Machine-Learning for Optimal Fitness Function Selection in Automated Testing

Daniela Toader
Supervisor(s): Annibale Panichella, Pouria Derakhshanfar, Mitchell Olsthoorn
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

ABSTRACT

The perpetual desire for more qualitative software has been an excellent incentive for software engineers to create automated tools to ease and improve the process of software testing. EvoSuite is an example of a state-of-the-art tool that synthesises test cases automatically. It uses a genetic algorithm to produce test cases based on given search targets. Previous studies have analysed the performance of single or combinations of targets but have not yet explored the differences between various combinations. In this research, we compare the Weak Mutation + Branch setting to Branch and the Default (combination of eight separate targets) of EvoSuite. We aim to provide insightful information about their differences in branch coverage and mutation scores. Moreover, we discuss machine-learning models that can predict which combination has the highest score (i.e., branch coverage, mutation score) based on characteristics of the tested classes, such as the number of lines of code. Our results highlight that the Weak Mutation + Branch combination outperforms Branch for the mutation score metric and Default for the branch coverage metric. They also show that Weak Mutation + Branch is outperformed by the branch criterion for Branch Coverage and by the Default combination for mutation score. Our findings also cover the performance of the models, having concluded that the Random Forest and Decision Tree classifiers produce the best results and are feasible options for predicting the best combinations from the ones analysed. Finally, static code metrics such as 'wmc', 'loc', and 'mathOperationsOty' often appear as relevant features for our models. We visualise how they influence the most suitable combination of criteria through our Decision Trees.

1 INTRODUCTION

Testing is an essential step in assuring the quality of software, as its leading goal is finding potentially dangerous faults in source code. Writing adequate and meaningful tests is an error-prone and costly task that requires a significant amount of development time [1]. For this reason, different tools have been created to synthesize test cases automatically. For example, EvoSuite [2] is a state-of-the-art tool that generates unit-level test suites for code written in the ubiquitous Java programming language. EvoSuite seeks to optimise (a combination of) coverage criteria either at the test suite or at the single test case level. For this research, test case level optimisation is performed.

EvoSuite uses a genetic algorithm, DynaMOSA [3], which optimises for multiple objectives simultaneously; each corresponding to one test goal to achieve (e.g., branch coverage, statement coverage). It aims to solve optimisation problems by generating tests based on certain fitness functions (i.e., criteria). EvoSuite can optimise eight different criteria (e.g., branch, line, mutation, etc.) [4]. Various approaches have been proposed in literature to combine these criteria with the ultimate goal of discovering combinations that will lead to finding more bugs. However, the differences between such combinations have not been extensively explored.

In this research, EvoSuite has been run alternatively with three different optimisation criteria: Default, Branch Coverage and Branch Coverage + Weak Mutation (`bcwm`). The criteria and their respective impact on the generated test cases are analysed and their results employed to infer patterns between input static code metrics and qualitative output measurements. It is not yet clear which metrics are best given a specific class under test. Defaults have been chosen mainly based on developers' empirical experience.

The lack of concrete information that relates objectives to class properties and consequently to coverage and fault-finding capabilities has given rise to a research gap that needs further investigation. The main aim of this research is to find *how effective the different state-of-the-art fitness functions are at guiding the search process toward finding bugs*. The goal is to create a model that predicts which fitness function is best suited for the classes under test, based on empirical results. Optimising the function selection can lead to a better performance of the algorithm, which can, in turn, improve the fault-finding capabilities of EvoSuite. This research is relevant as tools for automated test generation can enhance the quality assurance process by improving fault detection and saving considerable amounts of coding time [5].

The goal of this research is to answer the following question: *when and how does Weak Mutation increase the number of bugs detected when combined with Branch Coverage?* It is reasonable to answer by breaking it down into two sub-questions:

RQ1. To what extent does Weak Mutation in combination with Branch Coverage affect structural coverage and fault detection capabilities for different search budgets?

RQ2. What is the relationship between class metrics and the structural coverage and fault detection capabilities of the test suites when using Weak Mutation in combination with Branch Coverage?

For this research, we focus on two coverage criteria (and related objectives), namely, Branch Coverage and Weak Mutation. The former is always targeted, as by covering more branches, we increase the chance of finding bugs; the latter is specifically targeted for this research because by making use of mutation, the aim is to optimise fault detection over coverage, which can result in finding more bugs. Weak Mutation also has an advantage over Strong Mutation because it is significantly less expensive to perform while providing good capabilities.

The empirical study we conducted firstly consists of data processing: data gathering, data balancing, and class metrics extraction. The class metrics become machine-learning model features. Before training the models, we perform feature selection to reduce the dimensions of our metrics. Furthermore, we focus on training the models for inferring patterns. The patterns concern relationships between the collected metrics and the best performing combination of optimisation criteria for EvoSuite. All of the discussed models are classifiers that analyse the best performing combination based on branch coverage or mutation score results. We create classifiers for every search budget initially used with EvoSuite. Finally, we evaluate our models and provide visualisations of the inferred relationships.

Our study provided results in terms of both statistical significance of score metrics results and also of F1-scores for every classifier. These two result categories delivered the answers to our research questions and provided new information that can improve function selection in EvoSuite. We investigated the significance of the branch coverage and mutation score differences between the chosen combinations and found out that over 200 classes showed statistically significant differences with non-negligible (i.e., large, medium and small) effect sizes across all configurations. Given that a reasonable amount of our data proved to be significant, we have continued by training four main models for predicting the best combination from our selection. The assigned labels described which combination was better and, if equivalent, which tuples are equivalent. Labels were assigned by assessing the median scores (i.e., branch coverage and mutation score) and picking the best performing combination(s). We evaluated the models with average F1-scores across different configurations of (0.91, 0.73, 0.94, 0.56) for (Decision Tree, Support Vector Classifier, Random Forest and Logistic Regression). Visualisations of the models provide information on how to pick a combination based on the properties of the tested classes.

