Using Machine Learning to Prioritize Automated Testing in an Agile Environment

Laurie Butgereit
Nelson Mandela University
South Africa
laurie.butgereit@mandela.ac.za

***Abstract*—Automated software testing is an integral part of most Agile methodologies. In the case of the Scrum Agile methodology, the definition of *done* includes the completion of tests. As a software project matures, however, the number of tests increases to such a point that the time required to run all the tests often hinders the speed in which artifacts can be deployed. This paper describes a technique of using machine learning to help prioritize automated testing to ensure that tests which have a higher probability of failing are executed early in the test run giving the programmers an early indication of problems. In order to do this, various metrics are collected about the software under test including Cyclomatic values, Halstead-based values, and Chidamber-Kemere values. In addition, the historical commit messages from the source code control system is accessed to see if there had been defects in the various source classes previously. From these two inputs, a data file can be created which contains various metrics and whether or not there had been defects in these source files previously. This data file can then be sent to Weka to create a decision tree indicating which measurements indicate potential defects. The model created by Weka can then then be used in future to attempt to predict where defects might be in the source files and then prioritize testing appropriately.**

***Keywords*—machine learning, automated testing, Weka, Cyclomatic, Halstead, Chidamber-Kemere, agile**

I. Introduction

Testing is an integral part of Agile software development projects [1]. The second Agile value as documented in the Agile Manifesto is that “working software [is valued more than] comprehensive documentation”. The first principle of the Agile Manifesto is that “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” and the third principle is to “Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale” [2].

In order to adhere to these Agile values and principles, software testing is just as important as software coding. In many Agile methodologies, the definition of *done* must include the fact that tests are written and running successfully.

ISBN 978-1-5386-7365-2/19/$31.00 ©2019 IEEE

In some Agile methodologies, the actual tests are written *before* the code itself. In order to satisfy the fast development times expected on a Agile project to deliver a new software increment every few weeks, this testing must be automated.

However, as the project matures the number of tests increases thereby increasing the execution time of the test suites. At some point, the tests are no longer finishing in a time short enough to cater to the expectations of continuous delivery.

Data mining is the extraction of previously unknown information, patterns, and relationships from data which could be potentially useful [3]. This involves the application of various statistical methodologies such as the development of statistical models. This allows for visible and/or usable traits and characteristics to be drawn from the underlying data [4]. Machine learning uses many of the same algorithms as data mining uses. In addition, however, machine learning attempts to improve performance and make more accurate predictions [5].

This paper describes a research project where machine learning techniques are used to prioritize the writing and the subsequent execution of automated tests by attempting to predict which classes and methods may possibly have software defects in them. By prioritizing the writing of test programs to test suspect software, the possibility of uncovering defects in such software early is increased. In addition, by prioritizing the execution of such test programs, the actual defects are found earlier in the Agile iteration.

II. Clarification of Terms

There are a number of similar terms used in this paper which need to be clarified. These include *agile testing metrics*, and *static source code metrics*. The term *agile testing metrics* includes measurements such as defect cycle time (how long it takes to correct a defect), defects found in production (how many defects actually make their way into production), code complexity (how complex the source code is), and cumulative flow (measurements such as agile burn down velocity) [6]. The term *static source code metrics* refers specifically to characteristics of the source code at compile time. This includes measurements such as number of lines of code, number of comments, depth of nested constructs, etc.

This paper specifically looks at the use of static source code metrics in an agile testing environment. The paper does not deal with agile testing metrics themselves.

III. Previous Work on Predicting Defects

NASA (The US National Aeronautic and Space Administration) has released data sets consisting of software metrics and whether or not there were defects in that particular software unit. The data is released in Weka ARFF format [7]. This data includes defect data on more than 200 NASA projects spanning over 15 years [8].

The NASA project used a number of machine learning algorithms including PART, C4.5, Naïve Bayes, and the association rule mining (which was the basis of their research paper). The results were between 90% and 95% accurate in predicting defects using the association rule mining [8]. These results were nearly 25% higher than using PART, C4.5, and Naïve Bayes indicating that these other three methods were providing accuracies of between approximately 72% and 76% [8]. In 2013, the researchers published some concerns with their data set [9]. Despite these concerns, in a survey of 42 studies on defect detection and prediction using approximately different 600 datasets, 59% of the studies used the NASA data sets [10].

Gray *et al.* noted that open source systems could be used for training data if the defect information was correctly and consistently entered in a defect tracking system or source code control system [11]. In addition, static source metrics have been used to attempt to predict source files which are in need of refactoring [12].

