Striving for Failure: An Industrial Case Study About Test Failure Prediction

Jeff Anderson *
* Microsoft
North Dakota State University
jeffrey.r.anderson@ndsu.edu

Saeed Salem †
† North Dakota State University
Computer Science Dept.
saeed.salem@ndsu.edu

Hyunsook Do ‡
‡ North Dakota State University
Computer Science Dept.
hyunsook.do@ndsu.edu

***Abstract*—Software regression testing is an important, yet very costly, part of most major software projects. When regression tests run, any failures that are found help catch bugs early and smooth the future development work. The act of executing large numbers of tests takes significant resources that could, otherwise, be applied elsewhere. If tests could be accurately classified as likely to pass or fail prior to the run, it could save significant time while maintaining the benefits of early bug detection. In this paper, we present a case study to build a classifier for regression tests based on industrial software, Microsoft Dynamics AX. In this study, we examine the effectiveness of this classification as well as which aspects of the software are the most important in predicting regression test failures.**

***Index Terms*—Test failure prediction, data-mining software repositories, regression testing, case study**

I. Introduction

Most software projects help ensure quality with regression testing integrated throughout the development cycle [1]. Regression testing provides many benefits, from reducing the number of defects to allowing automated quality checks after refactoring and other feature works. These quality benefits are offset by the, often, high cost of performing full regression suite runs. For large projects, these regression test suites may contain many thousands of tests and may take hundreds to thousands of hours to execute. During these runs, only a small percentage of the tests actually fail, and generally, it is only the failures that are interesting because they indicate potential quality issues.

To date, much research has been done while attempting to predict defects in software based on various traditional software attributes, such as software changes, complexity, and the number of lines of code [2], [3], [4], [5]. Recent work has utilized more sophisticated information, such as dependency graphs, defect history, or dynamic performance data, to improve failure prediction [6], [7], [8]. Fault prediction by itself, however, does not ensure software quality or savings. In order to save development costs or achieve higher project quality, the fault prediction must be properly analyzed so that software engineers can apply the scarce testing resources to the correct product area. To achieve the proper application of fault prediction, some researchers have investigated how to maximize benefits in scarce environments [9].

Despite abundant study in this area, the existing research has only considered a limited number of attributes rather than the broader set of available attributes that can allow us to apply a holistic approach. Each piece of research shows that individual attributes can help with test failure prediction, but the studies do not delve into the interplay between those attributes and the software project itself. More recently, researchers have begun applying data mining techniques to test the failure prediction. In our prior work on an industrial product, we used a relatively naive approach that only considered historical pass and fail information from tests and then combined the information with concepts such as test result recency to show that data mining of multiple attributes can improve test case prediction [10].

While our previous research was shown to be promising, it, too, 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], [12]. These techniques have recently been applied to software engineering areas, such as bug triage and predicting bug bounces [13], [14]. 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.

The rest of the paper is organized as follows. Section II describes the Approach used in this research. Sections III and IV present our case study design, the results of the study, and data analysis. Section V discusses the impact of our results on software projects. Section VI presents Related Work. Finally, Section VII presents Conclusions and Future Work.

II. Approach

In this section, we describe the approach used in training, applying, and analyzing the effectiveness of the classifiers. This research was done in the context of the *Microsoft Dynamics AX 2012 R2* release, but the same approaches could be applied to other software projects. We plan to more broadly apply these techniques in future research.

Figure 1 illustrates a standard build and regression test cadence employed with many software projects, including the *Microsoft Dynamics AX 2012 R2* release. Changes are checked into the source control repository 24 hours a day, 365 days a year because the team works from various countries around the world. A nightly build is then performed and contains all the changes until the time when the build was started. Based upon hardware availability, a full regression test run is started about once every three days, using the most recent build available at that time.

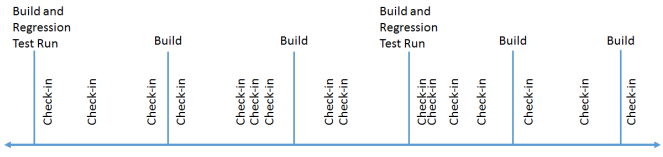

Fig. 1. Regression Test Cadence

This build-and-test cadence yields a large amount of data that can then be used as input when building classifiers. Figure 2 illustrates the set of 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 aspect is complexity. Complexity measurements are more traditional, such as 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.

