UNDERSTANDING CONTEXTUAL FACTORS IN REGRESSION TESTING TECHNIQUES

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Jeffrey Ryan Anderson

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Computer Science

April 2016

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

UNDERSTANDING CONTEXTUAL FACTORS IN REGRESSION TESTING TECHNIQUES

By

Jeffrey Ryan Anderson

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Hyunsook Do
Co-Chair

Dr. Saeed Salem
Co-Chair

Dr. William Perrizo

Dr. Gary Goreham

Approved:

8 April 2016	Dr. Brian Slator
Date	Department Chair

ABSTRACT

The software regression testing techniques of test case reduction, selection, and prioritization are widely used and well-researched in software development. They allow for more efficient utilization of scarce testing resources in large projects, thereby increasing project quality at reduced costs. There are many data sources and techniques that have been researched, leaving software practitioners with no good way of choosing which data source or technique will be most appropriate for their project.

This dissertation addresses this limitation. First, we introduce a conceptual framework for examining this area of research. Then, we perform a literature review to understand the current state of the art. Next, we performed a family of empirical studies to further investigate the thesis. Finally, we provide guidance to practitioners and researchers.

In our first empirical study, we showed that advanced data mining techniques on an industrial product can improve the effectiveness of regression testing techniques. In our next study, we expanded on that research by learning a classification model. This research showed attributes such as complexity and historical failures were the most effective metrics due to a high occurrence of random test failures in the product studied. Finally, we applied the learning from the initial research and the systematic literature survey to develop novel regression testing techniques based on the attributes of an industrial product and showed these new techniques to be effective. These novel approaches included predicting performance faults from test data and customizing regression testing techniques based on usage telemetry. Further, we provide guidance to practitioners and researchers based on the findings from our empirical studies and the literature survey. This guidance will help practitioners and researchers more effectively employ and study regression testing techniques.

ACKNOWLEDGEMENTS

Many thanks go to my advisors, Hyunsook Do and Saeed Salem for their advice, guidance, and continual support in my research. I would also like to thank William Perrizo and Gary Goreham for serving on my advisory committee. I am grateful to the entire NDSU computer science department and the many professors who have educated me over the years. I would also like to thank my wife, Jennifer, and my children, Nate and Joe, for allowing me to embark on this large investment time for so long. I also thank my parents, Kurt and Deb Anderson, for inspiring continued education throughout my life. Finally, I'd like to thank my boss, Randy Gerhold, for supporting my pursuit of a degree while also continuing my career at Microsoft.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Regression testing is one of the core activities employed in software projects to ensure quality and to minimize risk when making software changes. Without regression testing, the maintenance cost in software would rise dramatically. At the same time, regression testing itself is not without cost. The tests require hardware resources such as computers to run on. Regression tests also have a personnel cost related to analyzing and reacting to the results of regression test runs.

Numerous techniques and tools have been proposed and developed to reduce the costs of regression testing and to aid regression testing processes, such as test suite reduction, test case prioritization, and test case selection. Test suite reduction consists of removing unneeded or duplicated tests. Test selection is the process of identifying which regression tests are applicable to a given software change. Test case prioritization is the process of ordering those regression tests to either find faults earlier or to maximize the number of faults found without running all the tests. Recent surveys on regression testing techniques [13, 28, 83] provide a comprehensive understanding of overall trends of the techniques and areas for improvement. Because the underlying motivation for regresion testing techniques is to improve the cost effectiveness of regression testing, research has also been done showing that proper thresholds and weightings must be used in regression testing techniques in order to improve economic benefits [18].

In the field of regression testing techniques, a wide range of software metrics have been employed. These range from static code attributes such as complexity and code coverage metrics, to more dynamic metrics like code change (churn) and historical test results. Some techniques even consider metrics outside of the software itself such as team metrics (e.g., full time or vendor resources), temporal attributes, and requirements.

This wide ranging set of metrics has been analyzed in a variety of ways. Early research showed that the use of simple attributes could be effective in improving regression testing techniques. For instance, code complexity can effectively prioritize test cases and simple coverage information can help improve regression test selection techniques [59]. More recently, increasingly complex and sophisticated data mining techniques have been employed including regression analysis, association rule mining, and classification techniques [6, 77]. As might be expected, each new publication of a combination of software metrics and data mining techniques shows that it is more effective in selecting or prioritizing tests than a naïve baseline.

A limitation of this body of research is a lack of a fundamental understanding of in which circumstances and environments each software metric and technique should be employed in order to improve regression testing approaches. For instance, our recent research showed that code churn information was a far less effective metric for test case prioritization than simple code complexity if the stability of test cases in a given environment was high enough compared to the overall test failure rate [6]. Without this understanding of how the environmental factor of test case stability impacts the effectiveness of the test case prioritization technique, the previous research showing churn to be a good predictor in test case prioritization may have been incorrectly applied to this software product.

1.1. Goal of this Dissertation

Our thesis is that the data mining and metrics employed in regression testing techniques must take into account both internal and external contextual factors in order to be useful and effective in practice.

Failure to account for these influencing factors can lead to less effective application of regression testing techniques and ultimately lose economic benefit. If this thesis is true, then researchers and practitioners in this area of research can understand how metrics and techniques interact with the internal and external environmental factors in order to properly engage in regression testing techniques.

1.2. Approach to Meet this Goal

To address this thesis, we have proposed an approach that guides us to empirically investigate our research problem. First, we introduce a conceptual framework for understanding and evaluating regression testing techniques as described in Section 4.1. Next, we performed a literature survey to understand the state of the art. Finally, we performed a family of empirical studies as described in Chapter 5.

In the first empirical study, we applied data mining techniques to an industrial software project using test history metrics to verify that advanced data mining techniques can improve effectiveness over naïve baselines. We found advanced data mining techniques to improve effectiveness, though the improvement is small and the cost of applying the techniques is significant. This research in Section 5.1 is based on research published in the *Proceedings of the Working Conference on Mining Software Repositories (MSR)* in 2014[6].

Following that, we applied classification techniques to understand which metrics and techniques yield the best results, and contrasted those findings with the existing regression testing techniques corpus. We found that attributes such as complexity and historical failures were the most effective metrics due to a high occurrence of random test failures in the product studied. We also found that the effectiveness of various data sources varied throughout the product life cycle. The results of this research shows that our thesis holds, and contextual factors of the product have a significant impact on the effectiveness of both data sources and techniques. This research in Section 5.2 is based on research published in the *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE)* in 2015[7].

Finally, we applied the learning from the initial research and the systematic literature survey to develop novel regression testing techniques based on the attributes of an industrial product and showed these new techniques to be effective. We were able to predict performance faults based on test response times. This research in Section 5.3 is taken from research published in the *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)* in 2015[5]. We have also shown that telemetry from software usage can be used to improve the effectiveness of regression techniques by customizing regression testing based on software use. That research is pending publication at this time.

1.3. Organization of this Dissertation

The rest of this paper proceeds as follows. Chapter 2 discusses the background and related work. Chapter 3 describes the current state of the art in the areas of research. Chapter 4 describes the approach used in researching this dissertation, while Chapter 5 discusses the empirical studies performed to extend the state of the art. Chapter 6 provides guidance to industry and researchers. Chapter 7 summarizes the results and discusses avenues of future research.

2. BACKGROUND AND RELATED WORK

In this section, we discuss background and related work of regression testing techniques.

2.1. Motivation

This work was motivated by personal experience at Microsoft working on the *Microsoft Dynamics AX*[60] product. This is a very large Enterprise Resource Planning (ERP) product that enterprises use to run their business, including everything from accounting to retail to supply chain management capabilities. As with any large software product many thousands of tests exist, and those tests take considerable amounts of time and resources to run on a regular basis. Since research has shown the economic benefits of applying regression testing techniques[20], we wish to effectively employ those techniques for our product.

There are two major classes of tests for this product, unit tests and integration tests. Unit tests focus on a single class or piece of logic and usually take less than a second to execute. Integration tests test the combination of multiple pieces of functionality and often take longer to execute. Many thousands of both unit tests and integration tests are in active use in the product.

During each code submission, a subset of unit tests are run, consisting of tests marked as important by developers, together with a subset of tests selected automatically by the checkin system based on changes being made (also referred to as "churn"). The unit tests are run on every checkin so usually it is not possible for a failing test to exist on the checked in version of code.

The integration tests take significantly longer to execute, and as such are only executed once every few days, currently requiring multiple days to execute and spanning over 100 computers. Since many changes have been made between any two integration test suite runs, all integration tests are run during every regression test run. Since these tests are not run as part of the checkin process, it means it is also possible (and common) for test failures to exist based on checked in code. It is these integration tests where we wish to apply regression testing techniques to reduce costs and increase economic benefits.

The challenge is selecting which regression testing techniques to apply and what data to use. Many different data sources and techniques have all been shown to be effective with varying levels of effectiveness. Sometimes this research even appears contradictory. For instance, code coverage has been shown to be one of the most effective techniques[62], but Tonella et al.[78] showed that in some situations, it is outperformed by manual identification of important tests. In another example, Walcott et al.[80] showed that the cost of applying regression techniques can be more costly than testing without them, while Do et al.[20] showed economic benefits to applying regression testing techniques. Seemingly the only consistent message from existing research is the only bad choice is to do nothing at all.

From this seemingly confused corpus, we need a way to determine which data sources and techniques will be most effective for our particular product and environment. This gives rise to our thesis. We need to be able to **recognize the unique context of our product and how that context affects the successful application of regression testing techniques**. To research this thesis, we employ a combination of literature review and empirical studies on the *Microsoft Dynamics AX* product.

2.2. Existing Research

Regression testing techniques are important areas of research in reducing the cost of quality activities in software projects [56]. Identifying test cases for elimination or prioritization can decrease the time to finding and fixing bugs, as well as minimize the resources required to run test suites. A wide variety of regression testing techniques have been studied, and an overview of many of these techniques is discussed by Yoo and Harman [83], Engstrom et al. [28], and Catal and Mishra.[13]

The foundation for our work starts from the basic concepts on regression testing techniques and their economic impacts from various researchers [10, 22, 17, 21, 19, 23]. This includes the foundational research on the use of test prioritization [71], as well as the first empirical evidence that these techniques were economically advantageous [21].

A study relevant to our work was done by Nagappan and Ball [63]. The proposed approach has been used in many companies including Microsoft and is often referred to as *churn based testing*. It is the act of correlating the code changes made in a given build with the tests that should be run on that build. Another related area of study receiving attention is in the area of applying network analysis to software defects. These techniques range from network analysis of dependency graphs [87] to applying dependency graphs in determining testing strategies [50].

More advanced techniques have recently been evaluated, including linear programming [37, 42, 61], genetic algorithms [81, 53], and advanced classification techniques [7]. Many of these techniques begin to introduce more dynamic measures of churn [9, 19, 73, 70], an indication of which changes have been made since a previous release.

Beyond the regression testing area, in the software engineering field, data mining has recently been used to analyze the voluminous amounts of data generated from version control systems or fault reporting systems [54, 64, 88]. Nagappan et al. [64] presented an approach to mining data to predict error-prone components. To investigate their approach, they retrieved various software metrics and failure information from the version control system for five software projects developed at Microsoft. Zimmermann et al. [88] presented an approach that applies data mining to provide information related to code changes to programmers, such as suggestions or predictions of likely changes. Livshits and Zimmermann [54] presented an automatic way to discover common error patterns that reside in software revision histories by combining data mining with dynamic analysis techniques. All of the above approaches have shown that data mining could be useful in finding patterns and relationships that can help various software engineering tasks from massive software repositories.

Zimmermann and Nagappan have done a sizable amount of research about software dependency metrics and their impact on defects [87, 89]. Their research shows that software defects may be accurately predicted by looking at the interplay between software components. Further, Zimmermann et al. [88] have shown that software repositories can be mined to obtain this information that can then be used for fault prediction.

A recent literature review by Hall et al. [35] examined research on data mining software sources where the results were used to predict fault locations, severities and quantities. Yoo and Harman [84] performed a survey of techniques used in test selection and prioritization. Catal and Mishra[13] performed a mapping study of test prioritization, focusing on which aspects have been studied and current trends. Engstrom et al.[28] performed a survey of selection techniques along with qualitative analysis.

2.3. Regression Testing Techniques

In this section, we provide background information on three widely used regression testing techniques.

2.3.1. Reduction, Selection, and Prioritization

Regression testing is an important aspect of software development projects, ensuring that the quality of software remains high as changes are made and allowing developers to be confident about making those changes. While important, there are various costs associated with regression testing. For example, the regression tests may be costly to create, and false failures in the tests may involve costly analysis time. However, most of the cost associated with regression testing in many projects is the amount of time the tests themselves take to execute. In some situations, such as reported by Anderson et al. [7], this may involve hundreds of computers over multiple days. Even in smaller projects in which regression tests take just minutes to run, further reducing the time involved can be beneficial by allowing those tests to be run even more frequently as changes are being made.

There are three primary techniques that seek to reduce the cost of regression testing.

- **Test Suite Reduction:** Test suite reduction seeks to identify tests that can be removed from the test suite without reducing the quality of the test suite beyond some threshold. This technique finds test cases that cannot be applied to a new version of the program any longer (obsolete test cases), or test cases that produce the same coverage of the program as other test cases (redundant test cases) [38]. By removing these test cases, engineers can reduce the cost of exercising, validating, and managing these test cases over time.

- **Regression Test Selection:** Regression test selection is similar, but does not permanently remove the tests from the regression suite. Rather, test selection picks which tests should be rerun in any given test suite execution. This is often based on information like "churn," the code that has changed since the last regression test suite run. This technique relies on the assumption that a test that was previously passing should continue passing if no code executed by the test has been changed. Often, test selection techniques will include support for executing tests again that have not been run for a longer period of time as well.

- **Test Case Prioritization:** Test case prioritization offers an alternative approach to improving regression testing cost-effectiveness. Instead of selecting or removing test cases, prioritization provides an ordering of tests in an attempt to execute more important tests earlier in the test suite. By doing this, engineers can reveal faults early in testing, which allows them to begin debugging earlier. Alternatively, if the regression suite run needs to be stopped due to time constraints, the most important tests have been executed, lowering the number of faults that might otherwise be missed through less appropriate runs of partial test suites.

2.3.2. Evaluation Metrics of Regression Testing Techniques

The primary goals of the three regression testing techniques are different from each other, and therefore their effectiveness is evaluated using different metrics.

Test suite reduction techniques seek to reduce the size of the test suite; thus, the rate of test suite reduction has been used to evaluate test suite reduction techniques. Because of the reduction in the size of the test suite, fault detection ability or code coverage could be lost. Therefore, in addition to measuring the rate of test suite reduction, fault detection rate and code coverage were used to evaluate test suite reduction techniques.

Regression test selection techniques try to find a subset of test case to be rerun for the modified version of the program. There are two classes of regression test selection techniques, safe and unsafe. Safe techniques guarantee that test cases not selected could not have exposed faults in the modified program. For safe techniques, typical evaluation metrics are the number of test cases selected, the test case reduction rate, or the time required to execute the selected subset of the test suite.

For unsafe techniques, the unselected test cases might cause a loss in fault detection abilities, so there are two aspects to selection that can be traded off. By increasing the number of tests selected, more faults will be detected. But, increasing the number of tests selected reduces the execution cost savings. Unsafe techniques have recently gained more popularity, as the benefits gained in further reducing the suite size often make a slight decrease in fault detection a viable tradeoff.

When using unsafe selection techniques, frequently used measurements are precision and recall. Precision measures the ratio of selected tests that yielded faults to selected tests that did not. So, an increase in the number of tests selected will *decrease* precision. Recall measures the ratio of tests with faults that were detected. So, an increase in the number of tests selected will *increase* recall. These two measurements can be combined into a single value call the F-measurement or F-score, which is the harmonic mean of the two values.

$$F - measure = \frac{2 \times Precision \times Recall}{(Precision + Recall)}$$

A good test selection technique is one that has a higher F-score, meaning that both precision and recall have been improved without negatively impacting each other.

As an example, consider the set of tests shown in Table 2.1, and three different selection techniques. Of these nine tests, five will find a fault in the current test suite run. Selection Technique A selects seven tests to run, of which three detect faults. So the recall for technique A would be 3 / 5 = 0.600, and the precision would be 3 / 7 = 0.429.

Selection Technique B selects three tests to run, of which two detect faults. This yields a precision of 2 / 3 = 0.667 and a recall of 2 / 5 = 0.400. Comparing techniques A and B is difficult, since one has better precision and the other better recall. By computing the F-score for each, we find that A has an F-score

Table 2.1. Sample Regression Test Selection

Test Name	Finds Fault	Selection Technique A	Selection Technique B	Selection Technique C
T1	Yes	Yes	No	Yes
T2	No	Yes	No	No
T3	Yes	Yes	No	Yes
T4	No	Yes	No	No
T5	Yes	No	Yes	No
T6	No	Yes	Yes	No
T7	Yes	No	Yes	Yes
T8	No	Yes	No	Yes
T9	Yes	Yes	No	Yes

of (2 * 0.600 * 0.429) / (0.600 + 0.429) = 0.500, while B has an F-score of (2 * 0.667 * 0.400) / (0.667 + 0.400) = 0.500. This allows a direct comparison between the techniques and we can say that they are roughly equivalent.

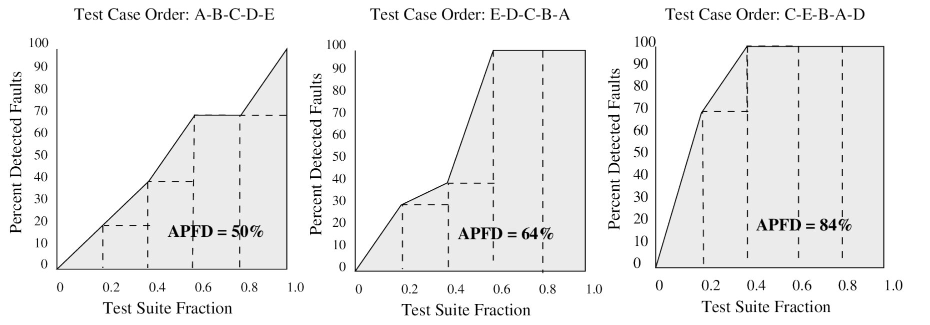
Extending this example, Technique C selects five tests, of which four find faults. The precision is 0.800 and the recall is 0.800, yielding an F-score of 0.800. Comparing this with Techniques A and B, we see that C is clearly a better test selection technique as it maximizes both precision and recall.

The main goal of *test case prioritization* techniques is to increase the rate of fault detection of a prioritized test suite. To measure the rate of fault detection, a metric called Average Percentage Faults Detected (APFD) has been introduced [71]. APFD is a value ranging from 0 to 100, where higher numbers imply faster (better) fault detection rates. Formally stated, let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF$_i$ be the first test case in ordering T$'$ of T, which reveals fault i. The APFD value for test suite T$'$ is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

To illustrate an example of APFD, consider ten faults in a program that has a test suite of five test cases, **A** through **E**. The fault detecting abilities of each test case are shown in Figure 2.1.A. In this example, we order the test cases **A–B–C–D–E** to form prioritized test suite $T1$. Figure 2.1.B shows a graph of the percentage of detected faults versus the fraction of $T1$ used. Test case **A** reveals two out of ten faults, meaning that 20% of the faults have been detected after 0.2 of $T1$ has been executed. Test case **B** reveals two more faults, meaning that 40% of the faults have been detected after 0.4 of $T1$ has been executed. The

test	fault									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B						x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x



A. Test suite and faults exposed **B. APFD for prioritized suite T1** **C. APFD for prioritized suite T2** **D. APFD for prioritized suite T3**

Figure 2.1. Example of APFD taken from Do and Rothermel [21]

area under the curve (AUC) is then the weighted average of the percentage of faults detected over the life of the test suite, referred to as APFD. In this particular example, the APFD is 50%.

Figure 2.1.C extends from this example with an alternate test case ordering of **E–D–C–B–A**. In this case, the APFD measurement is 64%, meaning that this prioritization is faster at detecting faults than $T1$. Figure 2.1.D shows another test suite $T3$ with a test case ordering **C–E–B–A–D**. This particular ordering yields the earliest possible detection of faults, with an optimal APFD of 84%.

Some variations of APFD have been proposed. APFD$_c$ accounts for varying test case and fault costs [27], and NAPFD [68] considers cases in which the rate of fault detection of different-sized test suites is being compared.

3. THE STATE OF THE ART OF REGRESSION TESTING TECHNIQUES

To investigate the state of the art of the use of data mining in regression testing techniques, we performed a literature review of research papers from the years 2000 to 2015. This review includes an overview of trends in research papers over these years, as well as an in-depth review of thirty three selected papers from this period which discuss attributes of software projects and how they are related to the effectiveness of regression testing techniques.

The results of this survey indicate that the only bad choice in selecting regression testing techniques is to do nothing at all. Any technique will provide some benefit. Data sources which provide more variation in values within the project are shown to be the most effective, as long as the data sources do not suffer from biases such as excessive false failures.

3.1. Measurement Techniques

When evaluating test case prioritization techniques, the average percentage of faults detected (APFD) score is the de facto standard. The APFD metric was first introduced in 1999 by Rothermel et al. [71] and has since been used in the vast majority of prioritization evaluation. Of the twenty papers focusing on test case prioritization we surveyed in detail, sixteen of them rely on APFD as the metric for determining the efficacy of the heuristic. The remaining papers focused more on the costs and benefits of the prioritization.

In the areas of test case selection and reduction techniques, the results are analyzed in various ways. Fault detection rate (FDR) is similar to APFD but involves a subset of tests as opposed to all the tests. Precision and recall, as well as the combination of precision and recall (F-measure) are also popular. In all of these cases, the rates of detection, precision, and recall are measured in terms of coverage data for the test. There does not appear to be any overall trend in these techniques, as they are spread evenly across papers.

3.2. Object Programs

The most widely used object programs for the empirical studies in the regression testing area are in the Siemens Suite. These programs were assembled at Siemens Corporate Research by Monica Hutchins and colleagues [43] These include TCAS, Schedule2, TotInfo, PrintTokens, and Replace. Other frequently used programs are Space, JTopas, XmlSecurity, JMeter, and Ant. Of the thirty-four papers surveyed, just over half (eighteen of them) relied primarily on programs from these suites. While the benefit of commonality is the ability to directly compare techniques on the same applications, we find that very few papers directly compare their techniques. This is discussed more later.

The concern with the use of common small programs is that the patterns observed in these programs may not generalize. The programs are fairly small compared to industrial applications, with only a few having thousands of lines of code. Considering that large industrial applications may have many millions of lines of code with individual modules larger than these applications, the applicability may be questioned. It should be noted that more recent studies have shown the applicability of these techniques in large industrial applications as well [6, 73, 69]. However, the effectiveness of data and techniques has varied widely depending on the types of applications [7].

The wide variation in effectiveness among programs in these suites is an important consideration. For instance, data from Mei et al. [58] showed that the programs that benefited the most from their technique (JUPTA) were the same programs that benefited the most from their baseline technique (prioritizing based on coverage data). Similarly, data from Hao et al. [37] shows very large variation in effectiveness between applications.

The implication of this variation in effectiveness is that some applications are just better suited to prediction than others. Mirarab et al. [61] pointed out that how much a given variable varies between applications appears to impact its effectiveness in prediction. For instance, techniques based on coverage data will be more effective when the level of coverage varies widely between test cases for the application. This is important when selecting both data sources and techniques to use for a given application.

3.3. Data Sources

Coverage is by far the most common data source for prediction techniques, used in twenty-eight of the papers surveyed. This was seen equally across prioritization, reduction, and selection. After coverage, churn is the second most common attribute, used in five of the papers, including a few in combination with coverage. Complexity, test granularity, fault history, and a variety of others such as output uniqueness make up the remainder of the data sources.

Coverage is popular, as it is easy to obtain in most applications. The coverage data for the Siemens programs as well as other common applications is readily available. Standardized tools, available in most languages including C++ and Java, allow for the collection of code coverage data in cases where it is not initially available.

One aspect that is not well studied is how the data sources vary in effectiveness for a given application. As previously mentioned, more variation in the values for a given data source is advantageous. But Tonella et al. [79] showed that user knowledge of important tests significantly outperformed coverage-based prioritization. Similarly, Anderson et al. [7] showed that churn (traditionally considered a good predictor) performed poorly in environments with a high occurrence of flaky tests. Flaky tests are non-deterministic tests which can fail for reasons other than regressions or bugs, and can fail even without changes to the underlying code [55]. The impact of data sources on the effectiveness of predictions is an area that needs additional research.

More recently, some researchers have begun using specifications, state models, and other software artifacts to implement regression testing techniques. For example, Kim et al. [49] utilized reliable mapping between the model element and the regression tests for that element to select which tests should be run. Another example is Hemmati and Briand [40], who used similarity measures between models for test case selection. In some situations, this is a manual mapping, or in the case of model-driven systems the mapping may be directly available from the tooling that builds the system. In either case, the mapping must be reliable to ensure that mapping errors do not negatively impact the effectiveness of the regression testing techniques.

3.4. Techniques

Techniques are tightly related to data sources because the techniques are reliant upon the data available. For instance, greedy-coverage-based ordering of tests is only applicable if coverage data is available as a data source. However, the techniques are more diverse than data sources because for the same data sources, many different algorithms can be applied. Again using the example of coverage data, greedy-coverage-based ordering is a relatively simple algorithm. More complex examples include genetic algorithms and the use of classification algorithms.

