Predicting Regression Test Failures using Genetic Algorithm-Selected Dynamic Performance Analysis Metrics

M. Mayo and S. Spacey

Waikato University, Hamilton, New Zealand
mmayo@waikato.ac.nz
sspacey@waikato.ac.nz
WWW home page: http://cs.waikato.ac.nz/

Abstract. A novel framework for predicting regression test failures is proposed. The basic principle embodied in the framework is to use performance analysis tools to capture the runtime behaviour of a program as it executes each test in a regression suite. The performance information is then used to build a dynamically predictive model of test outcomes. Our framework is evaluated using a genetic algorithm for dynamic metric selection in combination with state-of-the-art machine learning classifiers. We show that if a program is modified and some tests subsequently fail, then it is possible to predict with considerable accuracy which of the remaining tests will also fail which can be used to help prioritise tests in time constrained testing environments.

Keywords: regression testing, test failure prediction, program analysis, machine learning, genetic metric selection

1 Introduction

Regression testing is a software engineering activity in which a suite of tests covering the expected behaviour of a software system are executed to verify a system's integrity after modification. As new features are added to a system, regression tests are re-run and outputs compared against expected results to ensure new feature code and system changes have not introduced bugs into old feature sets.

Ideally, we would like to run all regression tests as part of the normal development process when each new feature is committed. However, regression testing the large number of tests required to cover (an ever expanding) previous feature set can take considerable time. For example, the regression test suite [14, 15] used in Section 4 of this work takes approximately 12 hours to execute fully in our environment which makes on-line regression testing difficult.

Recently authors concerned with regression testing have began looking to performance analysis and machine learning to aid software engineering [1] and in this paper we propose a method that joins performance analysis [2], machine

learning [3] and genetic algorithms [4] to predict the outcome of unexecuted regression tests in a large regression test suite. A contribution of our work is the inclusion of a set of unique execution metrics measured from the dynamic execution path of the program to compliment the pass/fail and coverage metrics used in previous work [5]. Since the dynamic execution paths corresponding to different test inputs on the same program, to greater or lesser degrees, overlap, this information is useful for modelling the interrelationships between tests and therefore for predicting test outcomes as we will soon show.

One problem with modelling interactions based on execution paths is that even small programs can have a very high execution path trace length and therefore there can be an extremely large number of dynamic metrics describing even a simple test's execution [6]. We solve this problem by using dynamic analysis tools to compress the program's execution trace information into a set of key metrics that we consider could be important determiners and then we use a genetic algorithm to select the best subset of these metrics for a final predictive model as detailed later in this paper.

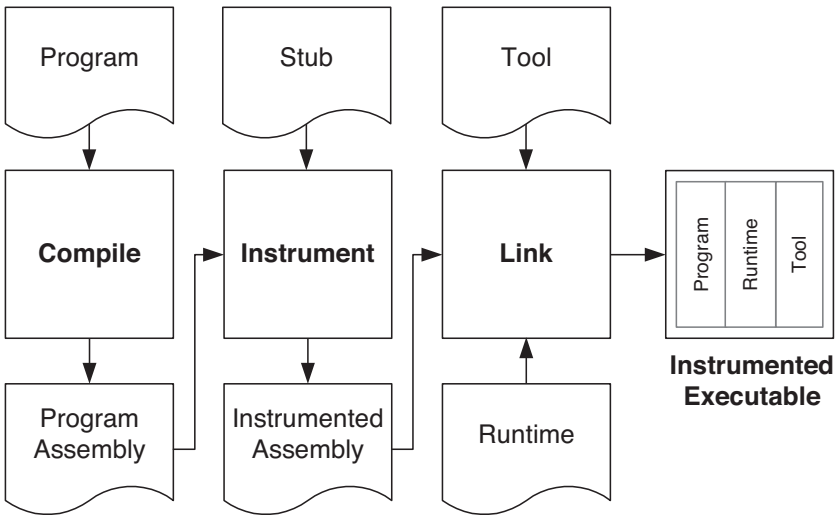
Our results show that it is indeed possible to predict, with high accuracy, which future regression tests will fail. Additionally we present results showing which of our measurement metrics has the greatest impact on model prediction accuracy. For software engineers, the results presented in this paper demonstrate that the proposed approach could be useful for either ranking tests (e.g. in order to execute those most likely to fail first) or for skipping tests (e.g. in order to avoid executing both tests in a pair if the test outcomes have high correlation). For dynamic instrumentation tool makers and quality assurance professionals the results indicate the value of key performance analysis metrics and could help focus future research in dynamic measurement tool development.