Classification models take a set of training instances and learn a classification model [15]. The classification model can then be applied to a new set of input attributes to make predictions as shown in Figure 3. 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 [16]. 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.

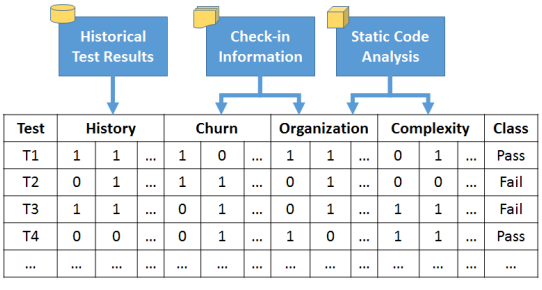

Fig. 2. Constructing Attributes for Tests

The result of this prediction is compared against which tests actually failed for test run 15 to determine the prediction's effectiveness.

Classification Algorithms: Many classification algorithms exist. This research is not intended to exhaustively compare classification algorithms because many previous studies have done this comparison [17], [18], [19]. Rather, this study seeks to understand how the attributes apply to predicting failures for a software project and what those attributes explain about the software project itself.

Some models, such as tree models, are human readable. These models are represented as decision trees where each node of the tree corresponds to an attribute value and determines which tree branch should be used. In these models, the closer an attribute is to the root of the tree, the more important that attribute is in the classification. Because this research seeks to understand attribute importance, tree models are helpful. The RepTree algorithm is used as the decision tree learning algorithm throughout this work [20].

Other classification algorithms are more mathematical. One example is the Naive Bayes classifier which makes an assumption that all available attributes are independent in nature and assigns weightings to each attribute which, mathematically, can then be used to compute the class. While this assumption is not accurate for the attributes used in this study, research has shown that the Naive Bayes classification method approaches or surpasses other classification algorithms [21].

Mathematical classification algorithms are more difficult for humans to read and are, therefore, more difficult to use in this research. However, the results of using these classification methods, specifically Naive Bayes, are included in the study to ensure that classifier selection is not skewing the results.

III. Empirical Study

As stated in Section I, 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?

A. Object of Analysis

In this study, we used the *Microsoft Dynamics AX 2012 R2* product release, the same product discussed in our previous research [10]. This release contains hundreds of new features beyond the original *Microsoft Dynamics AX 2012* release, many of which were regional regulatory features [22].

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.

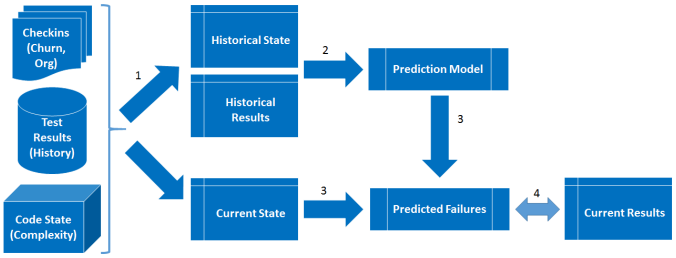

Fig. 3. Experimental Process

B. Variables and Measures

1) Independent Variables: This study manipulated two independent variables: input attribute and classification method.

Variable 1. Input Attribute: We considered four categories of input attributes as we described in Section II (Figure 2): 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 due to a failure in the last run

Note that we examined several number of runs (2, 5, 10, 15, and 20), but the manual inspection showed very little difference in predictive abilities across this range. Thus, we present the experiment results with 10 runs due to space limitations.

Organization attributes:

1) Primary Editing Team: Which product team made the most changes in this test run
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 the 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.

Variable 2. 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.

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.

Precision is defined as the ratio for the number of true positives to the total number of predicted failures. If TP is the set of true positives and FP is the set of false positives, then precision is defined as follows.

$$Precision = \frac{|TP|}{|TP| + |FP|}$$

Recall is the associated measure for the ratio of true positives to the total number of actual failures. Again, if TP is the set of true positives and FN is the set of false negatives, then recall is defined as follows.

$$Recall = \frac{|TP|}{|TP| + |FN|}$$

Recall can be artificially increased by randomly including more tests in the predicted set of failures, but this inclusion would reduce precision. Similarly, precision can be artificially increased by removing any predictions without extremely high confidence of failure. This trade-off between the two measures makes direct comparison of varying approaches difficult. Therefore, we combine the two measures to compute the harmonic mean, also known as the F-measure. A higher F-measure indicates that precision or recall has been increased without negatively impacting the other measure. The F-measure is the primary measurement used for comparison in this study and is defined as follows.

