Analysing Static Source Code Features to Determine a Correlation to Steady State Performance in Java Microbenchmarks

Jared Chad Swanzen
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
jaredswanzen@gmail.com

Kyle Thomas Botes
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
kylethomasbotes@gmail.com

Husnaa Molvi
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
husnaamolvi@gmail.com

Omphile Monchwe
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
omphile05monchwe@gmail.com

Dan Phala
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
mrphalad@gmail.com

Dustin van der Haar
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, Gauteng, South Africa
dvanderhaar@uj.ac.za

ABSTRACT

Source code analysis is an important aspect of software development that provides insight into a program's quality, security and performance. There are few methods for consistently predicting or determining when a written piece of code will end its warm-up state and proceed to a steady state. In this study, we use the data gathered by the SEALABQualityGroup at the University of L'Aquila and Charles University and extend their research of steady state analysis to determine whether certain source code features could provide a basis for developers to make more informed predictions on when a steady state would occur. We explore if there is a direct correlation between source code features on the time and ability of a Java microbenchmark to reach a steady state to build a machine learning-based approach for steady-state prediction. We found that the correlation between source code features and the probability of reaching a steady state go as high as 10.9% for Pearson's correlation coefficient, whereas the correlation between source code features and the time it takes to reach a steady state go as high as 21.6% for Spearman's correlation coefficient. Our results also show that a Random Forest Classifier with features selected with either Spearman's or Kendall's correlation coefficient boasts an accuracy of 78.6%.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation.*

KEYWORDS

steady state, machine-learning, static source code analysis, correlation coefficient, Java microbenchmark, ANTLR, correlation study, Pearson's r, Spearman's roh, Kendall's tau

ACM Reference Format:
Jared Chad Swanzen, Kyle Thomas Botes, Husnaa Molvi, Omphile Monchwe, Dan Phala, and Dustin van der Haar. 2023. Analysing Static Source Code Features to Determine a Correlation to Steady State Performance in Java Microbenchmarks. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3584692

1 INTRODUCTION

Microbenchmarking is a form of measurement-based software performance engineering that evaluates execution time and software features such as methods. By using microbenchmarking, developers can analyse the performance of their written source code to ensure the timely functioning of the system.

The main goal of this study is to try to find a potential correlation between the steady state of a program and the source code features present within the data to derive relevant features that can be used to train a machine learning model that can predict the ready state. We show that source code features correlate well with both the ability of a benchmark and the number of iterations required to reach a steady state and that these features can successfully be used to predict if a written piece of code will reach a steady state.

The article starts with a literature review covering the relevant aspects of automated source code analysis and similar works used to derive steady state. We then discuss the relevant experiment details and considerations, followed by the results and discussion and the study is concluded.

2 LITERATURE REVIEW

Source code analysis is the process of using automatic tools to extract information about a program from its source code or artefacts, such as Java byte code or execution traces [3]. Unlike dynamic

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584692

source code analysis, which evaluates code behaviour during execution, Static Code Analysis can be performed while the software is still in production and does not require the code to be running and this allows for detecting vulnerabilities earlier in the software development life cycle [11].

Steady-state performance is achieved when the software in execution is no longer subject to its initial fluctuations in performance. These fluctuations in Java software occur due to the JVM optimising frequently executed parts of the code, a phase referred to as the warm-up phase [16]. Detection of the presence of a steady state and when it occurs has become important because in JIT compiler benchmarking methodologies, the data gathered during the warm-up phase is discarded, and steady-state performance measurements are used to evaluate the performance of the system or program [2].

Although static code analysis and steady-state assessment are frequently discussed in literature, works that combine the two are uncommon and hard to come by. Laaber et al. [13] conducted a study using machine learning to identify unstable software benchmarks, focusing on benchmarks written in the Go programming language. Their approach uses the benchmarks' result variability as its measure for stability using thresholds proposed in previous works [8] [6]. They found that for the Random Forest model to predict benchmark's stability accurately, required the combination of all features. In contrast, the combination of features in the benchmark itself was found to be less significant. Their research highlighted the potential use of static code analysis to predict steady-state performance.

Another study on machine learning-based inspection with static code features was done by Tribus in [17] that analyses the source code instead of considering meta-values. The aim was to identify at least one minimal set of features to automatically detect faults within source code where a feature selection model and parser would translate the source code into compatible instances fed into the WEKA data mining framework. About 71 classifiers were compared using WEKA, and the six best performers included MultiLayerPerceptron, instance-based learner and logistic model tree and "AdaBoost" on a "BFTree" with an accuracy of over 90%. A second investigation confirmed their findings, decreasing the accuracy of "AdaBoost" on a "BFTree". This research validates the efficiency of machine learning techniques for static source code analysis.