2 Dynamic Performance Analysis

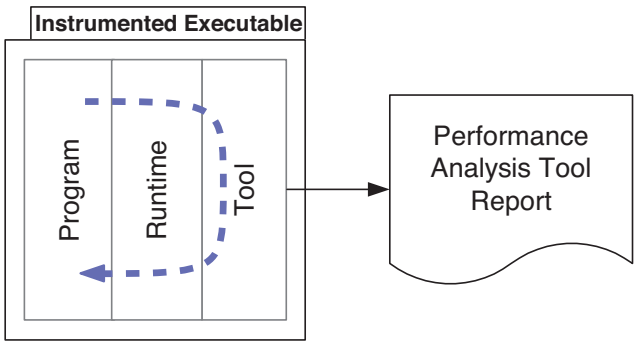
Before we introduce our algorithm to predict regression test correlations, we first present an overview of the Open Performance Analysis Toolkit (OpenPAT) [2] which provides the dynamic execution path measurements we need. OpenPAT [2] is an open source performance analysis toolkit that analyses program execution paths.

OpenPAT is derived from 3S [7,8] and, like its predecessor, OpenPAT instruments programs by inserting measurement stubs at critical points in a program's assembly code as illustrated in Figure 1(a). At runtime, the stubs back-up the main program's state and measure characteristics such as timing information which they pass on to one or more analysis tools for consolidation and later reporting as illustrated in Figure 1(b).

The approach of static assembly instrumentation and dynamic analysis can be used with any program that compiles to assembly [7,8] and combines the low execution overhead advantages of traditional one time instrumentation toolkits such as SUIF [9] with the dynamic performance measurement accuracy advantages of modern frameworks such as Gilk [10], Valgrind [11] and Pin [12].



(a) Source assembly is statically instrumented with stubs at critical points.



(b) Stubs pass dynamic measurements (dotted line) on to analysis tools at runtime.

Fig. 1. The Open Performance Analysis Toolkit (OpenPAT) approach of combining static instrumentation with dynamic analysis.

OpenPAT extends the 3S meaning of a "program that compiles to assembly" to include programs that compile to assembly bytecodes running on a virtual machine [2] such as .NET and Java so that the same tools can be used to analyse programs targeted for native and interpreted environments. In addition to allowing the same tools to gather runtime measurements for a wide range of languages, OpenPAT adds several other features that can be of value for regression testing including test case code coverage metrics, ranged instrumentation and assembly block to source code correspondence information [2].

OpenPAT comes with a number of analysis tools that provide unique dynamic and structural analysis metrics. We will concentrate on the metrics provided by just one of these tools, the OpenPAT version 3.0.0 `hotspot` tool discussed further in Section 3.2, which will be sufficient for demonstrating that we can provide high regression test prediction accuracy with our approach of combining dynamic performance metrics with machine learning and genetic algorithms which we describe next.

3 Approach to Predicting Regression Test Failures

We now describe the framework we have constructed for building predictive models of regression test outcomes based on dynamic execution measurements. We begin with an overview of the framework followed by a subsection describing the role of evolutionary search in our approach.

3.1 Framework Overview

Our framework for predicting test failures and ranking tests is explained with reference to Figure 2. The hypothesis underlying our framework is that dynamic execution metrics measured for a correct execution of a program contain characteristics that describe how the program needs to operate internally in order to produce valid outputs and that machine learning can use this information to better discern the relationships between different tests than simply relying on the test code coverage intersection metrics of previous work [5].