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

3) Experimental Process: To apply our approach, we need to collect the input attributes explained in Section III-B1. 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 3.

When investigating regression test suite run n, the first step is to collect the historical data from all regression test suite runs, 1 to $n - 1$. These measures are collected for each individual regression test that was executed during each historical suite run. For instance, if we are evaluating suite run 9, then each test from run 8 will have individual measures of associated code churn, organization, complexity, and other measures as they existed during test run 8. The same will be done for test run 7, etc. All these measures are combined with the test cases' pass and failure results during those previous runs to create the historical states and results as shown in Figure 3.

The second step is to run these historical states and results through a classification algorithm to develop a prediction model. Many classifiers exist, and for this study, the RepTree classifier was primarily used. Information about how this classifier was selected is discussed later. The classification model is a set of rules that can be applied to a given test case's attribute to yield a prediction of either pass or fail.

The third step is to apply this classification model to the current values of the input attributes for each test case in regression test suite run 9. The classification model uses all the attributes to generate a pass or fail prediction for each test. The set of predicted successes are ignored, because the goal of this research is to predict failures. The vast majority of tests do pass in each run, so the F-measure would appear artificially high if successes were also included in the results.

The fourth and final step is to compare the set of predicted failures with the set of tests that actually failed in that regression test suite run. The precision, recall, and F-measure are computed for this regression test suite run and are averaged with the F-measure from the other 63 regression test suite runs during the project to compute the average effectiveness of prediction using each set of attributes.

It is important to note that the prediction model for each regression test suite run varies slightly, because the model is re-generated based on all available historical information prior to the current run. While the model remains relatively stable, there are differences over time. The goal of this research is to examine the attributes' effectiveness in generating prediction models, not the effectiveness of any given model by itself.

One of the problems with many attributes used in the classification is a bias toward test success. With any given test run, roughly 90 percent of all the tests pass. This situation means that any attribute by itself will likely always predict pass, leading to meaningless results. To remove this bias, we used the standard technique of balancing [23] the number of test passes and fails in the training data. With a given set of training data, if 5,000 tests had failed and 45,000 had passed, we would include the 5,000 failures in the training set and only use 1 of every 9 passing results in the training set.

IV. Data and Analysis

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

A. Overall Results

TABLE I
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 I shows the average F-measure for the failure predictions resulting from each possible 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.

B. 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 I. 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.

C. 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 4 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 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, 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.