The structure of the paper further consists of the following six sections. Firstly, Section 2 describes different terms used throughout the research. Secondly, Section 3 shows the methods used for answering the main questions. Thirdly, Section 4 reveals findings that emerged from applying the methods, which are discussed in Section 5 along with some limitations. Moreover, threats to validity frequently appear in research, and this paper is no exception, particularly receiving attention in Section 6. Furthermore, the responsible research topic points to ethical implications outlined in Section 7. Lastly, Section 8 draws conclusions and explores future work.

2 BACKGROUND

This section summarizes the main background concepts used in this paper and describes their meaning in the context of the research.

Mutation testing. A form of testing that involves making syntactic changes to statements to generate "mutants" of the original source code. They are meant to behave similarly to real commonly occurring faults. Each one of the mutants contains a single fault and their goal is to make the tests fail to show the effectiveness and robustness of the test suite. A mutant is said to be "killed" if the test suite passes on the original program but fails on the mutant. This indicates that the test suite is of "higher quality" as it can detect the artificial fault. Instead, a mutant is "alive" if the test suite passes on both the original and the mutated program [6].

Weak and Strong mutation. In weak mutation, a mutant is killed "*if the execution of the test on the mutant is observably different from its execution on the original SUT, that is, if state infection is reached by the test*" [7], where SUT is the software under test. In comparison, strong mutation additionally requires the program to output a different result than expected. Thus the infected state must propagate to the output and to the test assertion to be detected by the test. It is more computationally intensive compared to weak mutation [6].

Mutation score. The mutation score is defined as the number of killed mutants (M_k) divided by the number of total mutants (M_t) minus the equivalent mutants (M_{eq}): $\frac{M_k}{M_t - M_{eq}}$. The equivalent mutants are the ones that cannot be killed by any test and are said to be equivalent to the original program. They "*change the program's syntax, but not its semantics, and thus are undetectable by any test*" [6]. The weak mutation objectives measure the distances of a test case t to weakly-kill each mutant injected in the program under test. Hence, there is one search objective (to be optimised) per mutant. The mutation score provides information about the fault detection capabilities of the test suite.

DynaMOSA. The DynaMOSA algorithm is a many-objective search-based evolutionary algorithm which dynamically focuses the search on a subset of the uncovered targets based on the control dependency hierarchy [3]. Its distinguishing feature is that it approaches each target (coverage criteria) independently, utilising a hierarchy of dependencies to dynamically focus on targets during execution. More recently, Panichella et al. [8] proposed a multi-objective version of DynaMOSA that leverages improved hierarchial dependency analysis. The latter variant has been shown to produce results with higher structural and mutation coverage [9], and it is the default search algorithm employed by EvoSuite.

Related work. Research such as [4] and [10] combined different testing criteria and looked at their effectiveness. [11] showed that there is no combination of criteria that performs best for finding faults and that firstly, the code structure needs to be observed (through a criterion like `branch`) and secondly, supported by additional targets. They also showed that most criteria detect different faults better and that depending on the types of faults, certain criteria find more bugs. However, these insights have not yet provided complete comparisons of combinations, while also showing clear patterns where specific criteria can have better results. This corresponds to the research gap we are investigating in this paper.

3 METHODOLOGY

This section highlights the main steps we performed in the research process. The methodology for measuring the impact of fitness function selection on the resulting branch coverage and mutation score is discussed. Every step of the workflow is outlined in the subsections below.

3.1 Investigation and data gathering

Firstly, we have researched and analysed the EvoSuite fitness functions. We investigated Weak Mutation, Branch Coverage and the Default combination to allow an easier understanding of the results. The `default` combination consists of the line, branch, weak mutation, output, method, method no exception, and cbranch criteria. Also, using the static code metrics, our goal was to link the `bcwm` (i.e., Weak Mutation + Branch Coverage) combination to the classes that can benefit most from it in terms of structural coverage and mutation score.

We ran EvoSuite on a set of 338 different classes belonging to the SF110 corpus [1] as well as the Apache Commons [2]. The former

[1] https://www.evosuite.org/experimental-data/sf110/
[2] https://commons.apache.org/

is composed of 110 projects of SourceForge, a popular open-source repository. The latter is a project consisting of the most common Java components. EvoSuite's output provided the branch coverage of the generated test suites as well as the mutation score results. We also used the set of classes to extract static code metrics, process described in Section 3.4. We ran the experiments using search budgets of 60, 180 and 300 seconds for calculating the branch coverage and a budget of 60 seconds for the mutation score for the selected classes. The budgets have been chosen based on literature recommendations from [3], [12], [13], and [14]. EvoSuite has been independently run ten times for each class to prevent outcome consistency issues. Potential inconsistencies could come from the stochastic nature of the evolutionary algorithm used by EvoSuite. Occasional differences in outputs can influence our results if variations are high, thus we chose to alleviate such variations by reiterating the run.

3.2 Medians comparison

Secondly, we investigated EvoSuite's output. We highlighted the differences between the `bcwm`, `branch` and `default` combinations. We did this by comparing median values of the branch coverage and mutation score over ten runs for each class between `bcwm` and the other two combinations. We first compared the forenamed differences to gain insights into the overall performance without looking at the properties of the examined classes. We did this for all considered budgets of 60, 180 and 300 seconds for branch coverage (`branch_60`, `branch_180`, `branch_300`) and of 60 seconds for the mutation score (`mutation_score`).

3.3 Statistical significance and effect size analysis

Furthermore, to construct a model that predicts which fitness function is most effective, we had to reason about the significance of the score differences between `bcwm` and the others. We performed the Wilcoxon signed-rank statistical test per class over ten runs on the coverage and mutation score of `bcwm` vs `branch` and `bcwm` vs `default`. This test provided the statistical significance of each batch of ten runs per class of score differences. We were only interested in the classes with a p-value smaller than 0.05, i.e., in the classes with statistically significant differences in mutation score (or branch coverage) among the different configurations.