The most popular technique is also the most simple, ordering by coverage. This was used in fifteen of the thirty-four papers surveyed. How the tests are ordered may vary slightly, such as selecting tests to maximize coverage with each additional test selected (greedy) or to maximize the increase in covered paths that are new (additional), as well as minor variations on these techniques. Generally, greedy coverage algorithms have been shown to be among the best techniques, as shown by Mirarab et al. [61].

With simple techniques such as ordering by coverage being fairly effective, many researchers have begun applying more advanced techniques such as linear programming [37, 42, 61], genetic algorithms [53, 80], and other techniques. A common theme in this research is more complex techniques yield a slight improvement in the prediction, but often not enough to justify the cost of the technique.

The costs of data collection, data curating, algorithm execution, and result analysis must be considered when evaluating techniques. For instance, Walcott et al. [80] found that it was more expensive to run the genetic algorithm in some cases than to run the full test suite, thereby negating the benefits of increased predictive power. Similarly, Anderson et al. [6] saw an increase of only 4% in effectiveness at the cost of very expensive computations for association rule mining. The collection of coverage data and the execution time of ordering by coverage data is often less costly in terms of both human time and machine processing time than more advanced data sources and techniques.

The implication from the results discussed in this section is that most benefits can be obtained with relatively simple and low-cost prioritization, reduction, and selection techniques. More advanced techniques may yield slight improvements but are often not worth the increased cost of applying the techniques. Often, the variation in effectiveness between applications is greater than the increase in effectiveness of moving to a more complex technique.

3.5. Baselines

The most common baseline used in evaluating regression testing techniques is random ordering or random selection, followed by running all tests. Thirteen of the thirty-four papers only compared against selection of random or all tests. Of those that did compare against less naïve baselines, the most commonly selected baseline was greedy coverage based. This appears to be a good baseline because, as previously discussed, the greedy coverage technique is one of the most effective simple techniques.

An interesting result was seen by Do et al. [22] in which occasionally random selection was shown to actually outperform untreated test prioritization. This is likely due to the way test suites grow over time. The implication of this finding is that selecting test cases randomly should not be considered equivalent with a project that has simply not performed selection, reduction, or prioritization of tests. In fact, sometimes randomizing the test cases can even improve results.

The key message when evaluating baselines, as previously discussed, is that the vast majority of benefits from prediction can be gained with relatively simple data sources and techniques. As such, a comparison against random ordering, retesting all or initial orderings is likely to always show an improvement, regardless of the effectiveness of the data sources or techniques. Given that greedy coverage is an easy technique to apply, and that coverage data is almost universally available, we recommend using greedy coverage as a standard baseline as new data sources and techniques are analyzed.

3.6. Trends

Because only thirty-four papers were selected for detailed review, not enough data is available to perform statistical analysis on trends in the data. However, looking at the details of the papers over time does yield some apparent trends in regression testing research. Figure 3.1 illustrates the object programs of research by year. As can be seen, in the early 2000s, the Siemens Suite together with Space were the primary object programs used in empirical studies. This is partially due to the strong working relationship between the authors of the papers, and partly due to the availability of these applications together with tests and other data sources for research. More recently, there has been a shift as more research is focusing on industrial applications and other open source programs. The most widely used open source programs for more recent studies include Ant, JMeter, XmlSecurity, and JTopas.

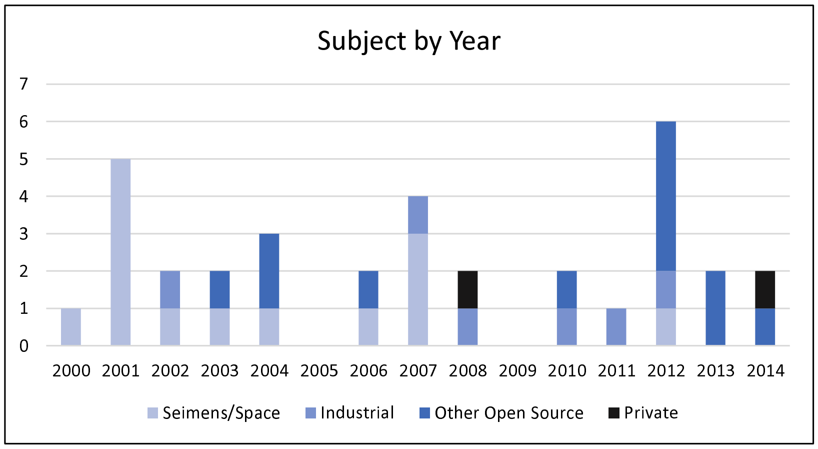


Figure 3.1. Paper Subject(s) by Year

Figure 3.2 illustrates another apparent shift over time in research. A decade ago, most research focused on coverage-based techniques, including the “greedy” and “additional greedy” algorithms. More recently, research has branched out to more varied and advanced techniques including linear programming, genetic algorithms, and others.

From these trends, it appears that the field of test case selection, reduction, and prioritization is maturing. Early research focused on a small number of applications with a small number of techniques.

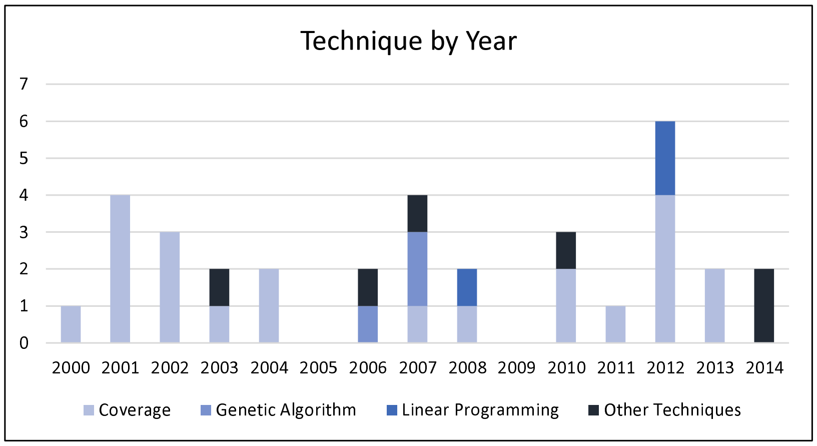


Figure 3.2. Paper Technique(s) by Year

Research in this area has now expanded and become more advanced, and it is being applied in an industrial setting. As these techniques start shifting out of research and into practice, it is important to understand the interplay between applications, data sources, techniques, and other project aspects to ensure successful technology transfer from academic research to industry.

4. APPROACH

This section describes the approach used in conducting the proposed research. Our motivation is to determine how contextual factors affect the effectiveness of regression testing techniques. To do this, we start by introducing a conceptual framework that provides a foundation for conducting our research. Using that conceptual framework, we performed a literature review to understand the corpus in this area as described in Chapter 3. We then performed empirical studies to understand the impact of internal and external contextual factors on the regression testing techniques. In particular, we utilized a large industrial application, *Microsoft Dynamics AX*, which involves several challenges that arise in testing. The following list summaries the steps in our approach.

- Construct a conceptual framework with which to evaluate regression testing techniques
- Conduct a literature survey of the current state of the art
- Investigate whether advanced data mining techniques are applicable to this product
- Determine which product attributes are most important in employing regression testing techniques
- Investigate the applicability of these techniques for use in finding non-functional bugs
- Develop new ways of applying regression testing techniques based on the unique attributes of the product

4.1. Conceptual Framework

This section describes a conceptual framework that provides guidelines about how the proposed research and application of data mining in regression testing techniques can be conducted. This framework is depicted in Figure 4.1. The first box in the figure is the source application. This is the software project for which the regression testing techniques are being applied. Data sources from this application are obtained, such as code coverage data or code churn information. A technique is applied to the data sources, such as a greedy algorithm based on coverage, or a classification technique using all available data sources. The output of the technique may be a recommended prioritization, or a selection of tests to remove from the suite, or a selection of tests to execute. This output is then evaluated against some ground truth using measures such as the average percentage of faults detected or the number of test cases to be rerun.

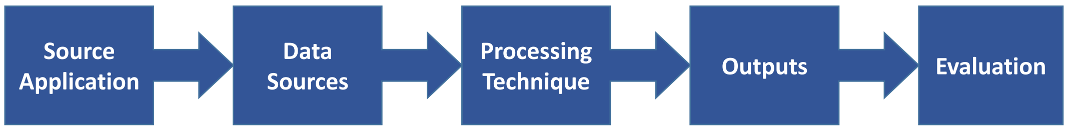


Figure 4.1. Formalized Conceptual Framework for Regression Testing Techniques

All of our research, as well as the guidance in Chapter 6, follows the conceptual framework in Figure 4.1. While each step by itself is important in the process, regression testing techniques are most successful when considerations are made of all steps in the framework.

4.2. Advanced Data Mining Technique Applicability

The first step in our approach is to determine whether advanced data mining techniques can be used to more effectively employ regression testing techniques in a large industrial application such as *Microsoft Dynamics AX*. We also need to determine if more advanced data mining techniques improve the effectiveness of the regression testing techniques.

To do this, we used test result data from 64 full regression test suite runs spanning over a year of development time. Using this data, we applied multiple test case prioritization techniques ranging from naive to more advanced data mining-based.

- Random ordering (Very naive, but also current practice in this product)
- Order by most frequent failure (Simplistic technique)
- Order by association rule mining (Advanced data mining technique)

We performed this study in two ways: once with all data available, and again looking only at a recent window of data. This additional dimension of the study allows us to determine if more advanced data mining techniques involving temporal restrictions further improve the test prioritization.

This empirical study showed that regression techniques *can* be successfully applied to the *Microsoft Dynamics AX* product to improve the effectiveness of test case prioritzation. Further, it showed that more advanced data mining techniques such as association rule mining *do* improve the effectiveness of those techniques. Also, the application of temporal constraints on data sources further enhance the regression testing technique effectiveness.

Through qualitative analysis, we also learned why some of these techniques were so effective. For instance, the temporal restrictions on source data were effective as different parts of the product development life cycle yielded different patterns of regression test failures. Areas under active development were more likely to see additional failures, while areas being stabilized saw fewer. From these qualitative discoveries, we learned that even within a single product release, the effectiveness of different attributes varies over time.

A more detailed discussion of this empirical study is available in Section 5.1.

4.3. Determine Most Influential Attributes

Once we determined that advanced data mining and regression testing techniques could be used in the product, our next step was to understand which attributes were the most influential. In our first study, the attributes were limited to only test failure information. As described in Section 2, other research has shown that many different software attributes can be used in regression testing techniques.

To investigate which attributes were most applicable in this case, we used the same data from the first study with several additional attributes. These attributes include churn, organizational attributes, complexity attributes, and historical results. Using these attributes, we learned a classification model and used it to again prioritize test cases.

The results of this study were surprising, as we found many of the attributes traditionally considered to be the most effective in prioritization, like churn, had very low predictive power. Other attributes not normally considered as effective, like historical failure trends, were shown to be far more powerful. With further investigation, we discovered this was due to specific contextual factors of the product, such as a high incidence of false test failures. Similarly, the source control gate system and use of extensive testing during code submission made churn less predictive.

From this study, we were able to show explicitly how contextual attributes of a product have a significant impact on the effectiveness of different data sources and techniques in applying regression testing techniques. More detailed information about this study is available in Section 5.2.

4.4. Application to Non-Functional Bugs

Next, we wished to see if the techniques and learning from the previous studies could be extended to new applications. Traditionally, regression testing techniques such as test case prioritization are used to order functional tests, looking for code defects. Many products, such as *Microsoft Dynamics AX*, have a need for a more optimal way of finding non-functional faults, such as performance problems. To study this, we used the same advanced data mining and prioritization techniques from the previous study to try to predict performance tests that can detect missing index performance faults.

The results of this novel approach to using regression testing techniques proved effective, but it was less useful than we hoped. While the techniques did properly prioritize performance tests, the costs of obtaining the data sources for prioritization left the technique untenable for actual use. This is due to the high configuration and execution costs necessary to obtain stable test execution timings on tests which take just seconds to execute. From this study, we learn that not only context of the product, but also context of the data sources themselves is important in effectively and efficiently applying regression testing techniques.

More detailed information about this study is available in Section 5.3.

4.5. Applying Regression Testing Techniques Based on Product Context

In this empirical study, we seek to apply the learning from previous empirical studies and develop a novel approach to regression testing techniques based on the unique context of a product. Two unique attributes of the *Microsoft Dynamics AX* product are the fact that it is delivered as a Software-as-a-Service (SaaS), and the availability of pervasive telemetry on product usage.

Based on the findings from previous studies, we used highly-discriminative data sources which are inexpensive to consume, specifically usage telemetry. Using these data sources, we compute highly-tailored test case prioritizations for each software installation, something important in a SaaS environment where downtime is measured in minutes. We applied multiple techniques ranging from simplistic to advanced data mining techniques, measuring the effectiveness of the resulting prioritizations.

In this study, we were able to show that the selection of the data sources and processing techniques based on the unique context of the product were effective in producing effective prioritizations.

More detailed information about this study is available in Section 5.4.

5. EMPIRICAL STUDIES

To investigate the approaches described in Section 4, we performed a family of empirical studies. We describe each of them in the following subsections.

5.1. Apply Advanced Data Mining Techniques to an Industrial Product

In an empirical study on an industrial product, we seek to determine if advanced data mining techniques for test case prioritization can improve results. This study provides interesting learning on the use of association rule mining and sliding time windows in the area of test case prioritization and was presented at MSR (Mining Software Repositories) of ICSE 2014. It also serves as a foundation for future research on this product by demonstrating that data mining techniques can be successfully applied to this industrial product to improve test case prioritization. Future research builds on this by examining the interplay between different metrics and classification techniques.

While many current regression testing techniques focus on static analysis and explicit relationships among tests and product elements, we believe that there are many additional relationships among software artifacts (e.g., software code, test code and historical test results) that are not obvious or even not visible to the owners of the software artifacts. If these underlying relationships about software systems are able to be discovered, regression testing processes can be improved by selecting and running more important test cases for failure detection.

To investigate this possibility, we apply a repository mining approach to several types of data commonly found in software repositories, or available by performing calculations on the contents of those repositories. These data sets include software test coverage data, prior defect-revealing behavior related to test cases, previous test results, build history information, and smoke tests.

Using these data sets, we propose and develop two techniques. The first approach is referred to as *most common failures* in which test cases that failed the most previously are recommended as test cases that are likely to fail in the future. The second approach is referred to as *failure by association* where failures in certain subsets of tests are used to determine other subsets that are likely to fail. Both of these techniques are then paired with an examination of age of data we refer to as *windowing*. The techniques are run once using all historical data, and then again using only data from the most recent runs.

To investigate the effectiveness of the proposed techniques, we have designed and performed an empirical study using an industrial product, Microsoft *Dynamics AX* that contains real failure information. Our results show that using information about historical failures can better predict future failures. Specifically, using techniques such as frequency of failures and failure by association based on a window of recent builds outperforms the same techniques on the entire historical dataset.

5.1.1. Approach

The goal of this research is to select effective regression tests that are likely to detect failures and to run them, giving development extra time to fix any issues discovered while the remaining tests are run. If a more useful set of tests runs earlier, problems found by such tests can be fixed up to two days sooner based on a three day test run.

Figure 5.1 demonstrates the mechanism by which test case selection occurs. At the start of any test run, the test cases for that regression test run are selected. This is accomplished by running a selection algorithm which reads from all test artifacts as well as historical test results from previous builds. The output of that algorithm is a set of tests deemed more likely to fail in the current regression test run. That set of tests runs first on the pool of test computers. Once execution of the selected tests completes and the test machines are free again, then the remaining tests are executed.

A final note in Figure 5.1 is that a small random set of tests out of the pool of all tests are pre-executed as part of the build process. These are referred to as "smoke tests" and are used to help determine the general quality of the build and whether or not it can be used for a full regression test run. The selection algorithm may also consider the results of those smoke tests in determining the selected tests deemed most likely to fail for that test run.

This study can be described abstractly as follows. Consider a set of n regression tests labeled T_1, T_2 through T_n. We define A_i to be the set of all tests runs in the i^{th} regression suite run: $A_i = \{T_{i1}, T_{i2}, ...T_{in}\}$

The union of all tests from all test runs is defined as A. Note that A differs from A_i since a given test may or may not be run in a given regression test run due to infrastructure issues, time pressure, active development causing the test to be temporarily disabled, or various other reasons.

Let F_i be the set of all failed tests in a given regression test run. So if m failures existed in regression test run i, we would define: $F_i = \{T_{i1}, T_{i2}, ...T_{im}\}$

Only tests which were run in a given regression test run may actually fail, i.e., $F_i \subseteq A_i$.

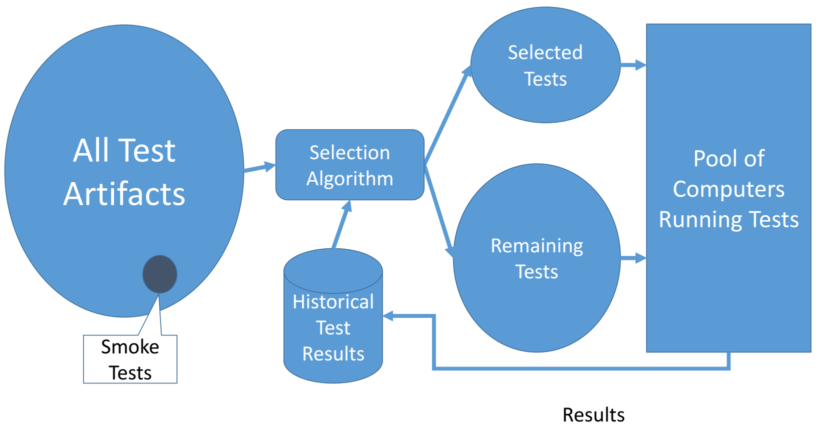


Figure 5.1. Test Selection Process

The goal of this study is to predict F_i as accurately as possible for a given regression test run. We will therefore define the set of k predicted failures for regression test run i as P_i such that: $P_i = \{T_{i1}, T_{i2}, ...T_{ik}\}$

Only tests that have failed in previous regression test runs can be predicted to fail in the current test run. So, we have that: $P_i \subseteq \bigcup_{j=1}^{i-1} F_j$

There is however no guarantee that P_i and F_i overlap in any way. If we were able to perfectly predict the failures in regression test run i, then we would have a case where $P_i = F_i$, but this perfect prediction is likely not possible. We can define the set of correctly predicted test failures C_i as the intersection between the predicted tests and the test that actually failed, i.e., $C_i = P_i \cap F_i$.

Increasing the set of correctly predicted tests is important, as it will increase the bug finding abilities of regression test run i. However, it should be noted that by increasing the size of P_i, the size of C_i can also be increased, to the point where if $P_i = A_i$, then we would have $C_i = F_i$. While all failed tests would have been correctly predicted, this is not a good situation, since $|P_i|$ would be significantly larger than $|F_i|$. These extra predicted failures that did not occur would require additional test resources to run, but would not find any bugs. Therefore what we actually want to do is maximize $|C_i|$ while minimizing $|P_i|$.

5.1.1.1. Most Frequent Failures

We refer to the first algorithm for predicting failures as *Most Frequent Failures*. In this approach, a threshold is used, and any test which failed at least as many times as the threshold in previous test runs is predicted to be likely to fail again.

5.1.1.2. Failure by Association

The second algorithm used in predicting likely failures is referred to as *Failure by Association*. In this technique, we use concepts from association rule mining to predict failures. Association rule mining is a technique by which a database of historical transactions is analyzed, and a set of rules are determined which indicates associations between items in the transactions [2]. Association rules mined from previous transactions can be utilized for predicting future associations. For simplicity we often refer to failure by association by the acronym ARM, since it relies on association rule mining concepts.

Our failure by association approach is similar to standard association rule mining, with one major difference. Instead of determining the association rules ahead of time, we run the rule mining algorithms separately for each test run being analyzed. We did this for two reasons. First, the number of transactions in our system is relatively small, being based on only 64 total regression test runs. So if a static database were used such as starting at halfway through the project, this greatly reduces both the number of test runs we can analyze as well as the number of transactions being considered in the rules. Also importantly, association rule mining techniques are very computationally intensive. The running time of association rule mining mainly depends on the number of items in the dataset and the minimum support and confidence. The number of tests cases which correspond to the number of items in association rule mining is over 65,000 test cases to analyze.

In *Failure by Association*, relationships between test failures are found and then used to predict failures in a given regression test run. Unlike the most frequent failures approach in the previous section, the *Failure by Association* approach requires some information from the current regression test run in addition to historical data. Fortunately, this information is available in the form of smoke tests. As described by Kaner et all [47], smoke tests are a small sample of tests that are run after each build to determine whether or not the build is high enough quality to continue running additional tests. We define the set of failed smoke tests in regression test run i as the set of regression tests which were run in i, but failed. We define S_i as: $S_i = \{T_{i1}, T_{i2}, ...T_{ir}\}$

The association rules are of the form $LHS \implies RHS$. The support of a rule is the number of transactions in which the left hand side items appear along the right high side items. Support is used to filter out weak rules that show the association in a very small number of transactions. The confidence of a rule is the ratio of the number of times the right hand side items appear when the left side items appear. Confidence is the likelihood that the right hand side items appear given that the left hand side items appear. Confidence is used to filter out rules where there is not a strong correlation between the left and right hand side items appearing. For a given dataset and support and confidence threshold, a set of 'interesting' rules is mined, such that: $Rules = \{(LHS_1 \implies RHS_1), ...(LHS_x \implies RHS_x)\}$

The LHS is the left hand side of each rule, also referred to as the antecedent. Given the failed tests in previous regression test runs, we can mine all the rules whose support and confidence exceed user-specified thresholds. However, we only have a small set of tests for the regression test run for which we are trying to predict the tests that are likely to fail. Therefore, we are not interested in all the rules that are valid in the previous test runs. We are only interested in the rules in which the LHS tests are part of the smoke tests, i.e., $LHS_j \subseteq S_i$.

To mine only the sought-after rules which are applicable for the current test run, we propose a different method for mining these rules. For a given support threshold, $minsup$, we mine the frequent tests in previous test runs. Only these frequent tests can be on the left hand side of any interesting rules since by definition the tests in an interesting rule have to appear in at least $minsup$ transactions. The transactions that have at least one of these frequent tests are retained and other transactions are pruned.

The second step is to mine for associations between tests and the frequent smoke tests in these transactions. Once all the antecedents have been determined based on support, the next step is to find the associated RHS or right hand side of the rule, also referred to as the consequent. Similar to how a support level is used in determining the frequent LHS itemsets, a threshold called the *confidence* is used to determine which itemsets occur in the RHS. Unlike the LHS which examines all previous test runs looking for when that test occurs, the RHS only looks at test runs in which the LHS also occurs.

5.1.1.3. Test Age

The final approach examined in this research is the applicability of test age on the accuracy of the predictions. The age of a test is the number of test runs that have occurred since that test run until now. We define a window of data to be the set of all test runs with $age \leq window$.

5.1.2. Empirical Study

As stated in Section 5.1, in this research, we investigate whether the use of failure by association, most frequent failures, and test result age can help better predict the likely test failures in a given test run. These methods are applied to an industrial product. This section describes the empirical study performed.

In our study, we investigate the following research questions:

RQ1: Can learning from previous test runs improve the effectiveness of selecting tests in terms of fault prediction?

RQ2: Can restricting the set of previous test runs based on age help increase the effectiveness of selecting tests in terms of fault prediction?

5.1.3. Variables and Measures

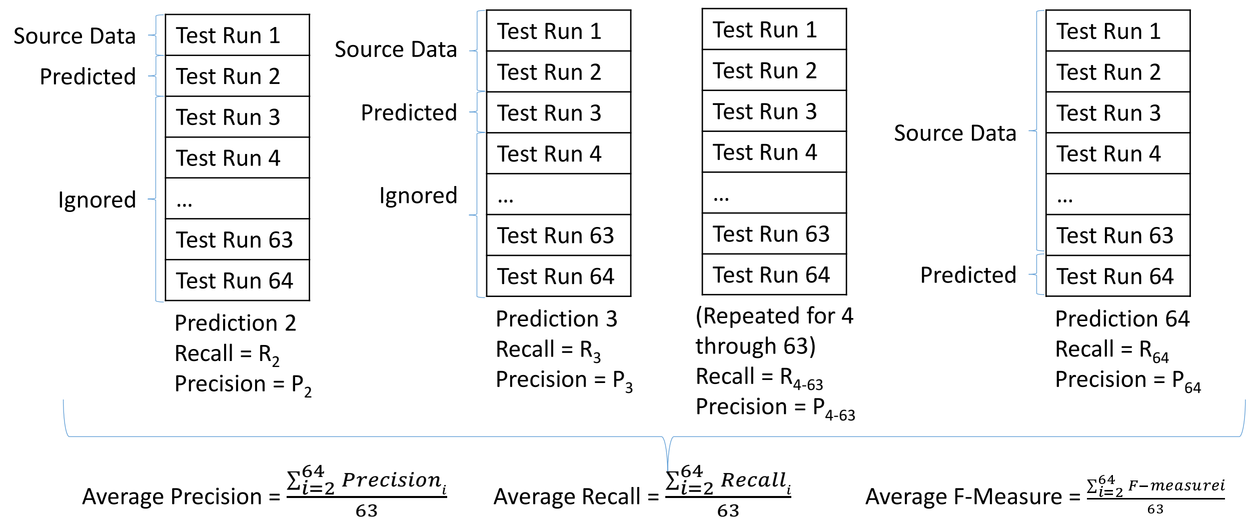


Figure 5.2. Experiment Process

5.1.3.1. Independent Variables

This study manipulated two independent variables: selection technique and test result age window. We consider two *control* technique and four *heuristic* techniques as follows:

- Control: A random set of tests is selected out of the set of previously failing tests and used as the prediction. When recommending the tests in the control method, the number of recommended failures needs to be determined as well. In this study, we predicted the same number of failures as the average number of failures seen in previous test runs.
  - $Trandom, full$: This technique randomly selects a set of tests from the full set of previous failure data.
  - $Trandom, recent$: This technique randomly selects a set of tests from the most recent previous failure data.