The works of Barrett et al. [2] and, most recently, Traini et al. [16] have challenged the traditional assumption that all benchmarks will reach a steady state. This has implications for the current state of the art and practice in steady-state detection, which commonly require the execution of a benchmark of the software under evaluation to gather performance measurements [16]. Using static source code features to predict the steady-state performance of Java software could potentially have significant advantages compared to traditional microbenchmarking methods. The results of benchmarks that do not reach a steady state do not correctly reflect the "real" performance of the software under test [13], as such time and resources would be saved if these benchmarks could be identified before their execution.

3 METHODOLOGY

In the study experiments, a secondary dataset was created to assess the impact of source code features in a correlation study and derive the results from steady-state classifications using conventional machine learning models.

3.1 Data sampling

We used the annotations that Traini et al. [16] produced in their paper that identified whether (and when) a steady state was reached. These classifications defined a 'run' as either a 'steady state' when all the forks of the particular benchmark reached a steady state or 'inconsistent' when only a few reached a steady state. It was important to note that no benchmark failed to reach a steady state on all forks. From 'steady_state_starts', we could see the iteration point where the benchmark reached a steady state.

3.2 Source Code Features

Once the source code was acquired, all the files imported to the benchmark class within the same package were recursively collected[1]. This was done to ensure that the source code features used were representative of the imports the benchmark relied on.

Afterwards, the ANTLR library, a powerful parser generator [14], was used to walk the Abstract Syntax Tree (AST) for each file. After a node was visited on the AST, the respective source code feature count increased. We considered all 127 features that ANTLR was capable of evaluating[2]. Finally, once all the nodes were visited, the results were considered and analysed within the correlation section of the study.

3.3 Correlation Study

After deriving the source code features, the study continued by using Pearson's r [7], Spearman's rho [20], and Kendall's tau [12] to determine if there was a link between the steady state and the feature under consideration. A correlation coefficient is a number between -1 and 1 that tells us the strength and direction of a relationship between variables. For example, a correlation coefficient of 0.92 describes a 92% positive correlation between the variables. This leads us to believe that as a variable (x) increases, the other variable (y) would also increase. A value of -0.92 describes a 92% negative correlation, leading us to believe that as x decreases, y will increase.

When a correlation coefficient is less than desirable, we will only consider features above a certain threshold, allowing for enough features to be considered. If this value is large enough, we will increase the threshold sparingly to ensure that we consider the most significant features. We will also allow for more features since the correlation coefficients are higher.

3.4 Steady State Prediction

Using the correlation study, all the features that could not make a significant contribution towards the steady state of a benchmark will be filtered out. These features are then fed into machine learning models for training and testing with a 90/10 split. To predict

[1]We were unable to acquire any of the benchmarks tested for apache-tinkerpop within their source code files and therefore apache-tinkerpop was not considered as part of the study

[2]A full list of these features can be found in the project files [15]

Table 1: Parameters Used for Machine Learning Models

Module	Parameter	Values
Logistic Regression	C	0.01, 0.1, 1, 10
Decision Tree	Criterion	gini, entropy
Random Forest	Criterion	gini, entropy
K Nearest Neighbours	n_neighbours	3, 5
Naive Bayes	-	-
Support Vector Machine	C	0.01, 0.1, 1, 10

whether a benchmark would reach either a consistent or inconsistent steady state, the following binary classifiers were used: (See Table 1 for parameters)

(1) Logistic Regression: A binary predictor that estimates the probability of an event's occurrence given a dataset of independent variables [5].
(2) Random Forest: A two-group classification algorithm that yields a single result by combining multiple decision trees [10].
(3) Support Vector Machine : Another two-group classification for limited samples [4].
(4) K Nearest Neighbours: A non-parametric classifier that aids in the grouping of individual data points using proximity [1].
(5) Decision Tree: A non-parametric algorithm used for classification and regression with a hierarchical tree structure [19].
(6) Naive Bayes: A binary and multiclass classification where the training data is already labelled with a class [18].

4 RESULTS

Results and source code to replicate findings can be found in the project files [15].

According to the methodology we defined, we could derive the following results.

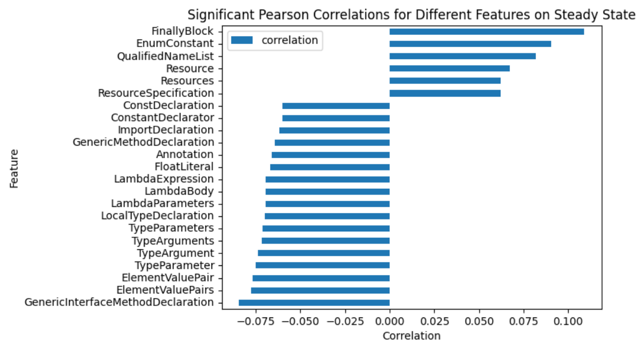

Figure 1: Significant Pearson Correlations on Steady State

In Figure 1 the highest correlations we observed are:

