Training a Machine-Learning Model for Optimal Fitness Function Selection with the Aim of Finding Bugs

Stoyan Dimitrov
Supervisor(s): Annibale Panichella, Mitchell Olsthoorn, Pouria Derakhshanfar
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

To ensure that a software system operates in the correct way, it is crucial to test it extensively. Manual software testing is severely time-consuming, and developers often underestimate its importance. Consequently, many tools for automatic test generation have been developed during the past decade. EvoSuite is a state-of-the-art tool for automatic generation of unit tests. It can produce test suites based on chosen coverage criteria, also known as a fitness function. Previous studies have widely assessed the performance of the different fitness functions available in EvoSuite. However, the combination of various coverage criteria has not been considered. In this paper, we assess the effectiveness of the combination of Branch coverage and Output diversity fitness functions. We compare it to two of the most popular fitness functions in EvoSuite - Branch coverage and the Default configuration (combines eight coverage criteria) to estimate its performance. We developed a machine learning tool that determines which fitness function will achieve better results based on class characteristics. The assessment criteria we consider are branch coverage and fault detection, represented by mutation score. We further examined how the time limit affects the performance of the considered fitness functions. The results have shown that the combination of Branch coverage and Output diversity outperforms the Default configuration significantly in branch coverage but has worse performance in fault detection capabilities. We have also found that the Branch and Output diversity coverage criteria achieve better results when compared with only using Branch coverage in terms of mutation score. Additionally, the static software metrics, especially CBO, LCOM and LOC, are highly correlated with the performance of the fitness functions and can predict which coverage criteria will achieve better results.*

1 Introduction

Software testing is an important activity for assuring the quality of software systems. Test development requires producing various program inputs that cover as much the potential use cases of the system as possible, as well as validating the obtained output results. However, this is an effort-intensive task and requires a significant time budget [44], [9]. Nowadays, as the complexity of modern applications is increasing, it is essential to produce meaningful tests that achieve high code coverage. Many developers underestimate the effort needed for software testing [11]. As a consequence, the testing produced by software developers is usually ineffective. Subsequently, there is a need for automation of test generation. Such optimisation can decrease the effort and time required for testing [44], [51]. Following the need for reliable automatic test generation, during the past decade, various techniques and tools for test case generation have been developed. Furthermore, recent research show that Artificial Intelligence performs better when compared to developer-written tests in the case of code coverage [2], [41].

EvoSuite is a state-of-the-art tool for automatic generation of unit test cases in Java. It is able to produce various test suites, based on the selection of fitness functions, with high code coverage [16], [17]. The EvoSuite fitness functions compute the distance of the generated test suites to cover the desired test targets [41]. The test targets are represented by various coverage criteria, such as branch coverage, line coverage and weak mutation coverage. Furthermore, as it requires only code that compiles the EvoSuite tool allows investigation of its performance on many popular open-source projects containing large sets of software systems, such as SF-110 corpus [17].

In recent years, many different fitness functions have been added to EvoSuite [47]. However, as individual fitness functions have been created to satisfy various coverage criteria, it is currently unclear what is their contribution in case of finding bugs [3]. Moreover, as in previous research, the effect of choosing various single fitness functions has been widely assessed, there is a need for further investigation into the performance of the combination of different coverage criteria [18], [49].

Previous studies have shown that the Branch coverage criteria in EvoSuite achieves the best results among all possible fitness functions in EvoSuite [18], [49]. Output diversity is a black-box technique that maximises differences between observed output. Previous research has shown that output diversity is a promising new black-box testing criterion, and test suites selected to satisfy this criterion are often more consistent in detecting faults when compared to the one selected from white-box criteria [4].

This research will focus on creating a model based on empirical results that predict which fitness functions are best suited for the class under test. In particular, we will investigate the possible benefits of using the combination of Branch coverage and Output diversity as a fitness function for test generation when compared to the Branch coverage fitness function and the EvoSuite Default configuration, which combines eight different coverage criteria. In this paper, we will abbreviate the fitness functions as follows: DC for the Default configuration, BC for Branch coverage and BO for the combination of Branch coverage and Output diversity.

To that end, the main research question is:

- ***When and how does Output diversity affect the number of bugs detected when combined with Branch coverage?***

The corresponding sub-questions are:

- *How does the Output diversity in combination with Branch coverage affect the branch coverage in comparison with using only Branch coverage/ Default configuration fitness functions?*
- *How does the Output diversity in combination with Branch coverage affect the fault detection in comparison with using only Branch coverage/ Default configuration fitness functions?*
- *How does the time budget affect the achieved branch coverage, when using the combination of Output diversity and Branch coverage, in comparison with using Branch coverage/ Default configuration fitness functions?*

Our results showed a correlation between specific static class analysis metrics and the performance of the different EvoSuite fitness functions. In general, the results have shown that the combination of Output diversity and Branch coverage achieves better results when compared to Branch coverage in terms of fault detection capabilities. However, it has worse performance than the DC fitness function. In terms of achieved branch coverage, the BO significantly outperforms the DC criteria and obtains almost identical results as the BC fitness function. However, based on various class analysis metrics, different fitness functions may be more appropriate in specific cases. The time limit positively affects the fitness functions' performance in the range of 1 to 3 minutes. After the third minute, the increase becomes almost insignificant. Additionally, adding more coverage criteria to a fitness function can lead to worse performance in many cases, in the interval of 3 to 5 minutes.

The research paper has the following structure. Section 2 introduces the background of the research and related work. Section 3 describes the research methodology and explains what approach we have chosen to answer the research question/ sub-questions and why we decided to follow this method. In section 4, we present the results of the experiment. In section 5, we reflect on the research findings. Section 6 is dedicated to the ethical aspects of our research and its reproducibility. In the final Section 7, we provide our conclusions and discuss open issues and possible improvements.

2 Background

2.1 Search-based Software Testing

Search-based Software Testing, also known as SBST, combines automatic test case generation and search techniques [23]. Previous studies have considered SBST techniques on different testing levels [33]. For instance, Fraser et al. [16] examined SBST for unit testing, while Derakhshanfar et al. [14] considered search-based testing on the integration test level. In this study we focus on Search-based unit testing. Search-based unit testing is one of the topics that has been extensively studied in previous research [2], [17], [47]. In the automatic unit test generation, the search process consists of generating various test suites that satisfy different coverage criteria. In SBST, these coverage criteria are also known as fitness functions. The purpose of the fitness functions is to guide the search to promising areas of the search space by evaluating candidate solutions, in our context – test cases. The fitness function is problem-specific and should be defined when considering new problem, in our case - the class under test (CUT) [34].

2.2 EvoSuite for automatic test generation

EvoSuite is a Search-based Software Testing tool that automatically generates unit test suites for Java classes based on chosen code coverage criteria, such as branch coverage, line coverage and others. It uses an evolutionary approach based on a genetic algorithm to derive test suites. In the context of SBST, each coverage criterion is considered a fitness function that guides the unit test generation to achieve optimal code coverage for that particular criterion [38]. This study focuses on Branch coverage, Output diversity and the EvoSuite Default configuration fitness functions. The Default configuration in EvoSuite combines eight coverage criteria. In particular, it combines – Branch coverage (BC), Direct Branch Coverage, Line coverage, Exception coverage, Method coverage, Method coverage (Top-Level, No Exception) - MNEC, Output coverage/ diversity and Weak Mutation coverage [19], [42].

2.3 Combining coverage criteria

For the first time, Rojas et al. [47] combined multiple coverage criteria for SBST. The results showed that when combining different fitness functions, the test generation performance does not suffer. Furthermore, in some aspects combining various coverage criteria can boost the performance of SBST. Later Gay et al. [18] provided evidence that multi-objective suite generation could be more effective than single coverage criteria. However, there is still a need to understand how and when a combination of various fitness functions could outperform the single coverage criteria.

3 Methodology

To answer the research questions mentioned in Section 1, we developed a machine learning data analysis tool that aims to find the extent to which the combination of Output diversity and Branch coverage affects the branch coverage and the fault detection capability of the test suites generated by EvoSuite. To assess the effect of this combination, we compared its performance to two fitness functions - Branch coverage and Default configuration in EvoSuite. A good model for determining the fault detection effectiveness of fitness functions is mutation score [36]. Mutation score is a quantitative measurement of the test quality based on the ability of the test suites to detect faults [63], [31]. Thus, in this paper we will use mutation score to measure the fault detection capability of the generated tests.

In this research, we strive to find a correlation between specific program characteristics and the performance of the different fitness functions. To discover such characteristics, we rely on static software metrics and machine learning. The tool we have developed uses a pre-determined set of static software metrics to decide in which cases it is better to use a combination of Branch coverage and Output diversity (BO), only Branch coverage (BC) or the Default configuration (DC). As the general structure of the machine learning system is the same, its configurations and parameters vary for the different datasets (based on the two functions that we are comparing and what we are examining - branch coverage or mutation score). The formation of the datasets is described in Section 3.1.

3.1 Data Collection

For the research, we collected data for different classes from the open source projects SF110 Corpus [1] and Apache Commons[2]. Following the same methodology as Panichella et al. [41], we chose 346 Java classes for analysis. For every class we obtained 49 static code metrics by using the CK tool [6]. The EvoSuite tool uses evolutionary algorithms to generate test suites, that are by definition stochastic algorithms. Hence, the produced test suites may vary when running the tool multiple times with the same configuration/setting. Therefore, we ran the tool ten times per class to address the stochastic nature of the tool. For every class and fitness function, we considered the branch coverage and mutation score achieved by the generated test suites. Following the approach of Derakhshanfar et al. [15] and Olsthoorn et al. [39], we run EvoSuite with three different search budgets (time out for the generation process) of 60s, 180s and 300s for the branch coverage. This is because the runtime of EvoSuite has a significant effect on the overall coverage achieved by the generated tests. As computing mutation score is highly time-expensive, we considered only the 60s time limit for the fault detection capabilities.

3.2 Statistical analysis

After obtaining the ten test results per class, we performed statistical analysis. We first applied the normality test on the ten test runs for every fitness function configuration. As we ran the test on only ten observations, we used Shapiro–Wilk normality test, which has good performance on distributions with a small sample size [45], [50]. The null hypothesis of the normality tests states that data follows the normal distribution when the p-value is greater than some predetermined value [1]. Opposite, when the p-value is smaller or equal to the chosen threshold value, the null hypothesis is rejected. In this research, we decided to set the p-value to 0.05, which is commonly used in studies [54]. Based on the normality of the data, we compared the scores of the different fitness function configurations using the t-test or Wilcoxon test [60] accordingly. T-test has a good performance for establishing significant differences when the data is normally distributed, while the Wilcoxon is appropriate when the data is not normally distributed [12]. When the p-value obtained from the t-test/ Wilcoxon test is lower than 0.05, the two distributions in the comparison are significantly different. Following the results from the statistical test, we saved for consideration only the classes that have a significant difference for every configuration. Despite using different fitness functions, most of the classes had very similar branch coverage and mutation score. Consequently, the resulting datasets for analysis were minimal. The full results from the statistical significant test can be observed in Table 1, 2.

In this study, we are not just interested in the statistical significance but also in the magnitude of the differences between the results (coverage and mutation scores) achieved by the different fitness functions. To this aim, we applied the Vargha-Delaney effect size ($\hat{A}_{12}$ statistics) [59], [53]. The full results of the $\hat{A}_{12}$ statistics are presented in Appendix A.

To determine for which classes one fitness function would achieve better results than the other, we apply a supervised machine learning model [37]. Hence, for binary classification we built a dataset where the class metrics are the features, while the labels to predict are $\{0, 1\}$. Based on the $\hat{A}_{12}$ statistics, if the combination of Branch and Output coverage performs better, we labelled the class with "1". Opposite, when the Default configuration or Branch coverage functions achieve a higher score, we labelled the CUT with "0". When the magnitude of the effect size test for the class is "negligible" or "small", the difference in the performance of the two fitness functions is not significant enough to bring some meaningful information. Therefore, we did not consider these classes in the analysis. The code we used for the Vargha-Delaney effect size test is from an open-source project [3], and represents an adapted Python version of the Vargha-Delaney effect size test from the R package "effsize" [56].

From the Vargha-Delaney test results, we observed a large class imbalance for some datasets. In particular, these are all sets comparing BO and DC. Additionally, there is a significant class imbalance for all datasets resulting from assessing the fault detection capabilities of the fitness functions. Such differences in the number of observations for the different classes and, in general, the small size of the datasets may lead to inaccurate predictions when using the machine learning approach. Thus, we decided to manually validate the obtained results from the data analysis tool for these datasets. Furthermore for the comparison of BO and DC in terms of branch coverage we performed qualitative analysis, presented in Appendix C.

[1] https://www.evosuite.org/experimental-data/sf110/
[2] https://commons.apache.org/
[3] https://gist.github.com/f9b19d65b7f16603c837024d5f8c8a65.git

3.3 Data analysis tool

The data analysis tool consists of three main stages: data pre-processing, classification and evaluation. Figure 1 represents the general structure of the tool. For every step on Figure 1 we performed parameter tuning. Then for every different configuration, we evaluated its performance and chose the one with the highest accuracy for the data analysis. For the tool development, we used the Scikit-learn library in Python [43].