- Heuristics: We consider four heuristics representing the different combinations of failure by association and data recency.
  - $Tfrequent, full$: This technique predicts failures based on most frequent previous failures, examining the full set of previous build data.
  - $Tfrequent, recent$: This technique predicts failures based on most frequent previous failures, examining only the most recent previous test runs.
  - $Tarm, full$: This technique predicts failures based on failure by association of previous failures, examining the full set of previous build data.
  - $Tarm, recent$: This technique predicts failures based on failure by association of previous failures, examining only the most recent previous test runs.

5.1.3.2. Dependent Variables

We consider two dependent variables, the precision and recall of the predicted test failures. Precision refers to the percentage of predicted failures that actually failed. Recall refers to the percentage of actual failures that were predicted. To measure the overall effectiveness across both precision and recall, we also compute the F-measure which is the harmonic mean of the precision and recall.

The formulas and more complete explanations of precision and recall are presented in Section 5.1.1.

5.1.4. Experiment Process

This experiment was performed on historical data from the *Dynamics AX 2012 R2* release. As previously mentioned, this release contained 64 regression test runs. The techniques being studied were applied to each test run, simulating what would have happened if they had been applied during the actual development cycle.

We started with regression test run 2 since the first regression test run had no historical data which could be used for prediction. At the time regression test run two was about to begin, only regression test run 1 had previously happened, so only data from regression test 1 could be used for prediction. Based on that data, we predicted the failures for regression test run 2. Once those failures were predicted, we then compared them with the actual results for regression test run 2 and calculated the precision and recall for that regression test run, P_2 and R_2.

We then applied the same process to examine regression test run 3. Again, simulating the data available at the time regression test run 3 was about to begin, we only had historical data from regression test runs 1 and 2. So that data was used to predict the failures in test run 3. We then compared those predicted failures from test run 3 against the actual failures in test run 3 and again calculated precision and recall, P_3 and R_3.

Following this same process, the failures for test run 4 were predicted based on the results of runs 1 through 3 and used to calculate P_4 and R_4. Test run 5 was predicted based on results of runs 1 through 4, yielding P_5 and R_5. This continued all the way up to predicting the failures in test run 64 based on the results of test runs 1 through 63, yielding P_{64} and R_{64}.

The only difference in this technique when applying test age to examine only a window of data was only the most recent test runs were examined. So with a window size of 10, the predictions for test run 37 would be based on the results of test runs 27 through 36.

Once precision and recall had been calculated for each test run using each technique being studied, the results were then averaged across the 63 results for each technique.Using the average precision and recall for each technique, we then calculated the F-measure of the technique. (See the equations below.) Averaging the precision, recall, and F-measure across all of the test runs is important, as there is high variability in the precision and recall values found in each test run across the data set. Based on the average values, we can determine if one technique is more effective than another in general, ignoring local variability.

5.1.5. Data and Analysis

5.1.5.1. Failure by Association and Frequent Failures

The first research question (RQ1) addressed was whether or not learning from previous test runs can improve the effectiveness of test case selection in future runs. Figure 5.3 shows precision, recall and F-measure for the control and heuristic techniques (failure by association (ARM) and frequent failures). Both the frequent failures and ARM techniques are much more effective at predicting failures than the control technique, the control having an F-measure of 0.179 and the frequent and ARM approaches at 0.441 and 0.460 respectively.

It is also of note that while the ARM approach yielded a better result than frequent failures, the difference was negligible. Overall, it was only 0.04 more effective based on F-measure. Based on the individual analysis of the association rules, it was found that very few rules contained more than a single test on the RHS. This means that the confidence measure in failure by association becomes very similar to the confidence measure in the frequent failures approach for determining predicted tests, leading to very similar prediction effectiveness. In systems where there is higher coupling between tests, the size of the RHS in the rules would be expected to increase, and therefore the effectiveness of the ARM approach should also increase compared to that of the frequent failures approach.

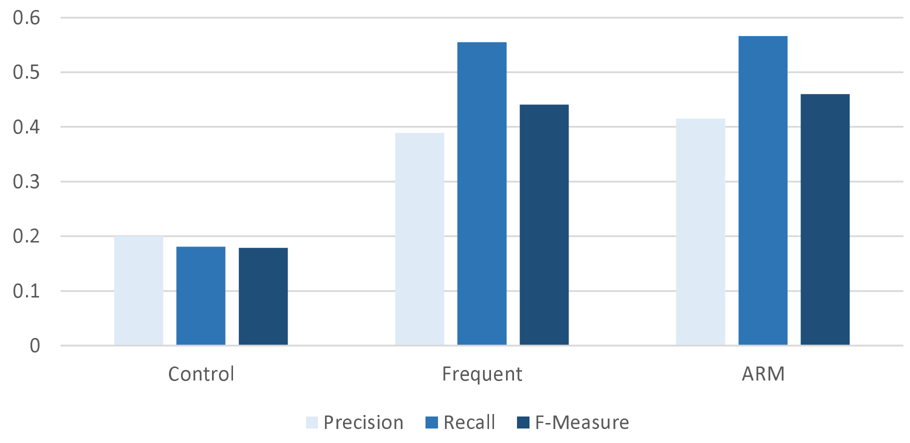


Figure 5.3. Effectiveness of ARM and Frequent Failures

5.1.5.2. Test Age and Analysis Windows

The second research question (RQ2) was whether restricting the set of previous test runs based on age help increase the effectiveness of the techniques we investigated in RQ1.

Figure 5.4 shows precision, recall, and F-measure across all three approaches both with and without restricting the set of previous test runs. It is clear from this data set that using recent test runs as opposed to all historical test runs greatly increases the effectiveness of fault prediction of the identified tests. This even applies in the case of the control, which randomly selected tests from any that had previously failed.

The cost of applying this windowed approach is also negligible. Unlike approaches such as failure by association which required several data processing steps to make predictions, applying a window based on age actually reduces the amount of data to be processed. The fact that reducing the amount of data sets to be processed increases the effectiveness of the approach is important because it means this is a very valid technique that can be applied with no additional cost to other methods.

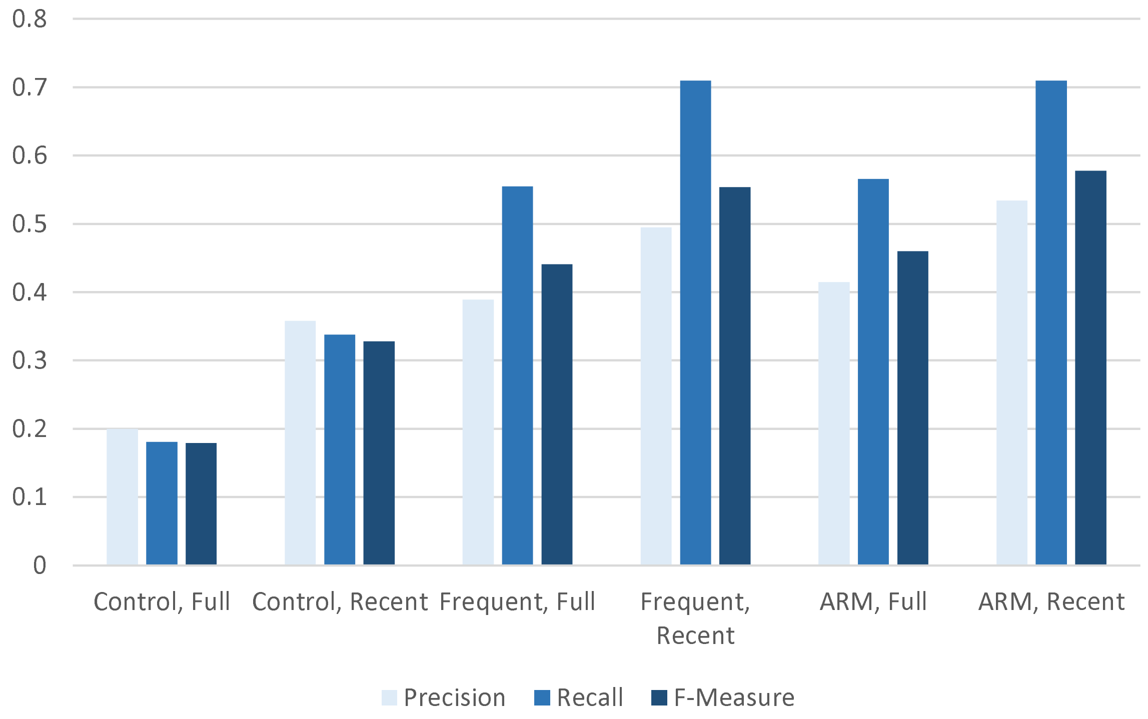


Figure 5.4. Overall Effectiveness of Prediction Approach

5.1.6. Discussion and Implications

Our results indicate that the effectiveness of test case failure prediction can be improved through the use of historical test results, together with an application of a concept of test result age. As shown in Figures 5.3 and 5.4, the use of frequent failures and failure by association produced better precision and recall values (between 0.38 and 0.56) compared to the control technique (less than 0.20) This essentially doubles the effectiveness of prediction. Similarly, applying a concept of test age to use only the 25 most recent test results showed increases in the effectiveness of not only the frequent failures and failure by association approaches, but also in increasing the effectiveness of the control approach.

While the overall results show the effectiveness of the proposed approaches, there are additional observations and implications, and we discuss these in the subsections.

5.1.6.1. Cost of Implementation

Implicit in this research but not yet discussed is the fact that performing analysis such as failure by association or frequent failures to predict test failures is an engineering activity which in itself has a cost. As discussed earlier in this section, spending cost on one quality activity in product development necessarily takes away time and resources that could have been spent on other, potentially more important activities. The cost of performing the research in this paper was not explicitly captured, so it will not be discussed quantitatively. A qualitative discussion is still helpful however.

The two primary techniques used for predicting based on previous failures were the failure by association and frequent failures approaches. Between these two approaches, failure by association demonstrated approximately a four percent benefit over that of frequent failures as shown in Figure 5.4. This does not necessarily mean that failure by association is a better approach. One challenging task is how to efficiently mine the applicable association rules for every test run. This is especially significant considering the fact that most frequent itemset mining algorithms (e.g., Apriori [2], ECLAT [85], FP-growth [36]) enumerate the entire extremely large frequent itemsets search tree.

This means that as the number of items in the itemsets increases, the cost of running these algorithms increases dramatically. In our case, it was only after about a large amount of pruning and performance tuning that the data and approach of failure by association were efficient enough to complete in a less than a few days. One such performance tuning activity was searching only within test runs which contained the same failures as occurred in the smoke tests.

Agrawal and Srikant [2] discuss many other association rule mining algorithms, many of which are more efficient than Apriori. But compared to performing analysis of most frequent failures, all of these algorithms are significantly more complex to write, as well as significantly more costly to execute on a computer. The most frequent failures approach took only around a day to write and less than an hour to execute.

Based on this experience and the fact that failure by association only yielded an improvement of around four percent in effectiveness, we recommend applying a most frequent failures approach in industrial application. The additional resource savings by using the frequent failures approach can be applied to other quality activities during the software release cycle.

5.1.7. Threats to Validity

This study focuses on the relationships between different tests and their propensity to fail. Since the amount of coupling or other inter-test relationships is purely based on the actions of the engineers building those tests, other products built by other companies and other groups of developers may exhibit stronger or weaker relationships than those observed in this study.

Similarly, the release cycle for this product was approximately one year. As discussed earlier, the changes in development focus are believed to contribute significantly to the validity of relationships between test failures. Thus, other products with different release cadences may experience different results.

As no test or product development environment is ideal, the set of tests which could be and were run varied in each test run. This study ignores all tests that did not exist prior to a given test run, or which stopped existing after that test run due to standard ongoing development during the release. Since a test being added or removed from the test suite is not an indication of previous or future failures, it was ignored. Because of this, not all test results were considered in the study. Exclusion of results for this reason may have had an impact on the results of the study.

5.1.8. Conclusions

As shown in this research, there are hidden relationships between test failures which may be mined to more accurately predict failures in future runs. It was also shown that the majority of these relationships are simple frequency relationships as opposed to more complex interactions between groups of test failures. This likely suggests that the tests themselves exhibit low coupling, which is advantageous. The tests are intended to not overlap each other in functionality, and this research suggests that they largely do not.

The other interesting result of this research is the drastic increase in prediction abilities when using a relatively small window of recent historical data, as opposed to full historical data. This suggests that not only are more recent failures better predictors than less recent failures, it also suggests that using less recent failures can be detrimental in prediction abilities. This is likely a result of different phases of development focusing on different areas of the product. While an area is being actively developed, tests that exercise that area are more likely to fail more often. When that development ceases, little variation in test failures is expected.

The most important result of this study for our overall research is the knowledge that data mining techniques can be used on this industrial product to improve test case prioritization. The next step of our research is to get an understanding of how the different metrics available on this product interplay with each other and the software environment. This is addressed in our next empirical study.

5.2. Using Classification to Understand Attribute Importance in and Industrial Product

While our previous research was shown to be promising, it, did not aggregate the various attributes to understand the interactions between these factors. Do the attributes have a causal impact on each other? Are they mutually exclusive in predicting test case failures? Are they indicative of larger, unseen forces in the software project?

To investigate these questions, we propose a holistic approach to test failure prediction based on heterogeneous multi-level data attributes. These attributes include historical test case pass and fail information, organizational information, code complexity information, and code change information. We used classification algorithms that learn a classification model from these input attributes, and using that model, we predicted future test case failures based on the current attribute values. An additional benefit of using classification model techniques is that the learned model is also a source of information. The model can be examined to understand how the input attributes impact each other and how the attributes impact the failure prediction results.

Classification techniques have been used with many problems and industries, such as image processing or genome characterizing [11, 72]. These techniques have recently been applied to software engineering areas, such as bug triage and predicting bug bounces [33, 67]. In this research, we apply similar classification techniques to the area of test failure prediction and use the resulting models to understand the interplay between various software attributes and the software project itself.

To evaluate our approach, we performed a case study for the release of an industrial product where many attributes were available. We used multiple classification algorithms across differing attribute sets, examining the effectiveness of the test failure predictions for each case. Our results indicate that, while each attribute can help improve failure prediction as shown by other research, some attributes, such as historical failure information, have much stronger prediction power. At the same time, other attributes, such as churn information, which are usually considered good predictors of test failure were not as effective for this project. By examining how these attributes impact each other in the classification models, we discovered that the attributes' effectiveness for prediction actually exposes problems with the software environment where the testing occurred. One of the important findings from this study was that churn information that has been considered to be the best predictor of failure prediction may actually be less helpful than other attributes such as historical test results based on the testing environment for the project. The implication of this finding is that much previous research on test case selection and prioritization may not be applicable in environments with a relatively minor amount of test instability.

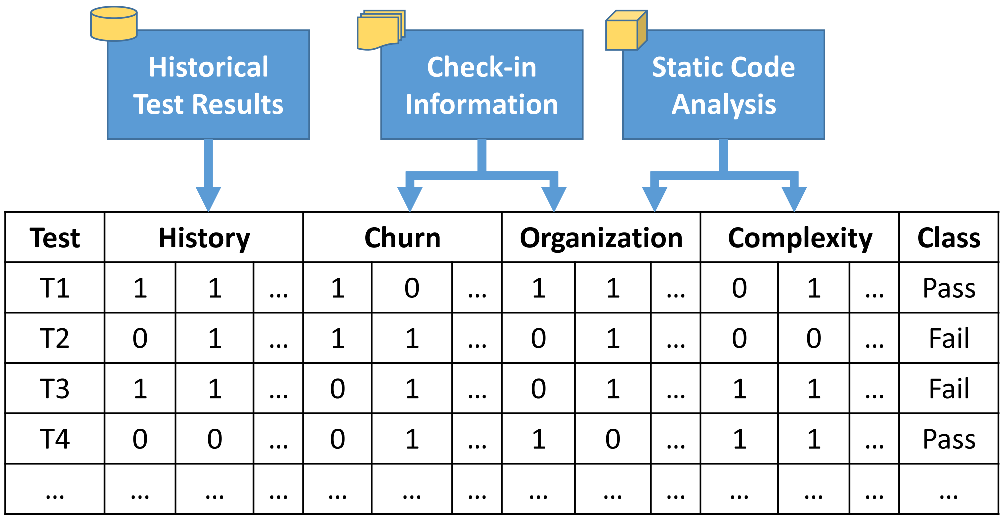
5.2.1. Approach

For this study we relied on the same data from our previous study, from *Microsoft Dynamics AX 2012 R2*. Figure 5.5 illustrates the inputs for the classification problem and how each set was derived. The historical test result features are based on the pass/fail information for every test in each regression test run.

The churn-based information from the source control system provides multiple features. They are split into two categories, churn measurements and organizational measurements. The churn measurements specifically count how many elements touched by the test have been modified since the last test run. This count is calculated by comparing the check in information with static code coverage data detailing which tests call into which code elements. The total number of elements called by the tests with modifications are then tallied to provide a numeric value.

The other aspect derived from churn-based information is organizational, specifically which teams have made changes and whether the people making changes are vendors or full-time employees of the company. These data are, again, cross referenced with static code coverage data to determine which team made the most modifications to the code called by each test. Similarly, this information provides metrics for whether any vendors made changes to the code covered by the test and how many changes they made.

The last measurement, complexity, is more traditional. It includes the number of lines of code in the test, the number of references made from the test to other code (fan-out), and other similar metrics.



Test	History			Churn			Organization			Complexity			Class
T1	1	1	…	1	0	…	1	1	…	0	1	…	Pass
T2	0	1	…	1	1	…	0	1	…	0	0	…	Fail
T3	1	1	…	0	1	…	0	1	…	1	1	…	Fail
T4	0	0	…	0	1	…	1	0	…	1	1	…	Pass
…	…	…	…	…	…	…	…	…	…	…	…	…	…

Figure 5.5. Constructing Attributes for Tests

Classification models take a set of training instances and learn a classification model [14]. The classification model can then be applied to a new set of input attributes to make predictions as shown in Figure 5.6. A wide variety of classification algorithms exist. Different classification models yield varying levels of prediction accuracy depending on the data set and the input parameters chosen [44]. Therefore, we also need to examine various classification algorithms, such as decision tree models and Bayesian models.

These data yield a large number of features that are grouped into the four categories of history, churn, organization, and complexity so that we can analyze which feature type is the most important. We can then analyze how combinations of these measurements impact the results.

For each regression test run performed during the *Microsoft Dynamics AX 2012 R2* release, a classification model is built and is based on the information available at that point in time. For instance, if we are examining regression test run 15, then we will consider historical results from runs 1 through 14, together with the churn-based information that occurred between runs 14 and 15. This classification model then predicts which tests are likely to fail in test run 15. The result of this prediction is compared against which tests actually failed for test run 15 to determine the prediction's effectiveness.

5.2.2. Empirical Study

In this study, we investigate the effectiveness of classification approaches to predict test case failures when applied to an industrial product. In this study, we investigate the following research questions:

1. RQ1: Can a classification model be learned for predicting test failures from software attributes?
2. RQ2: Which attributes are the most important in predicting test failure?

5.2.2.1. Object of Analysis

The data sets for this study came from the internal check in, bug, regression test, and cross reference databases associated with the release. The product itself contains approximately 5.5 million lines of code and has around 65,000 regression tests that are run regularly.

The full regression test suite is run, on average, once every 3 days because it takes close to 3 days for all 65,000 regression tests to run across the pool of available test computers. During the R2 release, there were 64 full regression test suite runs, yielding approximately 3.5 million distinct test results.

On average, approximately 10 percent of the regression test suite failed in any given regression test run. This statistic does not mean that 10 percent of tests found faults because many of the failures were due to environmental issues, timeouts, test bugs, or other issues. For instance, of all the regression test failures, fewer than half of them were related to bugs logged in the bug tracking system. Of those bugs logged, many were associated with multiple test failures, or were simply infrastructure or test bugs.

The goal of this study is to accurately predict the roughly 10 percent of regression tests that will fail in a given regression test suite run. We do not seek to place meaning on those failures beyond classifying their state as passed or failed. Some failures will be due to infrastructure or other issues and may not be due to underlying software faults. Even without fault or severity information, an accurate prediction about which tests will fail is still valuable. By avoiding the time spent running tests which pass, failures can be identified and analyzed more quickly.

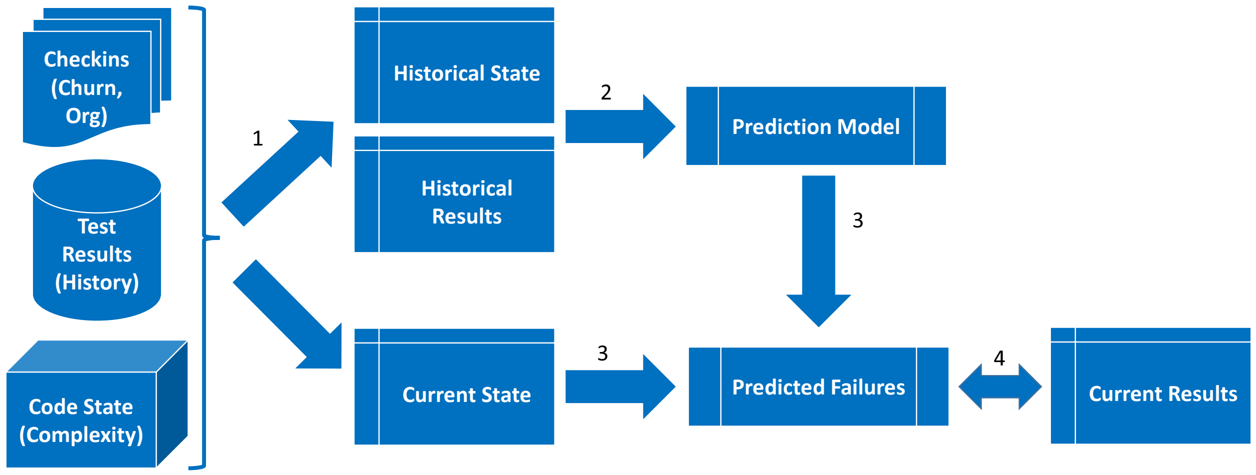


Figure 5.6. Experimental Process

5.2.3. Variables and Measures

5.2.3.1. Independent Variables

This study manipulated two independent variables: input attribute and classification method.

For input attribute, we considered four categories of input attributes as we described in Section 5.2.1 (Figure 5.5): history, organization, churn, and complexity. Each category contains multiple individual attributes as described in this section.

History attributes:

1. Last Run Status: Whether the test passed or failed the last time it ran
2. Passes in Last 10 Runs: How many times the test passed in the last 10 regression test runs
3. Bugs Found Last 10 Runs: From the failures in the last 10 runs, how many resulted in bugs being logged
4. Bug Found Last Run: Whether a bug was logged from a failure in the last run

Organization attributes:

1. Primary Editing Team: Which product team made the most changes
2. Checkins by Vendors: The total number of changes made by vendors in this test run as opposed to full time employees

Complexity attributes:

1. The total number of elements which are executed by the test case

Churn attributes:

1. The total number of files modified since the last run
2. The total number of checkins since the last run
3. The total number of checkins in code which the test executes since the last run

These attribute categories are used to generate classification models individually and in combination to determine their effectiveness. For instance, a model is generated and analyzed using all the history attributes and nothing else. Another model uses all the organizational attributes and nothing else. A third model uses both history and organizational attributes. Yet another model uses all attributes from all categories. This set of models yields 15 total combinations of attribute categories for which the research was performed.

The second independent variable is classification method. There are many classifiers available, each with many options and constraints for supported data types. It is not possible to test all classifiers. Instead, we focus on two common classifiers: Bayesian and tree methods. Since the goal of the study is primarily around examining the input attributes and not the classification mechanisms themselves, little classifier discussion is provided beyond what is necessary to rule out classifier preference as skewing the results of the primary independent variable.

5.2.3.2. Dependent Variables

We consider two dependent variables, the precision and recall of the predicted test failures. Precision refers to the percentage of predicted failures that actually failed. Recall refers to the percentage of actual failures that were predicted. To measure the overall effectiveness across both precision and recall, we also compute the F-measure, the harmonic mean of the precision and recall.

There are four result classes possible based on the predictions made in this study.

1. True Positive: The test was predicted to fail and actually failed.
2. True Negative: The test was predicted to pass and actually passed.
3. False Positive: The test was predicted to fail and then actually passed.
4. False Negative: The test was predicted to pass and then actually failed.

To apply our approach, we need to collect the input attributes and from these input attributes, we learn a model via a classification algorithm and use that model to predict the test failures for the current regression test run. These predicted failures are compared against the actual failures for that run to determine the prediction's effectiveness. This approach is repeated for each regression test run as shown in Figure 5.6.