(1) FinallyBlock: 10.9% positive correlation
(2) EnumConstant: 9% positive correlation

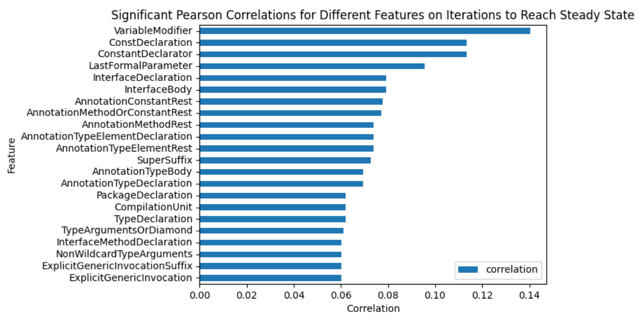

Figure 2: Significant Pearson Correlations on Iterations

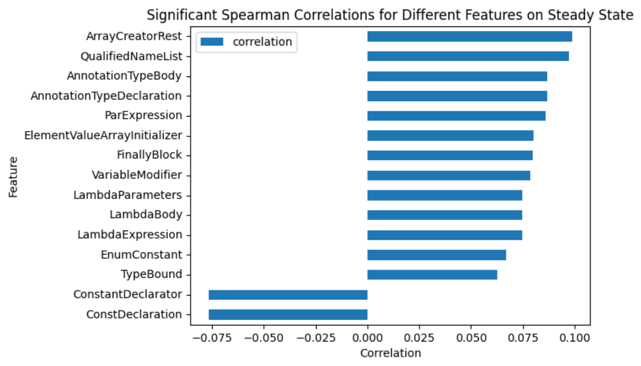

Figure 3: Significant Spearman Correlations on Steady State

(3) GenericInterfaceMethodDeclaration: 8.4% negative correlation

In Figure 2 the highest correlations we observed are:

(1) VariableModifier: 14% positive correlation
(2) ConstDeclaration and ConstantDeclarator: 11.4% positive correlation
(3) LastFormalParameter: 9.6% positive correlation

In Figure 3[3] the highest correlations we observe are:

(1) ArrayCreatorRest: 9.9% positive correlation
(2) QualifiedNameList: 9.7% positive correlation
(3) AnnotationTypeBody: 8.7% positive correlation

In the Figure for Spearman's correlation to iterations [4] the highest correlations we observe are:

(1) AnnotationConstantRest: 21.6% positive correlation
(2) LastFormalParameter: 20% positive correlation
(3) SuperSuffix: 19.1% positive correlation

Similarly, in the Figure for Kendall's correlation to iterations [5]:

(1) AnnotationConstantRest: 18.1% positive correlation
(2) SuperSuffix: 16.1% positive correlation
(3) LastFormalParameter: 15.2% positive correlation

[3]Spearman's rho in Figure 3 had identical results to that of Kendall's tau and was therefore omitted

[4]Due to space constraints some images were omitted. These can be found in the project files [15] under /correlation-study/graphs/

[5]See footnote 4

Table 2: Results for predicting whether a benchmark will reach steady state or not

Model	Parameters	Unfiltered	Pearson	Spearman	Kendall
Logistic Regression	C=0.01	42.9%	46.4%	50%	51.8%
	C=0.1	42.9%	46.4%	50%	51.8%
	C=1	50%	44.6%	50%	51.8%
	C=10	42.9%	46.4%	50%	51.8%
Decision Tree	criterion=gini	73.2%	71.4%	67.9%	67.9%
	criterion=entropy	69.6%	69.6%	71.4%	71.4%
Random Forest	criterion=gini	73.2%	66.1%	73.24%	69.6%
	criterion=entropy	75%	67.9%	73.2%	69.6%
K Nearest Neighbour	n_neighbours=3	66.1%	67.9%	**78.6%**	**78.6%**
	n_neighbours=5	67.9%	67.9%	62.5%	62.5%
Naive Bayes	-	48.2%	50%	48.2%	48.2%
Support Vector Machine	C=0.01	55.4%	55.4%	55.4%	55.4%
	C=0.1	55.4%	55.4%	55.4%	55.4%
	C=1	53.6%	53.6%	51.9%	51.9%
	C=10	53.6%	53.6%	55.4%	55.4%

5 REMARKS

When considering features and the ability of a benchmark to reach a consistent steady state, we found that the correlation reaches a maximum of 10.9% for Pearson's correlation coefficient. This leads us to believe that source code features could help in predicting the ability of a benchmark to reach a consistent steady state.

When considering benchmarks that consistently reach a steady state, we found that the correlation of a feature to the number of iterations it takes to reach a steady state is generally much higher at a maximum of 21.6% for Spearman's correlation coefficient. This could indicate that there is a stronger influence of source code features on the time it takes to reach a steady state.

