and E we also had to change the database configuration, to cause a performance regression. For regression-cause F, the logging library was configured to write directly to disk, so that a performance regression with blocking I/O occurs. Normally, most production logging systems use asynchronous I/O instead, which does not cause a blocking I/O regression. For regression-causes A, B, and E, we introduce the problems in two different execution paths of the software. For example, A1 is adding fields in long living objects on the execution path for the searching transaction. A2 is adding fields in long living objects on the execution path for the ordering transaction. We tag A1 and A2 as two different regression-causes, because they are on two different execution paths from two different transactions, each of which has a different performance profile. Then, for each code location of each regression-cause, we conduct a test run. We calculate the violation ratios of each run by scoring that run against the 'Normal' runs (G), which we also run six times. We also calculate the violation ratios for the normal runs by scoring that run over the other normal runs. At the end, we have a table with 54 rows of violation ratios for the nine different causes (six test runs each, for the normal case, and eight problems - A1, A2, B1, B2, D, E1, E2, and F).

In both case studies, we collect only the standard counters from the OS's monitoring infrastructure, which produces about 32 counters per process, for each test run. Then, we calculate the violation ratios using that run as the target and one of the 'Normal' regression-cause runs (G) as the baseline. As a result, we have a table for each case study that is similar to the example in Table 2. We use these two tables for the analysis.

5. RESULTS

5.1 RQ1: How Accurate is our Approach?

5.1.1 Approach

We use learners from the Weka data mining software. We use learners that accept numeric independent variables (since violation ratios of counters are numeric) and a categorical dependent variable (identified regression-cause) as inputs to determine the learner with the best accuracy. If the learner can take our inputs, we use it for our approach. We found 17 such learners in Weka. Table 3 lists all the used learners.

For evaluation, we perform a leave-one-out evaluation for both case studies (see Section 4). We first pick one random test run out of all the runs of all the regression-causes. The rest of the runs are used for training the learner. Then, a learner is used to suggest the regression-cause of this test run, by finding the closest matching regression-cause of all the other test runs that were used as training data. Since regression-causes are known for each test run, we compare the suggested and the actual to evaluate the accuracy. If they match, then it is a success. For example, if the test run's actual regression-cause is "Normal", meaning no regression, and if the suggested regression-cause is also "Normal", then it is a successful identification. Otherwise, it is a failure. For example, if the suggested cause is "Adding hot code", then it is a failed identification. We repeat this procedure for all test runs.

For each learner, we report the accuracy as a percentage. Equation (1) defines the accuracy measure [29] used in our study:

$$Accuracy = \frac{|Success|}{|N|} \quad (1)$$

In (1), *Sucess* is the number of test runs where the prediction of the learner matches the actual regression-cause. N is the total number of runs. A high accuracy means our approach can reliably suggest the correct regression-cause to the performance engineers, thus potentially saving time.

We also report the gain in overall accuracy [29] over a random predictor, which is defined in Equation (2):

$$Gain = \frac{Accuracy}{r} \quad (2)$$

In (2), r is the accuracy [29] of the random predictor, which is defined in Equation (3):

$$r = 100 * \sum_{i=1}^{|\text{Regression-Causes}|} (\frac{|\text{Runs for regression-cause}_i|}{N})^2 \quad (3)$$

Table 3: **Success rates of different learners for identifying performance regression-causes using violation ratios (RQ1).**

Machine learners	**Class**	**Comm.**		**DS**		**L.G.**	**C.G.**
		A.	**G.**	**A.**	**G.**		
Random r (Eq.(3))		20%	0	11%	0		
J48 [14]	D.Tree	**80%**	**4.00**	56%	4.95	4.48	4.60
LMT [18]	D.Tree	57%	2.83	67%	5.94	4.39	
R.Forest [19]	D.Tree	70%	3.50	72%	6.48	**4.99**	
R.Tree [20]	D.Tree	67%	3.33	64%	5.76	4.55	
N.Bayes [13]	Bayes	53%	2.67	62%	5.58	4.12	4.39
N.Bayes Mul.	Bayes	63%	3.17	68%	6.12	4.64	
BayesNet	Bayes	70%	3.50	59%	5.31	4.41	
Dec.Tab. [21]	Rule	60%	3.00	39%	3.42	3.21	3.70
JRip [22]	Rule	47%	2.33	57%	5.13	3.73	
PART [23]	Rule	77%	3.83	50%	4.50	4.17	
IBk [24]	Lazy	63%	3.17	63%	5.58	4.37	3.82
KStar [25]	Lazy	63%	3.17	63%	5.58	4.37	
LWL [26]	Lazy	53%	2.67	31%	2.79	2.73	
Logistic [27]	Reg.	57%	2.83	72%	6.48	4.66	4.44
Sim.Logis. [18]	Reg.	50%	2.50	67%	5.94	4.22	
Mul.Perc. [15]	Neu.Net.	57%	2.83	**74%**	**6.66**	4.75	**4.87**
SMO [28]	Neu.Net.	73%	3.67	70%	6.30	4.98	