To measure the effect size of the differences between `bcwm` and the other two settings, we used the Vargha-Delaney $\hat{A}_{12}$ statistical method [15]. This metric allows us to understand the magnitude of the score differences between the fitness functions. For example, when looking at the branch coverage for a search budget of 60 seconds, $\hat{A}_{12}(\texttt{bcwm}_{10runs\_class1}, \texttt{default}_{10runs\_class1})$ will provide the $\hat{A}_{12}$ statistics estimate and the magnitude of the effect size. If the estimate is larger than 0.5, then `bcwm` dominates the other set of results, which translates to `bcwm` having better results (i.e., branch coverage and mutation score). The magnitude provides an insight into how great the dominance is. Thus, samples with a "negligible" magnitude are not of importance to our analysis, while a "large" magnitude has high relevance. We simultaneously applied both tests on each batch of ten runs per class. Through this method, we gathered the significant classes with large, medium, and small

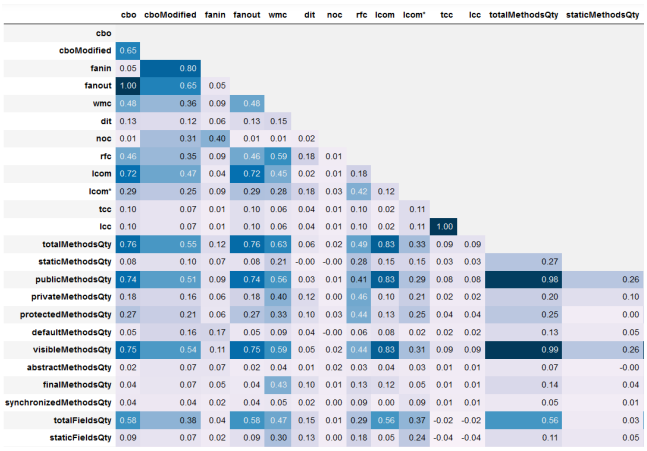


Figure 1: Sample of the class metrics correlation matrix

effect sizes for `bcwm` vs `branch` and `bcwm` vs `default` for all search budgets for branch coverage and mutation score, respectively.

3.4 Class metrics extraction and feature selection

To answer the research questions, it is important to understand in which contexts (program characteristics) one configuration is better than another. To discover these patterns, we relied on software metrics and machine learning. To reason about the influence of the input classes' characteristics on the fitness function selection, we have taken into account the input class metrics. We ran two metric extraction tools to obtain the class properties: `ck` [3] and `ckjm` [4]. The former functioned better for our set of classes and resulted in more metrics computed for most of them. We noticed the metrics have a high dimensionality as there are 49 in total. They are also highly correlated, as seen in the correlation matrix in Figure 1.

To avoid introducing bias into the models through highly correlated properties, we performed feature selection by using the `SelectKBest` method of the `sklearn.feature_selection` module. It has been applied with the `f_classif` parameter, which compares samples by calculating the ANOVA F-value for the provided sample. The F-value estimates the degree of linear dependencies between two random variables. Thus, it would calculate the difference between the means of the class distributions [16], [17]. The higher the difference, the better the feature discriminates between the given classes. We chose this method thanks to its ability to discriminate well between features. We have also examined the `mutual_info_classif` function, which measures the joint mutual information, i.e., the amount of information about one random variable one can extract from another random variable. The `mutual_info_classif` relies on entropy estimations from k-nearest neighbours distances [18], [19]. Empirical observations of our models and datasets have shown similar scores for the final models between the two selection approaches. As such, we mainly

[3] https://github.com/mauricioaniche/ck
[4] https://github.com/dspinellis/ckjm

considered the first one for the described experiment. We focused on 3, 5, 10 and 15 features to maintain model visualisation accessible.

3.5 Data balancing and deciding on models

Lastly, we chose to train four models to predict which fitness function to choose based on class properties: Decision Tree (DT), Random Forest (RF), Support Vector Classification (SVC), and Logistic Regression (LR). The class metrics are encoded into features and the criteria combinations were turned into labels. The label assigning procedure is detailed in Section 3.6. The dataset was constructed from the analysed significant classes and their properties. To tackle the issue of especially imbalanced class distributions, we performed data balancing techniques to prevent low accuracy/F1-scores for the infrequent classes. The two performed techniques were over and under sampling using the `RandomOverSampler` and `RandomUnderSampler` from the `imblearn.over_sampling` library. Oversampling is a technique that adds samples for the classes that are in minority and undersampling removes samples of the majority class samples to equalize the amount of samples across the classes [20].

3.6 Assigning labels

When constructing classifiers, an essential step is assigning labels to each data point. For this work, we assign labels based on the median of the structural coverage for a specific fitness function across 10 runs. For example, if class `X` has the median coverages $< \texttt{bcwm}_m, \texttt{branch}_m, \texttt{default}_m > = < 0.9, 0.3, 0.8 >$, then that class would be assigned the label `bcwm`. However, ambiguity sets in when two values are equal. Since we pick classes based on the statistical difference between pairs of combinations, it is possible for any two coverages to be identical.

To assign labels to these ambiguous data points, we tested three options: (1) removing the points from the data set altogether, (2) assigning a random label, (3) deterministically assigning a label based on the ordering or the fitness functions. We discarded the first option on account of removing too many data points, which would affect the robustness and generalisability of the results. We also discarded the second option because of the flakiness of the obtained results: a model trained on randomly generated labels does not provide much value for real-world applications. Furthermore, we also discarded deterministically assigning labels for ambiguous points. This introduces certain biases according to the ordering we choose and it affects the model's performance. This ordering would be used as a tie-breaker, but it is still significant, as it occurs in 53.1% of the selected data points.

Finally, we decided to solve the ties by designing for the equivalence of the labels, namely approach (4). Thus, new labels have been assigned for the equivalent scores: `equivalent_bcwm_branch`, `equivalent_bcwm_default`, `equivalent_branch_default`, and `equivalent_equivalent_all`. They provide a bias-free method of breaking ties and are descriptive in terms of the ties occurring during prediction.