5.2.4. Data and Analysis

5.2.4.1. Overall Results

Table 5.1. F-Measure by Independent Variable Combination

Classifier/Attribute	RepTree	Naive Bayes
All Attributes	0.522	0.488
History	0.510	0.500
Churn	0.194	0.150
Complexity	0.231	0.216
Organizational	0.155	0.052

Table 5.1 shows the average F-measure for the failure predictions resulting from each combination of independent variables. For all attribute sets, the RepTree method slightly outperforms the Naive Bayes classification models. One benefit from a decision tree based model such as RepTree is that the resulting model is easily understood by humans and can provide meaningful information about the role each attribute plays in the predictions. For that reason, most analysis in this section is based on the RepTree classifier.

5.2.4.2. RQ1 Analysis

Research question one is whether a classification model can be learned for predicting test failure from software attributes. The first step in analyzing this question is to determine a baseline. Without a classification model, a naive baseline is just to randomly predict tests as failing. As previously discussed in the Object of Analysis section, approximately 10 percent of tests fail for any given test run. If all tests were predicted to fail, that result would yield a precision of 0.100, a recall of 1.000, and a corresponding F-measure of 0.182. As the number of tests randomly predicted decreases, the precision would statistically remain at 0.100. The recall would decrease, thus further lowering the F-measure.

Given a naive baseline of a 0.182 F-measure, at best, we see that the classification models learned using all attributes and history were clearly much better at predicting test failures than the baseline, up to an F-measure of 0.522 as shown in Table 5.1. Other measures, such as complexity, were also marginally better. From this analysis, we can say the answer to RQ1 is yes; a classification model can be learned to predict test failure from software attributes.

5.2.4.3. RQ2 Analysis

Research question two is about which attributes are most important to predict test failures. To answer this question, we examine the F-measure achieved by various classification models built using the different attribute categories.

Figure 5.7 shows the overall F-measure by attribute category across all test runs. While the F-measure for any given run is not a continuous amount, the differences between aspects are most easily visualized as a line chart because it makes the variation between runs more visible. The solid line represents the F-measure for predictions made with all available data. This value averages 0.522 with a standard deviation of 0.174. The history attributes alone average 0.510 with a standard deviation of 0.134, showing that history data alone are nearly as predictive as all attributes put together and have slightly less variation.

Looking at the other attributes, churn, organization, and complexity, these averages are between 0.155 and 0.231, with a standard deviation around 0.09. Of interest to note is that the F-measure for these

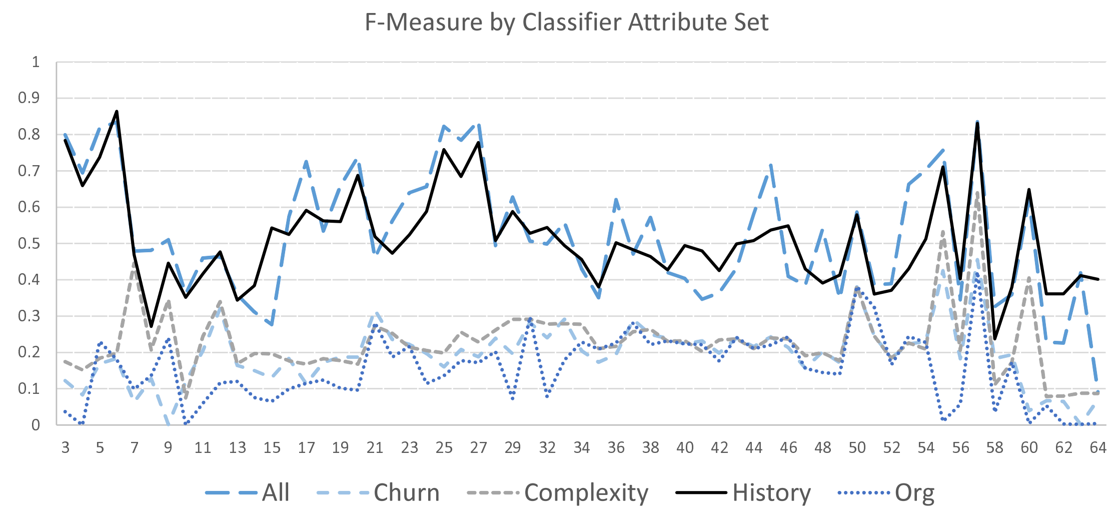


Figure 5.7. Overall F-Measure by Attribute Category

three attribute categories generally follows each other, with similar jumps in the F-measure for similar test runs.

A very surprising result from this research is the lack of predictive power from the churn attributes. Churn-based testing is generally considered to be one of the most accurate predictors for which test cases are likely to fail in future runs. Remember that churn data are a dynamic predictor because the data are specific to each run. Compare that dynamic data with complexity information, which is generally static. The complexity of a given test case generally does not change run by run. Naturally, one would expect that the dynamic churn data would be a much better predictor of failure because they are tied directly to which tests have changed.

If we compare build-by-build we see a very high correlation between the results of these two predictors, with the complexity attributed yielding a *higher* F-measure than the churn data. As shown in Figure 5.8, the trends for the F-measure values are similar build-by-build, with a correlation coefficient of 0.580.

From these results, we see that the most important attribute for predicting test failures in this software project is historical test failure attributes. Churn and complexity information is the next most important, although they do not have nearly as high of predicting power.

It is also interesting to note that classifications are based on historical data and therefore as historical data increases one would expect the prediction effectiveness to also increase. This would be represented as an increasing F-measure in Figure 5.7. Instead, there is a somewhat downward trend throughout the project. From this we conclude that lack of training data is not an issue. Rather it is weakness in the ability of the

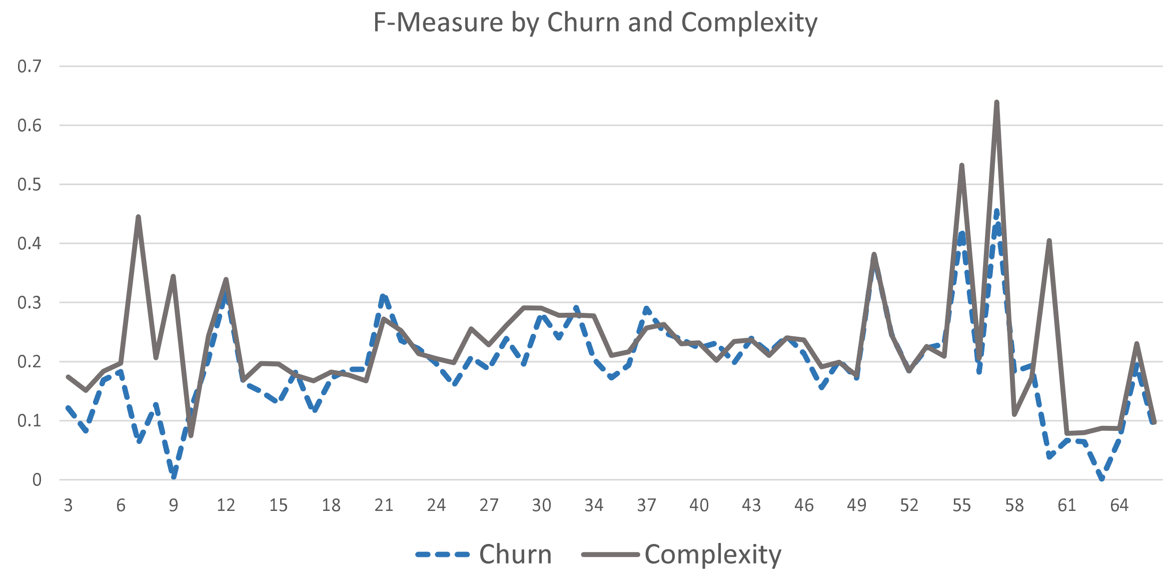


Figure 5.8. F-Measure of Churn vs Complexity

training data to more accurately predict test failures that inhibits better results. So simply increasing the volume of training data is not sufficient to increase effectiveness.

5.2.5. Discussion and Implications

5.2.5.1. Most Influential Attributes

There are two easy ways of determining which attributes are most influential in predicting regression test failures. The first one is by examining the F-measure by attribute category as shown in Figure 5.7. As shown in the graph, historical test data are a consistently better measure of future failure than any of the other attributes. Churn, organizational, and complexity measures all have roughly the same predicting power and are consistently less accurate than historical information.

The other way of determining influence is to examine the models generated by the classifier,. Classification models built as trees will contain the most selective attributes closer to the tree's root, and the less selective attributes at the leaves of the tree. In this case, all nodes in the top two levels of the tree are history-based attributes, meaning that the most important classification attributes are history. Only at level three and below do other attributes, primarily churn attributes, start to appear.

This result is very interesting for the Microsoft project because the development team normally thinks of churn information as being the best predictor of failure. If a test is currently passing and no changes are made to the test or any code it calls into, then one would expect the test to pass the next time it is run. This expectation of continued passing when no changes are made is not supported by the models or the results of this experiment. Based on these results, the prediction of future failures can be very effectively done without looking at churn or other measures.

This finding raises a question about why churn-based information is not a good predictor of test case failures. One possible conjecture is that environmental instability makes some tests more prone to random failure even if the associated code does not have any faults. An easy way to test this conjecture is to see if there are any tests which exhibit failures without *any* associated changes. Thus, we further examined historical test results. We found that there were 494,064 test results which had no changes from the previous test run. Of those results with no changes, 7,014 resulted in a test failure on the subsequent build after passing in the previous build. These 7,014 test results must be the result of environment or test randomness because code changes and associated defects cannot be the cause. This results in a random failure rate of approximately 1.4 percent.

As a comparison, there were 1,745,241 results which contained one or more code changes. Of them, 50,502 failed after having passed on the subsequent build, a rate of roughly 2.8 percent, telling us that randomness in the tests and environmental instability in the test system are responsible for at least half of the failures recorded during the project's regression testing. The number might be higher because it is possible that many failures which had code changes were still environmental in nature. At least half of the test failures are due to that instability.

We believe these random failures are why code churn, while seemingly a great predictor of test failure, ended up not being the best attribute. Because test environment instability caused so many failures, tests that had previously failed and may be more susceptible to the instability have a much higher likelihood of failing in the future.

5.2.5.2. Effectiveness over Time

In Figure 5.7, it is interesting to note that the F-measure for all attributes, except history, is very stable through most of the release, but shows a large amount of variation near the beginning and end of the release.

Figure 5.9 shows the number of tests that failed by test run. Notice that near the beginning of the project, the number of failures is relatively low. Anecdotally, major development did not start until around regression test run 20, where the majority of the development team started working on the project. Prior to this point, many of the failures were due to random instability in the test system.

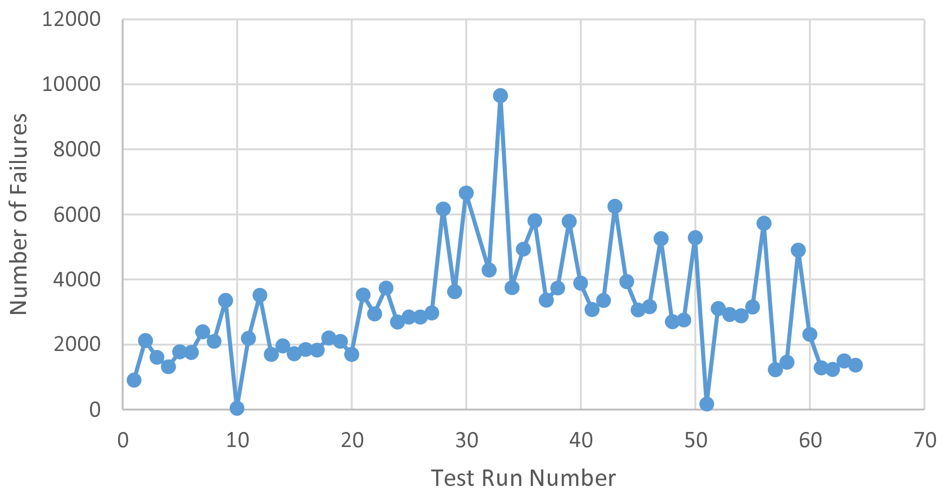


Figure 5.9. Number of Failures by Test Run

This instability due to branching, coupled with a lack of historical test results and little actual development at the time, seems to have led to some of the instability in the classification model's prediction capabilities. There are also some artificially low numbers for failures during some of the integration periods. For instance, for test run number 10, there were close to 6,000 files integrated but only 37 test failures. This result is likely due to the processes for branch integration.

5.2.5.3. Project Feedback

One obvious question arising from this study is about what parts are applicable to the *Microsoft Dynamics AX 2012 R2* development environment and what changes are being made because of this research.

The most important finding from this project is the large role that general test environment instability has played in reducing the predicting power of classification algorithms. While the Microsoft development team has always understood that false positives from test failures were a detriment to development, it was always assumed that churn and other metrics were much more important in terms of predicting failure. Indeed, many of the Microsoft development team's internal practices for determining which tests should be run prior to checkin are based on churn and other metrics.

What we find in this study is that churn is much less important than general test instability. If the goal is simply to predict which tests are likely to fail as is often stated in development practices, then identifying and tracking instability for tests is much more important than tracking churn. That being said, simply testing based on environmental instability does not make much sense because the underlying goal of testing is to detect defects, and environmental instability does not necessarily indicate underlying defects.

However, if the Microsoft development team wishes to improve upon our engineering systems and more accurately predict which tests will fail to minimize test pass time and to decrease defect detection time, we need to reduce the number of random failures for the tests. While they have historically used metrics related to the overall failure numbers in this analysis, the classification models shown in this paper indicate that test instability often occurs in blocks of time. Thus, when attempting to identify troublesome tests, it is useful to look not only at total failures, but also the recency of failures within a given window of time.

5.2.6. Threats to Validity

The primary threat to validity in this research is the fact that the techniques have only been applied to a single release of one application. As discussed extensively in the previous section, there are aspects about test environment instability, branching, and integration to that particular product and release which may not apply to other products or releases. These aspects may lead to different weightings for the classification attributes.

The attributes used in the classification algorithms to generate classification models represent those attributes which were tracked and available during this product release. Other products and other releases may have more or different attributes from which to draw, which may yield different results when used in classification.

A variety of classification algorithms were analyzed in this research, but many others exist and may yield slightly different results. It is not believed that selection of the classification algorithm has skewed the overall results because similar results were seen with each algorithm that was analyzed. There will be minor differences depending on the classification algorithm selected and the settings used while executing that algorithm.

5.2.7. Conclusions

We proposed classification-based test failure prediction approaches and presented a case study using an industrial application. Our results showed that classification models based upon attributes can be used to better understand the environment in which software is being developed and to identify shortcomings in that environment that are causing extra cost. The results also indicated that the traditionally relied upon measures, such as churn, may not be the best predictors of test case failure. Further, the results suggested that using additional attributes for classification models can improve the ability to predict future test case failures. We also learned that with the product release we utilized, some of those measures provided little benefit.

5.3. Mining Test Results for Reasons Other Than Functional Correctness

The previous two studies show that data mining techniques can be applied to the *Microsoft Dynamics AX* product to successfully prioritize test cases. We then looked into whether these same traditional techniques we had used to find functional issues could be applied in finding non-functional issues such as performance faults. From this study we found that this technique was possible, and was presented at ISSRE 2015 (International Symposium on Software Reliability Engineering).

Current research on regression testing techniques, such as test case prioritization, focus on the detection of functional issues in software. Functional issues are what is traditionally thought of as the most common class of "bug"; a fault in software code which causes an incorrect output or result. An class of non-functional issues also exists, and is just as important. Examples of non-functional issues include performance problems, security vulnerabilities, and usability deficiencies. Even if the software is providing the proper output, non-functional issues may render the software product unusable.

We theorize that many of the traditional regression testing techniques discussed earlier will be applicable in predicting non-functional software faults as well. These include commonly used data sources of churn, complexity, and lines of code [29, 32, 65, 66], as well as more advanced techniques such as software dependency graphs, historical defects, and performance data [57, 87, 16]. We focus these data sources and techniques on predicting performance faults in an industrial product, *Microsoft Dynamics AX*.

5.3.1. Approach

In this section, we describe the approach used in performance fault prediction based on an industrial product release currently under development, *Microsoft Dynamics AX*.

5.3.1.1. Current Performance Testing Practices

Performance is known to be a very important quality aspect of the *Microsoft Dynamics AX* product because performance issues slow down the productivity of users of the product. The primary reason customers purchase an ERP system is to increase productivity, so performance is very important to users. Performance faults can lead to problems including increased hardware costs for customers who deploy the software, decreased productivity for users, and in some cases, performance faults can even stop a business process from being completed if the problem is bad enough. For these reasons, much focus is placed on performance analysis and testing within this product.

As with any major product, one problem with performance testing is a very large number of configuration options that can have significant impacts on the performance of processes. These thousands of options in different combinations can yield different performance characteristics making for a virtually infinite number of configurations that obviously cannot all be individually tested. For example, in the act of confirming a purchase order, one must consider whether taxes are being calculated, whether budgeting is being checked, what level of financial detail is being tracked, whether encumbrance for public sector entities needs to be considered, whether the purchase is for stocked or service items, and so on.

To deal with this untenable number of configurations, customer research is performed to determine which configurations and data volumes are most common in certain industry segments (e.g., retail, public sector, manufacturing, etc.). Then, targeted performance tests are built for these configurations in the business processes known to be important for that industry segment. This yields a reasonable level of confidence in the most important business processes.

As the limited performance testing resources are applied to targeted business processes and configurations, some of the less important processes and configurations may have less than desirable levels of performance testing coverage. Any performance faults in these less common areas will still be a major issue for customers if that process is core to their business. Our research focuses on detecting performance faults in these less common process and configurations that may otherwise not have extensive performance testing coverage.

5.3.1.2. Missing Index Faults

The type of fault being predicted in this study is a missing index on a table that results in what will be referred to as a "bad query plan". *Microsoft Dynamics AX* is a multi-tier enterprise resource planning product consisting of a client that connects to one or more servers that execute business logic. Those servers then connect to a back-end database that contains the data. As *Microsoft Dynamics AX* is often deployed in large companies, the number of rows in many of the database tables can be very large. It is not uncommon to have tables containing over 100,000,000 rows.

A single business operation often requires reading and writing multiple times from multiple of these tables. Scanning all the rows in the table would be exceedingly expensive, so databases rely on indexes to quickly find only the rows necessary to serve a given query. Occasionally, a table will not have an index that can support the query, and the database will need to revert to a scan of the entire table, which can be many orders of magnitude slower. This impacts not only the current running operation, but all other operations on that database as well, as it consumes CPU and IO capabilities that could otherwise be used by other queries.

There are also cases where an index *does* exist that can be used by a query, but it is not selective enough. Database indexes are an ordered set of columns for which all values must be specified in the query restriction. If a column is not specified in the query restriction, then all following index columns are ignored and the database must revert to a scan of the rest of all the records in that range. For example, suppose we have a table containing columns $(Company, PurchaseOrderId, Quantity, UnitPrice)$. An index of $(Company, Quantity)$ exists on the table. When the system executes the following query:

SELECT UnitPrice

FROM PurchaseOrders

WHERE Company = 'Contoso' AND PurchaseOrderId = '1234'

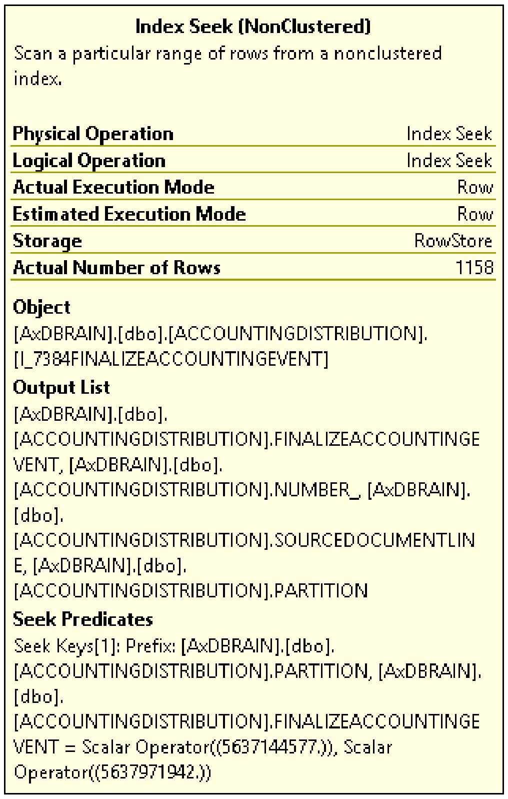
Only the $Company$ column of the index can be used for selectivity. If there are many companies in the table, this may be highly selective. But if, in this installation, only a single company exists with millions of purchase orders, then the index is no more useful than a table scan.

Figure 5.10 shows an example of a real index seek from a missing index fault that was found in the product through manual inspection. In this case, the index seeks by *[Partition, FinanizeAccountingEvent]*, which may be selective in some cases. But, in some cases, a large number of *AccountingDistribution* records (in some configurations) finalized as part of the same *AccountingEvent*. So this index seek may still end up reading a large number of rows. When many records have the same value for both the *Partition* and *FinalizeAccountingEvent* fields, many records will be returned by this index seek. In an installation with millions of rows in the table, this would yield a significant performance issue for users who executed this query.

As demonstrated by the example from Figure 5.10, the detection of bad query plans is largely a manual effort, requiring the knowledge of an expert developer who understands the different data distributions expected in installations of software as well as expected data volumes in each table and expected uses. One primary way of analyzing query plans is to take a trace of all the queries executed by some functionality and then manually analyze each query plan. This is relatively expensive as the cost of analyzing all queries executed by a single performance test may take 15 to 30 minutes for an expert developer.

5.3.1.3. Coverage Breadth Through Functional Tests

As previously mentioned, performance tests are often limited to the most highly-used portions of product functionality and lack product breadth. To gain that product breadth while minimizing engineering



Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	1158

Object
[AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].[I_7384FINALIZEACCOUNTINGEVENT]
Output List
[AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].FINALIZEACCOUNTINGEVENT, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].NUMBER_, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].SOURCEDOCUMENTLINE, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].PARTITION
Seek Predicates
Seek Keys[1]: Prefix: [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].PARTITION, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].FINALIZEACCOUNTINGEVENT = Scalar Operator((5637144577.)), Scalar Operator((5637971942.))

Figure 5.10. Sample Index Seek from Missing Index Fault

costs, we seek to use metrics from *passing* functional regression tests to predict functionality that is more likely to contain a missing index, which will yield a bad query plan.

The primary metric used in this study is the execution time of the functional regression test. This is under the assumption that bad query plans will cause functional tests to run slower than other tests without missing index faults. However, the relationship between test response time and bad query plans is not that simple. A bad query plan may make a query go from 5 milliseconds to 1500 milliseconds. But the test executing this query may take 5000 milliseconds total. A 5000 millisecond test with a 1500 bad query plan is not easily distinguishable from a 5500 millisecond test that does not invoke any bad query plans. Slow tests do not necessarily mean a bad query plan has been encountered. Further, the test dataset used by functional regression tests is very small in volume compared to a real customer deployment, as a real deployment may be many terabytes in size making unit testing unwieldy. To differentiate between tests that just take longer and tests that actually encounter bad query plans, we focus on the difference in execution time between base and volume datasets.

5.3.1.4. Experiment Approach

In this experiment, we will execute a large number of functional regression tests for the product, tracking the execution time of all the tests that pass. Only passing tests are evaluated, as a test failure leads to database cleanup processes and thus is less reliable as a timing. We then order the tests by descending execution time, expecting the tests with the higher execution time are more likely to have experience bad query plans due to a missing index.

If this ordering of tests proves reliable as a predictor of missing index faults, then an engineer could rank manual analysis tasks of the top tests for missing indexes. In this study, we analyze the effectiveness of the reordered rank through ROC analysis. The ROC curve is a graphical representation of the relative precision and recall as the number of selected items increases. The details about the experimental process will be discussed in Section 5.3.2.

5.3.1.5. Ground Truth

In order to evaluate the results of the research, we must first know which functional regression tests actually encounter bad query plans. As previously mentioned, manual analysis is exceedingly expensive and would not yield a large dataset of results. To work around this, we utilized fault injection. Fault injection is a standard technique in this area of research. The technique was originally described by Clark and Pradhan[15] and is now in common use.

In this experiment, we injected faults by removing an existing index. Before doing that, we first determined which indexes are used by each query of each test. This can be done in an automated way by first capturing all the queries that were executed during the functional regression test, then querying the SQL Server dynamic management views (DMV's) to retrieve the query plan that was executed for each. DMV's are internal data structures that Microsoft SQL Server uses to track server aspects that can be queried by the database administrator. From this process, we then have a list of all queries executed by each test, as well as an XML description of the query plan that each test utilized.