(A.) Accuracy (Equation (1)) (G.) Gain (Equation (2))
(L.G) Learner Avg. Gain (C.G.) Class Avg. Gain

For the Commercial system, we have five regression-causes in the test runs (four of them with performance regressions, and one is normal), with six test runs for each regression-cause. For the DS case study, we have nine regression-causes in the test runs (eight of them with performance regressions, and one is normal), with six test runs for each. Hence N is 30 and 54 and the value of r is 20% and 11% for Commercial and DS respectively.

5.1.2 Results and Discussion

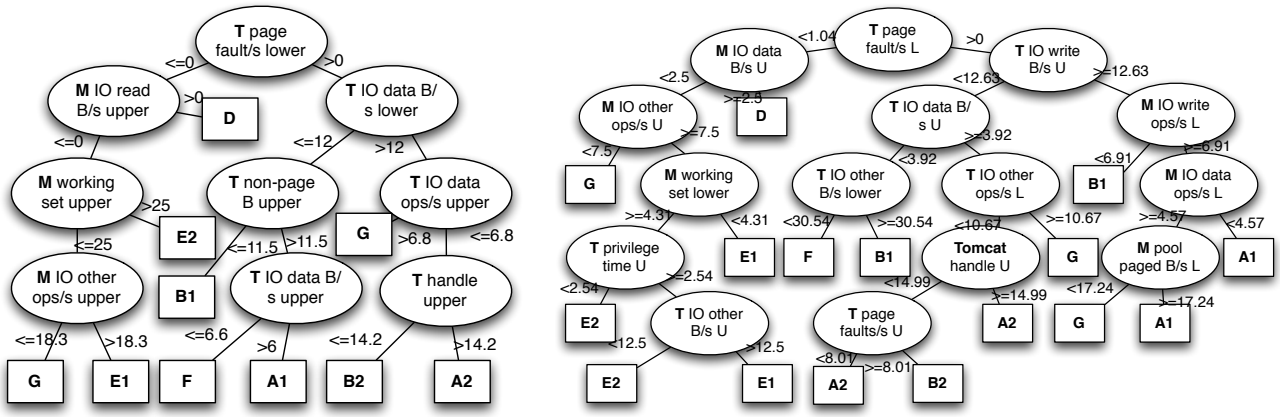
Table 3 shows the accuracy for each of the studied learners in both case studies. The first row shows the random accuracy r (Equation (3)). If a machine learner performs worse than r, then that learner is not useful.

The higher the accuracy of a learner compared to the random predictor, the better the learner is. The seventh column (L.G.) shows the average accuracy gained (Equation (2)) compared to the random predictor over the two case studies. This average gain indicates the usefulness of a learner. The last column (C.G.) shows the average accuracy gained of all the learners of a particular machine learning algorithm class (column two). This shows which class of learners is most suitable for our approach.

The first observation that we can make from the results is that: **most learners perform well compared to the random predictor**. The worst learner (LWL) has a gain of 2.73 times over the random predictor. The best learner (RandomForest) gains 4.99 times. So, at Step 2e of our automated approach, when the engineers try to confirm the regression-cause, they already have two to almost five times advantage compared to a random approach. This advantage would translate into saved time and effort.