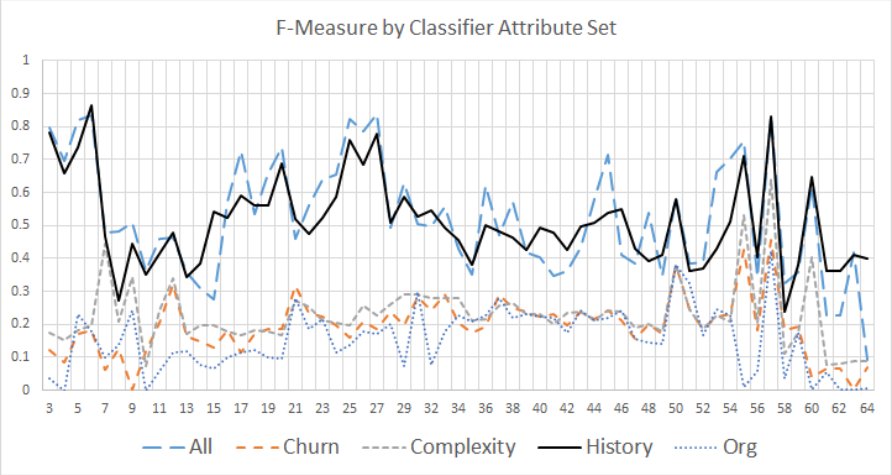

Fig. 4. Overall F-Measure by Attribute Category

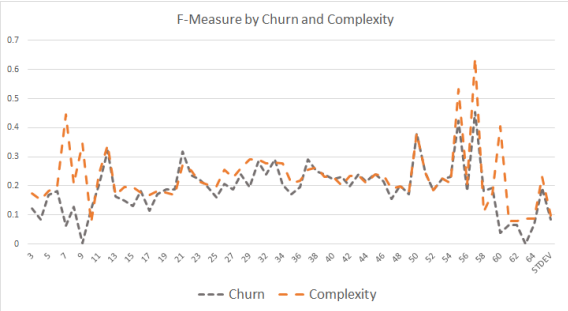

Fig. 5. F-Measure of Churn vs Complexity

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 4. 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 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.

D. Classification Models

The other interesting way of examining predictive power over time is to look at the classification models directly. The models vary for each run, so in this example, we look at the model for all attributes from 3/4 of the way through the project and the model at the end of the project.

Figure 6 shows the RepTree model with a maximum of 5 levels for regression test run 64. The first thing to notice is that all the root predictions in this model are based on historical information, things such as whether the test passed in the most recent run and how many times it has recently passed or failed. Only near the bottom of the tree does churn- and complexity-based information become useful. For instance, if the test passed in the last run but has failed at least once in the last 10 runs, then the number of changes is used with a relatively high threshold (2,238) in combination with the count

```
LastRunStatus = Pass
|   FailsInLast10Runs = 0
|   |   CheckinFileCountSinceLastRun = 0
|   |   |   BugsFoundLast10Runs = 0
|   |   |   |   TotalReferences <= 20 : Pass
|   |   |   |   TotalReferences > 20 : Fail
|   |   |   BugsFoundLast10Runs >= 1
|   |   |   |   PassesInLast10Runs <= 4 : Fail
|   |   |   |   PassesInLast10Runs > 4 : Pass
|   |   CheckinFileCountSinceLastRun >= 1 : Pass
|   FailsInLast10Runs >= 1
|   |   CheckinFileCountSinceLastRun < 2238
|   |   |   FailsInLast10Runs <= 1 : Pass
|   |   |   FailsInLast10Runs > 1 : Fail
|   |   CheckinFileCountSinceLastRun >= 2238 : Fail
LastRunStatus = Fail
|   FailsInLast10Runs = 0
|   |   PassesInLast10Runs <= 1
|   |   |   CheckinCountSinceLastRun <= 27
|   |   |   |   CheckinCountSinceLastRun <= 4 : Pass
|   |   |   |   CheckinCountSinceLastRun > 4 : Fail
|   |   |   CheckinCountSinceLastRun > 27 : Fail
|   |   PassesInLast10Runs >= 2
|   |   |   CheckinFilesForTestSinceLastRun <= 23 : Pass
|   |   |   CheckinFilesForTestSinceLastRun > 23
|   |   |   |   CheckinFilesForTestSinceLastRun < 397 : Pass
|   |   |   |   CheckinFilesForTestSinceLastRun >= 397 : Fail
|   FailsInLast10Runs >= 1 : Fail
```

Fig. 6. Prediction Model for RepTree with 5 Levels at Run 64

of failures to determine whether the test is predicted to pass or fail.

Also note that there appears to be an over fit of the model in this case. The top branch shows that, if a given test has passed all of the last 10 runs, then it will be predicted to fail only if there are no changes made to code hit by the test. This is very counter intuitive because a test without changes that has never failed is not expected to fail during a future run. This result is likely due to the fact that there are a small number of tests that have never failed in the 10 previous runs, and when they do fail for the first time, it is often due to an infrastructure or other non-code related issue.

This understanding is further strengthened by the fact that the top branch of the tree represents tests that passed last time, have never failed in the last 10 runs, and have no changes made to them. In these cases, if there is low complexity (fewer than 20 references), then they are predicted to pass. Cases with high complexity (more than 20 references) are predicted to