3.7 Evaluating the models

To validate and evaluate how well the models generalise to independent datasets, we used the k-fold cross-validation technique. This method randomly splits the data set into k equally sized so-called folds, one of which is used as the test set. The model is trained on the remainder $k-1$ folds, such that each fold serves as the test set in exactly one iteration of the algorithm. The reported results are the average F1-score of a nested k-fold cross-validation with Grid Search to also tune each model's hyperparameters. Grid Search is an exhaustive search technique that aims to find the optimal model hyperparameters and is widely used in practice [21], [22]. The outer cross-validation has been done with $k = 10$ runs performed over the data sets, and the inner with $k = 3$ to evaluate the Grid Search.

F1-score. The F1-score metric is used to compare the performance of classifiers. It is defined as the harmonic mean of the precision and recall of the model. Where precision is the number of true positives (TP) divided by the number of false positives (FP) plus true positives. Recall is the number of true positives divided by the number of true positives plus false negatives (FN).

$$F_1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

4 RESULTS

In this section, we answer the research questions put forward in Section 1 by presenting the results of our empirical evaluation.

4.1 Impact of function selection on structural coverage and fault detection capabilities

To capture how large the impact of choosing different fitness functions on branch coverage and mutation score is, we have analysed the performance and (per-class) significance of the score metrics. For three of the 338 classes, EvoSuite did not produce results for all configurations, and those are thus considered to yield 0.0 branch coverage. Appendix A provides boxplots that reveal the distributions of the branch coverage and mutation scores for the available search budgets for the analysed combinations. The results suggest that there is a noticeable difference both in terms of mean and IQR between `budget_60` and the higher budget results, across all criteria combinations considered. `budget_180` and `budget_300` show very similar results of all tested statistics, both outperforming the lowest budget dataset. Moreover, for all three budgets, it is apparent that the differences across targets are rarely significant over the three criteria. The highest discrepancy is when using the `branch` criterion in the `budget_60` setting, which marginally outperforms its counterparts. The mutation score data indicates that EvoSuite behaves similarly to the branch coverage scenario of the same time budget, with the difference that `default` combination of criteria marginally outperforms its counterparts in terms of mean score.

To better understand the (per-class) variations of the combinations' results, Table 1 shows the differences between the three considered ones. `bcwm` produces better structural coverage than `branch` in (43, 48, 44) instances, and worse in (105, 94, 84) for budgets (60, 180, 300), respectively. On the other hand, `bcwm` performs better than `default` in (124, 102, 82) cases, and worse in (33, 36, 30) for budgets (60, 180, 300) seconds.

Optimising for bcwm produces better mutation scores than for

Table 1: Comparison of median structural coverage and mutation score of bcwm against branch and default. The results are obtained after 10 runs of EvoSuite per class, for all score metrics.

Configuration	Comparison	bcwm better	Equal	bcwm worse
branch_60	vs branch	43	188	105
	vs default	124	178	33
branch_180	vs branch	48	194	94
	vs default	102	195	36
branch_300	vs branch	44	206	84
	vs default	82	217	30
mutation_sc	vs branch	134	113	72
	vs default	50	105	163

branch in 134 instances, and worse in 72. On the other hand, bcwm performs better than default in 50 cases, and worse in 163. This indicates that there exists a large number of classes which might benefit from the choice of an appropriate fitness function.

A notable point is the number of times bcwm gives similar results in terms of structural coverage (over half of the time for every budget) to branch and default. For configurations (branch_60, branch_180, branch_300, mutation_score) the bcwm was equivalent to branch in over (55%, 57%, 61%, 35%) of the comparisons and to default in over (53%, 58%, 64%, 33%) of the comparisons. There are many instances in which bcwm performs worse than the other two criteria. This indicates that adding more criteria (i.e. weak mutation) to optimise can harm the resulting coverage for certain classes. In numerous cases, bcwm gives higher coverage. These observations provide an incentive to further look into the properties of the input classes. Selecting the most effective fitness function can depend on them and not only on the number of criteria to optimise.

In terms of (per-class) significance of the scores differences between bcwm and branch and bcwm and default, the Wilcoxon test showed that in more than 59% of cases, the classes had a p-value below 0.05 for all configurations. The results are presented in Table 2. For configurations (branch_60, branch_180, branch_300),

Table 2: Comparison of the number of classes for which there are statistically significant differences in branch coverage and mutation score for bcwm when compared to branch and default, respectively. The Inters. column gives the number of classes for which there are statistical differences between both pairs of objectives.

Configuration	bcwm-branch	bcwm-default	Inters.
branch_60	228	213	182
branch_180	218	216	178
branch_300	275	200	178
mutation_sc	319	318	318

the class significance for bcwm compared to branch was (67%, 64%, 82%) of the analysed classes and for mutation_score, it was 100%. Compared to the default combination (branch_60, branch_180, branch_300), the significance results were (63%, 64%, 59%), while for mutation_score they where 99%. This demonstrates that a relatively large proportion of the initial dataset provides significant results and can thus further be analysed and used in the machine-learning models.

In terms of (per-class) significance of the scores differences between bcwm and branch and bcwm and default, over 170 classes of every configuration proved to not only be statistically significant ($p < 0.05$), but also to have a large effect size, given by the Vargha-Delaney $\hat{A}_{12}$ statistic. The numbers of statistically significant classes for each configuration that satisfy both of the aforementioned conditions are summarised in Table 3.

Table 3: Comparison of the number of classes for which there are statistically significant differences and large effect sizes in branch coverage and mutation score for bcwm when compared to branch and default, respectively.