3.3.1 Data pre-processing. As mentioned above, the training and analysis dataset has shrunk drastically after the statistical significance test. To that end, to develop a reliable machine learning tool, we should consider the risk of overfitting. Overfitting is the use of models or procedures that violate parsimony. In other words, include more terms than are necessary or use more complicated approaches than are necessary [21]. To mitigate the risk of overfitting as much as possible, we performed feature selection on the static class analysis metrics. We used L1-based or Tree-based feature selection, based on the dataset's properties [62], [48]. To assess which of these models performs better, we added various feature selection techniques (Linear SVC, DecisionTree, RandomForest and others) as the first term in the pipeline - Figure 1, which performance we examined following the approach in Section 3.3.3.

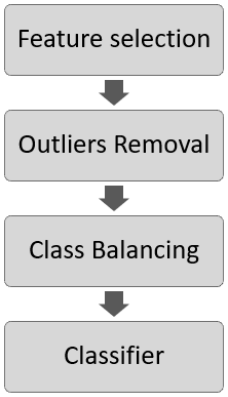


Figure 1. Pipeline of the data analysis tool

Another technique that we investigated and could enhance the performance of the data analysis tool is outlier removal. For this task, we decided to use Isolation Forest, which is a reliable algorithm for anomaly detection [28]. Then, we removed the detected outliers (anomalies) from the training data. However, for some datasets, this technique did not improve the classifier's performance. For this reason, we added it as the second term in the pipeline - Figure 1, and it was included only when it increased the accuracy of the classifier.

As there was a class imbalance in the datasets, we performed class balancing. There are two main techniques for class balancing - oversampling and undersampling. In general, undersampling refers to removing observation samples from the dominating class. Oversampling is the opposite - adding additional samples from the minority class by repeating observations from the dataset or creating artificial ones [35]. As our data is limited, we are not able to remove samples. Consequently, we used oversampling. In particular, we assessed two methods - Random Over Sampling and SMOTE. The first one adds random observations from the data set until the class is balanced, while the second one adds artificial samples [10]. We added both techniques as the third term in the pipeline - Figure 1, as their performance varies for the different datasets.

3.3.2 Classifier. As the datasets are very small, we were able to examine the performance of multiple classifiers. From Osisanwo et al. [40] and Althnian et al. [5], we have seen that SVM, Naive Bayes, Random Forest, Decision Tree and AdaBoost have good performance when trained on limited data. Consequently, as a final term in our pipeline, we included Gaussian Naive Bayes, Support Vector Machine, Decision Tree, Random Forest, and three boosting algorithms - AdaBoost, XGBoost and GradientBoost. In his research, Gómez-Ríos et al. [20] presents more information about AdaBoost, XGBoost and GradientBoost and their similarities and differences. Additionally, we considered Logistic Regression, as it is a simple and efficient method for binary classification [52]. As stated above, there is a severe risk of overfitting. Hence, we did not consider Neural Network classifiers as they are prone to overfitting because of their complexity [57].

To further boost the classifier algorithms, we performed hyper-parameter tuning. The classifier parameter tuning could often be more important than the choice of machine learning algorithms [26]. Therefore, we assessed many different parameters for every classification model. Additionally, we considered various parameters for all data pre-processing techniques. We have to mention that only the data balancing model and the classifier were tuned after the train-test split, using grid search. In total we considered more than 900 000[4] different combinations of pre-processing techniques and classifiers. For every dataset we chose the best combination of classifier and pre-processing techniques and used it for the data analysis.

3.3.3 Evaluation of the system. To reliably assess the performance of the machine learning system, we decided to use a cross-validation strategy. We used the Nested-CV model, as it provides unbiased performance estimates even for data with a limited sample size. Nested CV is performed in two layers to obtain the training and validation set [58]. In the first layer, we use a K-fold CV split with k=5. Thus we split the data into five equal segments, and on every iteration,

[4]Feature selection - 18 configurations; Outlier removal - 5 configurations; Data Balancing - 14 configurations; Classifiers - 733 configurations.

different segments are used for validation [46]. On the second layer, we used grid search with 5-fold cross-validation to assess the performance of the pipeline. To measure the performance of the classifiers, we used F1 score, which combines the precision and recall into a single metric by taking their harmonic mean. It is commonly used to assess the performance of machine learning algorithms [27].

Dataset	All classes	Classes with significant difference
BO/BC - branch coverage 60s	346	33
BO/DC - branch coverage 60s	346	60
BO/BC - branch coverage 180s	346	25
BO/DC - branch coverage 180s	346	73
BO/BC - branch coverage 300s	346	24
BO/DC - branch coverage 300s	346	70

Table 1. Classes with significant difference for all datasets considering branch Coverage

Dataset	All classes	Classes with significant difference
BO/BC - mutation score 60s	346	81
BO/DC - mutation score 60s	346	98

Table 2. Classes with significant difference for all datasets considering mutation Score

4 Results

In this section, we will discuss the obtained results and how they answer each of the research questions introduced in Section 1.

From Table 1, 2 we observe that out of the 346 classes we have selected for analysis, the classes with significant difference are just a small fraction. Thus, we can say that in most of the cases there will not be a considerable difference in the efficiency of using the combination of Branch coverage and Output diversity or Default configuration/ Branch coverage as a fitness function.

4.1 RQ1

- *RQ1: How does the Output diversity in combination with Branch coverage affect the branch coverage in comparison with using only Branch coverage/ Default configuration fitness functions?*

For assessing the effect of BO on the branch coverage we considered six different datasets. For each couple of fitness functions that we compare (BO and BC; BO and DC), we consider three time limits of 60s, 180s and 300s. Due to space limitation the results and the analysis for the 300s datasets is presented in Appendix D.

The Branch coverage fitness function achieves better results in all examined timeframes. However, the class distribution is almost even for the 180s and 300s. BO achieves a better score for almost all classes in all possible timeframes when compared with DC, in terms of branch coverage. From Appendix A, we see that there are too few classes for which the performance of DC fitness function is better. Because of this significant class imbalance, using the data analysis tool could be unreliable. For the 60s dataset there are only two classes in favour of DC. Thus for this timeframe we analysed the data only manually. As for the 180s and 300s datasets, we obtained results from the data analysis tool, which we validated through manual analysis and further performed qualitative manual analysis, which is presented in Appendix C. The top ten configurations, their parameters and the selected features, as well as all of the results from the data analysis tool are presented in Appendix B. On Table 3 we have presented the best classifier for every dataset and the achieved F1-score.

Dataset	Best performing classifier	F1-score
BO/BC - branch coverage 60s	XGBClassifier	0.899
BO/BC - branch coverage 180s	LogisticRegression	0.84
BO/DC - branch coverage 180s	GradientBoostingClassifier	0.922
BO/BC - branch coverage 300s	DecisionTree	0.947
BO/DC - branch coverage 300s	DecisionTree	0.904
BO/BC - mutation score 60s	RandomForestClassifier	0.925
BO/DC-mutation score 60s	GradientBoostingClassifier	0.887

Table 3. Best classifiers and F1-score for every considered dataset

4.1.1 60s. Figure 6 depicts top 10 best performing estimators for the 60s time limit. We note that all of the best performing configurations have the same feature selection technique - DecisionTree and the same set of features used for training. This provides a clear evidence that the static software metrics have a significant correlation with the performance of the fitness functions. Furthermore, from our observation of the results, we noted that the LCOM* (Lack of Cohesion of Methods) [22] metric is selected by the feature selection techniques in 88% of all combinations. This shows that the Lack of Cohesion of Methods is strongly correlated

to the performance of BO and BC fitness functions when considering branch coverage. From Table 3 you can see that the best performing classifier is XGBoost, with accuracy of around 90%. Its resulting decision tree is represented in Figure 7. The tree shows that LCOM* and the number of private fields decide which fitness function is better. It shows that BO performs better when the LCOM* ranges between 0.597 and 0.74 or when LCOM* is more than 0.74, but the number of private fields is more than five.

For the 60s timeframe, we observed that only two classes achieved better scores when using Default configuration as a fitness function. Furthermore, one of the classes has a medium difference; thus, its coverage is very close to the one achieved by BO. Therefore, we performed manual analysis following the procedure in Appendix C - C.1. From the manual analysis of the code, we cannot say that the two classes have some unique characteristics. Both classes contained only one complex method and many straightforward ones. As we could not collect any insights from the script why these two classes are anomalies, the class metrics provided some understanding.

In previous studies, Badri et al. [7, 8] showed that class metrics correlate with the testing effort. In particular, the metrics WMC, RFC, LCOM and LOC can predict the testing effort. In consequence, we firstly took a look at these class metrics. WMC represents McCabe's complexity [32], while LOC considers lines of codes, LCOM - lack of cohesion of methods and RFC counts the number of unique method invocations in a class. We observed that the average WMC is more than five times smaller than the mean for the whole dataset. Furthermore, the average LCOM for the two outlier classes is 3, while for the whole set is 1514.1. Additionally, the score for LOC, quantity of "return", loop, try-catch blocks, parenthesized expressions, numbers, assignments and mathematical operations is significantly lower than the average for all classes. In Appendix C, Figure 26 we have provided a table with specific class metrics, the average of these metrics for all significant classes under test and the average for the outlier classes. The classes in yellow have a "medium" difference.

Previous research has shown that LCOM is correlated to the complexity of classes [30]. Moreover, LOC and the number of assignments, mathematical operations, numbers, expressions and loops could be considered an indirect indication of the complexity of the classes. These values are much lower than the average for the classes where the DC fitness function performs better. As two classes are not enough to make reliable conclusions, from our observations, we can say that Default configuration will perform better for the 60s time limit only if the classes have very low complexity.

4.1.2 180s. The logistic regression classifier performs best for the 180 seconds time limit, when comparing BO and BC, with accuracy of 0.84 - Table 3. The best performing feature selection functions are Select from Percentile, with chi-squared as the scoring method [29] and XGBoost. Out of the ten best configurations, shown in Appendix B, in nine, the feature selection technique selected LOC, WMC and LCOM for training.

On Figure 9, in Appendix B you can see the coefficients and the interception of the obtained Logistic regression estimator. For the interpretation of the results we used the function presented in Figure 9 [25]. From these results, we can see that classes with a high number of mathematics operations and fields, as well as a high value of CBO (Coupling between objects), will be labelled as "1", and thus, the BO will achieve better coverage than the BC fitness function. Conversely, for the classes with high values of CBO (Modified), RFC and "return" quantity, the BC fitness function will perform better. Another factor we assess is that LCOM and LOC usually get very high values (a couple of hundreds or even thousands). Therefore, for the classes with many lines of code, the BO will choose better test suites, while for the classes with high LCOM, we should choose the BC fitness function for optimal results. Additionally, other coefficients could affect choosing the better fitness function, such as NOSI, WMC, number of parenthesized expressions and others, presented in Figure 9, but their influence is marginal.

For the feature selection, nine of the ten best configurations used DecisionTree, when comparing BO and DC. The only class metric present in all sets of selected features is the "number" quantity. The rest of the selected class metrics are "static field" quantity, number of comparisons, loops, maths operations and visible methods. To obtain the decision boundary of the GradientBoost - Table 3, we plotted and examined every DecisionTree, used as an estimator by the GradientBoosting. In total, we considered 100 different trees. We decided to study only the conditions that were present in ten or more different trees (at least 10% of all trees) to mitigate the randomness and extract only the essential rules. Eight conditions passed the 10% requirement - Appendix B, Figure 13. Then we analysed every condition manually, as some were very similar. After manually validating the decision tree results, we reduced the obtained conditions to five - Figure 14. We have found that DC outperforms BO in branch coverage when the number of loops is lower or equal to 4, the number of static fields is between 2 and 16, and the number of comparisons is between 2 and 131. At least two of these three conditions should be satisfied for the DC to outperform BO. Additionally, we performed manual analysis for this dataset, which is presented in Appendix C - C.2.

4.1.3 300s. Because of space limitations, the analysis for the 300s is in Appendix D. The decision trees for the comparison of BO and BC, and BO and DC are presented on Figure 11, 16.

To sum up, the combination of Branch and Output diversity performs better when compared to DC. For less than

180s, the complexity can affect the fitness function's performance; thus, for the classes with very low complexity, the Default configuration would achieve higher branch coverage. Specific software metrics determine which fitness function performs better for the datasets for three and five minutes and when comparing BO and BC.

4.2 RQ2

- *RQ2: How does the Output diversity in combination with Branch coverage affect the fault detection in comparison with using only Branch coverage/ Default configuration fitness functions?*

For the mutation score, we observed much more classes with significant differences than when considering branch coverage. From our observation of the effect size results, presented in Appendix A, we can say that BO is performing much better than the BC fitness function when considering fault detection capabilities. However, DC outperforms significantly BO.