The basic framework process is to run a suite of initial regression tests on a program at a time when the program is known to be correct and to save the correct test results for future comparison as in an unguided traditional testing approach. At the same time, we also record dynamic and structural metrics for the program's execution for every test case using OpenPAT. These additional OpenPAT metrics are referred to as "per test metrics" in Figure 2.

After the program has been modified we re-run the same tests. If the modification has introduced a bug in previous tested features one or more tests will fail. As each test completes, our framework takes the set of tests that have already been executed (where the executed tests are labelled with either "P" or "F" for pass or fail respectively), combines them with the dynamic metrics measured for the correct version of the program, and constructs a table of labelled examples suitable for machine learning.

The table consists of one row for each test that has been executed with columns corresponding to the metrics measured by OpenPAT for the original "correct" version of the program and the known pass/fail results from the "incorrect" version. The table is used to build a dynamically predictive model of test failure as shown in Figure 2. This model in turn is used to label all of the remaining outstanding tests with a failure probability based on the original OpenPAT measurements that can be used to rank tests.

With the test failure rank predictions, a developer is equipped to decide whether to:

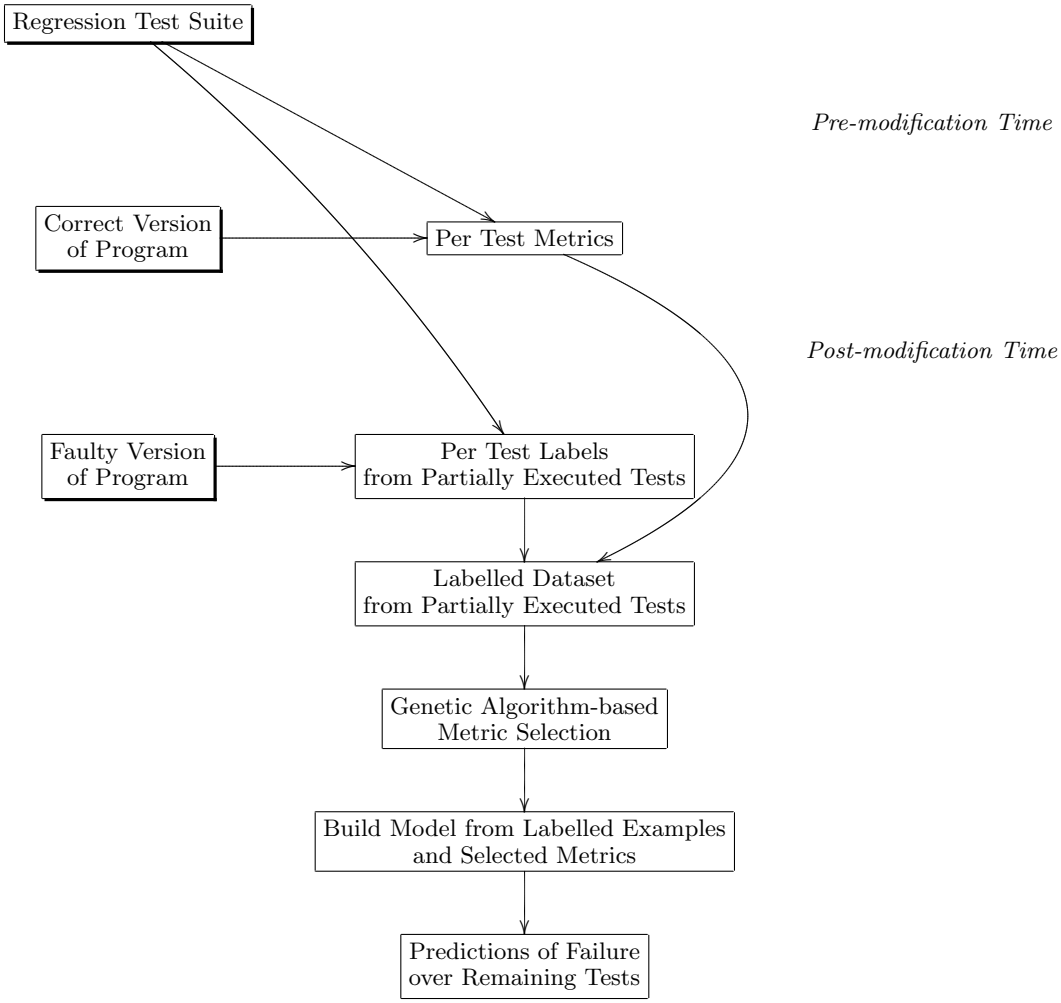


Fig. 2. Flow of information in the test failure prediction system. The inputs are the regression test suite and two versions of the program, one correct and one faulty.