Configuration	bcwm-branch	bcwm-default	Inters.
branch_60	189	174	139
branch_180	189	186	144
branch_300	238	176	137
mutation_sc	319	318	318

To further develop on the effect size of the analysed data, Appendix B (a), (b) illustrates the magnitude of the effects in terms of the number of classes with "negligible", "small", "medium", and "large" for each score metric and budget available. This visualisation represents the data before filtering on the p-value. For the (branch_60, branch_180, branch_300, mutation_score) configurations of bcwm vs default, over 65% of the classes have a large effect size. Between 9% and 19% of the classes show negligible magnitude and will not be taken into consideration for the machine-learning models. The small and medium effect sizes together account for between 4% and 13% of the data. For the branch coverage score metric, the number of classes with large effect sizes increases along with the search budget, reaching over 86% of the data. Appendix B (c) and Appendix B (d) show that after applying filtering on p-value < 0.05, most classes with negligible and small effect sizes have been excluded. Between 1 and 5 classes with small magnitude, and over 195 classes with large magnitude are kept and further used for the models. This means that over 60% of the data can be passed on for all configurations.

The $\hat{A}_{12}$ statistics provided not only the effect sizes but also the dominance relationships between the different combinations for all configurations. In Figure 2, we can observe that the branch setting is better for all configurations in terms of branch coverage, while for the mutation score, bcwm dominates in 113 cases, compared to 94 cases of being dominated. Against the default setting, as seen in Figure 3, bcwm dominates for all configurations for branch coverage, while for mutation score, it is dominated in 127 cases, compared to 83 cases where it dominates. The $\hat{A}_{12}$ statistics show similar

relationships with the results from Table 1, which backs up our findings in terms of the significance of our dataset.

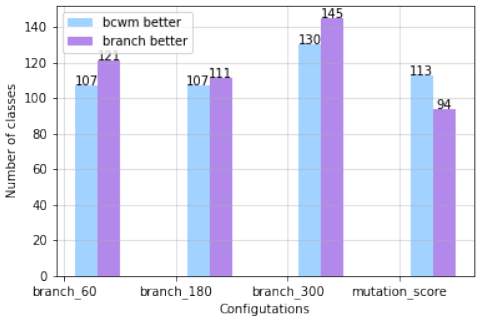


Figure 2: Dominating combinations in terms of numbers of analysed classes filtered by p < 0.05 - `bcwm` vs `branch`

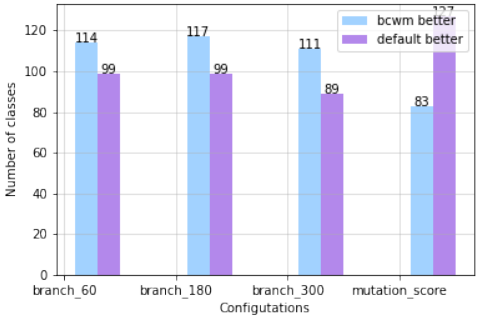


Figure 3: Dominating combinations in terms of numbers of analysed classes filtered by p < 0.05 - `bcwm` vs `default`

For the tested instances, `bcwm` produces results that are worse than or as good as `branch` in 86.5% of cases in terms of branch coverage. `bcwm` also produces results that are at least as good as `default` in 90.0% of cases. In terms of mutation score, `bcwm` is at least as good as branch in 77.4% of instances and equal to or worse than `default` in 84.2%.

4.2 The relationship between class metrics and the structural coverage and fault detection capabilities when using `bcwm`

To capture the relationship between the different fitness functions and structural coverage and mutation score, we trained four classifiers based on the static code metics of classes that show statistically significant differences from our data set. We trained the following models: a Decision Tree (`DT`) a Support Vector Classifier (`SVC`), a Random Forest (`RF`), and a Logistic Regression (`LR`). We compared their effectiveness using the standard F1-score. The results of k-fold cross-validation are provided in Table 4. These results are the product of approach (4) of assigning labels. The results are also a product of hyperparameter optimisation through Grid Search for every model and every configuration. The numbers of features (static code metrics) considered were 3, 5, 10, and 15. The best parameter settings can be found in Appendix C.

Table 4: Comparison of the the performance of four ML models at the task of classifying the statistically significant classes in terms of the best fitness function. The classification is done on the basis of which combination of criteria results in the best structural coverage for a class.

	F1-Score of Model			
Score Metric	DT	SVC	RF	LR
Branch 60	0.89	0.76	0.93	0.57
Branch 180	0.91	0.74	0.95	0.57
Branch 300	0.94	0.71	0.97	0.49
Mutation Score	0.90	0.71	0.93	0.63

The results indicate the `DT` and `RF` perform the best out of the four alternatives. They both perform well with a maximum difference in F1-score of 0.4 for all instances, with a minimum F1-score of 0.89. `SVC` and `LR` perform worse, with F1-scores of between 0.49 and 0.74. Across the different time budgets and also different score metrics, the performance of the models varies between 0.04 and 0.14 per classifier. The number of features selected in the preprocessing step does not tremendously influence the score of the DT and RF, having scores of at least 0.86 when selecting 3 features. On the other hand, when using 3 or 5 features for SVC and LR, the best scores are between 0.27 and 0.51 for all the configurations, which is significantly worse than the other presented options.

Given the DT model has shown valuable scores, we have chosen it to visualise the final patterns between static code metrics of the examined classes and the best combination to select between `bcwm`, `branch` and `branch`. To display two examples of the produced patterns, Figure 4 illustrates how metrics ("loc", "mathOperationsQty"), and ("loc", "wmc", "mathOperationsQty") lead to labels `default` and `bcwm`, respectively. We chose two of the best DTs with 5 and 3 features, respectively, to showcase the essential features in an easy-to-visualise fashion. The left model of Figure 4 had an F1-score of 0.90 and represents the `branch_180` configuration. The right model scored 0.87 for the `branch_60` configuration. Given such diagrams, one can efficiently discover relationships between metrics and the most suitable functions. Some of the best performing models also included the "cbo", "fanout", "rfc", "returnQty", "loopQty", "comparisonsQty", and "abstractMethodsQty" metrics. All results can be found under the project's public repository [5].