In Appendix C, Figure 17 shows the top performing configurations and the selected features of the data analysis tool for comparing BO and BC in terms of mutation score. Only one class metric is present in the selected features for all top ten classifiers - LCOM*. The classifier with the best performance is RandomForest - Table 3. We plotted every tree in the forest and extracted the most common decision points. The conditions that were present in most trees were LCOM*<=0.78 and LCOM*<=0.77, which proved the importance of the LCOM* metric. Because of the small number of classes in favour of BC, we further checked the conditions manually. In some cases, where there were similar decision points, such as LCOM*<=0.78 and LCOM*<=0.77, we combined them based on our manual analysis. The established conditions are presented on Figure 18. We found that for the BC fitness function to outperform BO, the conditions LCOM*>0.77, CBO>3 and "number" quantity greater than twelve, should always be satisfied. When the condition for methods quantity is not satisfied (the number of methods is less than 25), the requirement for the number of unique words (unique words quantity more than 87) should be satisfied. The opposite holds for the rest of the classes. The classes and its metrics are presented on Figure 19. In Figure 20 we have presented a decision tree that combines all discovered dependencies.

Figure 21 shows the best performing configurations for the mutation score dataset, when comparing BO and DC fitness functions. From Figure 21, Table 3 we can see that the best performing estimator is the GradientBoosting classifier, with accuracy of around 89%. Four class metrics are present in all top ten configurations. In particular, these are - the number of private fields, comparisons, assignments and DIT (Depth Inheritance Tree). To interpret the GradientBoosting classifier, we used the approach described in Section 4.1.2 when we compared BO and DC in terms of branch coverage. From the 100 analysed trees, the 10% requirement was satisfied by twelve conditions, shown in Figure 23. After the manual validation, we reduced them to six - Figure 22. We found that for BO to outperform DC, the number of comparisons should be between 14 and 34, while DIT should be lower or equal to 1. Additionally, the estimator suggested that the number of private fields should be less than or equal to 5, LOC should be less than 568, and the "return" quantity should be more than 12. At least four of these conditions should be satisfied for the class under test, in order BO to outperform DC.

4.3 RQ3

- *RQ3: How does the time budget affect the achieved branch coverage, when using the combination of Output diversity and Branch coverage, in comparison with using Branch coverage/ Default configuration fitness functions?*

In Figure 2 we have presented the average branch coverage for all considered fitness functions and classes for three time limits - 1, 3 and 5 minutes. In the interval of 1 - 3 minutes, the achieved branch coverage when using the BC or BO increases by around 3% on average. However, after the third minute, the improvement in the coverage is marginal - less than 1%. As the average coverage percentage of the DC fitness function increases at the almost same rate as the other two in the interval of 1 - 3 minutes, its coverage decreases after that time. Oppositely, we can see that, on average, the Branch coverage function continues to improve after the third minute. At the same time, BO has almost no visible improvement after the third minute. From Appendix A, we can see that the tendency when comparing the classes for the different fitness functions with the effect size test is very similar to the one in Figure 2. Consequently, we can claim that there is a correlation between the number of coverage criteria used as a fitness function and the achieved branch coverage. In particular, using more coverage criteria could lead to smaller branch coverage.

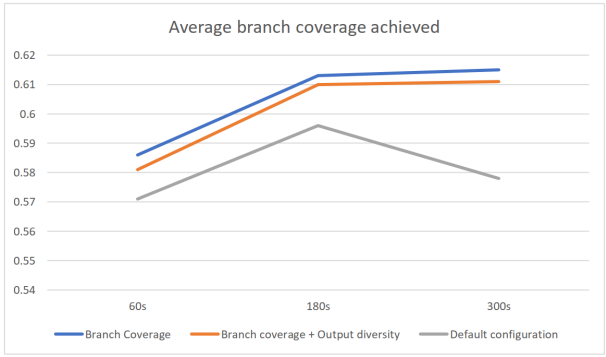


Figure 2. Average branch coverage achieved for different time limits

In terms of mutation score, we noted that the Default configuration fitness function did not manage to generate test results for the class com.soops.CEN4010.JMCA.JParser .JavaParser. This class distinguishes itself with its very high complexity; its WMC is 3301. However, BO and BC generated test suites and reported branch coverage for the class. As the Default configuration combines eight different coverage criteria, we suppose that it will not be able to produce results for very complex classes and short available time. In these cases, it will be better to use simpler fitness functions that are more effective. As BO performs much better than the BC fitness function, we suggest using BO for very complex classes and limited time. However, this claim is based on only one class observation and could be exclusion. Thus, further investigation to prove the claim is required.

5 Discussion and Threats to validity

Previous research has shown a correlation between static software metrics and testing effort [7], [8]. In our study, we observed a significant relationship between the class metrics and the achieved branch coverage/ mutation score by the different fitness functions. In particular, we have seen that the combination of Branch coverage and Output diversity performs much better than the Default configuration fitness function in terms of branch coverage. Additionally, it has significantly better results when compared to Branch coverage in terms of fault detection capabilities. However, in most cases, the mutation score of the DC fitness function is better than the one of BO. A study showed that the "Weak mutation" coverage criteria (WMC) could significantly impact the mutation score [55]. Therefore, the good mutation score achieved by the Default configuration could be explained by the fact that "weak mutation" is one of the eight coverage criteria that DC combines.

We observed that, on average, BO achieves almost identical results as BC for all time limits in terms of branch coverage. However, in terms of fault detection capabilities, BO dominates BC significantly. Therefore, for a tradeoff of less than one percent on average of the achieved branch coverage BO will provide significantly more extensive fault detection testing. Thus we suggest using the combination of Branch coverage and Output diversity instead of only using Branch coverage as a fitness function.

The class metrics CBO, LCOM* and LOC correlate significantly with the various fitness functions discussed in this paper. Previous research has shown that high values for WMC, LCOM and CBO increase the complexity and affect the testability of the classes [24]. Therefore, we can claim that the complexity of the CUT affects the performance of the fitness functions. However, the important class metrics and their values vary when comparing the various fitness functions and depend on the time limit. Additionally, we observed that, on average, the increase in the time limit after the 180s has a negative effect on the fitness functions that combine many coverage criteria when considering branch coverage.

Limitations to the current approach are mostly related to the data analysis tool. We should mention that different pre-processing configurations and classifiers may perform better than the one presented in Appendix B and used for analysis. Furthermore, we did not examine all possible tuning parameters because of the need for high computational power and time limitations. Consequently, there could be a combination of parameters that achieve higher accuracy than the one presented in Appendix B and Table 3. Another limitation of our tool is that despite taking all of the measures stated above to mitigate the risk of overfitting, our model could still be prone to overfitting because of the minimal size of some datasets. Additionally, the predictions may not be reliable for the datasets with high class imbalance because of the small data available for the minority class. In terms of manual analysis, it has to be noted that the results are subject to human bias and depend on the researcher's interpretation of the obtained statistical results.

6 Responsible Research

There are no significant ethical issues related to our research. No sensible or confidential information was used in this study. All of the data is from open-source projects, and the method followed for its collection is described in [41]. Reproducibility is an essential part of the research that ensures validity and credibility. To ensure that our study is legitimate, we should be transparent. To that end, we have provided the Python code corresponding to the development of the data analysis tool on Github [5]. Github supports version control and, in general, is a great tool for transparency [13]. Additionally, we added all data preparation and extraction scripts to the Github repository.

Another factor to consider for reproducibility is the random factor involved in the machine learning algorithms for pre-processing and classification. Due to the small sample size of the training sets, the randomness can cause fluctuations in the performance of the data analysis tool. Furthermore, despite the fact that we used a Nested CV for the evaluation described in Section 3.3.3, there is still a random factor in the K-fold split used for the first layer of the Nested CV [61]. This randomness affects the obtained accuracy of the data analysis tool and can lead to slightly different results in future recreation. We presented the top ten best performing configurations and their machine learning tuning parameters in Appendix B. All data pre-processing and classifier parameters that were not provided correspond to the default ones used by Scikit-learn.

[5] https://github.com/Stoyan4050/Training-a-Machine-Learning-Model-for-Optimal-Fitness-Function-Selection-with-the-Aim-of-Finding-Bug

7 Conclusions and Future Work

We have shown that static software metrics correlate with the performance of the fitness functions for branch coverage and mutation score. The metrics we established as most significant for choosing test coverage criteria are CBO, LCOM* and LOC. As all of these metrics could be considered a representation of the complexity of the classes, we state that when choosing a fitness function for automatic test generation, the complexity of the classes to be tested should be considered. BO significantly outperforms BC in mutation score and achieves almost the same results for branch coverage. Therefore, it is more appropriate to use BO instead of BC, as it will provide better fault detection testing and almost identical branch coverage. DC performs poorly when considering branch coverage, but it achieves a much better mutation score than BO and BC. Considering the time budget, increasing the time limit to more than three minutes leads to a decrease in the achieved branch coverage when adding more coverage criteria. However, increasing the time budget from 60s to 180s significantly improves the achieved branch coverage for all fitness functions.

Future work may include analysing different combinations of fitness functions available in EvoSuite. Additionally, further analysis into the factors that influence the performance of the coverage criteria could be performed. Additional study with more data may provide more insights into what factors lead to the observed branch coverage and fault detection capability performance of the combination of Branch coverage and Output diversity fitness function. Further investigation into other model parameters and configurations of pre-processing techniques and estimators could be tested to enhance the data analysis tool.

References

[1] Nor Aishah Ahad, Teh Sin Yin, Abdul Rahman Othman, and Che Rohani Yaacob. 2011. Sensitivity of normality tests to non-normal data. *Sains Malaysiana* 40, 6 (2011), 637–641.

[2] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[3] Hussein Almulla and Gregory Gay. 2022. Learning how to search: Generating effective test cases through adaptive fitness function selection. *Empirical Software Engineering* 27, 2 (2022), 1–62.

[4] Nadia Alshahwan and Mark Harman. 2014. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 181–192.

[5] Alhanoof Althnian, Duaa AlSaeed, Heyam Al-Baity, Amani Samha, Alanoud Bin Dris, Najla Alzakari, Afnan Abou Elwafa, and Heba Kurdi. 2021. Impact of dataset size on classification performance: an empirical evaluation in the medical domain. *Applied Sciences* 11, 2 (2021), 796.

[6] Maurício Aniche. 2015. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/.

[7] Linda Badri, Mourad Badri, and Fadel Toure. 2011. An empirical analysis of lack of cohesion metrics for predicting testability of classes. *International Journal of Software Engineering and Its Applications* 5, 2 (2011), 69–85.

[8] Mourad Badri and Fadel Toure. 2012. Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes. (2012).

[9] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[10] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. 2004. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter* 6, 1 (2004), 20–29.

[11] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 179–190.

[12] Patrick D Bridge and Shlomo S Sawilowsky. 1999. Increasing physicians' awareness of the impact of statistics on research outcomes: comparative power of the t-test and Wilcoxon rank-sum test in small samples applied research. *Journal of clinical epidemiology* 52, 3 (1999), 229–235.

[13] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*. 1277–1286.

[14] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2020. Generating Class-Level Integration Tests Using Call Site Information. *arXiv preprint arXiv:2001.04221* (2020).

[15] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie Van Deursen, and Annibale Panichella. 2020. 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)*. IEEE, 211–223.

[16] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[17] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[18] Gregory Gay. 2017. The fitness function for the job: Search-based generation of test suites that detect real faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 345–355.

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

[20] Anabel Gómez-Ríos, Julián Luengo, and Francisco Herrera. 2017. A study on the noise label influence in boosting algorithms: AdaBoost, GBM and XGBoost. In *International Conference on Hybrid Artificial Intelligence Systems*. Springer, 268–280.

[21] Douglas M Hawkins. 2004. The problem of overfitting. *Journal of chemical information and computer sciences* 44, 1 (2004), 1–12.

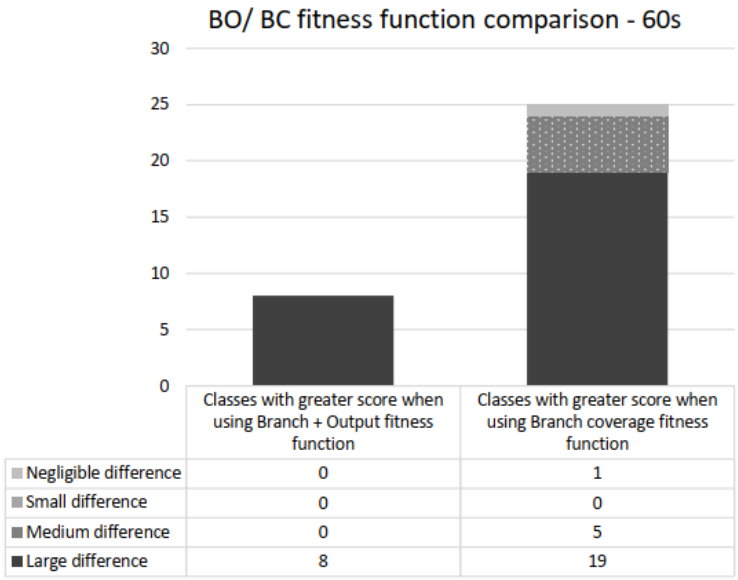
[22] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems* 3, 3 (1996), 143–158.

[23] Manju Khari and Prabhat Kumar. 2019. An extensive evaluation of search-based software testing: a review. *Soft Computing* 23, 6 (2019), 1933–1946.