Metric	Class	Description
`source_file`	Structural	Source file where a code section is located
`source_line`	Structural	Line in the source file being executed
`instructions`	Structural	Number of assembly instructions in a code section
`order`	Dynamic	Order in which code sections were first executed
`entries`	Dynamic	Number of times a code section was executed
`ticks_min`	Dynamic	Minimum CPU cycles a code section took to execute
`ticks_max`	Dynamic	Maximum CPU cycles a code section took to execute
`ticks_sum`	Dynamic	Total CPU cycles to execute a code section

Table 1. Structural and dynamic metrics measured by our version of the OpenPAT `hotspot` tool. We added the `ticks_min` and `ticks_max` timing metrics to the basic `hotspot` tool provided in the OpenPAT version 3.0.0 pre-release 1 distribution our selves by simply inserting min and max counters in the `_OP_MEASUREMENT_T` and per entrance updates in the OpenPAT version 3.0.0 `hotspot` `_OP_TOOL_ANALYSE` function.

1. execute the remaining tests that are most likely to fail first in order to provide additional information to support their debug process or
2. execute the remaining tests that are *not* likely to fail first to identify if unrelated features are affected or
3. execute the remaining tests that the learning classifier is not sure about (probability of failure around 50%) in order to strengthen the prediction probabilities for the remaining tests, which requires retraining the model.

The exact decision will be developer, context and business process dependent, but the decision is supported by our algorithm's predictions which we need to be of high quality to be of any practical value.

3.2 Program Test Metrics

The OpenPAT toolkit includes a wide range of metrics available at a fine-grained level for each code section aka "assembly basic block", in the program. Because a typical program can have a large number of code sections [22], we can quickly obtain an extremely high dimensionality (that being code sections times metrics per section) in the dataset. Unfortunately, many machine learning algorithms do not perform well when data dimensionality is high compared to the number of labelled examples used for training [21]. Therefore it is desirable to reduce the data dimensionality somehow and we discuss our use of a genetic algorithm for metric selection later in this work. First however, we describe the specific OpenPAT metrics that we used to measure each program's execution per-test.

The OpenPAT 3.0.0 `hotspot` tool provides a number of useful program analysis metrics that we use for machine learning. These metrics fall broadly into two categories: structural metrics and dynamic metrics. These are described in Table 1.

The `source_file` and `source_line` structural measures of the `hotspot` tool together with the dynamic `entries` measure give, in effect, test coverage infor-

mation. Specifically, each code section that is executed will have an `entries` figure of at least one and as the static metrics map the number of `entries` back to specific code lines we know that every code section that is executed by a test will have `entries` ≥ 1 and 0 otherwise.

The structural `instructions` metric can be considered a measure of source line complexity. Code that is more complex such as long formulas will, in general, expand to more assembly instructions than simpler code and so the number of assembly instructions associated with a code section can be an important indicator of potential logical issue points.

The dynamic `order` metric is a program execution path indicator. Program code sections that execute earlier in a program's execution path are assigned an earlier execution order by OpenPAT. For example, if a program consists of three code sections A, B and C, and A is executed first, then B executes in a loop 100 times followed finally by C, the OpenPAT execution order for the three blocks would be 1, 2 and 3 respectively (with B having an `entries` figure of 100 because of the loop). While it is true that a program's execution path can be test input data dependent, the internal path of tests is expected to provide information about dependent chains of sub-feature tests through predictable sub-path patterns. Thus (a possibly shifted) `order` metric chain can be useful in identifying test dependencies in addition to the traditional measure of the structural intersection of covered lines which is also identified by the `hotspot` tool as explained above.