The `DT` and `RF` classifiers can effectively predict which of the three fitness functions would produce the coverage for a given class with a mean F1-score of between 0.89 and 0.97. The `SVC` and `LR` models perform worse with scores between 0.49 and 0.76. The most commonly selected features are "loc", "wmc", "mathOperationsQty", "cbo", "fanout", "rfc".

[5] https://github.com/danielatoader/research-project

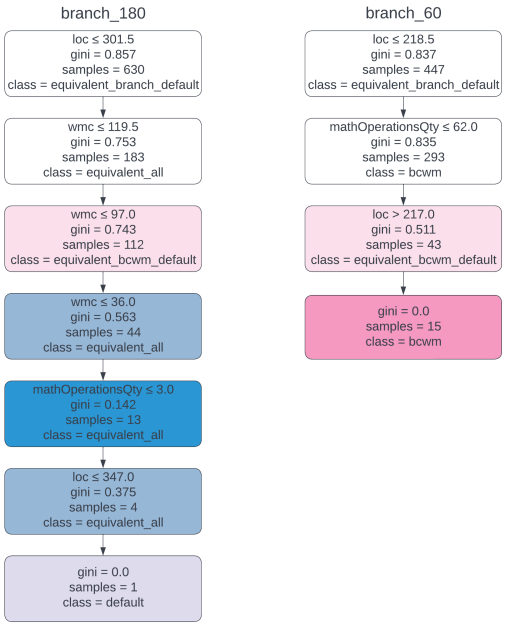


Figure 4: Decision Tree path examples of the `default` and `bcwm` labels for the `branch_180` and `branch_60` configurations, respectively

5 DISCUSSION

Our analysis indicates that based on the `bcwm`, `branch`, and `default` combinations, one can on average achieve the best results by using `branch` rather than `bcwm` when aiming to achieve the highest branch coverage. When the goal is achieving the highest mutation score, one should choose `bcwm` instead of `branch`. After observing the `bcwm` and `default` results, we recommend using `bcwm` when targeting the highest branch coverage and `default` for mutation score. We concluded that adding more functions to the combinations does not guarantee better performance, and the selection should depend on more factors than simply the number of optimised criteria. The models we trained, for instance, the Decision Trees, can provide insight into the conditions under which combinations of criteria are most effective. Such conditions reflect in class metrics such as the class complexity, their mathematical contents, their length, and so on. The models can be incorporated into EvoSuite and help software developers produce appropriate test cases for their specific needs. Like in most research, we encountered limitations that prevented us from having a more extensive analysis such as computational power and time shortages. These mainly showed up during training and performing the nested k-fold cross-validation with Grid Search. Our results can potentially see improvements once the scripts are run on more powerful machines. An additional limitation is not being able to link our models to EvoSuite, which means a lack of performance guarantees in practice. This could be overcome by extending the research, checking the linkage results, and reporting the outcome.

6 THREATS TO VALIDITY

Threats to internal validity concern factors that can significantly influence our results. One threat commonly shared by most ML approaches is the randomness associated with the underlying models and the distribution of the data used as training and testing instances. Even for deterministic models like decision trees, the manner in which the data is partitioned can significantly affect the quality of the results. To minimize those effects, we validated our models using the standard k-fold cross-validation technique. In total, we performed ten runs for each model. Another source of instability for many ML models originates from tuning the algorithm's parameters. To limit the influence of parameter settings, we opted for default parameter settings recommended by the ML framework wherever possible.

Threats to external validity concern the degree to which our results are generalisable. We selected a subset of 338 Java classes from the SF110 data set and Apache Commons and ran the models on the statistically significant subsets of sizes of approximately 200 samples with a "large" effect magnitude. This data set lends itself naturally to our work, as it has been used in many previous works on test case generation [12], [6], [23]. However, a broader set of benchmarks that extends beyond this data set would improve the robustness of the results.

Threats to conclusion validity concern factors that may influence the connection between our approach and our conclusion. We performed statistical tests in selecting classes from the SF110 data set that had statistically significant differences between at least one pair of different fitness functions. We also performed statistical tests to draw conclusions based on the F1-scores of the different models we obtained using k-fold cross-validation. The conclusions we drew are based on the statistical significance obtained using the Wilcoxon test and on the Vargha-Delaney A statistics.

7 RESPONSIBLE RESEARCH

Ethical implications of our software concern the manner in which end users might utilize our work. In a real-world setting, the decision-maker could either use the output of our models to choose a specific configuration for the intended classes under test manually or delegate the decision process to our application entirely. In either case, the user should be aware that our models are statistically driven and imperfect. As such, there are no guarantees of optimality for our results. However, given the stochastic nature of the evolutionary search algorithms underlying EvoSuite, we believe that this is a reasonable proposition: providing statistical guarantees for such stochastic algorithms on arbitrary inputs is beyond the scope of this work.

Reproducibility concerns the degree to which our experiments and results can be recreated. Our benchmark instances are selected from a widely used and open-source data set, and as such can easily be retrieved and reused. The tools used to extract the features from these instances are also open sources. Furthermore, we will make the code used to generate, train, and validate the models publicly available. Instances of the code where pseudo-random number generators are used will be seeded such that the results can be replicated precisely.

8 CONCLUSION AND FUTURE WORK

This research has analysed how class metrics of a subset of classes from the SF110 data set and the Apache Commons influence structural coverage and fault detection capabilities given the Weak Mutation combined with Branch Coverage fitness function `bcwm` when using EvoSuite.

In this research, we focused on bringing insights into the impact of the Weak Mutation + Branch Coverage setting of EvoSuite on the tests' capabilities of finding bugs. To reach such insights, we ran EvoSuite 10 times on a collection of 338 classes from the SF110 and Apache Commons datasets for budgets of 60, 180 and 300 seconds. We then compared Weak Mutation + Branch Coverage to the `branch` and `default` settings in terms of branch coverage and mutation score. Further, we analysed the significance and effect size of the comparisons. We filtered the classes on significance and on large, medium and small effect sizes, thus discarding the negligible effect size classes. Using the filtered data, we trained machine-learning models to infer relationships between static code metrics of the classes and their most suitable combination in EvoSuite. To gather the metrics, we ran extraction tools and performed data balancing techniques to prevent introducing bias into our models.