The second observation we can make from the results of Table 3 is that: **the most suitable learning technique is system dependent**. For the Commercial system, J48, which is a Java implementation of the C4.5 [14] learner, is the best learner with 80% accuracy. For DS, the best learner is MultilayerPerceptron [15], a lazy type learner, with 74% accuracy. This means that there is no universal best learner to suggest probable regression-causes for all software systems. The learner can be accurate in suggesting the probable regression-causes of one system but not the other. MultilayerPerceptron and J48 [14] had the opposite accuracy for the two case studies. MultilayerPerceptron can predict with 74% accuracy for DS (best out of 17) but can only achieve 57% for the Commercial system (11th out of 17). On the other hand, J48 is the best for Commercial (80%) but is the 14th best out of 17 learners for DS (56%).

The third observation is that: although there is no universal best learner, **there are few good learners (and classes of learners) for both case studies**. Those learners are identified by the high average gain column (column 7 - L.G.). RandomForest [19] achieves an average gain of 4.99 times compared to the random predictor (which is 70% and 72% for the Commercial and DS respectively). Similarly SMO achieves average gain of 4.98. Hence using SMO, we gain about three and a half times compared to the random predictor for the Commercial system, and more than six times for DS. In general, the neural network class, which SMO is part of, has a good class-average gain which is 4.87 (last column of Table 3). Note that we take the average to quantify the class level gain, but the median produced very similar results too. The decision tree class, which RandomForest is part of, also has good class-average gain of 4.60.



(a) J48 decision tree of the DS case study

(b) RandomForest decision tree of the DS case study

Figure 4: **Comparing the decision tree structure of the DS case study**

So, if it is not possible to find the best learner for a software system, learners from these two classes can be used.

The fourth observation is that: **there are unsuitable learners (and classes of learners) for both case studies**. In general, the rule and lazy based learners are the worst (class-average gain column of Table 3 - C.G.). While most of the classes have class-average gain greater than four, these two classes have less than four average gain. Rule based and lazy learners are the simplest learners. They are more suitable for a low number of features (i.e. fewer counters), with a large amount of data, and low amount of noisy data [30]. However, these learners can benefit from preprocessing techniques [11], to remove the noisy counters. The fact that both classes of learners perform badly suggests that the relationships among the performance counters are not simple.

To examine more on why certain class of learners performed very well (observation 2), we look at the decision tree class of learners in Table 3 for the DS case study. In particular we compare J48 and RandomForest. In the commercial case study, both learners have a similar performance. While in the DS case study there is a much larger difference in the performance of these two learners (almost 16%). Therefore, we compare the trees that are created by these two learners in the DS case study to understand why there exists such a difference in performance.

Figure 4 shows the decision trees for the J48 and RandomForest learners of the DS case study. As we can see, J48's decision tree is simple. On the left subtree of the root, the decisions are based on the MySQL related counters. On the right subtree, the decisions are based on the Tomcat related counters. RandomForest's decision tree, on the other hand, uses the data from both set of counters throughout its structure.

The J48 learner is not able to take advantage of all the relationships among the counters. So the gain and accuracy achieved is low. The RandomForest learner was introduced for this kind of situation [30]. It works by generating different decision trees on a subset of counters. The decisions are the most popular among all generated trees. The resulting tree was able to take advantage of more relationships among the counters. Thus, it has better gain and accuracy.

RandomForest learner can also be used to show importance of different counters for a specific classification problem. We run this analysis for the decision tree of the DS case study (Figure 4(b)). Table 4 shows the top 20 most important counters. If we look at the regression-causes of the DS case study in Section 4, most of the causes should mainly change the CPU and memory usages of the MySQL and Tomcat process with the exception of cause F. However, Table 4 shows that the top most important counters are mostly IO related. This is an example of the complex relationship among counters which the machine learners, such as RandomForest, are able to capture.

5.2 RQ2: How Much Training Data is Required?

The results of RQ1 shows good accuracy for both case studies. So we have evidence that our approach is useful. Assuming that a performance team wants to adopt our approach, they would first need to build a repository of test runs with identified regression-causes. Building such a repository could be time consuming. In RQ2, we want to understand how much training data is needed before one can start using our approach.