Finally, an expert developer with deep product knowledge selected three very important indexes to remove. The three index removals represent three independent experiment datasets. More could be performed, but we limited the study to three based on time availability. Because similar results were seen in all three experiment datasets, we feel confident that the results are generalizable across this particular application. The indexes removed are on tables with high data volumes in normal installations, which provide high selectivity and are highly used by queries. The three indexes selected are:

1. CustTrans.TransIdIdx [Partition, DataAreaId, InventTransId, InvoiceId, InvoiceDate]
2. PurchLine.SourceDocumentLineIdx [SourceDocumentLine, Partition, DataAreaId]
3. TaxTrans.SourceTableIdSourceRecIdVoucherIdx [Partition, DataAreaId, SourceTableId, SourceRecId, Voucher]

Finally, the query and query plan data were queried to find all tests that contained at least one query that relies on that index to the level of attributes that provides good selectivity. For instance, any query plan using *CustTrans.TransIdIdx* would need to be constrained by at least three fields of the index as any fewer would not provide enough selectivity to make this a proper index usage. *DataAreaId* in these indexes corresponds to a legal entity in the installation, so a customer with a single legal entity but millions of inventory transactions would see no selectivity from this index unless *[Partition, DataAreaId, InventTransId]* were all provided. Once all three of these segments are provided, the remaining segments of *[InvoiceId, InvoiceDate]* provide little if any additional selectivity.

Following this pattern, *PurchLine* is useful with a single segment (since *SourceDocumentLine* is a unique *Int64* value for each row in the table), and *TaxTrans* is useful after four segments. In *TaxTrans*, *[Partition, DataAreaId]* filters to a single company, which is not selective. *SourceTableId* is also not selective. Some customers will have all of their tax transactions in a single table, for instance if they are a distribution company without direct sales. The *SourceTableId* value is only selective once *SourceRecId* is added along with the three prefix segments.

5.3.1.6. Synthetic Volume

Another important aspect to consider when analyzing performance of database applications is the amount of volume in the tables. As mentioned earlier, most functional tests are executed on a artificially small dataset. One tool used in performance testing is synthetic volume, where a large number of rows is added to "important" tables prior to test execution to simulate a more realistic volume in a real installation. The *Microsoft Dynamics AX* application contains many thousands of tables, and the relationships between these tables is extremely complex. So synthetic volume is dummy rows added only to a small number of important tables. Further, this data is not used by any functional tests. It just expands the number of rows in the tables so any bad query plans will show up as more degraded than they would have on base volume, hopefully making them easier to find.

5.3.2. Empirical Study

In this study, we seek to understand whether metrics from *passing* functional regression tests can be used for fault prediction of non-functional performance faults, specifically missing indexes that yield a bad query plan. To investigate this research problem, we address the following research questions:

1. RQ1: Does ordering regression tests by execution time effectively predict which tests encounter a bad query plan?
2. RQ2: Can environment perturbation by the introduction of synthetic volume improve the effectiveness of that prediction?
3. RQ3: Can the effectiveness of the prediction be further improved by analyzing the differences in results between baseline and synthetic volume datasets?

5.3.2.1. Object of Analysis

This study is based on a pre-release version of *Microsoft Dynamics AX*. This is the same product that was used in the two previous studies, but is a new version currently under development. As part of that development, the product is undergoing significant architectural change to enable new deployment topologies, and as such the vast majority of functional and performance tests need to be migrated. In many cases, the tests are being rewritten during the migration to make them faster, more stable and more targeted. This provides a unique opportunity to use a set of very reliable and very fast functional tests with good product breadth to predict performance issues on an otherwise relatively mature product. Specific metrics for product size cannot be provided as the product is still in development. However, we are able to provide high level ranges of metrics as discussed below.

The database schema for the product contains over 8,000 individual tables with over 19,000 indexes. This is an average of just over two indexes per table. Some tables contain more than 20 indexes. The indexes have been added over multiple versions, often as the result of detecting performance faults in either performance tests or occasionally even in real customer environments.

5.3.3. Variables and Measures

5.3.3.1. Independent Variables

This study manipulates one independent variable in each of the three independent experiment datasets. For each experiment dataset, one heuristic is used and compared against the control.

1. Ordering tests by descending response time on regular data (Heuristic for RQ1)
2. Ordering tests by descending response time on volume data (Heuristic for RQ2)
3. Ordering tests by descending net difference in response time between regular and volume data (Heuristic for RQ3)

The control technique used is random ordering of tests (control for all RQs), as the research question is attempting to determine if mining data from test metrics allows for faster detection of bad query plans. While this may appear overly naive, this is effectively the currently used approach as no ordering of tests is currently used when searching for missing index faults.

5.3.3.2. Dependent Variables

The dependent variable is the accuracy of the ordering of tests. An ideal ordering would rank all tests that expose a bad query plan first, while the worst possible ordering would rank them last. A standard technique of analyzing the accuracy of ranking is the use of an ROC curve. ROC stands for "receiver operating characteristic", and is a standard statistical measurement plotting the true positive rate against the false positive rate. In an experiment such as this evaluating ranking, the ROC curve represents the precision and recall of the number of bad query plans found as the number of tests analyzed is increased.

From the ROC curve, the AUC or "area under curve" can be computed. As the ranking improves, the AUC is also increased. The measurement of AUC provides a value between 0 and 1 (with 1 being as good as possible) that provides a numeric method of determining the benefit gained from an ordering. With an AUC of 1, this would indicate that all tests that exposed a bad query plan were ranked first. Similarly, an AUC of 0.5 would be generated by a random ordering of tests.

The use of ROC analysis also provides a standard method of computing the probability that the difference between curves is due to random chance. Thus, this study seeks to maximize AUC from the ROC curve using the heuristics specified, and to show the probability of random chance causing the difference is sufficiently low.

5.3.4. Experimental Process

There are three separate experiment datasets used in this study. This is done by manually removing one of three indexes, and then running all tests on both base and volume data as shown in Figure 5.11. These three experiment datasets are independent from each other. The resulting datasets are then ordered based on the control and heuristics methods explained in the previous section. For each ordering, an ROC curve is generated and the AUC is computed. To evaluate the effectiveness of the heuristics, the ROC curves are compared with each other to yield both a difference in AUC as well as a p-value.

Figure 5.11 illustrates the overall process used in this study. As shown in the top part of the figure, the query plan data was captured with all existing indexes intact. Then, as shown in the lower part, one important index (as decided by an expert developer) was removed and all tests were executed both on base data and on volume data, capturing the response times of each. The results of this response time were then compared against the ground truth of which tests encountered a missing index by querying the query and query plan data. The index was then re-enabled and another index was disabled so the same process could be repeated.

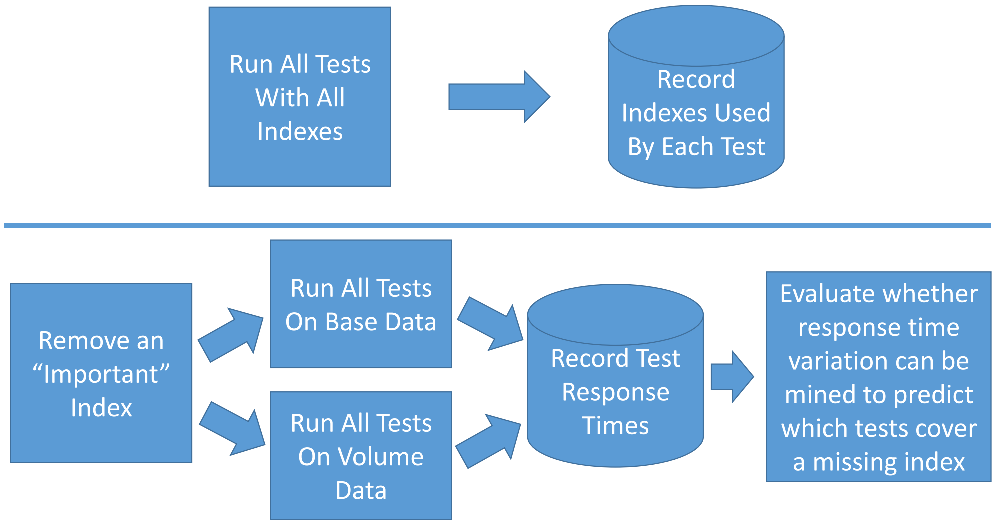
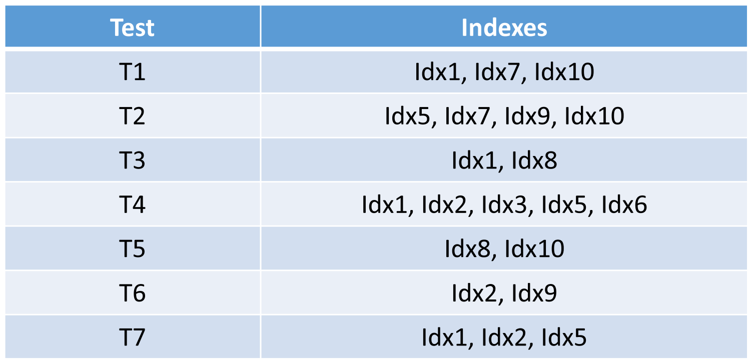

Run All Tests With All Indexes
Record Indexes Used By Each Test
Remove an "Important" Index
Run All Tests On Base Data
Run All Tests On Volume Data
Record Test Response Times
Evaluate whether response time variation can be mined to predict which tests cover a missing index

Figure 5.11. Experimental Process

The first step of the experiment process is determining which tests rely on which indexes. This step is done by executing the test with all indexes enabled and then querying the Microsoft SQL Server Dynamic Management Views to determine which indexes were used. A table is then created listing each test and the set of all indexes it used. An example is shown in Figure 5.12. For example, test $T1$ executes queries that rely on indexes 1, 7 and 10.

The next step is to remove an index. In this example, consider a removal of index $Idx5$ as shown in Figure 5.12. This means that tests 4, 6 and 7 are missing an index when run, simulating a missing index fault.

Within each experiment dataset where an index has been removed, the next step is to generate an ordering of tests. Figure 5.13 illustrates how this is done. The test is run twice with the index removed: first

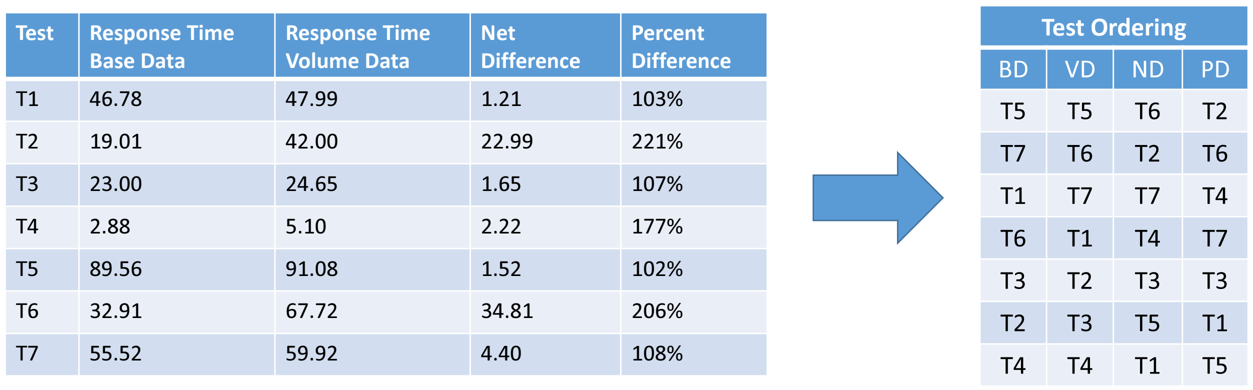


Test	Indexes
T1	Idx1, Idx7, Idx10
T2	Idx5, Idx7, Idx9, Idx10
T3	Idx1, Idx8
T4	Idx1, Idx2, Idx3, Idx5, Idx6
T5	Idx8, Idx10
T6	Idx2, Idx9
T7	Idx1, Idx2, Idx5

Figure 5.12. Index Usage

on the base data and then again on a synthetic volume dataset. The response times for each are tracked and used to also generate the net difference in response time and percentage difference in response time. The net difference is $volumetime - basetime$ and the percentage difference is $volumetime/basetime$.

This gives four different values for each test. The tests are then ordered by each of the values individually producing four orderings, shown in Figure 5.13 as BD (base data), VD (volume data), ND (net difference), and PD (percent difference).

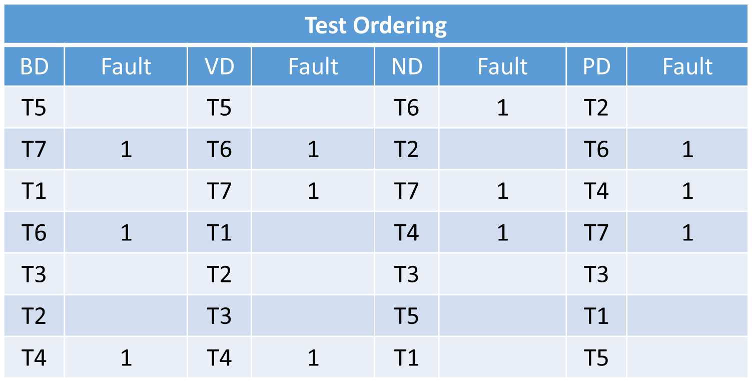


Test	Response Time Base Data	Response Time Volume Data	Net Difference	Percent Difference
T1	46.78	47.99	1.21	103%
T2	19.01	42.00	22.99	221%
T3	23.00	24.65	1.65	107%
T4	2.88	5.10	2.22	177%
T5	89.56	91.08	1.52	102%
T6	32.91	67.72	34.81	206%
T7	55.52	59.92	4.40	108%

Test Ordering			
BD	VD	ND	PD
T5	T5	T6	T2
T7	T6	T2	T6
T1	T7	T7	T4
T6	T1	T4	T7
T3	T2	T3	T3
T2	T3	T5	T1
T4	T4	T1	T5

Figure 5.13. Test Ordering

Figure 5.14 illustrates how the ground truth information is applied to the test orderings from Figure 5.13. The ground truth was determined in the first step when all tests were run with all indexes applied. By tracing the indexes used, we know which indexes should exist for a given test. When an individual index is removed for an experiment dataset, $Idx5$ in this example, we can cross reference this with the tracing information to determine which tests are now missing an index in that experiment dataset. As shown in Figure 5.14, tests 4, 6 and 7 are known to have a missing index in this experiment dataset because these tests exercise index $Idx5$. These faults are then marked in the test orderings.



Test Ordering							
BD	Fault	VD	Fault	ND	Fault	PD	Fault
T5		T5		T6	1	T2	
T7	1	T6	1	T2		T6	1
T1		T7	1	T7	1	T4	1
T6	1	T1		T4	1	T7	1
T3		T2		T3		T3	
T2		T3		T5		T1	
T4	1	T4	1	T1		T5	

Figure 5.14. Fault Detection By Test With Idx5 Removed

From the data shown in Figure 5.14, an ROC curve can be computed. Consider the net difference (ND) ordering with $Idx5$ removed. For the first ranked test, investigating this test leads to discovering a fault, one of the three faults that exist. Therefore, sensitivity (true positive rate) is “discovered-faults/all-faults”, $1/3$, 0.33. Specificity (true negative rate) is $4/4$, 1 since all tests that do not have faults were labeled as not having faults, below the first rank. For the first ranking we get the [1,0.33] point as shown in the figure.

For the top two rankings, sensitivity is $1/3$, 0.33, since one true fault out of the three total was discovered in the top two ranked tests. Specificity for the second point is $3/4$, 0.75, since the second ranked test was a false positive. So the second point in the ROC figure is [0.75, 0.33]. For the top three rankings, sensitivity is $2/3$, 0.66, and the specificity remains 0.75. So we get point [0.75, 0.66] in the ROC curve. Graphing sensitivity (true positive rate) vs specificity for all the different rankings generates the ROC curve shown in Figure 5.15 with an AUC of 0.833.

5.3.5. Data and Analysis

In this section, we present the results of this study as described in the previous section.

5.3.5.1. Overall Results

Table 5.2. AUC by Independent Variable for Each Experiment

	Control	Base Avg	Volume Avg	Net Diff
CustTrans	0.500	0.880	0.965	0.993
TaxTrans	0.500	0.882	0.900	0.906
PurchLine	0.500	0.948	0.967	0.988

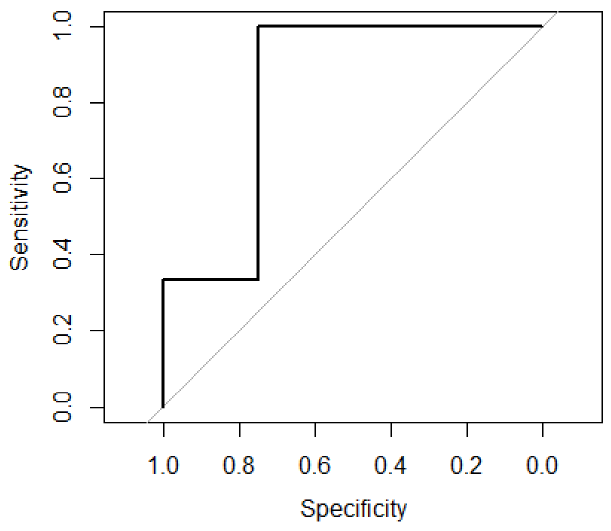


Figure 5.15. Example ROC Curve

Table 5.2 shows the AUC for the control and each heuristic for the three experiment datasets. Because the control technique is random selection, the AUC for each is 0.500. Higher numbers for AUC are better.

This data is based on three independent experiment datasets consisting of 950 tests for each. The 950 tests were the first 1000 executed by the test system with 50 removed that failed or showed extremely high variability between runs. The same tests were used in all three experiment sets, only the removed index was varied. For each heuristic in each experiment dataset, each test was executed a total of 6 times. The high and low response time dropped and the remaining 4 times averaged to yield the response time used. Standard deviation was calculated if the standard deviation was more than 25% of the average response time, that test was removed from the result set due to excessive variation.

For all experiment datasets the heuristics showed much higher AUC compared to the control. There were also slight improvements shown moving from base to volume data, and more slight improvements using the net difference between the two. This provides support for the goal of this study that was to show metrics obtained from passing test cases can be used in other quality activities. The remainder of this section will explain in more detail each of the research questions and the associated data.

5.3.5.2. RQ1 Analysis

RQ1 seeks to determine whether or not ordering tests by execution time can improve the prediction of tests that will invoke a bad query plan. The answer is yes: Ordering by execution time does effectively predict which tests encounter bad query plans. This can be seen in the first two columns of Table 5.2. The control for this research question is always 0.500, having an AUC of 0.500. The three index removals used in this study with the first heuristic ranged in AUC from 0.880 to 0.948, showing a large improvement over the naive baseline.

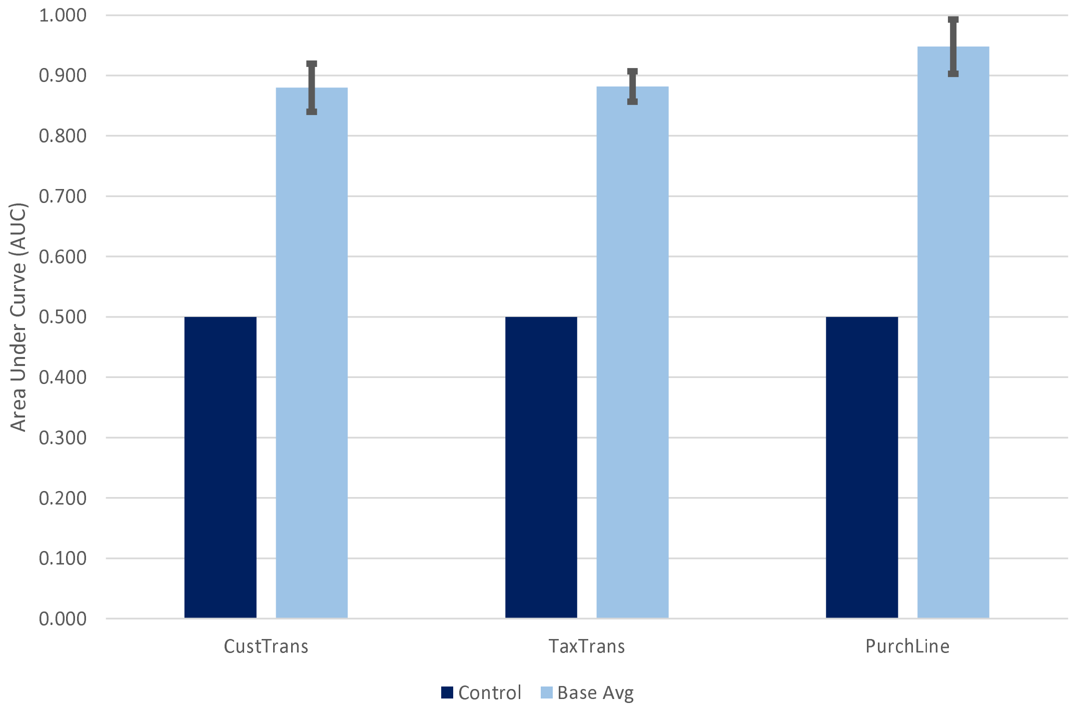


Figure 5.16. Base Data AUC with 95% Confidence Interval

Figure 5.16 illustrates the difference in AUC between the control and the heuristic of ordering by descending execution time on base data. The 95% DeLong confidence interval is also shown by the vertical bars on top of the “Base Avg” results. From this result, we can say with high confidence that the heuristic does improve test ranking for detection of bad query plans.

5.3.5.3. RQ2 Analysis

After showing that the technique shown in RQ1 was effective, we now seek to improve that ranking by gathering additional metrics. The answer for RQ2 is also yes: Additional metrics beyond response time on base data do improve predictions. In this case, the heuristic is to order the test cases based on their descending execution time when executed on a synthetic volume dataset. Because RQ2 is trying to determine what improvement can be made over the heuristic from RQ1, we are no longer comparing against random ranking and so raw AUC differences are not sufficient to determine if the technique has improved predictions. Column three of Table 5.2 shows that AUC has been increased using this heuristic, but we must determine if the improvement is due to chance or because of an actual improvement.

Using a null hypothesis that the difference in AUC is zero between the base average ordering and the volume average ordering, we find p-values of 5.78 e-10, 0.007 and 0.068 for the *CustTrans*, *TaxTrans*, and *PurchLine* experiment datasets, respectively. While the p-value for the *PurchLine* test does not meet the traditional 95% level, the other two tests clearly show the null hypothesis can be rejected. The difference in p-value between the tests appears to be primarily due to the difference in the number of positive values in each dataset. *PurchLine* has only 8 tests that exercise a bad query plan, while *CustTrans* has 50. Comparisons between the ROC curves for order by base average and volume average times are shown in Figure 5.17.

5.3.5.4. RQ3 Analysis

In RQ3, we seek to determine if more complex metrics can further improve the ranking of test cases. The answer is yes: More complex metrics calculated based on multiple other metrics do further improve predictions. The heuristic used here is the net difference in response time between base and volume test runs. This data is the most complicated to capture, as switching between base and volume data requires a redeployment of the software. During redeployment, a large percentage of disk space is impacted, thus minor things like disk fragmentation, remaining disk free space and even various hard drive performance metrics can have a noticeable impact on test response time. This research question then seeks to determine if the extra data available in this more complex metric outweighs the implicit randomness introduced by the increased complexity.

Figure 5.17 shows the overall ROC curves for all three experiment datasets. Sensitivity is the true positive rate while specificity is the true negative rate. As shown in Table 5.2, the AUC for all three has increased. Using the null hypothesis that the difference in AUC between ordering by volume average and net difference is zero, we find p-values of 0.0001, 0.018 and 0.145 for *CustTrans*, *TaxTrans* and *PurchLine*, respectively. We are unable to reject the null hypothesis for the *PurchLine* experiment dataset, but the *CustTrans* and *TaxTrans* experiment datasets both conclusively show that more advanced metrics can be mined to further improve the test case ranking.

The primary difference between these experiment datasets again appears to be due to the difference in the number of tests that exercise the bad query plans. With only 8 out of 942 positive tests in the *PurchLine* experiment dataset, there is not enough difference in the ranking to conclusively prove the results have been improved.