The dynamic `ticks_min`, `ticks_max` and `ticks_sum` metrics provide actual CPU execution cycle measurements for the blocks of code executing in a program. The `ticks_min` and `ticks_max` can be used to ascertain information about cache and data access patterns in a program's execution. For example, if code section B executes repeatedly in a loop on a single set of data obtained from main memory, then the second time B executes it could take considerably less time than the first if the data being operated on is still in the CPU's cache. As another example, if the same data is used by A and B and A executes before B then B's `ticks_max` figure could be close to its `ticks_min` figure because the overhead of initial cache loading was suffered by A. The `ticks_sum` metric is the total time a code section takes to execute and can compliment the `instructions` and `entries` figures in providing an execution time dimension to the computational complexity for a line of source code.

While the eight metrics of Table 1 are expected to provide useful information for test case outcome correlation prediction for the reasons outlined above, some can be expected to be of more predictive importance than others. Thus we will evaluate the importance of the different metrics as predictors of regression test set outcomes in Section 4.5 using a genetic search to find the best subset of the metrics to train a machine learning classifier with.

Program Name	Code Lines	Code Sections	#Faulty Versions	#Tests	#Failing Tests
print_tokens	472	315	7	4,072	69
print_tokens2	399	286	10	4,057	224
replace	512	442	31	5,542	107
schedule	292	244	8	2,638	96
schedule2	301	264	9	2,638	33
tcas	141	150	41	1,592	39
tot_info	440	259	23	1,026	84

Table 2. Siemens HR Variant v 2.2 [14, 15] programs and their test suites. The Code Lines column is the source lines excluding comments and blank lines in the program, the Code Sections column refers to OpenPAT version 3.0.0 compiled program basic blocks, #Faulty Versions is the number of faulty versions of the programs supplied, #Tests is the number of feature tests in the benchmark suite and #Failing Tests is the average number of failing feature tests for each of the faulty program versions.

4 Evaluation

In this section we describe the actual implementation of our framework concept and provide comprehensive practical evaluation results.

4.1 Programs and Regression Test Suites

In order to evaluate our method, we need one or more programs, each with a corresponding suite of regression tests. We require correct and faulty versions of the programs so that pass and fail test results can be used to test the predictive power of our approach. To this end, we used the public benchmark of faulty programs originally developed by Siemens and discussed in [14, 15].

Table 2 describes the program suite. There are in total seven different programs, all written in the C programming language, each program accompanied by over a thousand feature tests. Each program also comes with a varying number of faulty, buggy variants.

The figures in Table 2 reflect some minor adjustments we made to the original dataset. In particular, there are three faulty versions that fail no tests. This leaves 129 buggy variants of the seven programs in total. Many of the test suites also have a small number of duplicate tests, and the number of tests in the table reflects the test suite sizes after removal of these duplicates.

4.2 Metric Volume and Balance for the Benchmarks

We took measurements for the correct versions of each program as they executed every test. These measurements were used to compute the metrics for our datasets.

We then compiled each faulty program and re-ran the tests again, this time to determine which tests would pass or fail as a consequence of the bugs injected

into the faulty versions. These pass/fail outcomes became the ground truth labels for our datasets. Note that there are 129 datasets in total, one for each faulty program version, with the number of instances in each dataset being equal to the number of (non-duplicate) tests in the corresponding program's test suite.

The dynamic performance execution measurements that we used were acquired using OpenPAT's `hotspot` tool. Specifically, we measured the eight metrics of Table 1 for each code section in the program as described in Section 3.2. As the number of code sections in the benchmark set ranged from 150 for `tcas` to 442 for `replace`, the total metric measurements for each program test case ranged from 1,262 to 3,653. Such a large number of metrics can cause issues for machine learning algorithms [21] as introduced in Section 3.2 and is the justification for our inclusion of Genetic Algorithms in this work which we discuss further in Section 4.4.

In addition to the large number of metrics caused by the fine measurement granularity available in OpenPAT, there was also quite a large degree of class imbalance in the datasets as the number of failing tests for the faulty benchmarks of Table 2 is only a small proportion of the total number of regression tests for each program. This class imbalance has severe impact on how any predictive modelling scheme can be evaluated as discussed in the next section.