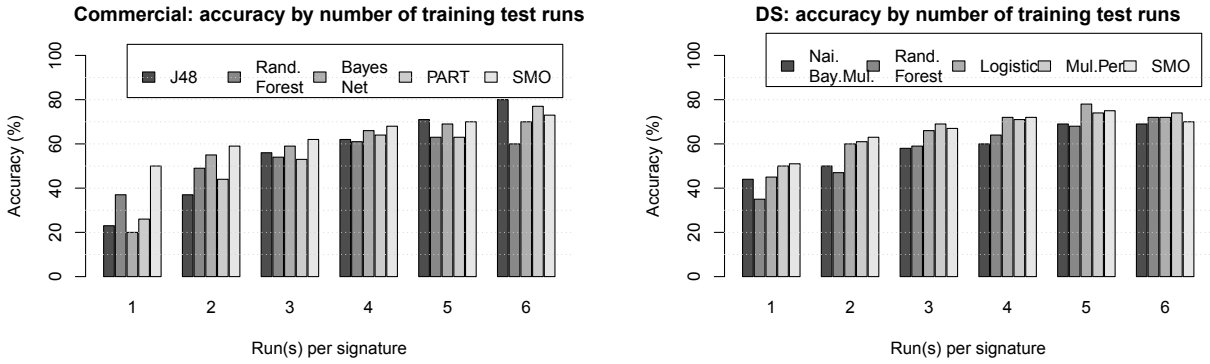
5.2.1 Approach

To determine the amount of performance test runs required in the repository (i.e., size of training set), we modify the leave-one-out procedure in RQ1. The goal is to observe the change in accuracy if we use one, two, three, four, or five runs of each regression-cause for training instead of using all six runs as we did in RQ1. So, after picking one

Table 4: **Variable importances (VI) of the counters in the RandomForest decision tree (in 10^{-2})**

VI	Counter	VI	Counter
.93	SQL IO R B/s U	.34	Tom. IO W B/s U
.78	SQL IO W B/s L	.33	Tom. pool paged U
.78	Tom. IO data B/s L	.31	Tom. pri. time L
.76	Tom. IO W B/s L	.31	Tom. working set L
.69	SQL work. set U	.28	SQL IO data O/s L
.66	SQL IO W O/s L	.28	SQL IO R B/s L
.64	Tom. IO data B/s U	.27	Tom. IO other B/s U
.53	SQL IO R O/s U	.24	SQL IO R O/s U
.47	SQL IO data O/s U	.23	Tom. page faults/s U
.46	Tom. page faults/s L	.22	Tom. pool nonpaged B U
.35	SQL IO data B/s L	.19	SQL User Time U

R=read W=write B=bytes O=operations U=Upper vio. ratio L=Lower vio. ratio



(a) Commercial

(b) DS

Figure 5: **Accuracy of the top five learners using different number of test runs per regression-cause for training (RQ2).**

random run for testing, we will only use n run(s), selected randomly (where n is one to six), of each regression-cause from the rest of the runs to train the learner. This will limit the size of training data to n for each regression-cause. We perform the procedure six times for each n and report the accuracy. We note that, similar to RQ1, under no circumstance is the testing run used for training.

5.2.2 Results and Discussion

Figure 5(a) and 5(b) shows the results for both case studies. We perform the procedure mentioned above using the top five learners according to the gain columns (G.) in Table 3. The first set of bars in Figure 5(a) and 5(b) show the accuracy when only one run of each regression-cause is used for training. The second set of bars is the accuracy when we use two test runs per regression-cause and so on.

We can make two observations from the results shown in Figure 5(a) and Figure 5(b).

Firstly, in most cases, **accuracy of the regression-cause identification increases with the increase in training data size**. This is an indication of good learners for any data mining application [29]. Since our approach (Figure 3) is iterative, the more iterations there is, the more historical data is available, and thus the better accuracy.

Secondly, for the Commercial case study, we did not reach the top accuracy with six runs in our case study. For J48, which is the best learner for this system, the accuracy increases steadily from 30% when one run per regression-cause is used, to 80% when six runs per regression-cause are used. So the result has not reached a plateau yet. This indicates that there is still room for improvement (when additional runs can be added to the repository) in the Commercial case study.

In contrast, we reach a plateau for the DS case study with about four runs. This probably means that we have enough runs for this case study. For Logistic, MultilayerPerceptron, and SMO, we reach 70% accuracy with four runs per regression-cause. At five runs per regression-cause, we do not have any additional gain in accuracy, thus indicating that we have reached the maximum accuracy for the three learners. At six runs per regression-cause, the accuracy of MultilayerPerceptron does not improve. Logistic and SMO's accuracy even decreases. This is evidence of over-fitting [29].