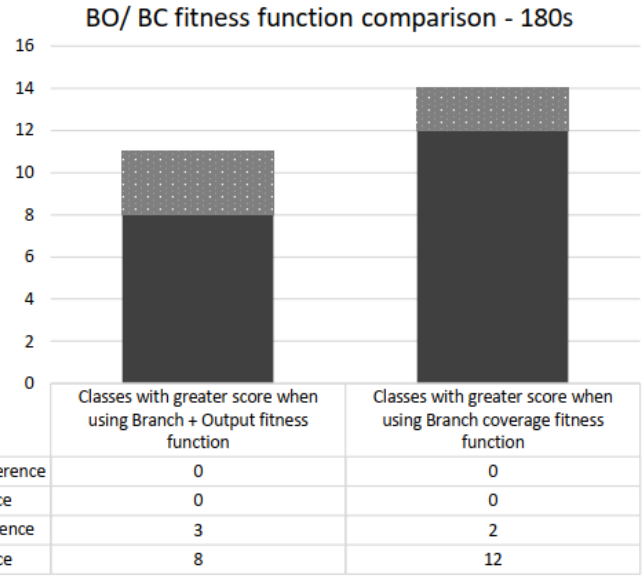
[24] Umesh L Kulkarni, YR Kalshetty, and Vrushali G Arde. 2010. Validation of ck metrics for object oriented design measurement. In *2010 3rd international conference on emerging trends in engineering and technology*.

IEEE, 646–651.
[25] Michael P LaValley. 2008. Logistic regression. *Circulation* 117, 18 (2008), 2395–2399.
[26] Niklas Lavesson and Paul Davidsson. 2006. Quantifying the impact of learning algorithm parameter tuning. In *AAAI*, Vol. 6. 395–400.
[27] Zachary Chase Lipton, Charles Elkan, and Balakrishnan Narayanaswamy. 2014. Thresholding classifiers to maximize F1 score. *arXiv preprint arXiv:1402.1892* (2014).
[28] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.
[29] Huan Liu and Rudy Setiono. 1995. Chi2: Feature selection and discretization of numeric attributes. In *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 388–391.
[30] Yutao Ma, Keqing He, Dehui Du, Jing Liu, and Yulan Yan. 2006. A complexity metrics set for large-scale object-oriented software systems. In *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*. IEEE, 189–189.
[31] Lech Madeyski. 2010. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology* 52, 2 (2010), 169–184.
[32] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.
[33] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.
[34] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.
[35] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. 2020. Machine learning with oversampling and undersampling techniques: overview study and experimental results. In *2020 11th international conference on information and communication systems (ICICS)*. IEEE, 243–248.
[36] Larry J. Morell. 1990. A theory of fault-based testing. *IEEE Transactions on Software Engineering* 16, 8 (1990), 844–857.
[37] Vladimir Nasteski. 2017. An overview of the supervised machine learning methods. *Horizons. b* 4 (2017), 51–62.
[38] Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. 2021. Hybrid Multi-level Crossover for Unit Test Case Generation. In *International Symposium on Search Based Software Engineering*. Springer, 72–86.
[39] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. 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)*. IEEE, 1224–1228.
[40] FY Osisanwo, JET Akinsola, O Awodele, JO Hinmikaiye, O Olakanmi, and J Akinjobi. 2017. Supervised machine learning algorithms: classification and comparison. *International Journal of Computer Trends and Technology (IJCTT)* 48, 3 (2017), 128–138.
[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.
[42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*. Springer, 309–324.
[43] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[44] Mauro Pezzè and Michal Young. 2008. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons.
[45] Nornadiah Mohd Razali, Yap Bee Wah, et al. 2011. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics* 2, 1 (2011), 21–33.
[46] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. Cross-validation. *Encyclopedia of database systems* 5 (2009), 532–538.
[47] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.
[48] Yvan Saeys, Inaki Inza, and Pedro Larranaga. 2007. A review of feature selection techniques in bioinformatics. *bioinformatics* 23, 19 (2007), 2507–2517.
[49] Alireza Salahirad, Hussein Almulla, and Gregory Gay. 2019. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability* 29, 4-5 (2019), e1701.
[50] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.
[51] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1294–1317.
[52] Abdulhamit Subasi. 2020. Chapter 3 - Machine learning techniques. In *Practical Machine Learning for Data Analysis Using Python*, Abdulhamit Subasi (Ed.). Academic Press, 91–202. https://doi.org/10.1016/B978-0-12-821379-7.00003-5
[53] Gail M Sullivan and Richard Feinn. 2012. Using effect size—or why the P value is not enough. *Journal of graduate medical education* 4, 3 (2012), 279–282.
[54] Matthew S Thiese, Brenden Ronna, and Ulrike Ott. 2016. P value interpretations and considerations. *Journal of thoracic disease* 8, 9 (2016), E928.
[55] Daniela Toader. 2022. Machine-Learning for Optimal Fitness Function Selection Using a Weak Mutation Branch Coverage Strategy with the Aim of Finding Bugs. Unpublished.
[56] Marco Torchiano. 2016. Effsize - A package for efficient effect size computation. https://doi.org/10.5281/ZENODO.1480624
[57] Jack V Tu. 1996. Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes. *Journal of clinical epidemiology* 49, 11 (1996), 1225–1231.
[58] Andrius Vabalas, Emma Gowen, Ellen Poliakoff, and Alexander J Casson. 2019. Machine learning algorithm validation with a limited sample size. *PloS one* 14, 11 (2019), e0224365.
[59] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[60] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968
[61] Tzu-Tsung Wong. 2015. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognition* 48, 9 (2015), 2839–2846.
[62] Xue Ying. 2019. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, Vol. 1168. IOP Publishing, 022022.
[63] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.

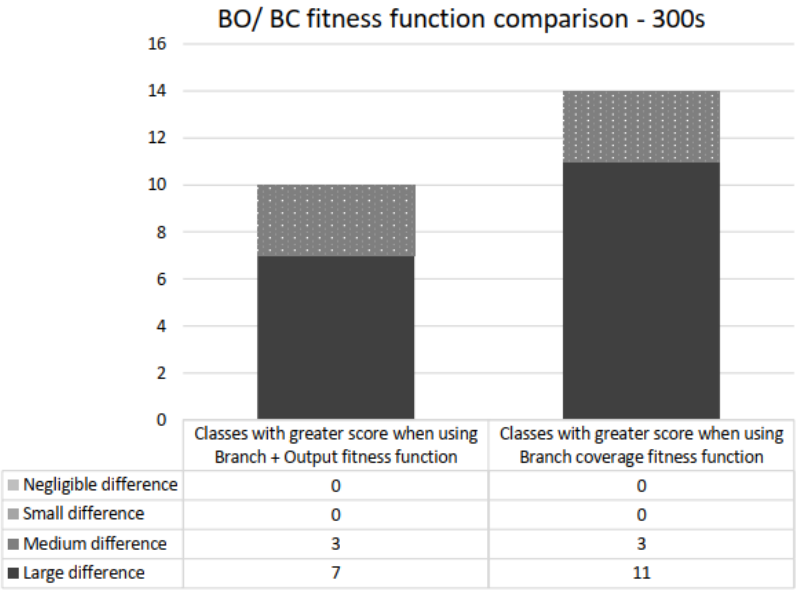
A Appendix A: Effect size



	Classes with greater score when using Branch + Output fitness function	Classes with greater score when using Branch coverage fitness function
Negligible difference	0	1
Small difference	0	0
Medium difference	0	5
Large difference	8	19

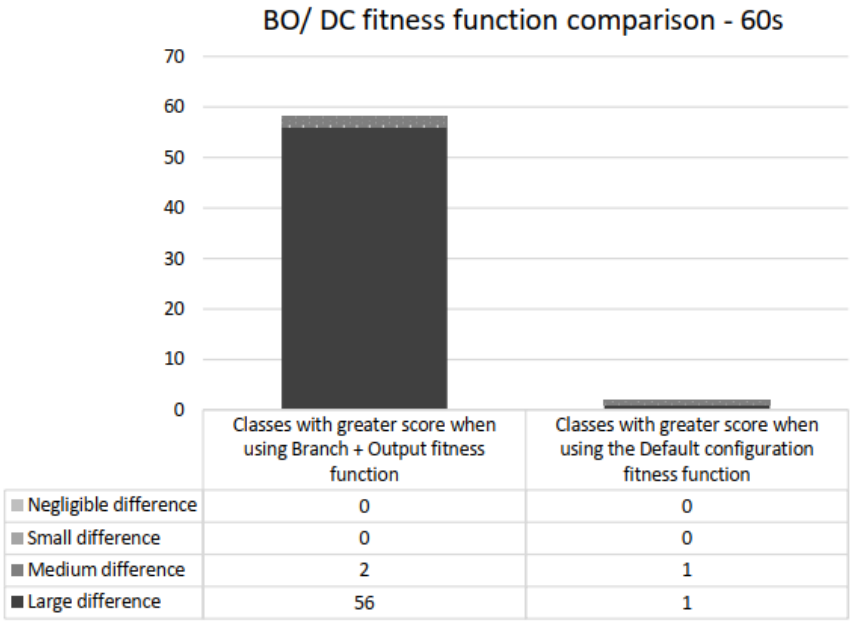


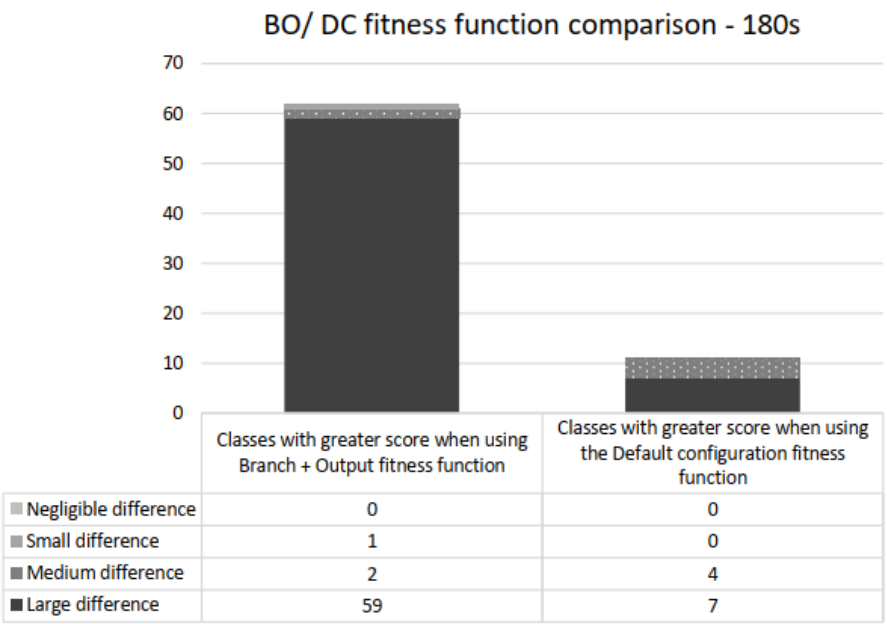
	Classes with greater score when using Branch + Output fitness function	Classes with greater score when using Branch coverage fitness function
Negligible difference	0	0
Small difference	0	0
Medium difference	3	2
Large difference	8	12



	Classes with greater score when using Branch + Output fitness function	Classes with greater score when using Branch coverage fitness function
Negligible difference	0	0
Small difference	0	0
Medium difference	3	3
Large difference	7	11

Figure 3. Effect size results for Branch + Output/ Branch coverage, in terms of branch coverage