We also performed feature selection to ensure the resulting relationships are visualisable. Our results highlighted that the Weak Mutation + Branch Coverage combination is a better option on average than the `default` for branch coverage, and better than `branch` for mutation score. The machine-learning models had F1-scores of between 0.89 and 0.93 for all configurations for the Decision Tree and Random Forest. The Support Vector Classifier and Logistic Regression had worse scores (SVC between 0.71 and 0.76, LR between 0.49 and 0.63. depending on the configuration. As they were clearly outperformed by the other two, they received less attention in this experiment, but their results can be found in the project repository [6]. The Decision Tree was chosen for means of visualisation, while the Random Forest had the highest F1-scores.

Further possible improvements include performing more feature selection or feature extraction methods on the calculated class metrics and adding more models to the analysis. It would also be intriguing to consider even more search budgets. The experiments could be performed on a broader range of instances to provide more robust and generalisable insight. The experiments could also be replicated with additional fitness functions for comparison purposes.

REFERENCES

[1] Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

[2] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[3] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[4] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[5] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *2008 The Third International Conference on Software Engineering Advances*, pages 252–257. IEEE, 2008.

[6] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[7] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, 2017.

[8] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*, pages 309–324. Springer, 2018.

[9] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.

[10] Gregory Gay. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*, pages 65–82. Springer, 2017.

[11] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019.

[12] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, 2018.

[13] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1224–1228. IEEE, 2020.

[14] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie Van Deursen, and Annibale Panichella. Good things come in threes: Improving search-based crash reproduction with helper objectives. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 211–223. IEEE, 2020.

[15] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[16] Kevin J Johnson and Robert E Synovec. Pattern recognition of jet fuels: comprehensive gc× gc with anova-based feature selection and principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 60(1-2):225–237, 2002.

[17] Hao Lin and Hui Ding. Predicting ion channels and their types by the dipeptide mode of pseudo amino acid composition. *Journal of theoretical biology*, 269(1):64–69, 2011.

[18] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Erratum: estimating mutual information [phys. rev. e 69, 066138 (2004)]. *Physical Review E*, 83(1):019903, 2011.

[19] Brian C Ross. Mutual information between discrete and continuous data sets. *PloS one*, 9(2):e87357, 2014.

[20] Bee Wah Yap, Khatijahhusna Abd Rani, Hezlin Aryani Abd Rahman, Simon Fong, Zuraida Khairudin, and Nik Nik Abdullah. An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets. In *Proceedings of the first international conference on advanced data and information engineering (DaEng-2013)*, pages 13–22. Springer, 2014.

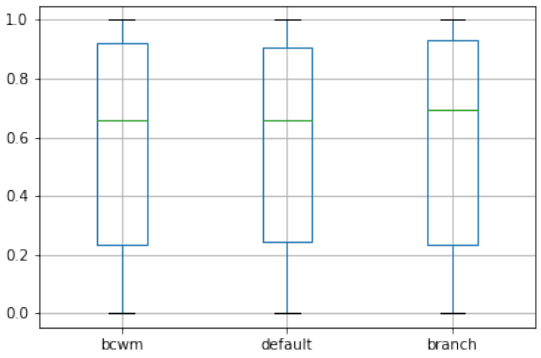
[21] Iwan Syarif, Adam Prugel-Bennett, and Gary Wills. Svm parameter optimization using grid search and genetic algorithm to improve classification performance. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 14(4):1502–1509, 2016.

[22] Fabrício José Pontes, GF Amorim, Pedro Paulo Balestrassi, AP Paiva, and João Roberto Ferreira. Design of experiments and focused grid search for neural network parameter optimization. *Neurocomputing*, 186:22–34, 2016.

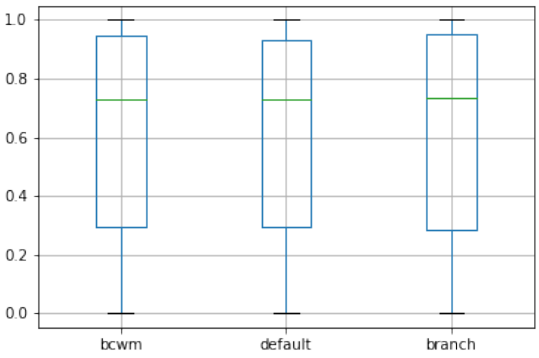
[23] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[6] https://github.com/danielatoader/research-project