fail, indicating that complexity is associated with what must be environmental factors causing tests to fail, because no code changes are associated with those tests.

```
LastRunStatus = Pass
|   FailsInLast10Runs = 0
|   |   CheckinFileCountSinceLastRun = 0
|   |   |   BugsFoundLast10Runs = 0
|   |   |   |   TotalReferences <= 20 : Pass
|   |   |   |   TotalReferences > 20 : Fail
|   |   |   BugsFoundLast10Runs >= 1 : Fail
|   |   CheckinFileCountSinceLastRun >= 1 : Pass
|   FailsInLast10Runs >= 1
|   |   CheckinFileCountSinceLastRun <= 1139
|   |   |   CheckinFileCountSinceLastRun <= 51 : Fail
|   |   |   CheckinFileCountSinceLastRun > 51 : Pass
|   |   CheckinFileCountSinceLastRun > 1139
|   |   |   CheckinCountSinceLastRun <= 68 : Pass
|   |   |   CheckinCountSinceLastRun > 68 : Fail
LastRunStatus = Fail
|   FailsInLast10Runs = 0
|   |   PassesInLast10Runs <= 3
|   |   |   CheckinCountSinceLastRun <= 27 : Pass
|   |   |   CheckinCountSinceLastRun > 27 : Fail
|   |   PassesInLast10Runs > 3
|   |   |   TotalReferences <= 702 : Pass
|   |   |   TotalReferences > 702 : Fail
|   FailsInLast10Runs >= 1 : Fail
```

Fig. 7. Prediction Model for RepTree with 5 Levels at Run 48

Figure 7 shows a very similar model at 3/4 of the way through the project. Note that the same over fit exists in the top branch of the tree. Similarly, it is only when multiple historical attributes are considered that churn-based and complexity information adds to the model's predictive power.

E. Model Selection

There is a large number of classification models which can be used. In this case, we selected RepTree because it generates results which are in line with other prediction models. The other benefit of a tree model is that it is easy to read in order to gain an understanding of the underlying associations between test attributes and their prediction power related to failures.

Other prediction models, such as Naive Bayes, are mathematical models which are not easy to read. Similarly, Random Forest models will generate a large set of trees which are all used to vote on the prediction. Neither model is easy to read. As shown in Figure 8, the F-measure is similar to the RepTree method both in magnitude and variation. For these reasons, this research focused more deeply on the RepTree method.

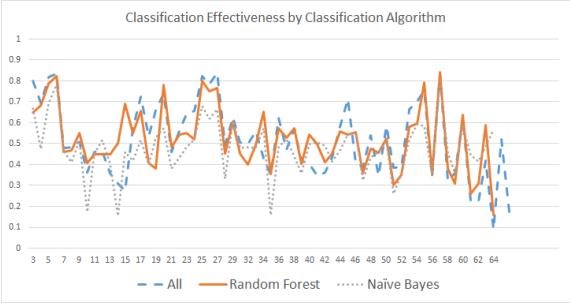

Fig. 8. Classification Algorithm F-Measure

F. Attribute Stability over Time

In this research, we discuss aspects of the classification models as if the attributes are static. Because the training data change with each run, there is some variation in the models used for each test run. An obvious question is how stable these models are over time for the project. It would be ideal if the model were perfectly static because that would mean that the attributes had a uniform ability to predict failure over time. As shown in Figure 9, this situation is not the case.

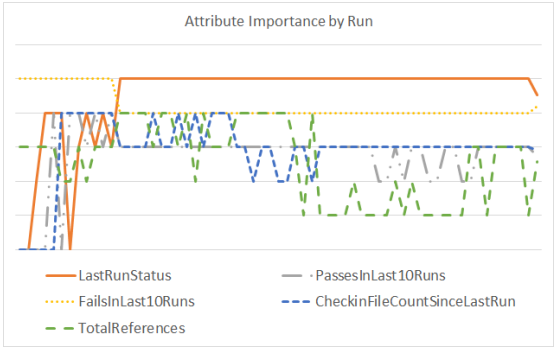

Fig. 9. Importance of Attributes by Build

This chart shows the importance of each attribute in the tree, with the value of 5 meaning the attribute is at the root of the tree and is, therefore, most important. The value of 1 means the attribute is a leaf of the tree, and 0 means the attribute does not appear in the tree which is pruned to a maximum of 5 levels.

While the attributes are not perfectly stable, the most important attributes are relatively stable. The last run status and fails in the last 10 runs are consistently the top two attributes. The other information, including complexity (total references) and churn (checkin file count since the last run), show up consistently in the bottom three levels of the tree, although they do vary slightly. From this visualization, we see that the generalizations discussed in this paper appear to hold, in general, across the software project.