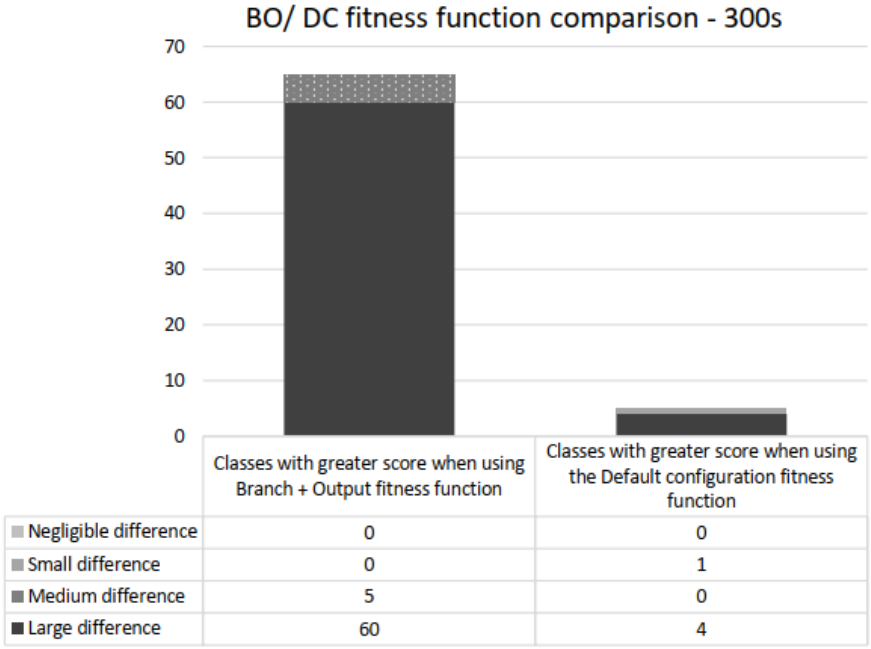
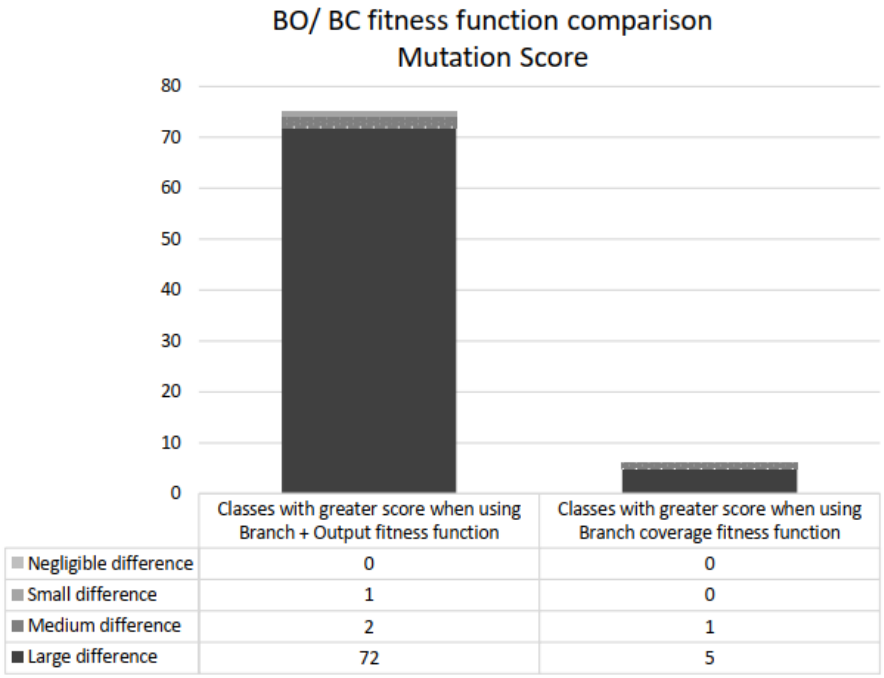
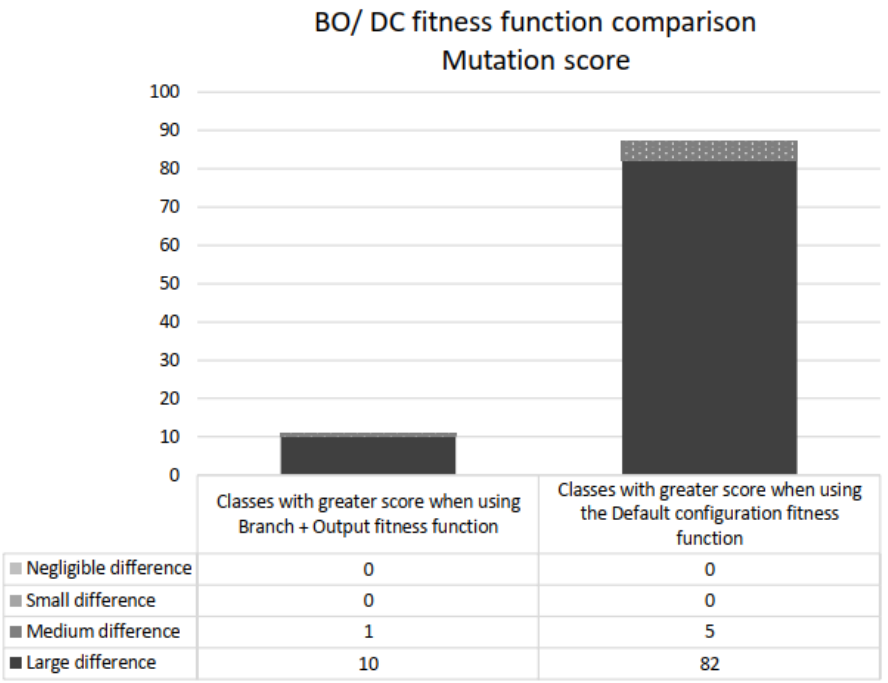


Figure 4. Effect size results for Branch + Output coverage/ Default configuration, in terms of branch coverage





Classes	Differences			
	Large	Medium	Small	Negligible
accessories.plugins.time.JDayChooser	X			
com.lts.io.DirectoryScanner	X			
com.soops.CEN4010.JMCA.JParser.JavaParser	X			
de.devcity.parseargs.ArgsParser	X			
ipac.BinaryCalculate	X			
net.kencochrane.a4j.file.FileUtil	X			
org.apache.commons.lang3.LocaleUtils	X			
org.apache.commons.lang3.builder.HashCodeBuilder	X			
org.apache.commons.math3.analysis.interpolation.TricubicSplineInterpolatingFunction		X		
org.jsecurity.util.AntPathMatcher	X			
twitter4j.TwitterBaseImpl	X			

Classes	Differences			
	Large	Medium	Small	Negligible
de.outstare.fortbattleplayer.model.impl.SimpleSector	X			
framework.util.jdbc.JDBCUtils	X			
jipa.Main		X		
twitter4j.OEmbedRequest	X			
twitter4j.TwitterImpl	X			
umd.cs.shop.JSState	X			

Figure 5. Effect size results for mutation score

B Appendix B: Data analysis tool results

FEATURES SELECTION	OUTLIER REMOVAL	DATA BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='minority')	XGBClassifier(max_depth=3, num_parallel_tree=3)	0.899047619	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=4, min_samples_split=8, n_estimators=10)	0.899047619	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	None	RandomOverSampler(sampling_strategy='minority', shrinkage=2)	AdaBoostClassifier(learning_rate=0.8, n_estimators=20)	0.882575758	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=20)	SMOTE(k_neighbors=2, sampling_strategy='all')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=2, min_samples_split=8, min_weight_fraction_leaf=0)	0.869047619	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=10, min_weight_fraction_leaf=0.5)	0.869047619	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	None	RandomOverSampler(sampling_strategy='all')	AdaBoostClassifier(learning_rate=0.1, n_estimators=20)	0.855	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='minority', shrinkage=1)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=2, min_samples_split=8, min_weight_fraction_leaf=0)	0.852380952	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=1, sampling_strategy='all')	XGBClassifier(max_depth=3, num_parallel_tree=1)	0.852380952	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority')	GradientBoostingClassifier(max_leaf_nodes=2, min_samples_split=10, min_weight_fraction_leaf=0.5)	0.852380952	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=1)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=2, min_samples_split=8, min_weight_fraction_leaf=0)	0.852380952	['lcom*' 'privateFieldsQty' 'returnQty' 'tryCatchQty' 'lambdasQty']

Figure 6. Hyper parameter tuning scores for the dataset BO/ BC fitness function comparison - 60s, in terms of branch coverage

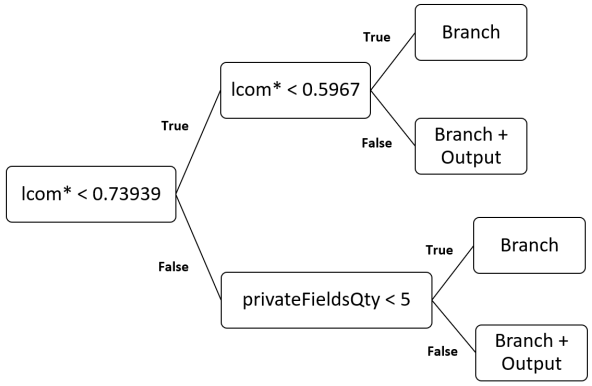


Figure 7. Decision tree for the top performing classifier - BO/BC comparison for 60s, in terms of branch coverage

FEATURES_SELECTION	OUTLIER_REMOVAL	DATA_BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=XGBClassifier(max_depth=5))	IsolationForest(n_estimators=20)	RandomOverSampler (sampling_strategy='minority' shrinkage=1)	LogisticRegression(C=1, max_iter=1000, penalty='none', random_state=42)	0.84	['cbo' 'cboModified' 'wmc' 'rfc' 'lcom' 'totalFieldsQty' 'nosi' 'loc' 'returnQty' 'parenthesizedExpsQty' 'mathOperationsQty' 'maxNestedBlocksQty' 'uniqueWordsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=5))	IsolationForest(n_estimators=20)	RandomOverSampler (sampling_strategy='all')	LogisticRegression(C=1, max_iter=1000, penalty='none', random_state=42)	0.84	['cbo' 'cboModified' 'wmc' 'rfc' 'lcom' 'totalFieldsQty' 'nosi' 'loc' 'returnQty' 'parenthesizedExpsQty' 'mathOperationsQty' 'maxNestedBlocksQty' 'uniqueWordsQty']
SelectPercentile(score_func=chi2)	None	SMOTE (k_neighbors=3, sampling_strategy='minority')	AdaBoostClassifier(learning_rate=1)	0.836666667	['wmc' 'lcom' 'publicMethodsQty' 'loc' 'returnQty']
SelectPercentile(score_func=chi2)	None	RandomOverSampler (sampling_strategy='all', shrinkage=1)	AdaBoostClassifier(learning_rate=0.8, n_estimators=5)	0.836666667	['wmc' 'lcom' 'publicMethodsQty' 'loc' 'returnQty']
SelectFromModel(estimator=XGBClassifier(max_depth=5))	None	RandomOverSampler (sampling_strategy='all', shrinkage=1)	LogisticRegression(C=1, max_iter=1000, penalty='none', random_state=42)	0.801269841	['cbo' 'cboModified' 'wmc' 'rfc' 'lcom' 'totalFieldsQty' 'nosi' 'loc' 'returnQty' 'parenthesizedExpsQty' 'mathOperationsQty' 'maxNestedBlocksQty' 'uniqueWordsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=2))	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=3, sampling_strategy='all')	DecisionTreeClassifier(criterion='entropy', max_features='sqrt', min_samples_leaf=2, min_samples_split=6)	0.8	['cbo' 'wmc' 'dit' 'rfc' 'lcom' 'lcom*' 'staticMethodsQty' 'loc' 'returnQty' 'stringLiteralsQty' 'mathOperationsQty' 'maxNestedBlocksQty' 'uniqueWordsQty']
SelectPercentile(score_func=chi2)	None	SMOTE(k_neighbors=1, sampling_strategy='minority')	AdaBoostClassifier(learning_rate=1, n_estimators=10)	0.793333333	['wmc' 'lcom' 'publicMethodsQty' 'loc' 'returnQty']
SelectPercentile(score_func=chi2)	None	SMOTE(k_neighbors=2, sampling_strategy='minority')	AdaBoostClassifier(learning_rate=0.5, n_estimators=10)	0.793333333	['wmc' 'lcom' 'publicMethodsQty' 'loc' 'returnQty']
SelectPercentile(score_func=chi2)	None	RandomOverSampler (sampling_strategy='minority')	AdaBoostClassifier(learning_rate=1, n_estimators=5)	0.793333333	['wmc' 'lcom' 'publicMethodsQty' 'loc' 'returnQty']
SelectFromModel(estimator=DecisionTreeClassifier (max_depth=4, max_features=10), prefit=True)	IsolationForest()	RandomOverSampler (sampling_strategy='all')	AdaBoostClassifier(algorithm='SAMME', learning_rate=0.1, n_estimators=75)	0.793333333	['fanin' 'publicFieldsQty' 'protectedFieldsQty' 'mathOperationsQty']

Figure 8. Hyper parameter tuning scores for the dataset BO/ BC fitness function comparison - 180s, in terms of branch coverage

Logistic regression coefficients

cbo	cboModified	wmc	rfc	lcom	totalFieldsQty	nosi	loc	returnQty	parenthesizedExpsQty	mathOperationsQty	maxNestedBlocksQty	uniqueWordsQty
6.72	-5.07	-0.4	-11.2	-2.06	3.95	2.77	1.74	-6.23	2.94	8.16	-2.48	1.24
After normalization												
0.12	-0.09	-0.007	-0.2	-0.04	0.07	0.05	0.03	-0.11	0.05	0.15	-0.05	0.02

Formula for Logistic regression interpretation of model:

Intercept: -1.53

$$\ln(P(Y=1)) = \beta_0 + \sum_{i=1}^{n} \beta_i X_i$$

$\beta_0 - interception;$

$\beta_i - feature\ coefficient;$

$X_i - feature\ value$

1 – Better performance of Branch + Output *0 – Better performance for Branch coverage*

Figure 9. Interpretation of Logistic regression functions - BO/BC comparison for 180s, in terms of branch coverage