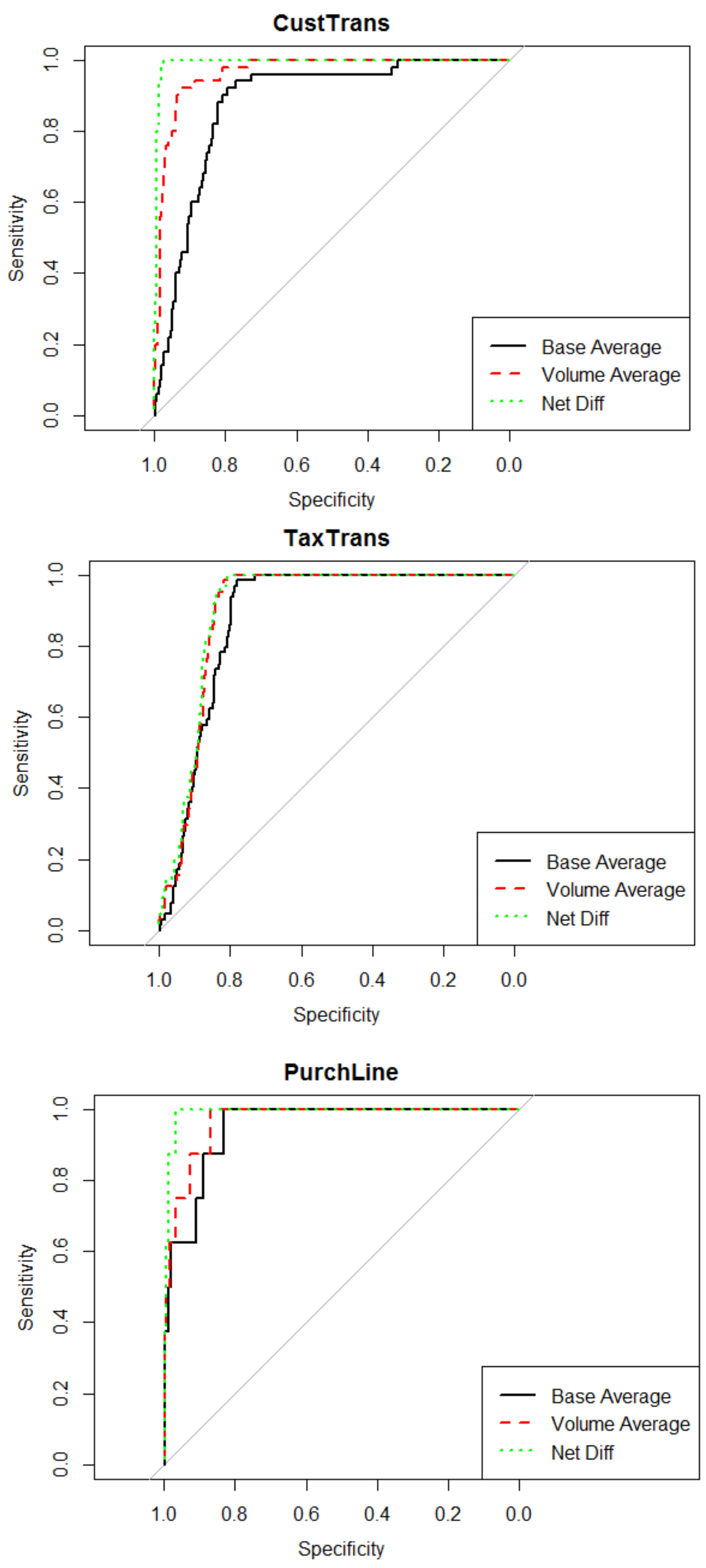


Figure 5.17. ROC Curves for Three Experiments

5.3.5.5. Other Metrics

The three metrics presented in this study so far of base response time, volume response time and net difference are obviously not the only metrics available. They were selected after manual analysis of numerous metrics that showed these three provided the best results. For example, it is also possible to calculate the percentage difference between the base average and volume average response times. However, using this metric significantly *decreases* the effectiveness of the ranking. This is shown in Figure 5.18. This and other less effective metrics thus show that the selection of proper metrics for use in ranking should be carefully considered, as more complex attributes are not necessarily better.

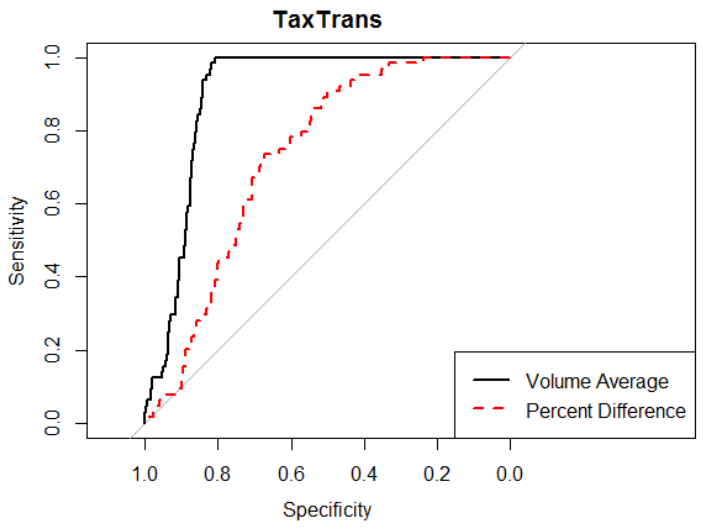


Figure 5.18. ROC Curves for Percent Difference vs Volume Average

5.3.6. Discussion and Implications

In this section, we discuss the implications of the results presented. As previously mentioned, the motivation for our research was a lack of testing resources to provide adequate non-functional test coverage across the breadth of the application. We have shown through this study that functional test metrics *can* be mined to better prioritize limited resources for performance analysis and performance testing, compared to a naive baseline of random application.

5.3.6.1. Usefulness of Approach

With this study, we have demonstrated a new technique for locating non-functional faults in software. As with any quality process, the usefulness is a factor of the benefits from using the technique, offset by the costs of employing that technique. In all three research questions, we find that the the majority of tests exposing the missing index faults occur within the first 30% of the tests run when ordered by any of the heuristics. So given a project with missing indexes such as those used in our experiment, it would be a useful technique in detecting these faults.

There are a number of caveats to this approach as described below. In order to get these results, the missing indexes had to be important, had to occur in a table with artificial volume, and the tests had to be run in a stable environment between versions.

In practice, we have found this technique difficult to use thus far. We attempted to find new, previously unknown faults with this technique, but were largely unable to for one primary reason. In the product we used, the only tables that contain synthetic volume are the ones deemed most important, which are also the same ones that have had extensive manual index analysis performed, and therefore are unlikely to have missing index faults. This technique would be much more applicable to a database product earlier in its life cycle before this extensive index analysis had been done. At that time, when more severe missing index faults exist, we believe that the technique would be much more applicable.

We also believe this technique can be further improved by including many more factors in the prediction model beyond just response time. Adding metrics to fault prediction models in functional testing has been shown to be beneficial, so we believe it will be beneficial here as well. The remainder of this section discusses the specific considerations that must be made when applying this approach in a software development project.

5.3.6.2. Environmental Purity

One of the primary issues involved in response time measurement is ensuring the test environment is identical in any comparison. Very minor environmental issues such as a difference in processor speed, a difference in RAM available, or even a change in virus scan settings can have significant and appreciable impacts on response time. Thus any response time numbers either must be performed on a highly controlled environment to ensure repeatability, or must be averaged over a significant number of executions in order to ensure environmental aspects are averaged out.

For this experiment, we used a single HyperV instance on the same physical hardware for all experiment datasets. As the RAM, hard drive space, processors and other aspects are not changed between tests, this reduces variability. Further, each test was executed multiple times in a row and the fastest and slowest results are thrown out leaving only the stable runs in between in the result set. By doing this, most test response times had a standard deviation of less than 5% of the average response time for that test.

Unfortunately, in order to switch experiment datasets, the software needed to be reinstalled between each run. This involves the re-writing of many gigabytes of data on the hard drive, which does cause variation in hard drive seek time and other aspects for different records. For this reason, we do still see variation for the same test across two experiment datasets where bad query plans and backing data were not changed between the two runs. This is similar to the issues faced in an industrial product where nightly regression test runs will similarly exhibit random variation. Even with this variation, our results show that the prioritization based on response time is still effective.

5.3.6.3. SQL Server Specifics

One particular challenge faced when examining the response time of workloads based on SQL Server is the extremely complex data caching architecture used by SQL Server to optimize disk access. As data is used, SQL Server moves data pages into RAM memory in very efficient ways that reduce future physical disk access. Since disk access is orders of magnitude slower than RAM access, this means “warm” queries that are able to pull data from RAM are often times orders of magnitude faster than “cold” queries regardless of the goodness of the query plan. This means that the order in which tests are run can have a significant impact on the observed response times.

To protect against this variation, two solutions were used in these experiment datasets. First and most simplistically, each test was executed multiple times in a row and the slowest and fastest results were ignored. This ensures that SQL has a chance to cache any required data pages prior to the execution. Secondly, the test framework was updated to call DBCC DROPCLEANBUFFERS between each test execution. DROPCLEANBUFFERS is a special command that tells SQL Server to drop all clean buffers from the buffer pool. This command is provided to allow simulating test queries on a cold buffer cache without needing to restart SQL Server between test runs. While this does not guarantee equivalent execution time between queries, it greatly reduces any impact that cold vs warm buffers have on the results.

The other SQL aspect that has a significant impact on query performance is the available memory size. Many production servers for large enterprise resource planning products like *Microsoft Dynamics AX* would have tens to hundreds of gigabytes of RAM available. In a system like this, the entire contents of the synthetic dataset can easily be held in RAM. At the same time, a typical customer deployment for an instance like this would have multiple terabytes of data. This makes it difficult to properly simulate the required database paging necessary when an extremely large dataset does not fit in RAM and encounters a bad query plan with a table scan.

To work around this issue, we used a configuration setting in SQL Server that allows a maximum RAM size to be specified. For all experiment datasets, the maximum RAM size was set to 1 gigabyte of memory. This is far lower than a customer environment would ever have. But it is large enough to still be functional, while being small enough to exhibit slowness when bad query plans are encountered. In the synthetic volume dataset used for expansion in this experiment, the top 10 tables each contain approximately 1 gigabyte of data. This forces any bad query plans on these tables to page in and out data.

5.3.6.4. Relative Time

An important aspect of this study is the fact that relative execution time between tests in a given run is a fairly accurate measure to use for prioritization. Cross-run comparisons were shown to increase the effectiveness of the prioritization somewhat, but the majority of the benefits compared to a naive baseline were obtained without the need for cross-run data. This means the issues caused by different environments, different installations, and different configurations can largely be ignored as the results are still significant (beyond the 95% threshold) even when only looking at the relative differences in response time.

5.3.6.5. Test Data Volume by Table

When we originally started this study, the plan was to apply the different heuristics and then manually analyze all query plans for the top one hundred or so tests from each heuristic, looking for bad query plans. Unfortunately, after analyzing approximately 100 tests in this way, we had only found a total of two bad query plans. We attribute this to the fact that the only around 40 (out of over 8,000 total) tables have a large number of records in the synthetic volume dataset. Additionally, less than 10% of all the tables in the system have at least 1,000 records even in the base dataset. In fact, over half of the tables in the system have three or fewer records in both the base dataset and the synthetic volume dataset. Unfortunately, for extremely small row counts, the execution time of a bad query plan that scans the table is roughly equivalent to the execution time of a good query plan that hits an index. In fact, based on statistics SQL will sometimes (properly) choose a query plan with a table scan if internal statistics indicate the table has a very small number of records.

What this means is the only tables that have a sizable number of records are also the most important tables in the system. For these tables, we have already done extensive index optimization and thus very few missing index bugs are likely to exist. The goal of this research was to provide wide product coverage in less important areas for which extensive performance testing has not been done, and thus more performance risk exists.

From this study, we understand that synthetic volume is a valuable tool in increasing the detection of bad query plans. An interesting extension to the research would be to artificially expand volume in the 90% of tables that have little or no existing data in the test datasets. With that synthetic data created, it would then be interesting to do manual query analysis ordered by execution time to see if previously undiscovered bad query plans could be diagnosed.

5.3.7. Threats to Validity

The primary threat to validity in this study is the manual selection of indexes to disable. Only three indexes were chosen due to time constraints, as the repeated runs in multiple configurations made each experiment dataset take approximately one week. Since we desired to collect all experiment datasets on the same hardware, this limited the number of indexes we could disable. The indexes that were disabled were also chosen based on the expert developer knowledge that they were heavily used throughout the system, and thus were likely to be exposed by multiple tests. This was necessary as shown in the PurchLine results because if an index is only exercised by a handful of tests the P-value of the experiment becomes too low to be useful. Were indexes chosen randomly, or were we able to detect "naturally" missing indexes, the results may be different. Unfortunately it was not possible to construct the experiment in this way.

Another threat is the variation in timing even when run on the same machine. If the running time of the test is compared across two different runs, we see much larger deviation from the mean than we see in repeated runs within the same installation. This is discussed earlier in the paper, but it adds to the possibility that run variation has had an unintended impact on the results. The consistency of the results across multiple indexes makes this less likely, but it is still a concern.

There is also a concern with the number of tests executed. 942 tests were executed, and were selected by running the first 1,000 tests detected through code reflection to have a dependency on the test dataset. 58 of these tests failed on one or more of the environmental deployments and were thus dropped from all six results sets. More tests exist than just these 1,000, so it would be interesting to see if the same results hold if the entire regression test suite were run. Unfortunately, this was prohibitively expensive in these experiment datasets due to the excessive time involved in capturing all query plans from all queries performed by all tests.

5.3.8. Conclusions

In this research, we have shown that the techniques normally applied in fault prediction and regression testing have further applications beyond functional bugs and have presented a case study using an industrial application. Non-functional fault classes, such as performance bugs, can be more accurately predicted by mining software metrics and applying similar analysis and ranking techniques. We have discussed industrial situations in which test ranking is not only economically advantageous, but also necessary based on project constraints to ensure proper non-functional quality. The major contribution of this research is the demonstration of a new application for data mining in regression testing, as well as learning about the considerations and constraints that must be taken into account when applying test ranking for predicting non-functional faults.

5.4. Test Suite Prioritization Through Telemetry Fingerprinting

The previous studies showed advanced data mining techniques can be applied to the *Microsoft Dynamics AX* product to improve regression test prioritization. We further were able to determine which attributes were the most crucial, and that these same data sources and techniques could be used to find non-functional faults as well. In this study we take those concepts forward into what we believe is the future of test case prioritization, customized prioritization based on deep product telemetry.

Traditionally, regression testing techniques have been applied statically, such as by using code churn to identify areas which most require testing. Two recent trends in software are a move to Software-as-a-Service (SaaS) and pervasive telemetry. With SaaS, software is provided via the Internet and not installed individually on a user's own computer. With this shift in how software is delivered comes a shift in how software issues are managed. When downtime occurs in traditional on-premises software, front line support is provided by IT professionals on site, and software fixes from the software vendor are expected to take some time to be implemented. With SaaS, the software vendor is expected to keep the software running at much higher levels of reliability than traditional on-premises software. When downtime occurs, service level agreements (SLAs) mean that the speed at which issues can be fixed can have major economic impacts on the profitability of a service.

Pervasive telemetry means that much more execution data is now collected than ever before. This is partially due to the move to SaaS, as execution logs can now be collected over the Internet as opposed to being written to local log files. It is also due to the decrease in the cost of storage memory; the collection of many gigabytes of data is now economically feasible for all products.

We feel that these two trends interact to provide a unique new opportunity in the area of test case prioritization. When an issue is found in SaaS products, we believe that pervasive telemetry from those products can be used to more accurately prioritize test cases based on actual usage. This will increase quality and at the same time reduce testing and repair times, both of which have positive economic benefits.

Telemetry has been used in the area of software engineering research. For example, Brooks and Memon used telemetry data to generate test cases for graphical user interfaces (GUIs) [12], and similarly, Amalfitano et al. [4] used telemetry from rich Internet applications to generate test cases. Elbaum et al. [26] have attempted to use profiling from deployed systems (telemetry data) in more traditional regression testing techniques. Their research was limited by the available telemetry at the time. The research described above has focused on regression testing for full product releases. In contrast, our research seeks to use pervasive telemetry across all deployments in order to provide custom prioritization of regression tests tailored for a single deployment when responding to critical situations requiring immediate patches.

In this study, we introduce the concept of telemetry fingerprinting. A telemetry fingerprint is an algorithm that determines how similar one set of telemetry is to another. If good fingerprinting algorithms can be identified, then test suites that closely match the telemetry fingerprint are known to be similar to the usage of that particular service. We hypothesize that algorithms that yield a higher fingerprinting score will be more effective at identifying test suites that are similar to the actual usage of the service. This allows for efficiently testing the aspects of the service that will be of most importance to users.

The primary contribution of this research is the introduction of the concept of telemetry fingerprinting for test suite prioritization. We apply this new concept to the same prerelease industrial product we used in previous studies, *Microsoft Dynamics AX*. From our empirical study, the technique of fingerprinting is shown to be effective in prioritizing test cases based on actual product usage. We also discuss issues encountered while attempting to apply this technique, as well as proposed future avenues of research.

5.4.1. Approach

Much work has been done on the current prerelease version of *Microsoft Dynamics AX* to improve regression testing. Thousands of tests that used to take minutes each to execute were rewritten to execute in mere seconds. Systems and frameworks were fundamentally changed to allow extremely rapid repeated execution of these tests, allowing for more tests to be run more often than ever before. Rather than running full regression test suites every few days, now virtually all tests are run on every code check in.

While great strides have been made to optimize regression test execution, the execution of those tests is still not without cost. Even at around one second per test, executing many tens of thousands of tests still takes multiple hours. As the software moves to a Software-as-a-Service (SaaS) model, each hour of delay during a site-down situation is now much more costly. When site uptime is measured to four- or five-nines uptime (99.99% or 99.999%) reliability, each minute of outage has a dramatic impact on monthly uptime measurements. Reliability of two-nines (99%) allows 7.2 hours of downtime per month. Going to three-nines (99.9%) allows only 43.8 minutes of downtime. At five-nines (99.999%) only 25.9 seconds of downtime is allowed per month.

In this environment, where reliability is measured such that minutes and even seconds matter, much more rigor must be placed in the selection of regression tests to execute when responding to site-down issues. An extra test suite that takes 5 minutes to run but does not find an issue can add significant cost depending on service level agreements (SLAs). In this environment, we believe that our novel approach of customizing regression test suite prioritization by deployment can yield significant cost benefits.

5.4.1.1. Proposed Approach

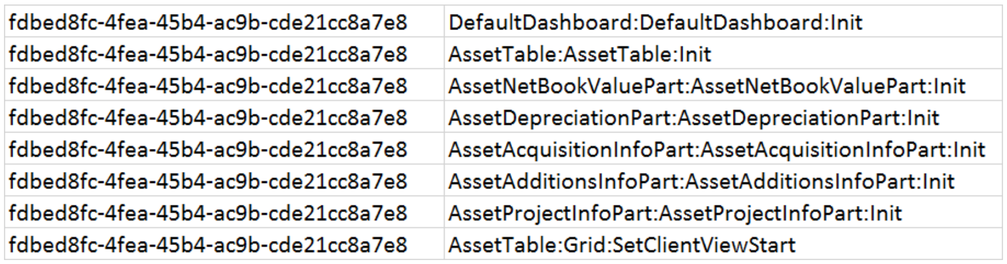
In this section, we describe the approach used in fingerprinting data processes and tests, and how that fingerprinting is utilized in test prioritization. This research was done using a prerelease version of *Microsoft Dynamics AX*. This version was installed and used in a live environment by multiple deployments. Telemetry was collected from these installations.

In this research, we use the term *telemetry fingerprint* or just *fingerprint* to mean the unique telemetry signature from a given usage of the product. In all telemetry sessions, the product and version are both the same, containing the same code. The difference is in the usage of that product in a given deployment. For instance, a distribution company will have a very different pattern of usage from a financial services company, focusing on very different areas of the product. We call these differences in usage *fingerprints*.

Figure 5.19 shows an example of the raw data. The first column is the session identifier, which identifies a single user navigating through the product. The second column is the set of distinct user interactions the user performed that caused a call to the server. The syntax of the interactions is of the format (**Form name**):(**Control name**):(**Command name**).

Using the example from Figure 5.19, the user performed the following steps:

- The user started on the default dashboard.
- The user clicked on the fixed assets form and launched it.
- While the fixed assets form was loading, a number of form parts loaded.
- The user clicked in the main grid on the fixed assets form.



fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	DefaultDashboard:DefaultDashboard:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetTable:AssetTable:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetNetBookValuePart:AssetNetBookValuePart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetDepreciationPart:AssetDepreciationPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetAcquisitionInfoPart:AssetAcquisitionInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetAdditionsInfoPart:AssetAdditionsInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetProjectInfoPart:AssetProjectInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetTable:Grid:SetClientViewStart

Figure 5.19. Sample Interaction Session

A collection of sessions similar to the one shown in Figure 5.19 exists in telemetry for each deployment. In this paper, we are trying to determine whether the collection of sessions provides a unique fingerprint for that deployment. To determine if a fingerprint exists, we need a way of determining how similar two sessions are. A variety of different session similarity functions are discussed later in this paper.

Let $I = \{i_1, i_2, \cdots, i_n\}$ denote the set of all possible actions that can be performed by a user, where n is the number of the total actions. We then define a session S to be an ordered list of actions. A single session comes from a single user and a single installation of the software product.

$$S = \{s_1, s_2, \cdots, s_l\}$$

where s_j is an action and l is the length of the session, i.e., $s_j \in I$ for $1 \leq j \geq l$.

We define a set of sessions G to be an unordered collection of sessions from the same installation of the software product.

$$G_i = \{S_1, S_2, ...S_m\}$$

where S_i is a session that belongs to the installation and m is the number of sessions in the installation. The set of all sessions from all installations is denoted as D and is defined as follows:

$$D = \bigcup_{i=1}^{l} G_i$$

To determine fingerprints, we introduce a similarity function $O : D \times D \to \mathbb{R}$ that measures the similarity between two sessions.

For sessions in an installation, G_i, we introduce two fingerprint measurements, internal and external. The internal similarity measures the average pairwise session similarity of all the sessions that belong to the same installation.

$$F_{internal}(G_i) = \frac{1}{|G_i| \times |G_i| - 1} \sum_{s_j \in G_i} \sum_{s_k \in G_i, k \neq j} O(s_j, s_k)$$

The external similarity measures the average similarity between the sessions in an installation and all the sessions in the other installations.

$$F_{external}(G_i) = \frac{1}{|G_i| \times |D \setminus G_i|} \sum_{s_j \in G_i} \sum_{s_k \in D \setminus G_i} O(s_j, s_k)$$

Finally, we produce a fingerprint score FP. The fingerprint score is the ratio of how similar sessions from the same installation are compared to sessions from other installations for a given similarity algorithm a.

$$FP(G_i) = \frac{F_{internal}(G_i)}{F_{external}(G_i)}$$

In this research, we seek to find session comparison algorithms that will maximize the value of FP for a dataset D. We hypothesize that similarity functions that yield a high FP value can be used to prioritize test suites.

Prioritization of test suites from fingerprinting algorithms can be accomplished if test cases emit the same telemetry events as those captured by interactive users. A test suite T_y yields a similar set of sessions as to those in G.

$$T_y = \{S_1, S_2, ...S_n\}$$

The similarity function can be used to calculate the average similarity between sessions in an installation G_x and the sessions in a test suite, T_y. Out of the available test suites $\{T_1, T_2, ...T_n\}$, we order the test suites from highest to lowest values of the similarity function $P(G_x, T_y)$.

$$P(G_x, T_y) = \frac{1}{|G_x| \times |T_y|} \sum_{s_j \in G_x} \sum_{s_k \in T_y} O(s_j, s_k)$$

We hypothesize that test suites T_y with the highest values of $P(G_x, T_y)$ are the most similar to the interactions performed regularly in installation G_x, and therefore should be prioritized first.

5.4.1.2. Example

This section provides an example of the computations explained in the previous section. For the purpose of this example, suppose we have two deployment datasets G_1 and G_2.

$$D = G_1 \cup G_2$$

For each of these deployment datasets, we have three sessions of data, labeled $G_1 = \{s_a, s_b, s_c\}$ and $G_2 = \{s_d, s_e, s_f\}$. This is a total of six sessions. Further suppose that the algorithm a being used for fingerprinting yields the pairwise similarity measurements listed in Table 5.3.

Table 5.3. Example Pairwise Similarities

Session 1	Session 2	Pair-wise Similarity
s_a	s_b	0.786
s_a	s_c	0.627
s_a	s_d	0.423
s_a	s_e	0.400
s_a	s_f	0.128
s_b	s_c	0.658
s_b	s_d	0.351
s_b	s_e	0.109
s_b	s_f	0.220
s_c	s_d	0.423
s_c	s_e	0.726
s_c	s_f	0.101
s_d	s_e	0.682
s_d	s_f	0.841
s_e	s_f	0.773

In order to compute the $F_{internal}$ score for G_1, we average all the pairwise similarities from G_1.

$$F_{internal}(G_1) = \frac{0.786 + 0.627 + 0.658}{3} = 0.690$$

Similarly, we compute $F_{internal}(G_2)$.

$$F_{internal}(G_2) = \frac{0.682 + 0.841 + 0.773}{3} = 0.765$$

The $F_{external}$ score is computed the same way, only using all pairwise similarities from two different deployments.

$$F_{external}(G_1) = \frac{0.423 + 0.400 + 0.128 + ...}{9}$$

$$F_{external}(G_1) = 0.320$$

In this case, $F_{external}(G_2)$ is also 0.320 since the same nine pairwise similarities exist for G_2. The final fingerprint score for each is the ratio of internal to external score.

$$FP_a(G_1) = \frac{0.690}{0.320} = 2.156$$

$$FP_a(G_2) = \frac{0.765}{0.320} = 2.391$$

These fingerprint scores are averaged to find the total fingerprinting score for algorithm a.

$$FP_a = \frac{2.156 + 2.391}{2} = 2.274$$

Specific examples of each fingerprinting algorithm used in this research are provided in Section 5.4.2.

5.4.2. Empirical Study

In this study, we seek to determine if a telemetry fingerprinting approach can be used to effectively prioritize test cases. There are two aspects that must be studied. First, we construct a way of calculating fingerprinting scores and determining their values. Once fingerprinting scores have been computed for various fingerprinting algorithms, we then determine the effectiveness of the prioritization by each fingerprinting algorithm. Our hypothesis is that a higher fingerprinting score will lead to a more effective prioritization. To investigate this hypothesis, we consider the following three research questions.

- RQ1: Can stable high fingerprinting scores be computed for different algorithms?
- RQ2: Can fingerprinting algorithms produce an effective prioritization of test suites?
- RQ3: Are high fingerprinting scores positively correlated with highly effective test suite prioritizations?