4.3 Prediction Quality Assessment

In order to evaluate the effectiveness of our method, it was necessary to decide on a scheme for measuring and comparing regression test outcome predictions made by different implementations of our basic approach. The simplest measure, prediction accuracy, is not ideal for the Siemens test suite [14, 15] because there are only a small percentage of failing tests for each program as shown in Table 2. In fact the average number of failing tests is only 3% for the programs of Table 2, so even a naïve prediction scheme that simply classified all tests as passing would already achieve an average accuracy of around 97% for the regression test suite.

An alternative, and the prediction quality metric we use in this paper, is to report Area Under the Curve (AUC) [16], a different machine learning performance metric that focusses on the trade-off between true positives and false positives as the classification threshold changes. AUC reports a number between 0.5 (for a random classifier) and 1.0 (for a perfect classifier). The advantage of this metric is that it is not sensitive to class imbalance, and therefore a classifier predicting that all tests pass will achieve the worst possible AUC of 0.5.

4.4 Predictive Algorithm Selection

In total, we evaluated nine different combinations of test class (pass/fail) prediction algorithms on our datasets, as detailed in Table 3. The prediction algorithms consist of three machine learning approaches: Naive Bayes [18], a simple bayesian classifier that assumes conditional independence of metrics; Sequential Minimal Optimization [19], a linear support vector machine classifier; and Random Forests [20], a state-of-the-art method based on an ensemble of trees.

Class Prediction Algorithm		Metric Selection Algorithm
`NB:` Naive Bayes		`ALL` eight `hotspot` metrics or
`SMO:` Sequential Minimal Opt.	×	`GA:` metrics selected by a Genetic Algorithm or
`RF:` Random Forests		`CV:` Coverage only metrics

Table 3. The nine algorithmic combinations used to predict regression test outcomes. Each of the machine learning algorithms has three forms: one with all eight OpenPAT dynamic metrics as input, a second with the metrics pre-selected by a Genetic Algorithm and the third with coverage-only metrics.

We used the implementations of these algorithms from Weka 3.7.7 [3] with all default settings, except for the Random Forests algorithm that had its number of trees set to 100.

To allow us to evaluate the relative importance of different dynamic measurement metrics on regression test prediction quality we used three versions of each dataset for each of the three machine learning approaches: the first dataset version comprised all of the OpenPAT metric measurement information; the second version comprised a subset of the metric information; and the third supplied only the coverage information from Table 1. To select the metric subset for the second variant of the datasets we used a Simple Genetic Algorithm [4] with a fitness function that rewards correlation of the metrics with the test outcomes (i.e. pass or fail) while explicitly penalising redundancy between metrics as described in Hall [13]. Our genetic algorithm (GA) was again a Weka 3.7.7 implementation and was a simple binary GA with "1" on a chromosome to indicate the presence of a particular OpenPAT metric or coverage feature and a "0" to indicate its absence. The GA was executed for 100 generations with all other default settings, and it was applied only to the training split of the dataset as described below.

We trained and tested each combination of class prediction and metric selection algorithm with randomly selected subsets of the regression tests for each of the faulty versions of the benchmark programs. In all cases, the selected training tests comprised 25% of the total tests (simulating 25% of the regression suite being executed), and the remainder of the tests (i.e. 75%) were used for evaluating the model's predictive power.

We performed ten randomised training and predictive power assessment runs for each of the nine algorithmic combinations of Table 3 and the 129 faulty program versions of our test suite yielding a total of $129 \times 10 \times 9 = 11,610$ individual experimental runs. Prediction quality figures were computed to assess the quality of each experimental run as discussed next.