IV. The Environment

This research was conducted at a listed company in South Africa. For the scope of this paper, this company will be identified as Company X. Company X is one of the largest vendors of secure tokens in South Africa which can be redeemed for commodities such as airtime, data, electricity, WIFI connectivity, and some gambling facilities. In addition, the company offers other non-tangible goods and services such as event ticketing, bus ticketing, bill payment services, and traffic fine payment services.

These facilities are provided through the company's proprietary switch which connects to the various cell phone providers, bus providers, bill payment aggregators, etc. In addition the company offers a number of client configurations such as a Windows based client, an Android based client, and a web based client to allow merchants (and, in some cases, end users) to access their services. All software is written in Java.

The software was and continues to be developed and maintained by Company X. The software methodology used by the company is a type of Water-Scrum-Fall. Dave West coined the term Water-Scrum-Fall to describe a situation where an organization originally utilized a Waterfall methodology and is attempting to migrate to an Agile methodology [13]. In a Water-Scrum-Fall methodology, the first few steps of a project such as requirements gathering are executed using Waterfall methodology. The development of the software and possibly the Quality Assurance step is done using an Agile methodology such as Scrum. The actual deployment of the software is again executed under a Waterfall methodology.

At Company X, only the development step of the project was done under an Agile methodology – specifically Scrum. Although the Quality Assurance step and User Acceptance step was done using a Waterfall methodology, the development team had extensive automated tests to ensure that the QA and UAT steps were flawless.

At the conception of this research project, the automated tests for the Android app were taking four to five hours to run. These tests included complete end-to-end integration tests which tested the flow of data from the Android app, to the central server, to the individual service providers (such as a cell phone company or bus company) and back. These tests were currently run in alphanumeric order with the newest tests written being usually the last to be executed in the suite. The priority of which tests were most important (or most likely to catch defects) needed to be decided.

V. Previous Research in Software Metrics

Throughout the history of software development, there have been a number of different software metrics which have been developed. This section describes some of the metrics which are available. In addition, this section describes which of these metrics were chosen for this project.

A. Cyclomatic Complexity

In 1976, Thomas McCabe published a paper entitled *A Complexity Measure* [14]. McCabe argued that since testing and maintenance of software often cost more than the original development of the software, some mathematical method was needed to indicate which software modules will be difficult to test and maintain.

McCabe proposed the use of mathematical graph theory. He proposed determining how complex a piece of software was by counting the number of possible paths through a piece of code. As the number of possible paths increased, then the complexity of the piece of software increased.

It is important to note that McCabe did his research in the mid-1970s in FORTRAN. Programming languages have changed since the 1970s but his underlying concept of complexity is still valid. Intuitively, if a Java method has two nested IF statements and a SWITCH statement, it is more complex than a method which only has one FOR loop.

B. Halstead Metrics

In 1977, Maurice Halstead noted that it is possible to count the number of *operators* and the number of *operands* in a program. From those measurements, he proposed that one could then calculate certain metrics such as the size of the *vocabulary* of the program, the *length* of the program, the *volume* of the program, the *difficulty* required to understand the program, the *effort* to reproduce the program, etc [15].

n1	Number of distinct Operators
n2	Number of distinct Operands
N1	Number of total Operators
N2	Number of total Operands
n = n1 + n2	Vocabulary
N = N1 + N2	Length
$V = N \times \log_2(n)$	Volume
D = n1/2 x N2/n2	Difficulty
E = D x V	Effort

Table 1: Halstead metrics

The metrics proposed by Halstead can be seen in Table 1.

As with McCabe, Halstead was working in the mid-1970s and programming languages have improved since that time. The definition of what is an operator and what is an operand can be a point of argument. For example, consider the Java code snippet found in Table 2.

It is clear that the variable *d* would be an operand and the symbol = would be an operator. But what of the type indicator *Double*? Is that an operator or an operand? In the expression *d += i,* clearly the += is an operator. But is it one operator or is it two operators? In addition, in that same expression, there is an implicit type cast from *int* to *Double.* Should that be also counted? *println* could be counted as an operator, but should *System.out* be counted as one operand or as two operands (*System* and *out*) and an operator (the full stop)?

```
Double d = 1.23;

for(int i=0; i<10; i++) {

    d += i;

}

System.out.println("d is now " + d);
```

Table 2: Java snippet

In addition, by looking at the literature on Halstead metrics and by looking at the various open source implementations of Halstead metrics, there appears to be some confusion on the definition of the terms *distinct* and *total*.

```
int a;
int b;

a = 7;
b = a;
```

Table 3: Java snippet

Looking at the Java snippet in Table 3, there are three separate uses of the variable *a*. Other operands include *b* and 7. Does the Halstead metric consider the *distinct* operands to be *a*, *b* and *7* for a count of 3 and the *total* operands to be three uses of *a*, two uses of *b*, and a *7* for a count of 6? Or does the Halstead metric consider the *distinct* operands to be three uses of *a*, two uses of *b*, and a *7* for a count of 6 and the total operands to be *a, b* and *7* for a count of 3?