When evaluating RQ1, the primary concern is the ratio of internal to external scores (the overall fingerprinting score), because high-scoring algorithms must exist for this research to be applicable. A random scoring would on average yield a fingerprint score of 1, so the higher the fingerprint score, the more effective the algorithm.

In addition to the overall score, stability is also important. In order to be effectively employed in prioritization, those fingerprinting scores must be stable with little variability when the dataset is randomized. If the scores are not stable, then application of that algorithm in prioritization will have high randomness.

Similar to RQ1, in RQ2, we seek a stable prioritization effectiveness within a fingerprinting algorithm. If one technique always provides an effective prioritization while another one does not, then we can identify successful fingerprinting algorithms while discounting unsuccessful ones. The fingerprinting algorithm is then used to prioritize test suites that yield the highest fingerprint score for a given software installation.

Finally, in RQ3, we examine the core of our hypothesis. Algorithms that produce a high fingerprinting score are hypothesized to be highly discriminating in identifying workload similarities, so they should be effective in prioritizing test suites. We therefore expect a highly positive correlation where high average fingerprinting scores yield a high average prioritization effectiveness.

5.4.2.1. Object of Analysis

For this research, we used telemetry from a prerelease version of the software product *Microsoft Dynamics AX*. The newest version of this software product is deployed in a Software-as-a-Service (SaaS) model. This means that the software is hosted on servers owned by Microsoft Azure, and the deployment and management of the service is handled by Microsoft or other service providers.

As part of the monitoring of the software service to ensure availability, a standard set of telemetry events is collected from the running service. One such event is the *UserInteractionMarker* event, which indicates an action performed by a user. While the specifics of the action are not directly available for analysis due to privacy concerns, a hashed value for each interaction is maintained. This means that unique patterns of interaction with the application can be identified even if there is no way of determining what those specific interactions were.

In this study, we utilized telemetry streams from four prerelease software deployments. Since we are working closely with these four companies to validate the prerelease software version, we were able to obtain more detailed telemetry than would otherwise be available to us. We will refer to these deployments as “Installation A” through “Installation D”. Additionally, we used telemetry streams from two “bug bash” events, where approximately fifty engineers from Microsoft used the software as a customer would over the course of a testing event.

Table 5.4 describes the specific datasets. As described in Section 5.4.1, a session is an ordered set of actions performed by a user. We do not have access to user information, so some users may have performed only one session while other users may have performed many sessions. This data was gathered from a fourteen-day period in November 2015.

Table 5.4. Session Information by Dataset

Dataset	Number of Sessions	Average Session Length
Installation A	864	24
Installation B	58	166
Installation C	931	27
Installation D	1739	40
Bug Bash 1	55	132
Bug Bash 2	245	23

5.4.3. Variables and Measures

5.4.3.1. Independent Variables

This study manipulates one independent variable, the fingerprinting algorithm. The fingerprinting algorithm is simply a function that returns a score between zero and one, given two sets of sessions. A score of zero indicates no similarity between the sets of sessions. Two random uncorrelated sets of sessions ideally will return a fingerprinting score of zero. A score of one indicates the highest correlation between the sets of sessions. Ideally, two sets of sessions from the same deployment will return a fingerprinting score of one.

In this study, we used five fingerprinting algorithms:

- By Length: A measure of the average ratio of short session length to long session length.
- By Form Match: A measure of the ratio of forms seen in both sessions to forms only seen in one of the sessions.
- By Action Match: A measure of the ratio of actions seen in both sessions to actions only seen in one of the sessions.
- By Markov Probability: The probability of seeing one set of sessions computed via Markov chains with a correction from the other set of sessions.

- By Count of E: A ratio of the number of occurrences of the letter "E" in the action names from one set of sessions to the other set of sessions.

5.4.3.2. Dependent Variables

There are two dependent variables in this study. For RQ1, we compute the fingerprinting score. We determine both the average fingerprinting score and the standard deviation of those scores across repeated random splitting of the data to evaluate the stability of the score. The mean of the scores is compared across algorithms to determine the discriminatory abilities of the algorithm.

For RQ2, the dependent variable is the area under curve (AUC) for the receiver operating characteristic (ROC) curve for prioritization by fingerprinting score. Again, the standard deviation and mean are examined to determine the stability and effectiveness of each algorithm.

RQ3 relies on these two dependent variables to see if a highly positive correlation exists between the first (fingerprinting score) and the second (AUC).

5.4.4. Experimental Process

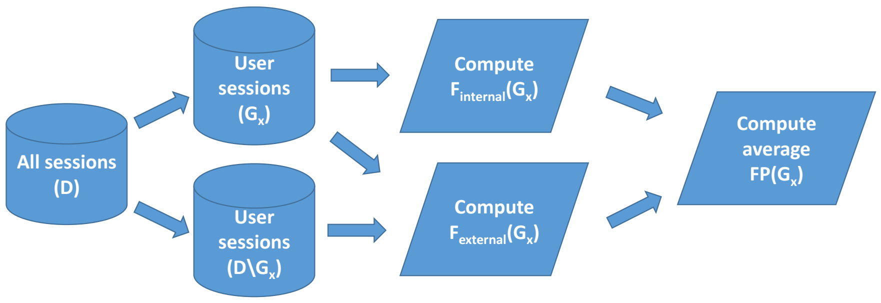

All sessions (D)
User sessions (G_x)
User sessions ($D \backslash G_x$)
Compute $F_{internal}(G_x)$
Compute $F_{external}(G_x)$
Compute average $FP(G_x)$

Figure 5.20. Experiment Process for RQ1

Figure 5.20 shows the process used for determining fingerprint scores for each similarity algorithm a. The sessions for each installation G_i are used to compute the internal and external fingerprint scores, from which the final FP_i for that installation can be computed. The fingerprint scores are then averaged to find the total fingerprinting score FP_a.

Once the fingerprint scores are known, we then wish to see if fingerprinting scores can be used to accurately prioritize test cases. Figure 5.21 shows how this process is performed. All combinations of G_x and T_y are compared for each company. The test suites T for each company are then prioritized by descending similarity score.

To obtain each test suite T_i, 20 percent of the sessions from G_x were selected as tests, then randomly grouped into five test suites each. The result is that a random selection of roughly 4 percent of the sessions from each installation were labeled as tests. The experiment was repeated thirty times to ensure that the random selection did not skew the results.

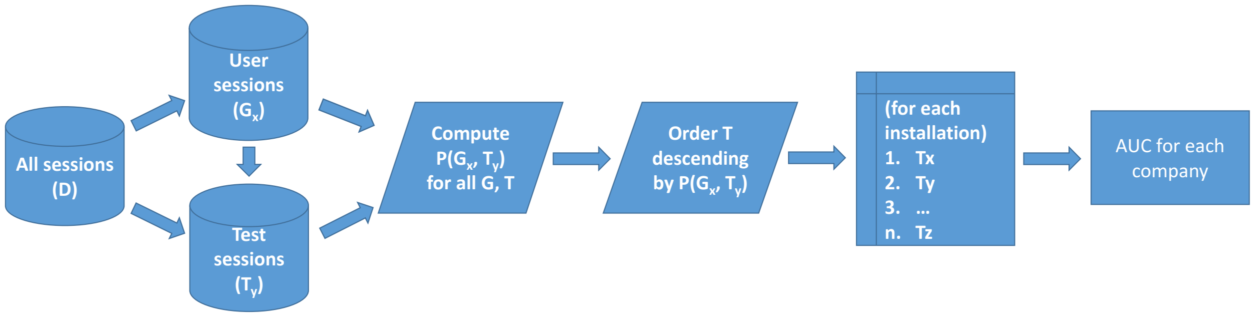


Figure 5.21. Experiment Process for RQ2

We then computed the receiver operating characteristic (ROC) curve for this set of tests. A true positive is a case in which a test came from the same original deployment dataset. A true negative is a case in which the test came from a different deployment dataset. The area under curve (AUC) for the ROC curve was then computed.

AUC is a good measurement of the performance of the prioritization, as it gives a score from zero to one. One represents a perfect ordering, in which all the tests from the deployment dataset were prioritized first. Zero is the worst possible ordering. A score of 0.5 represents a random ordering of tests.

A session comparison algorithm that provides a high FP score is theorized to also have a high AUC when prioritizing tests. Similarly, a lower FP score should yield a lower AUC, and therefore a worse prioritization.

5.4.4.1. Fingerprinting Algorithms

In this section, we describe the fingerprinting algorithms in detail using examples. In these examples, we will use the sample sessions shown in Figure 5.22.

In this example, session one shows a user starting from the main dashboard, launching the sales order form, switching rows, filling in a field, then switching rows again. Similarly, session two shows a user starting from the dashboard, going to the fixed assets form, opening the depreciation dialog, and clicking “ok”. Session three starts from the dashboard, opens the purchase order form, and confirms a purchase order. Session four again starts from the dashboard and opens the sales order form, but this time closes the enhanced preview and then switches rows twice.

```
Base Sessions                             Test Sessions

1: Dashboard:Init                         3: Dashboard:Init
   SalesTable:Init                           PurchTable:Init
   SalesTable:Grid:SelectRow                 PurchTable:Confirm:Click
   SalesTable:ItemId:SetValue
   SalesTable:Grid:SelectRow              4: Dashboard:Init
                                             SalesTable:Init
2: Dashboard:Init                            SalesTable:EnhancedPreview:Close
    AssetTable:Init                          SalesTable:Grid:SelectRow
    AssetTable:Depreciation:Click            SalesTable:Grid:SelectRow
    DepreciationDialog:OkButton:Click
```

Figure 5.22. Examples of Session Data

5.4.4.2. By length

Fingerprinting by length is done by taking the length of the shorter session and dividing by the length of the longer session. This value is averaged across all combinations in the two sets. So for this example, session three is length 3 and session one is length 5, giving a ratio of $3/5 = 0.6$. Sessions two and three yield a ratio of $3/4 = 0.75$. Sessions one and four yield $5/5 = 1$. Sessions two and four yield $4/5 = 0.8$. Averaging across all four combinations gives a final fingerprint score of $(0.6 + 0.75 + 1 + 0.8)/4 = 0.788$.

5.4.4.3. By form match

Form match is the ratio of the number of times each form from the small session exists in the big session. Using Figure 5.22 as an example, the length of session three is shorter than the length of session one. The first form in session three is the *Dashboard*, which does exist in session one. The second is *PurchTable*, which does not exist in session one. The third is again *PurchTable*, which does not exist in session one. So the overall score for this is $(1 + 0 + 0)/3 = 0.333$.

Using the same technique for sessions one/four, two/three, and two/four, we get scores of 1, 0.333, and 0.25, respectively. This yields an overall fingerprinting score of $(0.333 + 1 + 0.333 + 0.25)/4 = 0.479$.

5.4.4.4. By action match

Action match scoring is nearly identical to form match, only the full action is compared as opposed to just the form name. When comparing sessions one and four by form match we compute a score of 1, but action match only scores 0.8 because the *SalesTable: ItemId:SetValue* action is not found in session four. Completing this example, the scores for combinations one/three, one/four, two/three, and two/four are 0.333, 0.8, 0.333, and 0.25, respectively, for a total score of $(0.333 + 0.8 + 0.333 + 0.25)/4 = 0.429$.

5.4.4.5. By Markov chain

The first step in computing the Markov chain probability is to generate a matrix of how many times each navigation was observed in the base sessions. This matrix serves as the base emission matrix for the Markov chains. Figure 5.23 shows this initial matrix.

	Dashboard:Init	SalesTable:Init	SalesTable:Grid:SelectRow	SalesTable:ItemId:SetValue	AssetTable:Init	AssetTable:Depreciation:Click	DepreciationDialog:OkButton:Click
Dashboard:Init		1			1		
SalesTable:Init			1				
SalesTable:Grid:SelectRow				1			
SalesTable:ItemId:SetValue			1				
AssetTable:Init						1	
AssetTable:Depreciation:Click							1
DepreciationDialog:OkButton:Click							

Figure 5.23. Example Initial Emission Matrix

A correction must be made to account for transitions that do not exist in the base sessions but do exist in the test sessions. For example, the navigation from the *Dashboard* to *PurchTable* does not exist at all in the base sessions, nor does the navigation to *SalesTable:EnhancedPreview:Close*. So without modifying the emission matrix, the resulting probability would almost always be zero.

To account for these missing navigations, we apply a correction. This is illustrated in Figure 5.24. A small value is added to the cell for the *Dashboard:Init* to *PurchTable:Init* navigation to avoid yielding a zero probability. To avoid impacting the existing navigation ratios, that same value of 0.1 is also added to the two other navigations. This same process is used for all missing navigations.

Once all necessary corrections have been made to the emission matrix, a probability can then be computed. The probability of seeing session three given the emission matrix is the probability of each navigation multiplied together. The probability of the *Dashboard:Init* to *PurchTable:Init* navigation is $0.1/(1.1 + 1.1 + 0.1) = 0.0435$. The probability of *PurchTable:Init* to *PurchTable:Confirm:Click* is

	Dashboard:Init	SalesTable:Init	SalesTable:Grid:SelectRow	SalesTable:ItemId:SetValue	AssetTable:Init	AssetTable:Depreciation:Click	DepreciationDialog:OkButton:Click	PurchTable:Init	PurchTable:Confirm:Click	SalesTable:EnhancedPreview:Close
Dashboard:Init		1.1			1.1			0.1		
SalesTable:Init			1.1							0.1
SalesTable:Grid:SelectRow			0.1	1.1						
SalesTable:ItemId:SetValue			1							
AssetTable:Init						1				
AssetTable:Depreciation:Click							1			
DepreciationDialog:OkButton:Click										
PurchTable:Init									0.1	
PurchTable:Confirm:Click										
SalesTable:EnhancedPreview:Close			0.1							

Figure 5.24. Example Emission Matrix with Corrections

$0.1/0.1 = 1$. So the total probability of seeing session three given the emission matrix is $0.0435 * 1 = 0.0435$.

The same technique is used to compute the probability of session four.

$$(1.1/(1.1 + 1.1 + .1)) * (.1/(1.1 + .1)) * (.1/.1) * (.1/(1.1 + .1))$$

$$= 0.478 * 0.083 * 1 * 0.083$$

$$= 0.00302$$

The average fingerprint score for these sets of sessions is therefore $(0.0435+0.00302)/2 = 0.0233$. Note that the Markov chain scores are very small compared to other scores. But since both the internal and external scores are similarly small, the resulting fingerprint ratio is still directly comparable with other techniques.

5.4.4.6. By E count

The last technique is arguably very naïve, but is included to show the power of this fingerprinting approach. In this technique, the ratios of the number of occurrences of the letter *E* from one session to another session are compared. In session one, the letter *E* occurs fourteen times (twice in *SalesTable:Init*, four times in *SalesTable:Grid:SelectRow* and so on). It occurs eight times in session two, twice in session three, and seventeen times in session four.

The ratio of smaller to larger count in each comparison of one/three, one/four, two/three, and two/four yields a fingerprint score of:

$$((2/14) + (14/17) + (2/8) + (8/17))/4 = 0.422$$

While this technique may appear somewhat random, note that the number of times the letter *E* occurs in a given session is somewhat related to which forms are in use in that session. *SalesTable* has two, while PurchTable only has one. This means that even a silly technique like counting letters will encode some information about whether or not the same forms are being used in two sessions.

5.4.5. Data and Analysis

In this section, we present the results of this study as described in the previous sections. As shown in Tables 5.5 and 5.6, the standard deviation for both the fingerprinting score (Average FP) and the area under curve for the receiver operating characteristic curve (Average AUC) demonstrate that the results had little relative variance in repeated experiments with different random samplings.

From these results, we can see that the form name and action name fingerprinting methods were clearly superior to the others. Recall that the fingerprinting score is a ratio of internal similarity to external similarity, meaning these methods showed that sessions from the same dataset were seven times more similar than sessions from different data sets.

This high measure of similarity translated well to the AUC measure when the test sessions were prioritized by fingerprinting score. In the case of both form and action name algorithms, the resulting prioritization was nearly perfect.

5.4.5.1. RQ1 Analysis

RQ1 seeks to find algorithms for which there is a high internal to external ratio, yielding a high fingerprinting score. A purely random algorithm with no measure of similarity between sessions would yield a fingerprinting score (ratio) averaging 1 across repeated experiments. There is an upper bound on fingerprinting score as well, since not all sessions from the same deployment are identical. Theoretically, the upper bound approaches infinity as internal scores approach 1 and external scores approach 0. Pragmatically, however, this is not possible, unless internal sessions are nearly identical and external sessions are very different. In practice, since customer deployments are all using the same software, there will be an upper limit on any fingerprinting score. We do not have a quantification of that limit at this time.

Table 5.5. Fingerprint Scores by Algorithm

Fingerprint Method	Avg FP	Std Dev
By Session Length	1.131	0.021
By E Count	1.105	0.017
By Markov Chains	2.391	0.887
By Form Name	7.257	0.358
By Action Name	9.336	0.570

To determine the fingerprint scores, we compared 80 percent of the sessions from each installation with 80 percent of the scores from each other installation. Using the previously described techniques to compute internal and external scores, we determined the final average fingerprint. This experiment was performed thirty times and averaged to ensure that the scores remained stable and were not negatively impacted by anomalies in the data.

As shown in Table 5.5, all algorithms had a fingerprinting score greater than 1, meaning that all were capable of providing useful information on whether two sessions came from the same deployment. Form name and action name were clearly superior. We had expected Markov chains to be better than the data shows, however. In looking at specific examples, it appears that the high occurrence of missing navigations from the test sets led to greater variability in the scores and therefore decreased fingerprinting abilities. We attempted to work around this by changing the penalty for missing navigations, but for this dataset a 0.1 penalty yielded the best results.

It is also interesting to note that while session length and E count are seemingly meaningless measures, they did provide some benefit in session identification. We believe this is because they do encode some information about how similar two actions were. For instance, manually creating and posting a large business document may take thousands of steps, while approving an auto-generated invoice may take fewer than twenty steps. So, while the fingerprint score was not much above the minimum value of 1, we can see from the standard deviation that it was still better than average.

Finally, RQ1 examines the stability of the fingerprinting scores. We see from the standard deviations in Table 5.5 that the highest variability was seen with the Markov chains algorithm. Better stability was seen in the algorithms using form and action names.

The answer to RQ1 is yes, high-scoring yet stable fingerprint algorithms can be found that distinguish between internal and external sessions.

5.4.5.2. RQ2 Analysis

RQ2 attempts to apply the fingerprinting algorithms in order to prioritize tests (for our case, randomly selected sessions that are treated as tests). A perfect prioritization would order sessions from the same dataset first, followed by sessions from all other datasets. This would yield an AUC for the ROC curve of 1.0. A purely random ordering of sessions on the other hand would yield an AUC on average of 0.5. So any ordering with an AUC above 0.5 is a better prioritization than random ordering. Random ordering is commonly used in industry. As the AUC approaches 1.0, the prioritization increases in effectiveness.

Similar to RQ1, this was done as thirty separate experiments to avoid any impact of data anomalies. Each installation was divided randomly into 80 percent of sessions marked as user sessions and 20 percent marked as test sessions.

Table 5.6. AUC by Fingerprint Method

Fingerprint Method	Avg AUC	Std Dev
By Session Length	0.574	0.043
By E Count	0.597	0.039
By Markov Chains	0.655	0.049
By Form Name	0.972	0.011
By Action Name	0.973	0.012

Similar to the fingerprinting scores, Table 5.6 shows that order by form name and action name yield a highly effective prioritization. This means that if we apply this technique to prioritize tests based on fingerprinting, we would run the tests directly associated with that deployment customer's business practices ahead of all other tests.

Using Markov chains provided some benefit in prioritization, but not nearly as much as using form and action names. The session length and E count techniques, while better than average prioritizations, were not all that useful. As expected, these techniques would not be very useful in a production environment, as we would expect that even manual prioritization would greatly outperform them.

The answer to RQ2 is yes, fingerprinting algorithms can be used to successfully prioritize test suites.

5.4.5.3. RQ3 Analysis

RQ3 examines the core of our hypothesis, that finding algorithms with a high fingerprinting score will enable effective prioritization of test suites tailored to a given software installation. We evaluate this research question by computing the Spearman Rank correlation coefficient [74] between the fingerprint score and the AUC of the ROC curve for each fingerprinting method and installation dataset. This yields a value between -1 and 1. Uncorrelated values will on average have a correlation coefficient of 0, while the value approaches -1 or 1 as correlation increases.

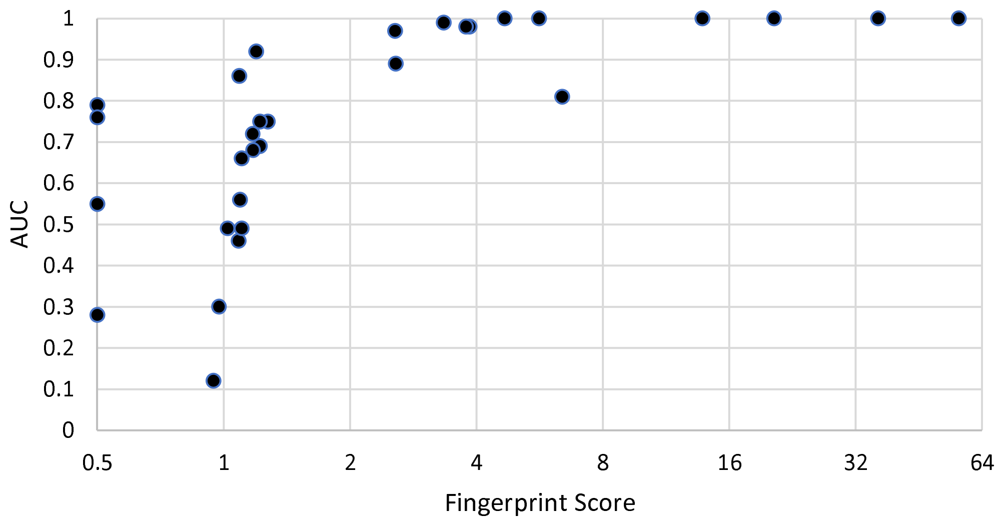


Figure 5.25. AUC by Fingerprinting Score

Computing the value across these five methods and six data sets as shown in Figure 5.25, we find a Spearman Rank correlation coefficient of 0.884, meaning that the fingerprinting score for an algorithm is highly correlated to the prioritization ability of that algorithm.

The answer to RQ3 is yes, having a high fingerprinting score means that an algorithm is more suitable to effective test suite prioritization.

5.4.6. Discussion and Implications

In this section, we discuss the implications of these results, as well as some of the considerations that must be made in attempting to apply the techniques.

5.4.6.1. Application of Techniques

The first obvious question arising from this research is how these results can be applied in practice. For this discussion, we will use *Microsoft Dynamics AX* as a canonical example.

Different installations of the software product will exhibit varying usage characteristics. For instance, a manufacturing company may heavily use shop floor control functionality, while a financial services company would not use it at all. That means that regressions in the shop floor control module would have little to no impact on the financial services company, while they would cause significant harm to the manufacturing company.

To apply this technique, we try to determine a usage pattern from an installation when testing critical one-off software fixes. Any given installation of the *Microsoft Dynamics AX* product in the cloud may span multiple servers, but it is marked by a single "tenant ID," which is the unique identifier for that installation. The source tenant is listed on each telemetry event, allowing a custom test prioritization to be created for each installation.

In a business-down critical situation, the risk of a change introducing a new bug must be weighed against the cost of remaining down, which in many cases may be tens to hundreds of thousands of dollars per hour. In these situations it may make sense to perform preliminary testing, fix the service even though regression testing is only partially complete, then complete the remainder of testing and analysis once the service is back up.

To do this, fingerprinting algorithms that have been shown to produce high fingerprinting scores for this product's telemetry stream can be used to quickly prioritize regression test suites based on their known telemetry stream. Those test suites with the highest resulting fingerprinting scores would be prioritized first, thereby reducing overall risk to that company's operations, since the tests match how that company actually uses the software on a daily basis.

5.4.6.2. Specialization of Workloads

Fingerprinting is easiest when there are very specialized aspects to a telemetry stream that make identification of similar sessions obvious. This is the case with two of the datasets examined in this study.

The company described as “Installation C” in Table 5.4 used the *Microsoft Dynamics AX* product as a platform and built a completely custom application on top of it. That means that the forms seen in each session never occurred in any sessions from other datasets. This makes the measures by form count and action count highly selective.

Similarly, the company described as “Installation A” only used a small portion of the product, a single module. So while these forms did occur in other datasets, they were a small subset of the forms seen in those datasets, while they made up a majority of the forms for this company’s sessions. While this may seem to discount the validity of this research at first, in practice this is how many companies use this product. And if they are only using one of the many modules, a prioritization that extensively tests that module first is valuable.

Because only a small number of datasets were available at the time of this research, we would like to examine additional datasets in the future. Markov chains were not nearly as useful as simple form and action name matching, but this was largely due to the high specialization of workloads. We believe that in workloads performing different business processes using the same forms, techniques such as Markov chain analysis would become much better than form and action name matching.

Because variation between datasets may significantly impact the effectiveness of fingerprinting techniques on prioritization, it is important to be able to determine which will be most effective for a given product. As we have shown through RQ3, the fingerprinting scores based on existing data can be computed prior to doing prioritization. Those fingerprinting scores are highly accurate in predicting which algorithm should best be used in prioritization.