FEATURES_SELECTION	OUTLIER_REMOVAL	DATA_BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=1, sampling_strategy='all')	DecisionTreeClassifier(max_depth=2, max_features='sqrt', min_samples_leaf=2)	0.946667	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='all', shrinkage=1)	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=4, min_samples_split=10, n_estimators=20)	0.946667	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=LinearSVC(C=0.015, dual=False, penalty='l1'), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='all', shrinkage=2)	RandomForestClassifier(max_depth=3, max_features=10, min_samples_leaf=4, min_samples_split=8, n_estimators=10)	0.9	['cboModified' 'wmc' 'lcom' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'mathOperationsQty' 'uniqueWordsQty' 'modifiers']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=3, sampling_strategy='minority')	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=3, min_samples_split=8, n_estimators=20)	0.9	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=3, sampling_strategy='all')	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=3, min_samples_split=8, n_estimators=10)	0.9	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=3, min_samples_split=8, n_estimators=10)	0.9	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=1, sampling_strategy='minority')	XGBClassifier(max_depth=3, num_parallel_tree=1)	0.885714286	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=2, sampling_strategy='minority')	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=3, min_samples_split=10, n_estimators=20)	0.885714286	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	SMOTE(k_neighbors=2, sampling_strategy='all')	SVC(C=10)	0.885714286	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='minority')	SVC(C=10)	0.885714286	['fanout' 'protectedMethodsQty' 'stringLiteralsQty' 'anonymousClassesQty']

Figure 10. Hyper parameter tuning scores for the dataset BO/ BC fitness function comparison - 300s, in terms of branch coverage

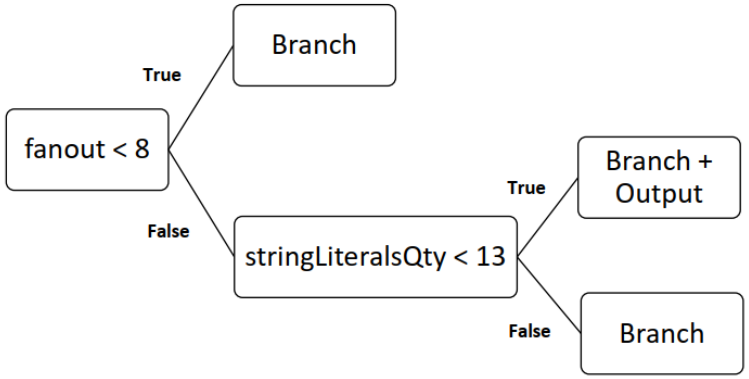


Figure 11. Decision tree for the top performing classifier - BO/BC comparison for 300s, in terms of branch coverage

FEATURES_SELECTION	OUTLIER_REMOVAL	DATA_BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='all', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0)	0.9223529	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=10, min_weight_fraction_leaf=0)	0.9147368	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=10, min_weight_fraction_leaf=0)	0.9147368	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest()	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0)	0.8837565	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0)	0.8766416	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0)	0.8766416	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectPercentile(score_func=chi2)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=0)	RandomForestClassifier(max_features=1, min_samples_leaf=3, min_samples_split=8, n_estimators=20)	0.8642105	['wmc' 'lcom' 'loc' 'numbersQty' 'assignmentsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='all', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=8, min_weight_fraction_leaf=0)	0.8642105	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=8, min_weight_fraction_leaf=0)	0.8380451	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='all')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=10, min_weight_fraction_leaf=0)	0.8380451	['visibleMethodsQty' 'staticFieldsQty' 'loopQty' 'comparisonsQty' 'numbersQty' 'mathOperationsQty']

Figure 12. Hyper parameter tuning scores for the dataset BO/ DC fitness function comparison - 180s, in terms of branch coverage

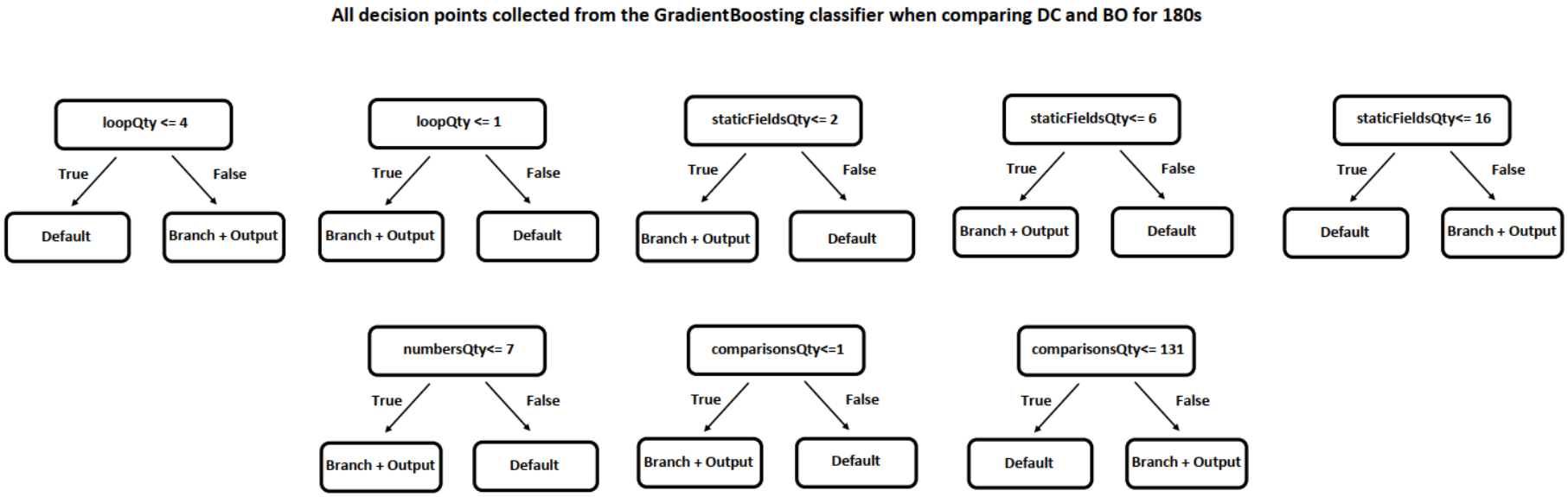


Figure 13. All established conditions from the GradientBoosting classifier - BO/ DC comparison for 180s, in terms of branch coverage

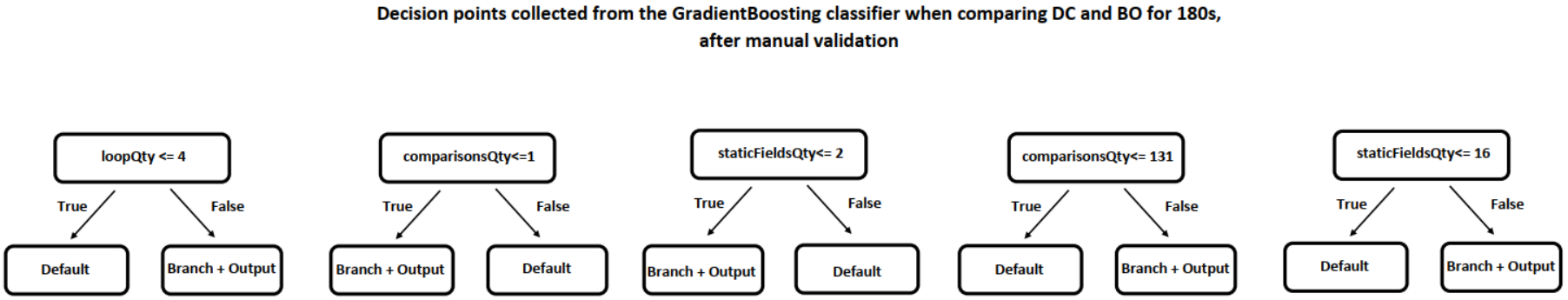


Figure 14. Conditions from the GradientBoosting classifier after the manual validation - BO/ DC comparison for 180s, in terms of branch coverage

FEATURES SELECTION	OUTLIER REMOVAL	DATA BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=XGBClassifier(max_depth=5) , prefit=True)	None	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	DecisionTreeClassifier(criterion='entropy', max_depth=4, max_features='log2', min_samples_leaf=4, min_samples_split=8)	0.904333333	['cbo' 'lcom' 'lcom*' 'totalFieldsQty' 'staticFieldsQty' 'publicFieldsQty' loc' 'loopQty' 'tryCatchQty' 'parenthesizedExpsQty' 'stringLiteralsQty' 'numbersQty']
SelectFromModel(estimator=RandomForestClassifier(n_estimators=2), prefit=True)	IsolationForest(n_estimators=20)	SMOTE(k_neighbors=1, sampling_strategy='minority')	AdaBoostClassifier(learning_rate=0.1, n_estimators=5)	0.895652174	['fanout' 'lcom*' 'abstractMethodsQty' 'publicFieldsQty' 'loopQty' 'mathOperationsQty' variablesQty' 'maxNestedBlocksQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority')	XGBClassifier(max_depth=3, num_parallel_tree=1)	0.89	['fanin' 'lcom' 'finalFieldsQty' 'tryCatchQty' 'parenthesizedExpsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	XGBClassifier(max_depth=3, num_parallel_tree=1)	0.89	['fanin' 'lcom' 'finalFieldsQty' 'tryCatchQty' 'parenthesizedExpsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all')	XGBClassifier(max_depth=3, num_parallel_tree=1)	0.89	['fanin' 'lcom' 'finalFieldsQty' 'tryCatchQty' 'parenthesizedExpsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=0)	XGBClassifiermax_depth=3, num_parallel_tree=1)	0.89	['fanin' 'lcom' 'finalFieldsQty' 'tryCatchQty' 'parenthesizedExpsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=2)	AdaBoostClassifier(algorithm='SAMME', learning_rate=0.1)	0.89	['fanin' 'lcom' 'finalFieldsQty' 'tryCatchQty' 'parenthesizedExpsQty']
SelectFromModel(estimator=RandomForestClassifier(n_estimators=2), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=2)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=8, min_weight_fraction_leaf=0)	0.870389016	['fanout' 'lcom*' 'abstractMethodsQty' 'publicFieldsQty' 'loopQty' 'mathOperationsQty' 'variablesQty' 'maxNestedBlocksQty']
SelectFromModel(estimator=RandomForestClassifier(n_estimators=2), prefit=True)	IsolationForest(n_estimators=20)	SMOTE(k_neighbors=1, sampling_strategy='all')	AdaBoostClassifier(learning_rate=0.1, n_estimators=5)	0.857556936	['fanout' 'lcom*' 'abstractMethodsQty' 'publicFieldsQty' 'loopQty' 'mathOperationsQty' 'variablesQty' 'maxNestedBlocksQty']
SelectFromModel(estimator=RandomForestClassifier(n_estimators=2), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0.25)	0.857556936	['fanout' 'lcom*' 'abstractMethodsQty' 'publicFieldsQty' 'loopQty' 'mathOperationsQty' 'variablesQty' 'maxNestedBlocksQty']

Figure 15. Hyper parameter tuning scores for the dataset BO/ DC fitness function comparison - 300s, in terms of branch coverage

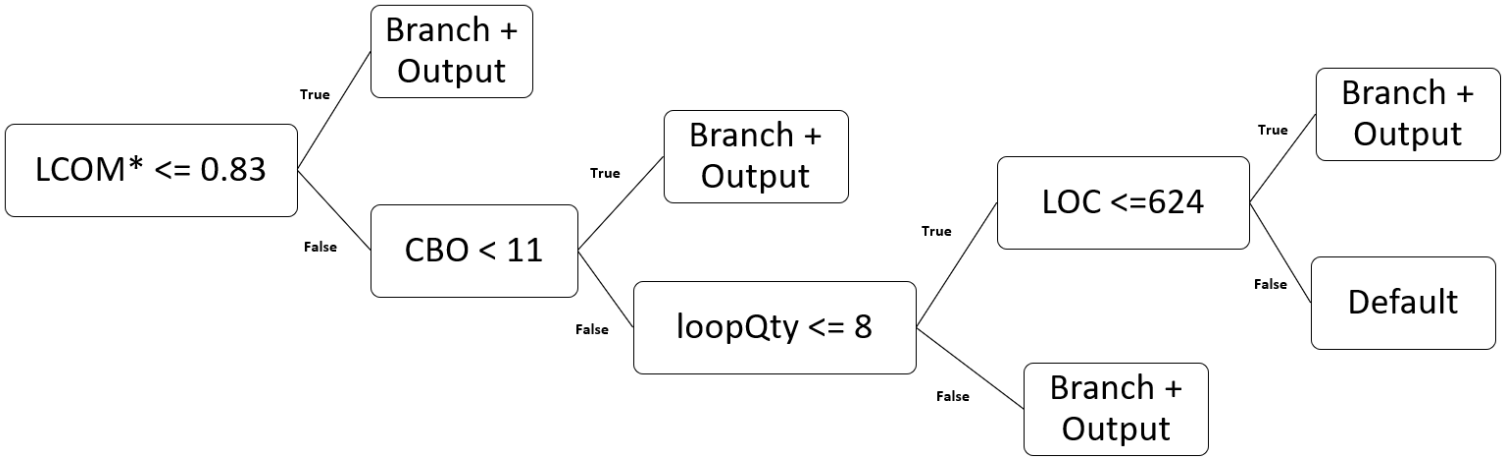


Figure 16. Decision tree for the top performing classifier - BO/DC comparison for 300s, in terms of branch coverage