Various open source implementations of Halstead differ in this respect. In other words, some open source implementations of Halstead has N1 >= n1 and N2 >= n2 and other implementations have n1 >= N1 and n2 >= N2.

This research is not about general complexity and it is not about verifying whether or not Halstead metrics actually define the complexity of a piece of source code. This research is whether or not there are patterns in source code which could assist in prioritizing tests. Tables 6 and 7itemsise how Halstead-based metrics were implemented in this research.

C. Chidamber Kenmere

In 1992, Chidamber and Kenmere published a report at Massachusetts Institute for Technology proposing new metrics to measure object oriented design [16]. This was followed in 1994 by a paper in IEEE Transactions with the same name [17]. They defined six metrics shown in Table 4.

WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Objects
RFC	Response For a Class
LCOM	Lack of Cohesion in Methods

Table 4: Chidamber-Kenmere metrics

These six metrics were based on measurement theory and, according to the authors, reflected the viewpoints of experienced object oriented software developers. Three of the metrics (WMC, DIT, and NOC) measured aspects of the object definition. Three of the metrics measured aspects of the object semantics (WMC, RFC, and LCOM). Two of the metrics measured relationships between objects (RFC and CBO).

D. *Implemented in this research*

This research was specifically for prioritizing testing of existing Java applications. This research was not about object oriented design. The research was also not about general software complexity. The research was about testing existing code. For this reason, the majority of the Chidamber-Kemere metrics were ignored with the exception of WMC. The Chidamber-Kemere metrics are about *design* and, at the time of testing, the *design* has been finalized in the existing software. The WMC metric was retained in order to calculate some class values and some average values per methods.

The Cyclomatic values (both for the entire class and the average per method) were used. The underlying calculations of Halstead values were also used. However, to avoid the argument on the difference between the definition of the terms *distinct* values and *total* values, expressions used will be OperatorInstances, OperatorNames, OperandInstances, and OperandNames.

VI. Tool Development

A tool was developed to calculate a collection of the metrics mentioned in Section V. The open source utility JavaParser was used to do the actual parsing of the Java source code. For each Java compilation unit file (usually a single class or interface), the metrics shown in Table 5 were calculated.

Lines of code
Lines of comments
Chidamber and Kenmere WMC
Cyclomatic value per class
Cyclomatic average per method
OperatorNames
OperatorInstances
OperandNames
OperandInstances

Table 5: Metrics chosen for this research

All metrics which were based on Halstead (in other words OperatorNames, OperatorInstances, OperandNames, and OperandInstances) were measured per class and not per method. This is justified by the fact that a per method calculation would not take into account global class variables. It is noted that the other Halstead-related metrics were ignored because if a pattern could be found using values derived from the four basic Halstead values, then the pattern could also be derived from the four basic Halstead values themselves.

It is also important to comment on which Java constructs were deemed to be operators in the terms of Halstead-based calculations in this research. These are itemised in Table 6

All Java unary, binary, and ternary operators
All Java statements such as if, while, do, for (both types), switch, case, break, continue, default, return, try/catch, throw
() used inside expressions but excluding () using in method declarations and calls
instanceof
Type declarations for variables, parameters, casts and method return types
{} used in array initializers and class initializers but excluding {} used in compound statements and method delimitation
[] used for array access
Full stop/period used for class field access
new
Method calls including super()
comma in method calls

Table 6: Halstead operators

The Halstead operands are itemised in Table 7

Variable names, parameter names, class names, interface names, class field names
super
this
int, character, string, boolean, float, long and double literals

Table 7: Halstead operands

Some of the constructs which were not counted as either operators or operands are listed in Table 8

Lambdas	They were not used in any of the software projects covered by this research
Package definition and import statements	Compiler instructions
Inner classes	All statistics for inner classes were added to the outer class
Streams	Streams were not used in any of the software projects covered by this research

Table 8: Constructs ignored by the developed tool

An additional tool was developed to access the Subversion (source control) logs of each file. The company under research had a reasonable discipline of flagging defect fixes using specific code words in the subversion commit message. By pulling the Subversion logs for each source file, it was possible to determine which source files had had defects in it during the past.