V. Discussion and Implications

In this section, we discuss the implications of our study's results considering three aspects: (1) most influential attributes, (2) effectiveness over time, and (3) project feedback. We also discuss threats to the validity of our study.

A. 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 4. 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, such as Figures 6 and 7.

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.

This instability may also explain why complexity starts showing up near the leaf nodes in the classification models as illustrated in Figures 6 and 7. Complexity is a static measure throughout the project. In a large-scale software system such as *Dynamics AX*, the general graph of code references changes very little during point releases. The fact that complexity shows up in models as a predictor indicates that higher complexity is related to a higher chance of failure. We believe this correlation is for the same reason discussed above, that test environment stability is negatively impacted by the added complexity and, therefore, makes tests more likely to fail.

B. *Effectiveness over Time*

In Figure 4, 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.

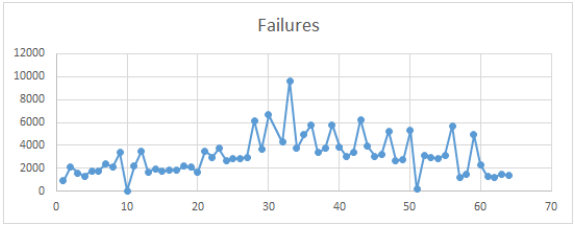

Fig. 10. Number of Failures by Test Run

Figure 10 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.

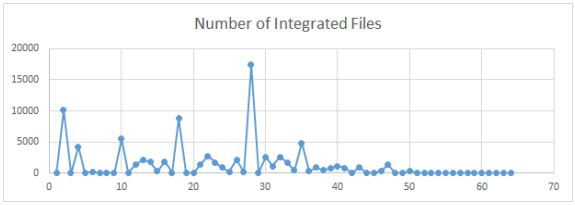

Fig. 11. Number of Integrated Files by Test Run

Figure 11 shows that the number of integrated files is another aspect of development that impacts test stability. During the initial phase of the release, many large integrations from hotfix, feature pack, and other branches were pulled into the main branch of code.

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.

When integrating one branch with another, the source control system is locked down. A private stabilization branch is used to integrate, test, and verify that the integration has succeeded prior to re-opening the main branch. Thus, when a large integration has happened, extra care is often placed on ensuring that all tests run reliably during the integration, and any test failures post integration are usually re-run to verify that there are no bugs after the integration. The impact of this

process is that the prediction power of the classification models drops significantly during these non-standard development periods.

C. 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.

D. 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.

VI. Related Work

There has been much research done in the area of fault prediction. An early survey about some of these techniques was done by Fenton and Neil [2]. In this critique, they examined techniques already in use by the industry using complexity, lines of code, and various testing metrics. Even in this early research, it was shown that holistic models with multiple metrics can be more reliable at predicting defects or predicting software reliability than single variables.

More recent research focused on advanced software metrics beyond simple complexity. Nagappan and Ball [4] showed that aspects such as software dependencies and churn (the changes made since the last test run) were successful in predicting defects as reported in the field in the Microsoft Windows Server 2003 release. In the study, they also showed that a single concept such as churn can yield multiple input variables: the number of changed lines of code, the number of changes, and the number of changed files. In our research, we similarly subdivided attribute categories into multiple input attributes.

Zimmermann and Nagappan have done a sizable amount of research about software dependency metrics and their impact on defects [7], [24]. Their research shows that software defects may be accurately predicted by looking at the interplay between software components. Further, Zimmermann et al. [25] have shown that software repositories can be mined to obtain this information which can then be used for fault prediction.

Software engineering researchers pulled these techniques into the area of fault prediction. For example, Podgurski et al. [14] used classification techniques to automatically classify software failure reports. In this case, the faults were reported in the field. The classification algorithms were used to perform first level triage on the bugs. More directly applicable to our area of research, Guo et al. [13] utilized classification techniques to predict bug bounces. Shihab et al. [26] took the idea of classification techniques for predicting bug bounces further, using four broad categories of classification attributes, each of which contained multiple measures. Similar to findings by Fenton and Neil [2] and Nagappan and Ball [4], Shihab et al. [26] showed that multiple attributes in concert had better predicting power than single attributes alone. While our research focused on different areas of prediction, we relied heavily on Shihab et al.'s techniques and categories of attributes.