6. APPLICATION AND FEEDBACK FROM INDUSTRY

6.1 Application

To collect feedback from the performance engineers, we applied our proposed approach on a few recently available test runs of the commercial system as a proof of concept.

Setup: Since we can not build a historical repository because of the unavailability of certain data, we decided to use the same injected runs as we used in RQ1 and RQ2 as our historical test repository. These test runs are hence considered as synthetic test runs.

We asked two performance engineers of the commercial system (besides the last two co-authors) for three test runs with actual performance regressions. All three runs have performance regressions as confirmed by the performance engineering experts of the commercial system in a recent testing cycle. The regression-cause has been confirmed to be similar to one of the regression-causes that exist in our repository. We also obtain three runs of the previous version to use as baseline.

For each of the three runs, we apply our approach to identify the regression-cause using the top five learners in Table 3. Similar to RQ2, we train the learners with one randomly chosen run per regression-cause. Then we train with data from two, tree, four, five, and all 6 runs for each regression-cause. For each of the three new test runs we measure the accuracy of each of the learners (i.e., whether the predicted regression-cause is the actual regression-cause). We repeat this procedure 6 times, so that in the case where just one run is chosen per regression-cause, there is chance for each of the 6 run for that regression-cause to be chosen.

Results: The accuracy is recorded in Figure 6. The first set of bars show the accuracy when we use only one run per regression-cause. The next set of bars show the accuracy when we use two runs per regression-cause and so on.

For these particular runs with actual performance regression, our approach can accurately identify the actual regression-cause using the repository of synthetic test runs. Both BayesNet and SMO reach 100% accuracy when only five test runs per regression-cause are used as training data. At four runs per cause, both learners reach 94% accuracy already.

The results support the possibility of adopting our approach in a commercial setting. Since not many software system have a performance test repository with identified

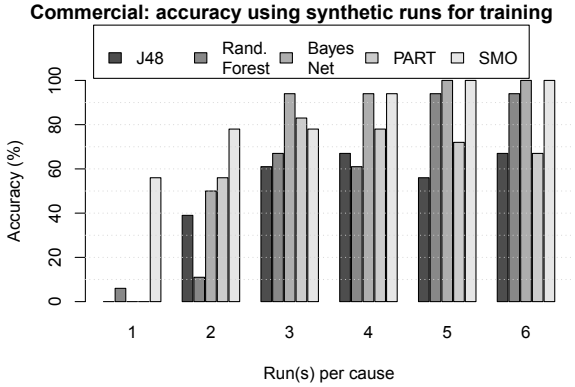


Figure 6: **Accuracy of the top five learners when using the synthetic runs to identify the regression-causes of three actual test runs of the commercial system.**

regression, one can boot-strap our approach by creating a synthetic repository of test runs with injected problems (due to various causes). Using this repository, our approach can be adopted without having to wait for test runs with actual performance regressions.

6.2 Feedback

Apart from the high accuracy of our approach, the feedback from the performance engineers was positive. The engineers were impressed with the potential time saving. They compared our approach with a push information mechanism such as Amazon's book suggestion. Having the suggestions would help them in completing the analysis earlier since they do not have to go through all the data.

More importantly, the engineers said that the violation ratios that we produced in Step A (Section 3) are simple and easy to explain. It is easier to adopt an approach which can be explained easily to other teams.

The engineers were also impressed that we only use the standard resource counters which are available on any operating system. In reality, the software also implements their own counters such as queue sizes or response-times of different processing threads. Such counters could potentially improve the accuracy of finding regression-causes by our approach. We are going to examine this in our future work.