These two utilities used together created a comma separated file which has the general format of:

```
filename, value1, value2, . . . value9, YES/NO
```

where 9 metrics were listed and the keyword YES or NO indicated if there had been a defect in that particular class at any point in time.

VII. Weka

Weka is a work bench for machine learning [18]. Weka's interactive work bench provides a wide range of tools which can be used for analyzing and visualizing data. The work bench is primarily designed to be used by domain experts and not necessarily machine learning experts. There is also a Java library providing the various machine learning techniques which can be embedded inside of Java code.

The filenames of the data file created and described in Section VI were removed and the required headers were inserted to create a file in the ARFF format required by Weka.

The J48 decision tree classifier was used to process the ARFF file. J48 is an open source Java implementation of the C4.5 algorithm [19]. C4.5 was developed by Ross Quinlan [20]. Both J48 and C4.5 generate a classification-decision tree for a given data set [21].

VIII. Results

Three different software systems developed Company X were used for this research. For the scope of this paper, the software projects will be simply named A, B, and C. All three projects were written in Java. The size of the three individual projects can be seen in Table 9.

Project	Total # Classes	Total # Lines of Code
A	598	108,977
B	768	136,723
C	509	97,332

Table 9: General project statistics

The first exercise was to extract the Subversion commit logs to determine which classes had had defects in the past. It was found that only Project A had the sufficient project discipline among developers to consistently flag defect corrections in the source code repository logs. Project B had recently had a reorganization of packages and many of the commit logs had been lost. Developers in Project C had never had the discipline to log those correction.

At this point in this research, it was decided to only use the source code from Project A in the creation of a defect prediction model.

The source code from Project A was then used as input to the tool described in Section VI. The tool created a Weka format ARFF format which looked similar to the data seen in Table 10.

Using the ARFF data unchanged, Weka (with the J48 Decision Tree) created a model which correctly predicted defects for 83.2776% (498 out of 598).

Weka is an interactive tool and it is possible to easily remove attributes from the data file. For example, it was found that there were, in fact, some inconsistencies in the commenting techniques of developers working on Project A. Some developers had left old code in the source but commented the old code out. This habit artificially increased the count of the number of comments. By removing the number of lines of comments from the data, the model correctly predicted defects for slightly more source files (501 out of 598).

```
@relation bug-prediction
@attribute wmc numeric
@attribute cc numeric
@attribute loc numeric
@attribute comments numeric
@attribute operatorNames numeric
@attribute operatorInstances numeric
@attribute operandNames numeric
@attribute operandInstances numeric
@attribute bug { 'yes', 'no'}

@data
8,1.375,71,17,[. . . .deleted . . . .],no
```

Table 10: Example Weka ARFF file

In fact, by playing with the data using Weka, it was found that just the one Halstead-based value operatorNames could provide a better model which correctly predicted defects in 508 out of 598 source files. This provided an accuracy rate of 84.9498%

Naïve Bayes provided a slightly higher accuracy of 85.1171% (509 out of 598). PART provided a slightly lower accuracy of 84.6154% (506 out of 598).

These values of 83% to 85% are higher than the 72% to 75% described in Section III but less than the 95% described in that section using their association rule mining.

The actual model created by Weka J48 can be seen in Table 11.

```
operatorNames <= 88: no
operatorNames > 88: yes
```

Table 11: J48 Model

The model created by Weka was then used to process all the source files in Project C. Project C was chosen because it had the largest test suite and was the project most at risk of having large test suites slow down sprint delivery. (Project B was ignored for the remainder of this project because it did not have any test programs to begin with.)

This generated a list of 33 source files (out of the 509 source files in Project C) which warranted having more test programs written and also warranted having those tests moved to the front on the test execution queue. At the time of writing this paper, these new test programs are being written and prioritized in the test execution queue.

IX. Conclusion

The Agile Manifesto is the basis of a number of Agile methodologies include Scrum, Kanban, and Lean. The manifesto insists that working software must be released frequently to customers. In order to *prove* that the software is working, automated tests should be written to exercise and examine the software. As the project grows, the number of tests also grows. Even with automated testing, it is often the case that tests can take hours to run.