FEATURES_SELECTION	OUTLIER_REMOVAL	DATA_BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority')	RandomForestClassifier(max_depth=3, max_features=1, min_samples_leaf=4, min_samples_split=10, n_estimators=20)	0.924637681	['cbo' 'lcom*' 'totalMethodsQty' 'numbersQty' 'uniqueWordsQty' 'logStatementsQty']
SelectKBest()	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='all', shrinkage=2)	AdaBoostClassifier(algorithm='SAMME', learning_rate=0.1)	0.904333333	['cbo' 'fanout' 'lcom*' 'publicMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'stringLiteralsQty' 'uniqueWordsQty' 'logStatementsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=8)	RandomOverSampler(sampling_strategy='all', shrinkage=2)	XGBClassifier(max_depth=10, num_parallel_tree=2)	0.867080745	['cbo' 'lcom*' 'totalMethodsQty' 'numbersQty' 'uniqueWordsQty' 'logStatementsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=2)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=8, min_weight_fraction_leaf=0)	0.858615137	['fanin' 'rfc' 'lcom*' 'totalMethodsQty' 'privateMethodsQty' 'defaultMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'stringLiteralsQty' 'numbersQty' 'mathOperationsQty' 'maxNestedBlocksQty' 'innerClassesQty' 'uniqueWordsQty'
SelectKBest(k=8)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=1, sampling_strategy='minority')	RandomForestClassifier(max_features=3, min_samples_leaf=3, min_samples_split=10, n_estimators=20)	0.85552381	['cbo' 'fanout' 'lcom*' 'publicMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'stringLiteralsQty' 'logStatementsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=5)	IsolationForest()	RandomOverSampler(sampling_strategy='all')	AdaBoostClassifier(learning_rate=0.5, n_estimators=20)	0.85552381	['fanin' 'lcom*' 'totalMethodsQty' 'protectedMethodsQty' 'defaultMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'returnQty' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'innerClassesQty' 'uniqueWordsQty'
SelectFromModel(estimator=XGBClassifier(max_depth=5)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority')	AdaBoostClassifier(learning_rate=0.5, n_estimators=20)	0.85552381	['fanin' 'lcom*' 'totalMethodsQty' 'protectedMethodsQty' 'defaultMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'returnQty' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'innerClassesQty' 'uniqueWordsQty'
SelectFromModel(estimator=XGBClassifier(max_depth=5)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	XGBClassifier(max_depth=6, num_parallel_tree=1)	0.85552381	['fanin' 'lcom*' 'totalMethodsQty' 'protectedMethodsQty' 'defaultMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'returnQty' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'innerClassesQty' 'uniqueWordsQty'
SelectFromModel(estimator=XGBClassifier(max_depth=5)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='all')	XGBClassifier(max_depth=6, num_parallel_tree=1)	0.85552381	['fanin' 'lcom*' 'totalMethodsQty' 'protectedMethodsQty' 'defaultMethodsQty' 'visibleMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'returnQty' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'innerClassesQty' 'uniqueWordsQty'
SelectKBest(k=8)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='all', shrinkage=0)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=10, min_weight_fraction_leaf=0)	0.854974359	['cbo' 'fanout' 'lcom*' 'publicMethodsQty' 'privateFieldsQty' 'finalFieldsQty' 'stringLiteralsQty' 'logStatementsQty']

Figure 17. Hyper parameter tuning scores for the dataset BO/ BC fitness function comparison, in terms of mutation scores

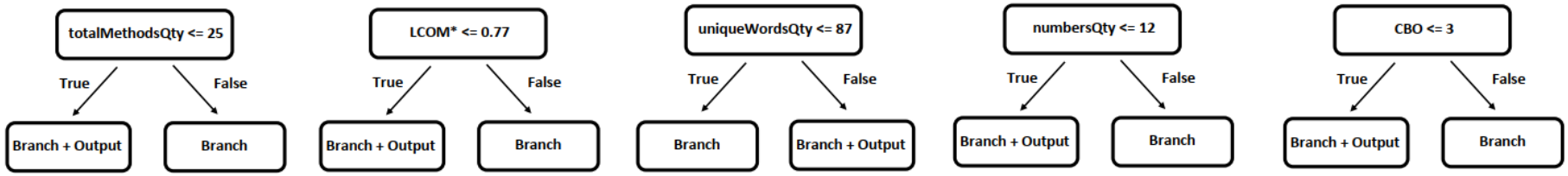


Figure 18. Established conditions from the RandomForest classifier - BO/BC comparison, in terms of mutation score

class	cbo	lcom*	totalMethodsQty	numbersQty	uniqueWordsQty
de.outstare.fortbattleplayer.model.impl.SimpleSector	10	0.81818	22	17	71
framework.util.jdbc.JDBCUtils	7	0.95982	32	19	261
jipa.Main	4	0.77778	12	110	84
twitter4j.OEmbedRequest	4	0.78667	25	28	66
twitter4j.TwitterImpl	56	0.99254	259	71	368
umd.cs.shop.JSState	11	1	8	13	56

Figure 19. Class metrics for the classes that achieve better mutation score when using Branch coverage as fitness function - BO/BC comparison, in terms of mutation score

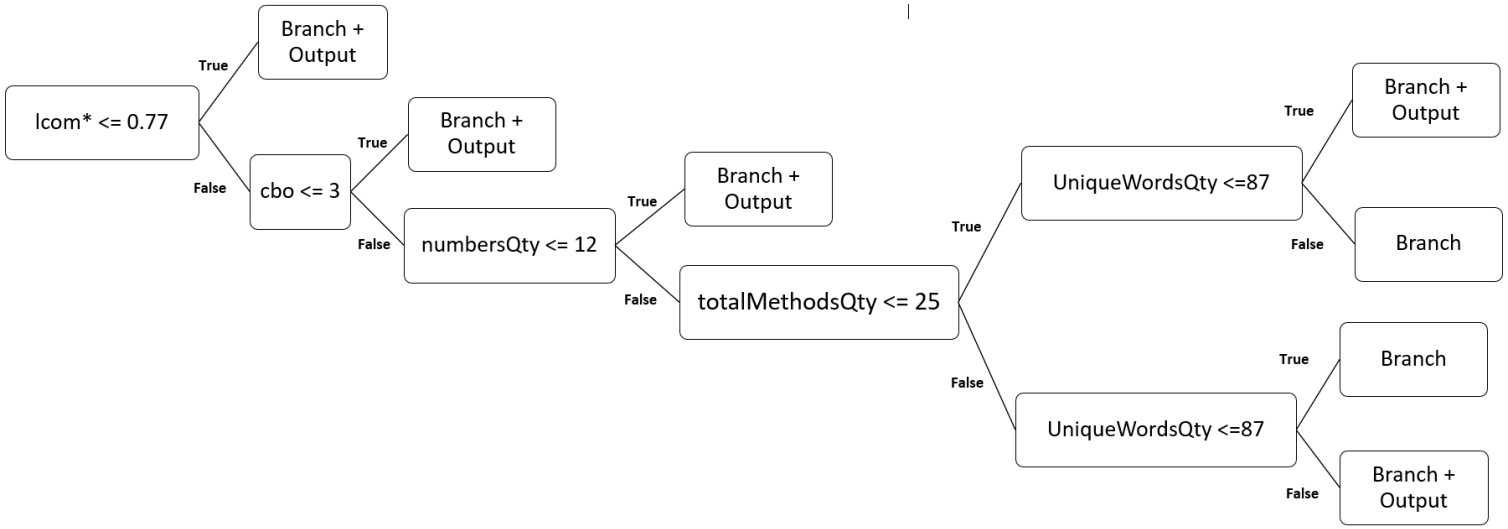


Figure 20. Suggested decision tree for mutation score, when comparing BO and BC fitness functions, in terms of mutation score

FEATURES_SELECTION	OUTLIER_REMOVAL	DATA_BALANCING	CLASSIFIER	SCORE	FEATURES
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=3, sampling_strategy='all')	GradientBoostingClassifier(max_features=3, max_leaf_nodes=10, min_samples_split=10, min_weight_fraction_leaf=0)	0.887077224	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=2))	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='all', shrinkage=1)	XGBClassifier(max_depth=3, um_parallel_tree=3)	0.83350659	['cbo' 'fanin' 'wmc' 'dit' 'rfc' 'visibleMethodsQty' 'staticFieldsQty' 'privateFieldsQty' 'nosi' 'loopQty' 'comparisonsQty' 'stringLiteralsQty' 'numbersQty' 'assignmentsQty' 'mathOperationsQty' 'variablesQty' 'maxNestedBlocksQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=2, sampling_strategy='minority')	AdaBoostClassifier(learning_rate=0.5)	0.830827224	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(n_estimators=20)	RandomOverSampler(sampling_strategy='minority', shrinkage=1)	RandomOverSampler(sampling_strategy='minority', shrinkage=1) GradientBoostingClassifier(max_features=3, max_leaf_nodes=3, min_samples_split=8, min_weight_fraction_leaf=0)	0.825836526	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=2, sampling_strategy='all')	GradientBoostingClassifier(max_leaf_nodes=10, min_samples_split=10, min_weight_fraction_leaf=0)	0.814287634	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=1)	GradientBoostingClassifier(max_features=3, max_leaf_nodes=2, min_samples_split=8, min_weight_fraction_leaf=0)	0.813763441	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=RandomForestClassifier(n_estimators=2), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	RandomOverSampler(sampling_strategy='minority', shrinkage=0)	RandomForestClassifier(max_features=1, min_samples_leaf=3, min_samples_split=10, n_estimators=20)	0.811282051	['cbo' 'fanout' 'dit' 'lcom*' 'visibleMethodsQty' 'privateFieldsQty' 'defaultFieldsQty' 'loc' 'comparisonsQty' 'stringLiteralsQty' 'uniqueWordsQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=1, sampling_strategy='all')	AdaBoostClassifier(learning_rate=0.5)	0.810934751	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']
SelectFromModel(estimator=XGBClassifier(max_depth=5))	IsolationForest(n_estimators=20)	SMOTE(k_neighbors=2, sampling_strategy='all')	AdaBoostClassifier(learning_rate=0.5, n_estimators=100)	0.804045615	['cbo' 'cboModified' 'dit' 'rfc' 'defaultMethodsQty' 'visibleMethodsQty' 'staticFieldsQty' 'privateFieldsQty' 'nosi' 'comparisonsQty' 'parenthesizedExpsQty' 'stringLiteralsQty' 'assignmentsQty' 'mathOperationsQty' 'variablesQty']
SelectFromModel(estimator=DecisionTreeClassifier(), prefit=True)	IsolationForest(max_features=5, n_estimators=10)	SMOTE(k_neighbors=3, sampling_strategy='minority')	GradientBoostingClassifier(max_leaf_nodes=10, min_samples_split=8, min_weight_fraction_leaf=0)	0.801149804	['dit' 'privateFieldsQty' 'loc' 'returnQty' 'comparisonsQty' 'assignmentsQty']

Figure 21. Hyper parameter tuning scores for the dataset BO/ DC fitness function comparison, in terms of mutation score

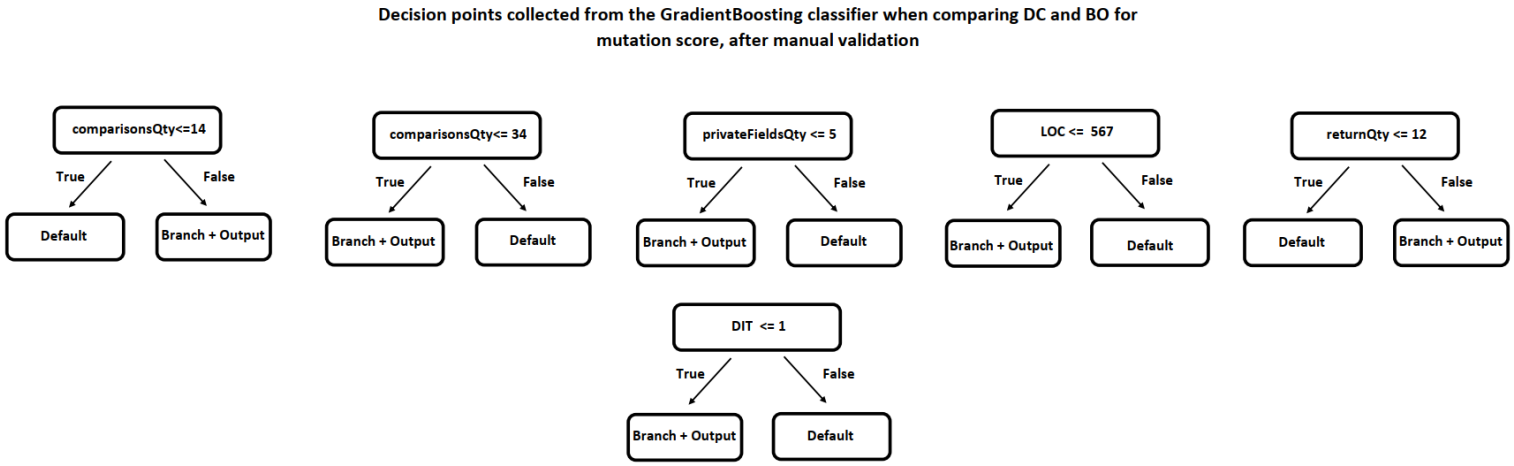


Figure 22. Conditions from the GradientBoosting classifier after the manual validation - BO/ DC comparison, in terms of mutation score