However, the engineers also noted some potential limitations of our approach. First, we demonstrated our approach on only one of their regressions. A larger industrial study is required to justify the investment cost of applying our approach. Second, we did not verify the accuracy of our approach when there are multiple regression-causes in the same test run. However, identifying standalone regression-causes are also helpful to developers. In future work, we will examine the case of multiple regression-causes. Third, although the results show that any machine learner performs better than the random predictor, the training set must have sufficient number of test runs for each regression-cause. If the regression-cause of the new test is new to the software system, there would be no similar runs in the historical test repository (Figure 3). Thus, the learner will output the closest regression-cause that it can identify. This can be misleading. The engineers can potentially mitigate this risk by introducing synthetic runs for possible regression-causes into the training set, but there is no guarantee. Fourth, while our approach can save time, from manually checking all the possible regression-causes, it might create unwanted distractions from the mismatched regression-causes when the approach fails to produce a correct match. Fifth, as the software evolves, the regression data in the repository might become invalid over time, which might cause the learners to produce misleading suggestions. A longitudinal study is required to understand the effect of evolution on the accuracy of older runs. Finally, using our approach we can identify the type of regression, but not the location of it. Although finding the location would be be even more helpful to the performance engineers, knowing the regression type is the first step in the right direction. In future work, we will explore how we can automatically find the code location from the regression-cause. Even with these limitations, the possibility of saving time makes our approach very attractive to them.

7. RELATED WORK

There are many related studies which aim to detect problems in performance regression tests automatically. For example, Foo et al. [16] detected changes among the performance counters using association rules. If the differences are higher than a threshold, the run is marked as problematic. Malik et al. [31] used a factor analysis technique called principal component analysis to transform all the counters into a small set of more distinct vectors. Then they compare the pairwise correlations between the vectors in the target run with those of the baseline run. Ghaith et al. [32] proposed the use of queuing networks to detect performance regression. Jiang et al. [5, 33] introduced approaches to automatically detect anomaly in a performance test. Their approaches automatically detect out-of-order sequences in the software log. As in our previous studies [12, 11], where we first proposed the use of control charts to detect performance regressions, these related works only leverage the data of the baseline and the new test run. Our approach in this paper aim to improve previous results by leveraging a repository of previous test data as well. Hence the related literature presented above can only detect if a performance regression has occurred, but not identify the regression-causes.

There are many studies [34, 35, 36, 37] in system engineering which use machine learning or statistical techniques on performance counters. However, their goal is to determine if a software system is performing well or not during a particular time period. On the other hand, Altman et al. [4] proposed an approach to detect the causes of performance regression, which is similar to what we want to achieve. While their approach can be very accurate, it uses call stacks sampled throughout the test run. We believe that our approach is more practical since call stacks are more difficult and costly to collect on all components of a large software system.

8. CONCLUSION

In this paper, we proposed a new type of software repository and demonstrated its value through an industrial setting. This repository of performance test runs allow us to automatically identify the causes of performance regressions in new test runs.

We conducted two case studies to develop and evaluate our approach. The results show that our approach can accurately suggest (up to 80% accuracy) the regression-cause with a very small training dataset (sometimes with as few as three or four test runs per regression-cause). Moreover, this approach can be boot-strapped using synthetic test runs with injected problems.

The results thus far are encouraging. For future research, we are planning to adopt a code mutation approach which randomly injects regressions into various parts of the code. Then, we can use the same technique we have here to determine the different regression-causes in different areas of the code. This eliminates the need of prior tests in our approach. The engineer can use this mutation injections to generate all synthetic test data that contains a large amount of possible regression-causes to rapidly create their repository.

9. ACKNOWLEDGMENT

We thank BlackBerry for providing support and data access for this study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of BlackBerry's products.

10. REFERENCES

[1] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Trans. on Soft. Eng.*, vol. 26, no. 12, pp. 1147–1156, 2000.

[2] J. Palmer, "Designing for web site usability," *Computer*, vol. 35, no. 7, pp. 102–103, 2002.

[3] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Conf. on Prog. Lang. Design and Impl.* ACM, 2012, pp. 77–88.

[4] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *Conf. on Object-oriented Prog., Sys., Lang., and Apps.* ACM, 2010, pp. 739–753.

[5] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Intl. Conf. on Soft. Main.*, 2008, pp. 307–316.

[6] L. Eeckhout, A. Georges, and K. D. Bosschere, "How java programs interact with virtual machines at the microarchitectural level," in *Conf. on Object-oriented Prog., Sys., Lang., and Apps.* Anaheim, CA: ACM, 2003, pp. 169–186.

[7] Z. Zheng, Y. Li, and Z. Lan, "Anomaly localization in large-scale clusters," in *Intl. Conf. on Cluster Comp.*, 2007, pp. 322–330.