As the test suite gets longer and longer, a mechanism is needed to prioritize tests in the queue such that tests which are more likely to fail are in the front of the queue so that they fail fast when necessary.

This paper looked at using machine learning techniques in order to determine which characteristics of a Java source file are indicative of potential defects in the source file. Three different sets of metrics were collected: Cyclomatic metrics, Halstead-based metrics, and one Chidamber-Kemere metric. The subversion commit logs were accessed in order to see if there had been defects inside the source file historically. This data was collated and then analyzed by Weka's J48 decision tree.

It was found that Halstead-based metric OperatorNames was the most important metric collected and could predict defects with just under 85% accuracy. This information can inform the creation of the test program queue to ensure that tests which test at-risk classes execute before tests which test classes which are lest at-risk.

References

[1] D. Talby *et al*, "Agile software testing in a large-scale project," *IEEE Software,* vol. 23, *(4),* pp. 30-37, 2006. Available: https://ieeexplore.ieee.org/abstract/document/1657936/.

[2] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development,* vol. 9, *(8),* pp. 28-35, 2001. Available: http://andrey.hristov.com/fht-stuttgart/The_Agile_Manifesto_SDMagazine.pdf.

[3] I. H. Witte*n et a*l, *Data Mining: Practical Machine Learning Tools and Techniques.* 2016.

[4] M. J. Zaki and M. J. Meira, *Data Mining and Analysis: Fundamental Concepts and Algorithms.* 2014.

[5] M. Mohri, A. Rostamizadeh and A. Talwalkar, *Foundations of Machine Learning.* 2012.

[6] (). *Doing Agile Testing Properly: How metrics can hel*p. Available: https://www.globalapptesting.com/blog/doing-agile-testing-properly-how-metrics-can-help.

[7] (Aug 9, 2016). *NASA Defect Datase*t. Available: https://github.com/klainfo/NASADefectDataset.

[8] Q. Son*g et a*l, "Software defect association mining and defect correction effort prediction," *IEEE Trans. Software Eng.,* vol. 32, *(2),* pp. 69-82, 2006. Available: https://ieeexplore.ieee.org/abstract/document/1599417/.

[9] M. Shepper*d et a*l, "Data quality: Some comments on the nasa software defect datasets," *IEEE Trans. Software Eng.,* vol. 39, *(9),* pp. 1208-1215, 2013. Available: https://ieeexplore.ieee.org/abstract/document/6464273/.

[10] M. Shepperd, D. Bowes and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Software Eng.,* vol. 40, *(6),* pp. 603-616, 2014.

[11] D. Gra*y et a*l, "Reflections on the NASA MDP data sets," *IET Software,* vol. 6, *(6),* pp. 549-558, 2012. Available: https://ieeexplore.ieee.org/abstract/document/6464273/.

[12] L. Zhao and J. Hayes, "Predicting classes in need of refactoring: An application of static metrics," in *Proceedings of the 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA,* 2006, Available: http://blankslatetech.com/project1/trunk/promisedata.org/pdf/phil2006ZhaoHayes.pdf.

[13] D. Wes*t et a*l, "Water-scrum-fall is the reality of agile for most organizations today," *Forrester Research,* vol. 26, 2011.

[14] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng., (4),* pp. 308-320, 1976. Available: https://ieeexplore.ieee.org/abstract/document/1702388/.

[15] M. H. Halstead, *Elements of Software Science.* 1977.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.,* vol. 20, *(6),* pp. 476-493, 1994. Available: https://ieeexplore.ieee.org/abstract/document/295895/.

[17] S. R. Chidamber and C. F. Kemere, "A metrics suite for object oriented design," Massachusetts Institute of Technology, Massachusetts, USA, Tech. Rep. CISR WP No. 249, Sloan WP No. 3524-03, December, 1992. 1992.

[18] G. Holmes, A. Donkin and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on,* 1994, .

[19] T. R. Patil and S. Sherekar, "Performance analysis of Naive Bayes and J48 classification algorithm for data classification," *International Journal of Computer Science and Applications,* vol. 6, *(2),* pp. 256-261, 2013.

[20] J. R. Quinlan, *C4. 5: Programs for Machine Learning.* 2014Available: https://books.google.co.za/books?hl=en&lr=&id=b3ujBQAAQBAJ&oi=fnd&pg=PP1&dq=c4.5+quinlan.

[21] Y. Zhao and Y. Zhang, "Comparison of decision tree methods for finding active objects," *Advances in Space Research,* vol. 41, *(12),* pp. 1955-1959, 2008.