It is interesting to note the commonalities between the three techniques to acquire a correlation coefficient. LastFormalParameter seems to have a consistently high influence on the iterations it takes to reach a steady state, while other features tend to appear in at least two techniques. Kendall and Spearman also have similar distributions across all their coefficients.

Table 2 contains the results of our machine learning models. In most cases, results on the models matched the performance on unfiltered features. After filtering the correlating features from Spearman's and Kendall's correlation coefficients on a K Nearest Neighbour classifier with 3 neighbours, we can observe an impressive 12.5% increase in accuracy. The largest decrease in accuracy with filtering features is 7.1% when considering Random Forest with an entropy criterion and Pearson's correlation coefficient.

Figure 4 shows the confusion matrix for one of the two highest-performing models. Both models had an identical accuracy (78.6%), f1 score (77% for inconsistent and 80% for steady state) and confusion matrix. This is likely due to the similarity in the correlation coefficients from Kendall's tau and Spearman's rho. The distribution of the confusion matrix in Figure 4 assures us that the model provides accurate predictions based on source code features.

This proves that source code features correlate with steady-state performance. To our knowledge, we could not find similar papers to compare our results, making this the current state of the art.

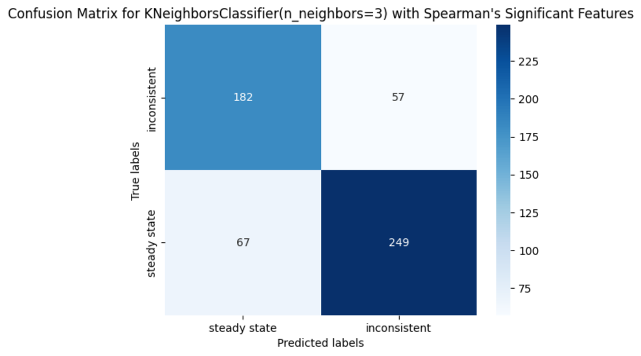

Figure 4: Confusion Matrix for KNearestNeighbours() and n_neigbours=3 with Spearman's Significant Features

6 CONCLUSION

For this research paper, we used the data gathered by Traini et al. [16] to extend their research of steady state analysis to determine whether certain source code features could provide a basis for developers to make more informed predictions on when a steady state would occur.

We found some correlation between source code features and their ability to reach a consistent, steady state and a good correlation between source code features and how long it would take for a benchmark to reach a steady state. Using the K Nearest Neighbour Classifier with features selected with either Spearman's or Kendall's correlation coefficient, we can obtain an accuracy of 78.6%.

As is the problem with most machine learning applications, we could always do with more data [9]. Replicating the study across larger benchmarks might yield better correlations and machine learning results. We aim to have this paper be another milestone in improving the assessment of Java software's steady state performance that Traini et al. proposed in their paper.

REFERENCES

[1] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

[3] David Binkley. 2007. Source code analysis: A road map. *Future of Software Engineering (FOSE'07)* (2007), 104–119.

[4] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[5] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.

[6] Charlie Curtsinger and Emery D Berger. 2013. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 219–228.

[7] David Freedman, Robert Pisani, and Roger Purves. 2007. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York* (2007).

[8] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.

[9] Wilhelmiina Hämäläinen and Mikko Vinni. 2006. Comparison of Machine Learning Methods for Intelligent Tutoring Systems. In *Intelligent Tutoring Systems*, Mitsuru Ikeda, Kevin D. Ashley, and Tak-Wai Chan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 525–534.

[10] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[11] Arvinder Kaur and Ruchikaa Nayyar. 2020. A comparative study of static code analysis tools for vulnerability detection in C/C++ and JAVA source code. *Procedia Computer Science* 171 (2020), 2023–2029.

[12] M. G. Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 30, 1-2 (June 1938), 81–93. https://doi.org/10.1093/biomet/30.1-2.81

[13] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. 2021. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering* 26, 6 (2021), 1–53.

[14] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[15] Jared Chad Swanzen. 2023. *Reproduce - Analysing Static Source Code Features to Determine a Correlation to Steady State Performance in Java Microbenchmarks*. https://doi.org/10.5281/zenodo.7646968

[16] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2023. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2023), 1–57.

[17] Hannes Tribus. 2010. Static Code Features for a Machine Learning based Inspection: An approach for C.

[18] Geoffrey I Webb, Eamonn Keogh, and Risto Miikkulainen. 2010. Na"ıve Bayes. *Encyclopedia of machine learning* 15 (2010), 713–714.

[19] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. 2008. Top 10 algorithms in data mining. *Knowledge and information systems* 14, 1 (2008), 1–37.

[20] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of Biostatistics* 7 (2005).