5.4.6.3. Test Suite Size

When we first performed this research, we found much higher Markov chain fingerprint scores, sometimes close to 20 or 25. But at the same time the AUC for the Markov chains was very poor, closer to 0.550. After analyzing specific instances, we determined that this was due to the high penalty for missing navigations in a given test session. Two or three missing navigations would yield a minuscule fingerprint score, even in cases where the rest of the session was clearly similar to the base sessions. This caused some test sessions to be artificially prioritized last, leading to very low AUC scores.

To work around this issue, we grouped individual test sessions into suites of ten tests. The result was that a few missing navigations in an individual test would not cause as much harm, as the other highly similar tests increased the prioritization.

This is similar to what occurs in practice. Tests in this product are already grouped by area of focus into test suites. Those suites are the smallest granularity at which test scheduling can be performed anyway. Thus, the technique of test case grouping by suite mirrors how product development works.

5.4.7. Threats to Validity

The primary threat to validity in this study is the relatively small number of datasets available. Only four came from real customer usage, with the other two coming from internal "bug bash" sessions performed by the manufacturer of the product. The data available for each was also relatively small, as these customers had only been live on the software for a short period of time when this research was performed. It is possible that the usage of the software may change over time. If this is the case, a windowed approach to the data may be necessary to maintain accuracy, as was shown by Anderson et al. [6].

Another threat is the reliance of this research on the ability to use actual form and action names. These values were available in telemetry from the prerelease version of the software, but in a released product they will likely be hashed to avoid storing any private or identifiable information. Hashing form names would make algorithms such as the E count technique that rely on the specific names of forms and actions not possible. Other techniques such as form name and action name matching would still be possible, as the hashed values can be checked for equivalence.

A major risk to validity is the way in which the ground truth for prioritization capability was measured. We took random sessions from production data and labeled them as test sessions. Test sessions do emit the same telemetry stream as production data. But there may be subtle differences in telemetry when testing the system to telemetry in real environments that would change the measure of an ideal prioritization. Unfortunately, there is no true ground truth that can be used in this case. Even if manual prioritization or prioritization by requirements specifications were to be used, past research has shown that these measures are not perfect either [1, 8, 30]. For this reason, we think that treating production sessions as analogous to test sessions is the most accurate measure available.

5.4.8. Conclusions

In this study, we investigated whether the use of telemetry can improve test case prioritization. The empirical results with an industrial application have shown that fingerprinting algorithms based on usage telemetry can be effective in test case prioritization. Further, our results have shown that these same algorithms can be employed to prioritize test cases to tailor test execution to how a given software installation is being used. Finally, the results have shown that higher fingerprinting scores are more effective in prioritizing tests, thereby allowing evaluation and selection of fingerprinting methods prior to applying them in a customer-down critical situation.

We believe that the proposed techniques are a promising way of applying traditional prioritization techniques in a new software world of Software-as-a-Service in which telemetry is pervasive and downtime is unacceptable. If software testing can be custom-fit to the usage in a particular installation, downtime costs can be reduced while quality is retained. We also believe that this will become increasingly important as software velocity continues to increase.

6. GUIDANCE TO PRACTIONERS AND RESEARCHERS

In this chapter, we discuss guidance to practitioners and researchers based on the findings from this dissertation. The guidance is based both on the state of the art as determined through the literature survey, and on the findings from the empirical studies we performed. Following this guidance will help practitioners and researchers more effectively employ and study regression testing techniques.

6.1. Data Sources

First and most importantly, the data sources selected have greater influence on the outcomes of regression testing techniques than other attributes. Studies have shown that measures that are more capable of distinguishing among tests are more effective. For example, using code coverage as a data source is much more effective when the code coverage levels vary widely among tests. When the coverage levels are very similar, the effectiveness of the techniques suffers greatly.

Second, when examining data sources, it is advised to be aware that a strong false signal from one data source can easily overwhelm other signals, thereby degrading the benefits of applying the techniques. Examples include false failures due to "flaky tests" as described by Anderson et al. [7]. Sometimes these false signals can be cleaned from the data as demonstrated by Herzig et al. [41] But the existence and levels of false signals must be considered when identifying data sources.

6.2. Regression Testing Techniques

As discussed earlier, many studies have shown that relatively simple techniques can improve the effectiveness of regression testing techniques. For example, in the case of test case reduction, when coverage is used as a data source, multiple studies have shown that greedy algorithms (e.g., selecting test cases by the number of additional blocks they cover) are generally the most effective. Similarly in prioritization, simply ordering by tests with the highest levels of code coverage first yields good results. But this varies somewhat depending on the programs being used for the experiment or the cost of applying the techniques. The results obtained from these simple methods have been repeatedly shown to be nearly as effective as more complex techniques.

The one primary case where more advanced techniques are required is when an explicit constraint must be met, such as ensuring that the test run time is a specific duration, or ensuring coverage levels are above a specific percentage. For example, if all tests must execute in less than one hour, then a simple ordering by code coverage is less successful than a more advanced linear programming solution that optimizes multiple variables at the same time. These constraint problems lend themselves well to the use of linear programming as an optimization technique due to its ability to easily encode the constraints within the problem.

Advanced data mining techniques such as classification, genetic algorithms, and others have been shown to be slightly more effective than simple techniques. However, the increase in effectiveness is very small, and empirical studies indicate that, typically, the cost of employing these techniques (i.e., implementation, data cleansing, model training, and selection) tends to be high. This means that in most cases the benefits of moving to a more advanced regression testing technique do not outweigh the costs.

There is one situation in which these more advanced techniques are advantageous. As previously discussed, often there is no way of knowing ahead of time which data sources should be considered for a specific project. If historical data is available to learn a classification model, this technique can be used to identify which data sources are effective for a project. Care must be exercised when using this approach because as previously mentioned, different project phases will correlate with different performance characteristics of data sources. This means that the training data used in classification and other machine learning techniques should ideally come from a similar development phase of a similar project.

6.3. Project Attributes

A variety of project attributes make a project more likely to exhibit good results for regression testing techniques. One attribute discussed in multiple research papers is the size of the program. Across various techniques, it was often seen that performance breaks down for extremely small programs, such as those under five hundred lines of code. This can be explained because for extremely small programs, most tests exercise almost all code in the program. In most cases, a program must be large enough where different tests execute different areas of functionality in order for regression testing techniques to be effective.

As discussed in the data sources section, the accuracy of the data source is also important. An obvious issue exists when tests exhibit false failures due to environmental instability. Some studies have also called out issues with accuracy around items like coverage data, in which obtaining full code coverage data from test runs is so costly that it is only done once in a while. This means that coverage data may not be accurate, particularly in the areas of higher churn, leading to a decrease in the effectiveness of regression testing techniques.

Beyond the global aspects of a project, such as its size or environmental stability, the current phase of a project must also be considered. Most of the literature we surveyed either considers empirical evaluations of regression testing techniques based on an artificial set of changes and/or bugs, or considers only changes between versions of a product. In many industrial applications, test suites are run many times within a single version of the software. A project relying on a traditional waterfall development process will exhibit significantly different results from different regression testing techniques at different project points. For instance, during the planning phase, code churn is minor. During the coding phase, both churn and architectural impacts are high. During the testing phase, churn may remain high, while the size and architectural impact of changes decrease. Considering the previous guidance with respect to selecting data sources with high variation, this may mean that churn is a better data source during the development phase when software code changes are more often associated with differences in functionality compared to during the planning phase when there is less variation between tests based on churn.

6.4. Guidance to Practitioners

From this survey, it appears that most of the benefits of test selection, reduction, and prioritization are gained from the removal of less important tests, not from discrimination between equally important tests. This is why doing something is usually better than doing nothing, even if naïve data sources and techniques are used. The implication to industry is that test selection, reduction, and prioritization can be effective techniques, but they should not be overemphasized. Based on the literature surveyed, we recommend the following steps to successfully employ the techniques of test selection, reduction, and prioritization.

- Start by determining which data sources are available in a given project. Common sources are churn information (from source control), coverage data, and bug information.
- Exclude data sources that may be misleading. For instance, if a high percentage of test results are known to be flaky, exclude historical test results as a data source.
- From these data sources, select data sources that have the highest variation between tests are the easiest to collect and maintain.
- Determine the constraints of the environment in which the tests will execute. For instance, take into account maximum run time or minimum required code coverage levels.

- Determine the attributes to focus on. This will often be coverage level but may also include identification of test failures or enhancement of bug detection.
- Build up a simple heuristic model based on the data sources, constraints, and attributes. Simple techniques will usually be the most effective when we account for the implementation cost. Linear programming can be used to encode constraints.
- If the relationships among variables are not obvious and historical data is available, consider learning a classification model to determine which data sources and heuristics yield the best results.

The research papers described in the previous section have shown that these steps will provide the most benefit at the least cost. Improving the effectiveness of regression testing techniques further will add greater cost, and will require attempting a wide range of expensive techniques on a wide range of data sources to determine which technique works the best for a specific project during a specific phase. Also note that the effectiveness of the data sources and techniques will also vary over the project life cycle.

6.5. Guidance to Researchers

The vast majority of research in the areas of test selection, reduction, and prioritization has used naïve baselines of test-all or random approaches. As each technique shows widely varying effectiveness across applications, direct comparisons with other techniques have generally been avoided. This leaves practitioners to try a variety of data sources and techniques on their own to find out what works for them. Future research in regression testing techniques would better serve practitioners by focusing more heavily on when a given data source or technique will prove more effective than other data sources or techniques. This can be accomplished by determining why a given data source or technique is effective in the first place. For example, if a technique detects and removes test cases that are largely redundant, then that technique will be most effective in a project with a high occurrence of redundant test cases, while being less effective in a project with low redundancy. By providing this deeper level of understanding, industrial practitioners will be able to more easily select and apply the techniques described in the research.

7. CONCLUSIONS AND FUTURE WORK

In this dissertation, we have performed the following steps in pursuit of our thesis.

- We developed a conceptual framework for understanding and evaluating regression testing techniques
- We have surveyed existing literature to understand the current state of the art in regression testing techniques.
- We have performed empirical studies to show that advanced data mining together with regression testing techniques are applicable to the *Microsoft Dynamics AX* product.
- We have extended traditional regression testing techniques to novel applications within the *Microsoft Dynamics AX* product.
- We have developed customized regression testing techniques based on the specific context of the *Microsoft Dynamics AX* and shown empirically that they are effective.

7.1. Merit and Impact of This Research

This research provides two primary benefits. First, it provides practitioners, such as a *Microsoft Dynamics AX* team, the first comprehensive understanding of how to choose and apply advanced data mining and regression testing techniques based on their unique product context. This knowledge will enable industrial applications to more effectively use these techniques and thereby improve economic benefits.

Second, this dissertation helps future researchers to better understand why and when various data sources and approaches will be more effective in applying regression testing techniques. This will hopefully provide more effective future research in this important area of study.

7.2. Future Directions

Similar to the merit and impact of this research, there are two primary future directions we will pursue. For the industrial application *Microsoft Dynamics AX*, we will continue working to implement and enhance regression testing techniques to improve our quality activities. Findings from this dissertation will help improve regression testing processes with that application.

From a research perspective, we wish to further investigate the impact of context on other industrial and open source products. With a more concrete understanding of how the contextual attributes of these programs impact the effectiveness of regression testing techniques, we hope to further refine the application of those techniques and improve regression testing approaches.

REFERENCES

[1] D. Aceituna, H. Do, and S. Lee. A human interactive approach to building requirements models. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2010.

[2] R. Agrawal and R Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, September 1994.

[3] Nadia Alshahwan and Mark Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 181–192. ACM, 2014.

[4] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Rich internet application testing using execution trace data. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 274–283. IEEE, 2010.

[5] Jeff Anderson, Hyunsook Do, and Saeed Salem. Experience report: Mining test results for reasons other than functional correctness. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 405–415. IEEE, 2015.

[6] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 142–151. ACM, 2014.

[7] Jeff Anderson, Saeed Salem, and Hyunsook Do. Striving for failure: An industrial case study about test failure prediction. In *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.

[8] Md Arafeen, Hyunsook Do, et al. Test case prioritization using requirements-based clustering. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013*, pages 312–321. IEEE, 2013.

[9] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, Laszlo Lango, and Tibor Gyimóthy. Code coverage-based regression test selection and prioritization in webkit. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 46–55. IEEE, 2012.

[10] J. Bible, G. Rothermel, and D. Rosenblum. Coarse- and fine-grained safe regression test selection. *ACM Transactions on Software Engineering Methodology*, 10(2):149–183, April 2001.

[11] Oren Boiman, Eli Shechtman, and Michal Irani. In defense of nearest-neighbor based image classification. In *IEEE Conference on Computer Vision and Pattern Recognition, 2008*, pages 1–8. IEEE, 2008.

[12] Penelope A Brooks and Atif M Memon. Automated gui testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342. ACM, 2007.

[13] C. Catal and D. Mishra. Test case prioritization: A systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.

[14] Archana Chaudhary, Savita Kolhe, and Raj Kamal. Machine learning techniques for mobile intelligent systems: A study. In *2012 Ninth International Conference on Wireless and Optical Communications Networks (WOCN)*, pages 1–5. IEEE, 2012.

[15] Jeffrey A Clark and Dhiraj K Pradhan. Fault injection: A method for validating computer-system dependability. *Computer*, 28(6):47–56, 1995.

[16] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 357–366. IEEE, 2011.

[17] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *International Journal on Empirical Software Engineering*, 10(4):405–435, 2005.

[18] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 71–82, November 2008.

[19] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 26(5), September 2010.

[20] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2006.

[21] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, September 2006.

[22] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 113–124, November 2004.

[23] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

[24] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, pages 329–338, May 2001.

[25] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[26] Sebastian Elbaum and Madeline Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[27] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338. IEEE Computer Society, 2001.

[28] E. Engstrom, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14 – 30, 2010.

[29] Norman E Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[30] Arnaud Gotlieb and Dusica Marijan. Flower: optimal test suite reduction as a network maximum flow. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 171–180. ACM, 2014.

[31] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering Methodology*, 10(2):184–208, April 2001.

[32] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[33] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer Supported Cooperative Work*, pages 395–404. ACM, 2011.

[34] Shifa-e-Zehra Haidry and Ted Miller. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, 39(2):258–275, 2013.

[35] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[36] J. Han, J. Pei, Y. Yin, and R Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. In *Data Mining and Knowledge Discovery*, pages 53–87, January 2004.

[37] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 34th International Conference on Software Engineering*, pages 738–748. IEEE Press, 2012.

[38] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[39] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, March 2001.

[40] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–150. IEEE, 2010.

[41] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 39–48. IEEE Press, 2015.

[42] S. Hou, L. Zhang, T. Xie, and J. Sun. Quota-constrained test case prioritization for regression testing of service-centric systems. In *Proceedings of the International Conference on Software Maintenance*, pages 257–266, September 2008.

[43] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[44] Pooja Jain, Jonathan M Garibaldi, and Jonathan D Hirst. Supervised machine learning algorithms for protein structure classification. *Computational Biology and Chemistry*, 33(3):216–223, 2009.

[45] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[46] J. Jones and M.J. Harrold. Test suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engingeering*, 29(3):193–209, 2003.

[47] Bach Kaner and pettichord. *Lessons Learned in Software Testing*. Wiley Computer Publishing, 2002.

[48] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, pages 119–129, May 2002.

[49] Mijung Kim, Jake Cobb, Mary Jean Harrold, Tahsin Kurc, Alessandro Orso, Joel Saltz, Andrew Post, Kunal Malhotra, and Shamkant B Navathe. Efficient regression testing of ontology-driven systems. In *Proceedings of the 2012 international symposium on software testing and analysis*, pages 320–330. ACM, 2012.

[50] B. Korel. The program dependence graph in static program testing. In *Information Processing Letters*, volume 24, pages 103–108, 1987.

[51] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *Proceedings of the International Conference on Software Maintenance*, pages 247–256, September 2008.

[52] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 442–453, November 2003.

[53] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[54] B. Livshits and T. Zimmermann. DataMine: Finding common error patterns by mining software revision histories. In *International Symposium on Foundations of Software Engineering*, pages 296–305, September 2005.

[55] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[56] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Proceedings of the International Conference on Software Maintenance: Frontiers of Software Maintenance*, pages 88–108, September 2008.

[57] Michael Mayo and Simon Spacey. Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics. In *Search Based Software Engineering*, pages 158–171. Springer, 2013.

[58] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[59] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[60] Microsoft. Microsoft Dynamics AX product description. https://www.microsoft.com/en-us/dynamics/erp-ax-overview.aspx. Accessed: 2016-04-06.

[61] Siavash Mirarab, Soroush Akhlaghi, and Ladan Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE Transactions on Software Engineering*, 38(4):936–956, 2012.

[62] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Fundamental Approaches to Software Engineering*, pages 276–290. Springer, 2007.

[63] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *International Symposium on Empirical Software Engineering and Measurement*, pages 364–373, 2007.

[64] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, May 2006.

[65] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *First International Symposium on Empirical Software Engineering and Measurement*, pages 364–373. IEEE, 2007.

[66] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[67] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *International Conference on Software Engineering*, pages 465–475. IEEE, 2003.

[68] X. Qu, M. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 75–86, July 2008.

[69] Xiao Qu, Mithun Acharya, and Brian Robinson. Impact analysis of configuration changes for test case selection. In *22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 140–149. IEEE, 2011.

[70] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering Methodologies*, 13(3):227–331, July 2004.

[71] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *IEEE International Conference on Software Maintenance*, pages 179–188. IEEE, 1999.

[72] Rickard Sandberg, Gösta Winberg, Carl-Ivar Bränden, Alexander Kaske, Ingemar Ernberg, and Joakim Cöster. Capturing whole-genome characteristics in short sequences using a naive bayesian classifier. *Genome Research*, 11(8):1404–1409, 2001.

[73] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 81–90, November 2007.

[74] Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[75] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[76] Matt Staats, Pablo Loyola, and Gregg Rothermel. Oracle-centric test case prioritization. In *IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 311–320, 2012.

[77] Boya Sun, Andy Podgurski, and Soumya Ray. Improving the precision of dependence-based defect mining by supervised learning of rule and violation graphs. In *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10. IEEE, 2010.

[78] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[79] Paolo Tonella, Paolo Avesani, and Angelo Susi. Using the case-based ranking methodology for test case prioritization. In *22nd IEEE International Conference on Software Maintenance*, pages 123–133. IEEE, 2006.

[80] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 1–12, July 2006.

[81] K. Walcott, G. Kapfhammer, R. Roos, and M. L. Soffa. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2006.

[82] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 140–150, July 2007.

[83] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation : A survey. *Software Testing, Verification, and Reliability*, March 2010.

[84] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[85] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335, 2003.

[86] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *35th International Conference on Software Engineering (ICSE)*, pages 192–201. IEEE, 2013.

[87] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering*, pages 531–540, 2008.

[88] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining versions histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, pages 563–572, May 2004.

[89] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 435–438. IEEE Computer Society, 2009.

APPENDIX. PAPER DETAILS

This section provides details of the literature survey. For ease of reading, the following abbreviations are used in the table.

1. Study Goals: TS = Test Case Selection, TR = Test Case Reduction, TP = Test Case Prioritization
2. Subject(s): I = Industry, O = Open Source, P = Private

Paper	Goal	Measurement	Subject	Data Source	Technique	Baseline
Elbaum 00 [23]	TP	APFD	O: Siemens Suite + Space	Coverage, Fault Exposing Potential	Greedy Coverage, Additional Coverage	Random, Optimal
Elbaum 01 [24]	TP	APFD (Including Fault Cost)	O: Space	Coverage, Test Cost, Fault Severity	Additional Coverage, Fault Index Prioritization	Random
Srivastava 02 [75]	TP	Similar to APFD	I: Large Microsoft Office Productivity Software	Churn, Coverage	Maximize Coverage	None (Tool Demonstration)
Elbaum 02 [25]	TP	APFD	O: Siemens Suite + Space, Grep, Flex, QTB	Coverage, Fault Probability	Order by Technique	Random, Optimal
Do 04 [22]	TP	APFD	O: Ant, XmlSec, JMeter, JTopas	Coverage	Greedy, Additional Coverage (On Total, Block and Method)	Initial, Random, Optimal
Tonella 06 [79]	TP	APFD	O: Space	User Manual Ranking Tuples	Machine Learning	Greedy, Additional Coverage, Random
Walcott 06 [80]	TP	APFD	O: Gradebook, JDepend	Execution Time, Coverage	Genetic Algorithm	Initial Ordering, Random
Li 07 [53]	TP	APFD	O: Siemens Suite + Space	Coverage	2-Optimal, Hill Climbing, Genetic Algorithm	Greedy Algorithms
Sherriff 07 [73]	TP	Percent Time Additional Selected Test Found Bug	I: Three Releases of IBM Internal Tool	Churn	Cluster Based on Churn	None (Discussed Comparison with Other Techniques)
Korel 08 [51]	TP	Similar to APFD	I: ATM, Cruise Control, Fuel Pump, TCP, ISDN, PrintTokens	Coverage, Model to Coverage Map	Order by Coverage	Random, Prioritize Tests Executing Marked Transition
Hou 08 [42]	TP	APFD	P: Travel Agent Web Service (Artificial Program)	Coverage	Linear Programming	Random, Total Branch, Additional Branch
Do 10 [19]	TP	Cost/Benefit	O: Ant, JMeter, XmlSec, NanoXml, Galileo	Coverage, Churn	Order by Coverage, Bayesian Network	Raw Cost/Benefit
Mei 12 [58]	TP	APFD	O: JTopas, XmlSec, JMeter, Ant	Static Coverage	Order by Coverage	Coverage of Seeded Faults and Mutants
Staats 12 [76]	TP	APFD	P: Altitude Switch, Wheel Brake System, Flight Guidance System (Simulink and Java)	Data Flow Analysis (Def-Use)	Order By Variables Checked	Random, Which Order Killed the Most Mutants Repeatedly
Zhang 13 [86]	TP	APFD	O: JTopas, XmlSecurity, JMeter, Ant	Coverage, Complexity, LOC	Coverage Factoring in Bug Probability	Coverage of Mutants
Haidry 13 [34]	TP	APFD, Time to Find All Faults	O: Elite, GSM, CRM, MET, Bash	Dependency Weight, Height	Order by Dependency Measures	Unordered, Random, Order by Coverage
Jones 03 [46]	TP, TR	Runtime, Reduction in Fault Detection	O: TCAS, Space	Coverage	Greedy, Greedy Additional	None
Rothermel 04 [70]	TP, TR, TS	APFD, Execution Time, Coverage	O: Emp-Server, Bash	Granularity, Churn, Coverage	Selection, Prioritization, Reduction	Test All
Leon 03 [52]	TP, TS	APFD	O: GCC, Jikes JavaC	Coverage	Greedy Approximation, Cluster Filtering, Failure Pursuit	Random
Beszedes 12 [9]	TP, TS	Recall	O: WebKit	Coverage, Churn	Order by Size, Coverage	Random
Hao 12 [37]	TR	Size by Acceptable Loss	O: Siemens Suite	Coverage	Linear Programming	HGS Algorithm
Harrold 01 [39]	TS	Coverage	O: Siemens Suite	Coverage	DejaVu, TestTube Tools	Random, Test All
Graves 01 [31]	TS	Precision, Time Savings	O: Siemens Suite + Space, Player	Coverage	Minimization, Dataflow, Safe	Random, Test All
Bible 01 [10]	TS	Precision, Time Savings	O: Siemens Suite + Space, Player	Coverage	Greedy Coverage	Test All
Kim 02 [48]	TS	Test Run, Fault Age	O: Siemens Suite + Space	Coverage, Time since Last Run	Coverage, with Boost for Time since Last Run	Random, Test All
Yoo 07 [82]	TS	Coverage, Fault Detection History, Execution Cost	O: Siemens Suite + Space	Coverage, Fault Detection History, Execution Cost	Pareto Genetic Algorithms	Additional Greedy

Continued on next page

Paper	Goal	Measurement	Subject(s)	Data Source(s)	Technique(s)	Baseline(s)
Jeffrey 07 [45]	TS	Size Reduction, Fault Loss	O: Siemens Suite + Space	Branch Coverage, All Uses Coverage	Chose Best Additional Coverage, Add Back	Randomly Added Back
Hemmati 10 [40]	TS	FDR	I: SUT (C++ Safety Monitoring Program)	Test Similarity	Choose Most Diverse	Random, Greedy, Coverage-Based Genetic Algorithm, STCS
Qu 11 [69]	TS	Percent of Impacted Functions	I: Make, ABB1 (Industrial Application)	Coverage	Determine Configuration from Coverage	Test All
Mirarab 12 [61]	TS	FDR, APFD	O: Ant, NanoXML, Galileo, XmlSec, JMeter	Coverage	Linear Programming	Coverage, Bayesian Network, Genetic Algorithm
Kim 12 [49]	TS	Precision, Recall	O: I2B2, GO (Gene Ontology)	Ontology Test Coverage	Order by Coverage	Test All
Gotlieb 14 [30]	TS	Processing Time, Optimum Result	P: Randomly Generate Test Cases, Requirements	Requirement to Test Mapping	Graph Search	Other Graph Search Algorithms and Linear Programming
Alshahwan 14 [3]	TS	Coverage, Spearman Rank to Fault Oracle	O: FAQForge, Schoolmate, Webchess, PHPSysInfo, Timeclock, PHPBB2 (all PHP)	Output Uniqueness	Order Maximizing Uniqueness	Test all