Other researchers have investigated fault prediction by relating to the problems that are often faced by industry. For instance, Monden et al. [9] have looked at the economic impact of fault prediction under the assumption that testing resources for industrial products is a scarce resource. Their research considers the fact that when less time is spent on testing

areas which do not have bugs, resources are freed to be used elsewhere with higher impact.

In our research, we build on this rich history. We know that software repositories can be mined, that multiple aspects are better than single aspects, and that classification techniques can be successfully applied for defect prediction. Further, we know that identifying important regression test cases based on mining software repositories is viable and economically important. With this research, we build on this foundation by looking holistically at the test failure prediction and regression testing. More importantly, we seek to understand the interplay between the range of attributes used in the classification algorithms and what they can tell us about the underlying software project.

VII. Conclusions and Future Work

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.

In future work, we plan to expand this research to multiple products and releases. Because many of the aspects seen in this product were specific to the environment and activities during the release, it would be interesting to see if the same trends also apply to other products. We believe that such an investigation will help us to determine if there is a global model which effectively predicts test case failures across products. Further, we will also investigate whether the aspects of the models produced for other products tell similar stories about the environment and practices for those projects.

Acknowledgment

This work was supported, in part, by NSF CAREER Award CCF-1149389 to North Dakota State University, grants from the National Center for Research Resources (P20 RR016471) and from the National Institute of General Medical Sciences (P20 GM103442) part of the National Institutes of Health.

References

[1] R. V. Binder, *Testing Object-Oriented Systems*. Upper Saddle River, NJ: Addison Wesley, 1999.

[2] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[3] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[4] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM*. IEEE, 2007, pp. 364–373.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 86–96.

[6] M. Mayo and S. Spacey, "Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics," in *Search Based Software Engineering*. Springer, 2013, pp. 158–171.

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

[8] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2011, pp. 357–366.

[9] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, and K.-i. Matsumoto, "Assessing the cost effectiveness of fault prediction in acceptance testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1345–1357, 2013.

[10] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 2014, pp. 142–151.

[11] O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification," in *IEEE Conference on Computer Vision and Pattern Recognition, 2008*. IEEE, 2008, pp. 1–8.

[12] R. Sandberg, G. Winberg, C.-I. Bränden, A. Kaske, I. Ernberg, and J. Cöster, "Capturing whole-genome characteristics in short sequences using a naive bayesian classifier," *Genome Research*, vol. 11, no. 8, pp. 1404–1409, 2001.

[13] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Not my bug! and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 conference on Computer Supported Cooperative Work*. ACM, 2011, pp. 395–404.

[14] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *International Conference on Software Engineering*. IEEE, 2003, pp. 465–475.

[15] A. Chaudhary, S. Kolhe, and R. Kamal, "Machine learning techniques for mobile intelligent systems: A study," in *2012 Ninth International Conference on Wireless and Optical Communications Networks (WOCN)*. IEEE, 2012, pp. 1–5.

[16] P. Jain, J. M. Garibaldi, and J. D. Hirst, "Supervised machine learning algorithms for protein structure classification," *Computational Biology and Chemistry*, vol. 33, no. 3, pp. 216–223, 2009.

[17] F. J. Provost and T. Fawcett, "Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions." in *KDD*, vol. 97, 1997, pp. 43–48.

[18] L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, L. D. Jackel, Y. LeCun, U. A. Muller, E. Sackinger, and P. Simard, "Comparison of classifier methods: A case study in handwritten digit recognition," in *International Conference on Pattern Recognition*. IEEE Computer Society Press, 1994, pp. 77–77.

[19] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural Computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[20] "http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/reptree.html."

[21] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI Workshop on Empirical Methods in AI*, vol. 3, no. 22, 2001, pp. 41–46.

[22] Microsoft Corporation, "New feature list for Microsoft Dynamics AX 2012 R2," http://www.microsoft.com/en-us/download/details.aspx?id=35824, Nov. 2012.

[23] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[24] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *ESEM*. IEEE Computer Society, 2009, pp. 435–438.

[25] 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*, May 2004, pp. 563–572.

[26] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *2010 17th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2010, pp. 249–258.