4.5 Results

The average Area Under the Curve (AUC) performance quality results of our nine experimental runs across the 129 faulty program versions and nine algorithm combinations are presented in Table 4 below. The best performing classifier

	Naive Bayes			Seq. Min. Opt.			Random Forests		
program	ALL	GA	CV	ALL	GA	CV	ALL	GA	CV
print_tokens	0.614	0.677	0.500	0.793	0.807	0.761	0.866	**0.872**	0.810
print_tokens2	0.820	0.858	0.500	0.934	0.937	0.931	**0.968**	0.967	0.936
replace	0.726	0.812	0.500	0.917	0.914	0.884	0.913	**0.916**	0.877
schedule	0.644	0.718	0.500	0.817	0.823	0.789	0.831	**0.843**	0.791
schedule2	0.613	0.714	0.500	0.856	0.851	0.778	**0.879**	0.872	0.845
tcas	0.820	0.857	0.500	0.868	0.864	0.734	0.860	0.857	**0.883**
tot_info	0.765	0.823	0.500	0.925	0.931	0.879	**0.944**	**0.944**	0.887

Table 4. Average Area Under the Curve (AUC) prediction quality results by program and algorithmic classifier. Each column provides results for one of the machine learning classifiers used with a subset of OpenPAT metric data from Table 3. For example the first three columns give the results for the Naive Bayes machine learning classifier used with all the OpenPAT metrics, a subset of the metrics selected by a Genetic Algorithm (GA) and just the OpenPAT coverage metrics (CV) respectively as discussed in Section 4.4.

algorithm for each program has been emphasised in the tables for the reader's convenience.

From the table, three observations can be made. Firstly, the Random Forests algorithm is clearly the best performing classification method. The linear support vector machine classifier Sequential Minimal Optimization frequently comes a close second, but overall it is unable to improve on Random Forests. The Naive Bayes classifier is universally the worst classifier. It is also apparent that the best algorithmic classifier's predictive abilities are always much more accurate than random guessing.

Secondly, the inclusion of the OpenPAT metrics improves performance in most cases compared to using simple code coverage. This is most obvious in the case of Naive Bayes, where code coverage (i.e. CV) features alone are insufficient to train the model at all, as demonstrated by the AUC measures being at 0.5 which indicates no predictive power.

The third observation is that for four of seven benchmarks, the best prediction quality was achieved using a Genetic Algorithm to select a subset of the OpenPAT metrics so as not to "overwhelm" the machine learning algorithm with all the OpenPAT metrics as discussed in Section 4.2. In fact, considering each of the machine learning algorithms in isolation, we see that the GA variant of the classification algorithm gives the best results for:

1. all seven of the benchmarks using the Naive Bayes algorithm
2. four of the seven benchmarks using Sequential Minimal Optimization and
3. four of the seven benchmarks using Random Forests

which indicates the value of the GA metric sub-selector in these tests.

In order to examine the performance of the GA more closely, we looked at the OpenPAT metrics selected by it during all runs for Naive Bayes over all 129 datasets. The results are shown in Table 5, and they give the probability of

Metric	Class	GA Selection Prob.
source_file	Structural	29%
source_line	Structural	29%
instructions	Structural	29%
order	Dynamic	30%
entries	Dynamic	31%
ticks_min	Dynamic	29%
ticks_max	Dynamic	29%
ticks_sum	Dynamic	31%

Table 5. Importance of the OpenPAT `hotspot` tool metrics as indicated by the probability that each metric was selected by the Genetic Algorithm for the Naive Bayes machine learning algorithm.

each metric being selected by the GA, averaged over faulty version and run. The results show that while the different metrics are selected fairly uniformly, there is a slight bias towards `ticks_sum`, `entries` and `order`. It is also valuable to note that while the GA only selects 30% of the available attributes for use with the Naive Bayes algorithm, the predictive quality of the GA variant is better than the Naive Bayes approach using all the available metrics as shown by the `ALL` column of Table 4 for every benchmark considered here.

As previously indicated, the genetic algorithm allows the Naive Bayes classifier to perform better with larger initial metric volumes (number of metrics used times program code sections the metrics are measured for) than would otherwise be possible. While the `GA` metric subset selection improvement was less pronounced for the other machine learning algorithms for these small benchmarks, the Genetic Algorithm improvements are already valuable for these benchmarks and are expected to become more pronounced for larger commercial and open source programs as the metric volume increases [22].