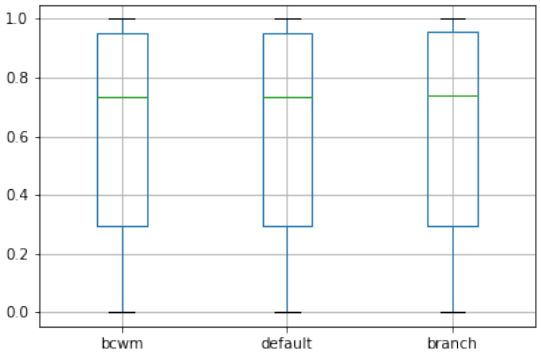
A COVERAGE AND MUTATION SCORE



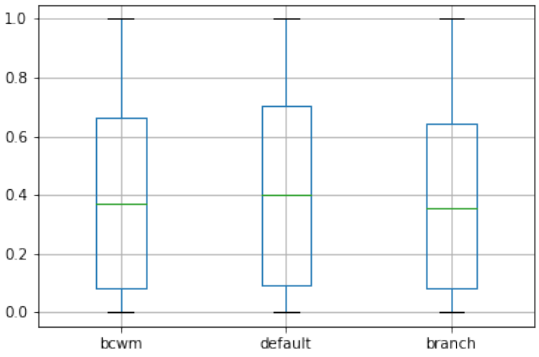
(a) Coverage 60



(b) Coverage 180

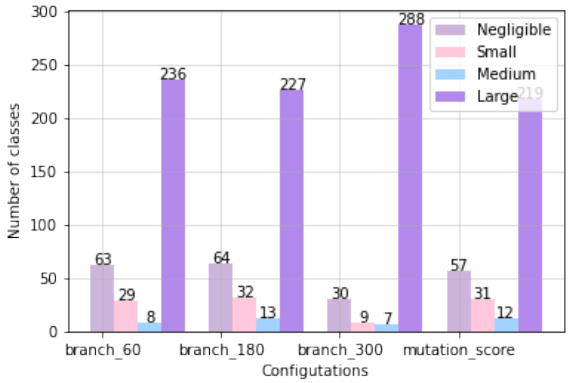


(c) Coverage 300

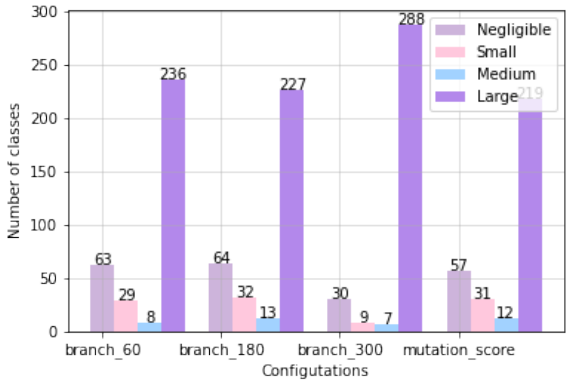


(d) Mutation Score

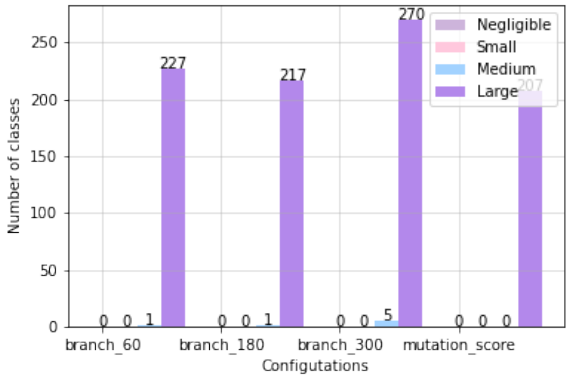
B EFFECT SIZE GRAPHS



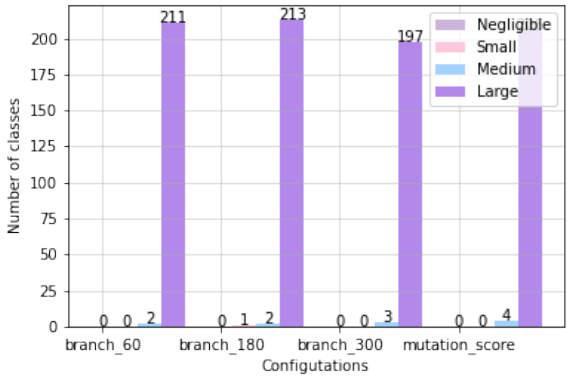
(a) Magnitudes of effect per configuration in terms of numbers of analysed classes - `bcwm` vs `branch`



(b) Magnitudes of effect per configuration in terms of numbers of analysed classes - `bcwm` vs `default`



(c) Magnitudes of effect per configuration in terms of numbers of analysed classes filtered by p < 0.05 - `bcwm` vs `branch`



(d) Magnitudes of effect per configuration in terms of numbers of analysed classes filtered by p < 0.05 - `bcwm` vs `default`

C BEST GRID SEARCH PARAMETERS

Table 5: Best Decision Tree parameters as evaluated by the Grid Search

	Best parameters of DT
Configuration	DT
Branch 60 - 15 features	"criterion" : "entropy", "max_depth" : 20, "max_features" : "sqrt", "max_leaf_nodes" : null
Branch 180 - 15 features	"criterion" : "entropy", "max_depth" : 15, "max_features" : "sqrt", "max_leaf$_n$odes" : null
Branch 300 - 15 features	"criterion" : "entropy", "max_depth" : 15, "max_features" : "sqrt", "max_leaf_nodes" : null
Mutation Score - 15 features	"criterion" : "gini", "max_depth" : null, "max_features" : 4, "max_leaf_nodes" : null

Table 6: Best Support Vector Classifier parameters as evaluated by the Grid Search

	Best parameters of SVC
Configuration	SVC
Branch 60 - 15 features	"kernel" : "linear", "C" : 1.0
Branch 180 - 15 features	"kernel" : "linear", "C" : 1.0
Branch 300 - 3 features	"kernel" : "linear", "C" : 1.0
Mutation Score - 10 features	"kernel" : "linear", "C" : 1.0

Table 7: Best Random Forest parameters as evaluated by the Grid Search

	Best parameters of RF
Configuration	RF
Branch 60 - 15 features	"max_features" : 2, "n_estimators" : 20
Branch 180 - 15 features	"max_features" : 2, "n_estimators" : 50
Branch 300 - 15 features	"max_features" : 2, "n_estimators" : 50
Mutation Score - 10 features	"max_features" : 2, "n_estimators" : 50

Table 8: Best Logistic Regression parameters as evaluated by the Grid Search

	Best parameters of LR
Configuration	LR
Branch 60 - 15 features	"C" : 0.1, "penalty" : "l2", "solver" : "newton − cg"
Branch 180 - 15 features	"C" : 1e − 05, "penalty" : "none", "solver" : "newton − cg"
Branch 300 - 15 features	"C" : 0.1, "penalty" : "l2", "solver" : "newton − cg"
Mutation Score - 15 features	"C" : 100, "penalty" : "l2", "solver" : "liblinear"