[8] J. S. Vetter and D. A. Reed, "Managing performance analysis with dynamic statistical projection pursuit," in *Conf. on Supercomputing*, Portland, OR, 1999, p. 44.

[9] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *Intl. Conf. on Dependable Sys. and Net.*, 2009, pp. 285–294.

[10] W. Shewhart, *Economic control of quality of manufactured product.* American Society for Quality Control, 1931.

[11] T. H. D. Nguyen, B. Adams, J. Zhen Ming, A. E. Hassan, M. Nasser, and P. Flora, "Automated verification of load tests using control charts," in *Asia Pacific Soft. Eng. Conf.*, Ho Chi Minh City, Vietnam, 2011, pp. 282–289.

[12] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Intl. Conf. on Perf. Eng.* Boston, MA: ACM, 2012, pp. 299–310.

[13] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Conf. on Uncertainty in Artificial Intelligence*, San Mateo, 1995, pp. 338–345.

[14] J. R. Quinlan, "C4.5: Programs for machine learning," *Mach. Learn.*, vol. 16, no. 3, pp. 235–240, September 1993.

[15] M. Minsky and P. Seymour, *Perceptrons.* Oxford, England: M.I.T. Press, 1969.

[16] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Z. Ying, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Intl. Conf. on Qual. Soft.*, 2010, pp. 32–41.

[17] H. W. Gunther, "Websphere application server development best practices for performance and scalability," *IBM WebSphere Application Server Standard and Advanced Editions - White paper*, 2000.

[18] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," *Mach. Learn.*, vol. 59, no. 1-2, pp. 161–205, May 2005.

[19] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[20] E. Frank and R. Kirkby, "R: Randomtree," 2012. [Online]. Available: http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/RandomTree.html

[21] R. Kohavi, "The power of decision tables," in *European Conf. on Machine Learning*, Heraklion, Crete, Greece, 1995.

[22] W. W. Cohen, "Fast effective rule induction," in *Intl. Conf. on Machine Learning*, Tahoe City, CA, 1995.

[23] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," in *Intl. Conf. on Machine Learning*, San Francisco, CA, 1998.

[24] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, Jan 1991.

[25] J. G. Cleary and L. E. Trigg, "K*: An instance-based learner using an entropic distance measure," in *Intl. Conf. on Mach. Learn.*, Tahoe City, CA, 1995, pp. 108–114.

[26] E. Frank, M. Hall, and B. Pfahringer, "Locally weighted naive bayes," 2003.

[27] S. L. Cessie and J. C. V. Houwelingen, "Ridge estimators in logistic regression," *Journal of the Royal Statistical Society*, vol. 41, no. 1, pp. 191–201, 1992.

[28] J. C. Platt, "Advances in kernel methods," B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. Cambridge, MA: MIT Press, 1999, ch. Fast training of support vector machines using sequential minimal optimization, pp. 185–208.

[29] Information Retrieval Project, *Information Retrieval: Ground Rules for Indexing.* University of Dayton Research Institute, 1963.

[30] T. Mitchell, *Machine Learning.* New York, NY: McGraw-Hill, 1997.

[31] H. Malik, "A methodology to support load test analysis," in *Intl. Conf. on Soft. Eng.* Cape Town, South Africa: ACM, 2010, pp. 421–424.

[32] S. Ghaith, M. Wang, P. Perry, and J. Murphy, "Profile-based, load-independent anomaly detection and analysis in performance regression testing of soft. systems," in *European Conf. on Soft. Maintenance and Reengineering*, 2013, pp. 379–383.

[33] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic performance analysis of load tests," in *Intl. Conf. in Soft. Main.*, Edmonton, 2009, pp. 125–134.

[34] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," in *Intl. Conf. on Machine Learning*, Williamstown, MA, 2001, pp. 202–209.

[35] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *Symp. on OS Design & Impl.*, San Francisco, CA, 2004, pp. 231–244.

[36] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *Intl. Conf. on Dependable Sys. and Net.*, 2005, pp. 644–653.

[37] G. Bronevetsky, I. Laguna, B. R. De Supinski, and S. Bagchi, "Automatic fault characterization via abnormality-enhanced classification," in *Intl. Conf. on Dependable Sys. and Net.*, 2012, pp. 1–12.