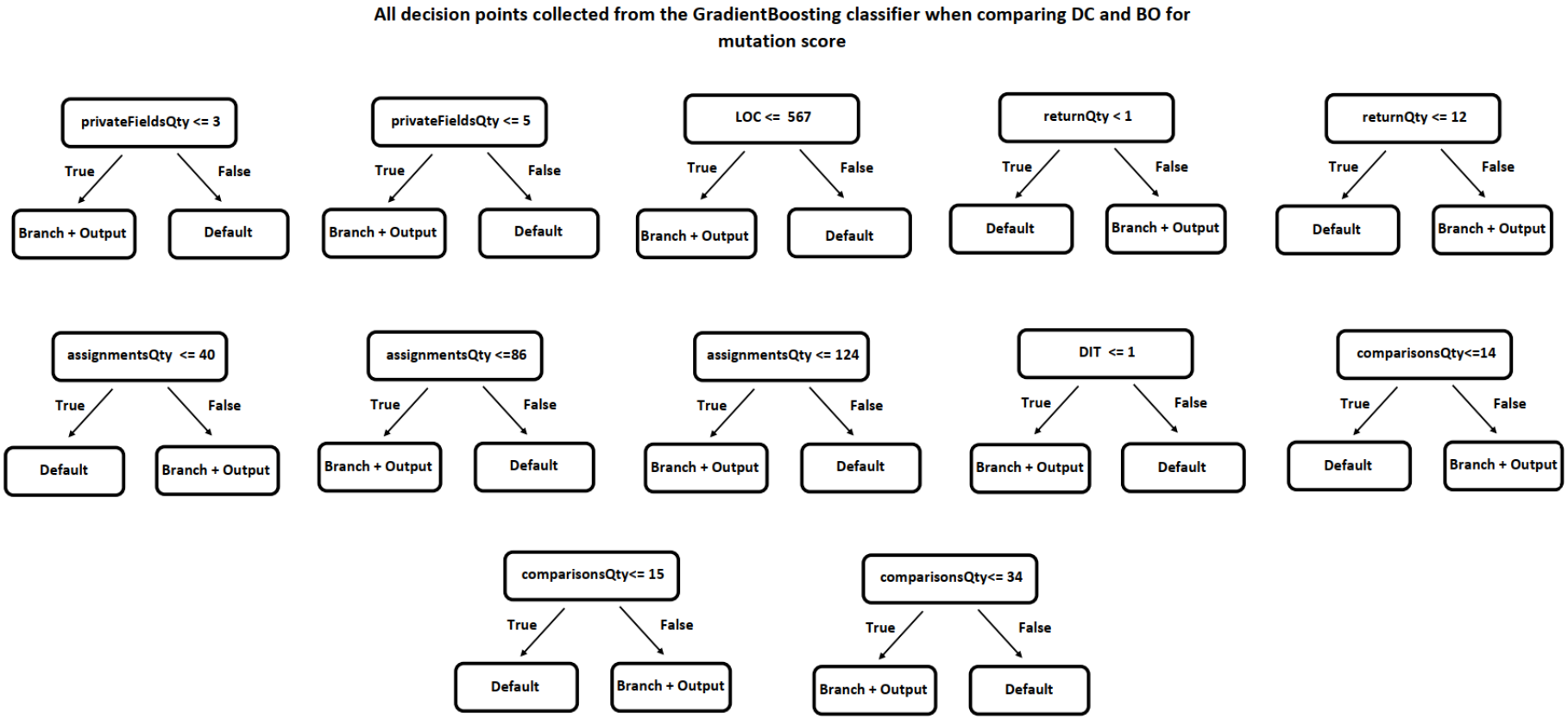


Figure 23. All established conditions from the GradientBoosting classifier - BO/ DC comparison, in terms of mutation score

C Appendix C: Manual qualitative analysis for the comparison of the combination of Branch coverage and Output diversity and the Default configuration in terms of branch coverage

C.1 Manual qualitative analysis

For the classes that we analysed manually, we looked into the source code of these classes and their static analysis metrics, as well as the results for every EvoSuite run. For the datasets with significant class imbalance, we analysed the minority classes by looking for specific patterns and class characteristics to explain why those classes are exceptions. Thus, we strive to explain the rare cases when it is more appropriate to use the fitness function corresponding to the minority class. In Appendix C, we have presented the classes that we analysed manually for the 60s and 180s datasets when comparing BO and DC in terms of branch coverage.

C.2 Comparing BO and DC in terms of branch coverage - 180s

For the 180s dataset, eleven classes achieved higher branch coverage when using DC as a fitness function, while BO dominated in 62. We first examined the static software metrics that we considered in the manual analysis for the 60s timeline, described in Section 4.1.1. However, this time some classes did not match our conclusions in Section 4.1.1 from the analysis of the 60s time limit. One such class is org.tartarus.snowball.ext.englishStemmer, which had a "medium" magnitude from the Vargha-Delaney test. This is why we decided to manually look into the test results for each of the ten runs of EvoSuite. For BO, all the values for the branch coverage varied between 0.806 and 0.82, while for the DC, they were between 0.81 and 0.82. Furthermore, in general, the two distributions were very similar. In contrast, for the ioproject.server.network.ClientGroup class, the branch coverage result for BO varies between 0.65 and 0.89, while for the DC, it is between 0.8 and 0.93. Consequently, we decided not to consider org.tartarus.snowball.ext.englishStemmer in the analysis. All other classes with "medium" magnitude had notable differences. The resulting table is showed in Appendix C, Figure 25. From our observation, again, we can say that the complexity affects the performance of the fitness functions. The class jigl.image.levelSetTool.LevelSetNudge has a very high "numbers", "assignments" and "mathematical operations" quantity. However, it has LCOM of zero and WMC and LOC of around three times less than the average for the all classes in the dataset. The class com.yahoo.platform.yui.compressor.JavaScriptCompressor does not correspond to the above assumptions; only the value of LCOM is low. Additionally, org.joda.time.DateTimeZone and org.joda.time.MutableDateTime have a similar code structure. To sum up, as in the general case the assumption that DC will perform better when the complexity of the classes is low holds, there are exceptions for the 180s time limit.

C.3 Comparing BO and DC in terms of branch coverage - 300s

For the 300s dataset, the above assumptions do not hold. It is interesting to note that org.joda.time.DateTimeZone and org.joda.time.MutableDateTime are again in the classes with better performance for the DC fitness function. As both classes are from the same project, their code structure does not contain any specific characteristics. Hence, we can propose that the good performance of DC for these classes could be related to inheritance or some internal code structure, specific for the "joda" project. However, no other common characteristics describing the five classes with better performance for the DC fitness function were found.

class	wmc	lcom	loc	returnQty	loopQty	tryCatchQty	parenthesizedExpsQty	numbersQty	assignmentsQty	mathOperationsQty
com.google.javascript.jscomp.ReferenceCollectingCallback	65	65	195	13	1	0	5	5	31	2
com.mentorgen.tools.profile.runtime.Profile	55	62	238	16	3	1	1	4	38	5
com.yahoo.platform.yui.compressor.JavaScriptCompressor	269	165	965	18	27	0	22	125	174	65
ioproject.server.network.ClientGroup	30	0	94	13	4	0	0	0	14	0
jigl.image.levelSetTool.LevelSetNudge	64	0	219	2	10	0	12	2269	119	109
org.apache.commons.math3.fraction.Fraction	79	163	275	33	1	0	15	73	81	30
org.jfree.chart.plot.ValueMarker	10	0	37	6	0	0	1	1	4	0
org.joda.time.DateTimeZone	158	957	615	64	1	11	13	43	114	46
org.joda.time.MutableDateTime	119	3474	442	29	0	1	3	20	35	2
org.quickserver.net.client.monitoring.HostMonitoringService	51	177	221	8	3	3	1	4	37	10
Average among classes that perform better when using the Default configuration fitness function	90	506.3	330.1	20.2	5	1.6	7.3	254.4	64.7	26.9
Average among all classes with significant difference – BO / DC 180s, in terms of branch coverage	179.301	2132.26	706.56	51.65753	16.301	4.054795	18.83562	147.83562	127.0410959	49.50685

Figure 24. Classes with higher branch coverage when using Default configuration as a fitness function in comparison with Branch + Output diversity for 180s

Classes with higher branch coverage when using the Default configuration as a fitness function – BO/ DC 60s

Classes	Differences			
	Large	Medium	Small	Negligible
org.javathena.login.parse.FromClient		X		
org.apache.commons.math3.distribution.fitting.MultivariateNormalMixtureExpectationMaximization	X			

Classes with higher branch coverage when using the Default configuration as a fitness function – BO/ DC 300s

Classes	Differences			
	Large	Medium	Small	Negligible
Newzgrabber.Downloader	X			
gnu.trove.impl.hash.TShortHash			X	
org.joda.time.DateTimeZone	X			
org.joda.time.MutableDateTime	X			
umd.cs.shop.JSJsho	X			

Classes with higher branch coverage when using the Default configuration as a fitness function – BO/ DC 180s

Classes	Differences			
	Large	Medium	Small	Negligible
com.google.javascript.jscomp.ReferenceCollectingCallback	X			
com.mentorgen.tools.profile.runtime.Profile	X			
com.yahoo.platform.yui.compressor.JavaScriptCompressor		X		
ioproject.server.network.ClientGroup		X		
jigl.image.levelSetTool.LevelSetNudge	X			
org.apache.commons.math3.fraction.Fraction	X			
org.jfree.chart.plot.ValueMarker		X		
org.joda.time.DateTimeZone	X			
org.joda.time.MutableDateTime	X			
org.quickserver.net.client.monitoring.HostMonitoringService	X			
org.tartarus.snowball.ext.englishStemmer		X		

Figure 25. Classes with higher Branch coverage when using Default configuration as a fitness function in comparison with Branch + Output diversity - differences comparison

class	wmc	lcom	loc	returnQty	loopQty	tryCatchQty	parenthesizedExpsQty	numbersQty	assignmentsQty	mathOperationsQty
org.javathena.login.parse.FromClient	25	1	90	10	0	0	1	37	15	2
org.apache.commons.math3.distribution.fitting.MultivariateNormalMixtureExpectationMaximization	33	5	173	3	16	0	3	38	74	17
Average among classes that perform better when using the Default configuration fitness function	29	3	131.5	6.5	8	0	2	37.5	44.5	9.5
Average among all classes with significant difference – BO / DC 60s, in terms of branch coverage	140.35	1514.1	587.68	52.06667	16.45	1.933333	14.31667	181.36667	109.7833333	40.86667

Figure 26. Classes with higher Branch coverage when using Default configuration as a fitness function in comparison with Branch + Output diversity - differences comparison

D Appendix D: Analysis for the comparison of combination of Branch coverage and Output diversity and Branch coverage/ Default configuration fitness functions in terms of branch coverage - 300s

D.1 BO/ BC

For the 300s time limit dataset, from Figure 10 we observe consistency with the selected features for the top ten best performing configurations. However, for this dataset, the selected features are very different from the ones we observed from the data for the one and three-minute time limit. The best performing classifier is the Decision tree, with an accuracy of around 0.95. The obtained tree can be observed in Figure 11. The class metric FANOUT counts the number of output dependencies a class has, i.e., the number of other classes referenced by a particular class [6]. From the obtained tree model, we can see that for FANOUT more than eight and a number of string literals less than thirteen, it is better to choose BO as a fitness function. In the other cases, the BC fitness function will achieve better coverage. Because the chosen class metrics have nothing in common with the one established for the 60 and 180 seconds datasets, the accuracy is around 0.95, and the model is prone to overfitting, we also checked the model manually. Interestingly, the decision tree correctly classifies all but one class and achieves an accuracy of around 95%. Therefore, we can claim that for the 5-minute dataset, we can choose the better performing fitness function for the class under test in terms of branch coverage using the proposed tree on Figure 11.

D.2 BO/ DC

For comparing BO and DC in terms of branch coverage, when the time limit is 300s, the best classifier is DecisionTree, with an accuracy of around 90%. In Appendix B, Figure 15 shows the top ten classifiers and the selected features for every configuration of the data analysis tool. From Figure 15 we can see that only the metrics considering lack of cohesion of methods are present in all configurations on Figure 15. Thus, the LCOM/ LCOM* should be very important for choosing the optimal fitness function. Because of the random factor in the Decision tree and the very imbalanced dataset, we run the data analysis tool for this set several times. We obtained similar trees with a very high accuracy of around 90%. However, they still had some differences. Therefore, after manually validating the trees, we combined their decision points into one tree, presented in Figure 16. From the tree, we can see that the classes that perform well when using DC as a fitness function have high values for LCOM*, more than 0.83. Also, the CBO of these classes is more than 11, while the number of loops is less than eight. The final condition considers the lines of code and requires the classes to have LOC less than 624 in order for the Default configuration to perform better than BO. If the classes satisfy all four conditions, DC will outperform BC for this dataset.

From our manual data analysis, we can claim that the decision tree on Figure 16 correctly classifies all available CUT. Hence, the tree in Figure 16 can correctly predict which fitness function - BO or DC will achieve better branch coverage when the time limit is 300s.