5 Conclusion

We have presented a framework for using dynamic execution measurements taken during the regression testing of correct versions of a program for predicting future regression test failures using genetic search and machine learning algorithms. Our experiments demonstrate that combining dynamic performance metric information with machine learning and genetic algorithms can provide improved test result accuracy predictions over approaches that use test code coverage metric intersections alone. This increased prediction accuracy could lead to reductions in regression testing time and allow regression testing to be more frequently applied to feature modifications to support on-line software quality assessment.

While we restricted our experiments to the well known Siemens test suite [14, 15] and eight OpenPAT metrics [2] in this paper, the approach as presented is directly applicable to larger software programs and additional dynamic program

analysis metrics. Future work may consider incorporating dynamic metric information gathered during testing (not just prior information gathered for the correct program version) into the method, adding new dynamic measurement metrics from OpenPAT including for example detailed internal control flow information, evaluating the accuracy of the approach with different training sets sizes, different prediction quality metrics, and different code section sizes, and evaluating the benefits of the GA metric selection feature with different machine learning algorithms on larger commercial and open-source programs [22].

References

1. A.V. Nori and S.K. Rajamani (2011). *Program Analysis and Machine Learning: A Win-Win Deal.* Microsoft Research India, In *Proc. 18th International Static Analysis Symposium (SAS)*, pp. 2-3.
2. The OpenPAT Project. The Open Performance Analysis Toolkit. http://www.OpenPAT.org [Online; accessed 20-March-2013].
3. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I.H. Witten (2009). The WEKA data mining software: An update, *SIGKDD Explorations*, 11(1), pp. 10-18.
4. D.E. Goldberg (1989). *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley.
5. M. Harman, P. McMinn, J. Teixeira de Souza and Shin Yoo (2012). Search Based Software Engineering: Techniques, Taxonomy, Tutorial. *Empirical Software Engineering and Verification*, (7007), pp. 1-59.
6. S. Spacey, W. Wiesmann, D. Kuhn and W. Luk (2012). Robust software partitioning with multiple instantiation. *INFORMS Journal on Computing* 24(3) pp. 500-515.
7. S. Spacey (2006). 3S: Program instrumentation and characterisation framework. Technical Paper, Imperial College London.
8. S. Spacey (2009). 3S Quick Start Guide. Technical Manual, Imperial College London.
9. G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy and C. Sapuntzakis (2000). An overview of the SUIF2 compiler infrastructure. Technical Paper, Stanford University.
10. D.J. Pearce, P.H.J. Kelly, T. Field and U. Harder (2002). GILK: A dynamic instrumentation tool for the Linux kernel. In *Proc. of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools* 37, pp. 220-226.
11. N. Nethercote and J. Seward (2003). Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2) pp. 44-66.
12. C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi and K. Hazelwood (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 190-200.
13. M.A. Hall (1998). *Correlation-based Feature Subset Selection for Machine Learning.* Ph.D. Thesis, University of Waikato, Hamilton, New Zealand.
14. Siemens, HR Variants v 2.2. http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/.

15. M. Hutchins, H. Foster, T. Goradia, T. Ostrand (1994). Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th International Conference on Software Engineering*, pp. 191-200.
16. T. Fawcett (2006). An introduction to ROC analysis, *Pattern Recognition Letters*, 27, pp. 861-874.
17. S. Yoo. (2012). Evolving human competitive spectra-based fault localization techniques. In G. Fraser (Ed.) *Proc SSBSE 2012*, LNCS 7515, pp. 244-258.
18. G.H. John and P. Langley (1995). Estimating Continuous Distributions in Bayesian Classifiers. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. pp. 338-345. Morgan Kaufmann, San Mateo.
19. J.C. Platt (1998). Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods – Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, Eds. MIT Press.
20. L. Breiman (2001). Random Forests. *Machine Learning* 45(1) pp. 5-32.
21. P. Domingos (2012). A Few Useful Things to Know about Machine Learning. *Communications of the ACM*, 55 (10), pp. 78-87.
22. S. Spacey, W. Luk, D. Kuhn and P.H.J. Kelly (2013). Parallel Partitioning for Distributed Systems using Sequential Assignment. *Journal of Parallel and Distributed Computing* 73(2), pp. 207-219.