Preprint: Bartosz Boczar, Michał Pytka, and Lech Madeyski, Which Static Code Metrics Can Help to Predict Test Case Effectiveness? New Metrics and Their Empirical Evaluation on Projects Assessed for Industrial Relevance, pp. 201–215. Cham: Springer International Publishing, 2022. DOI: 10.1007/978-3-030-77916-0_8

Preprint: https://madeyski.e-informatyka.pl/download/BoczarEtAl22.pdf

Which Static Code Metrics Can Help to Predict Test Case Effectiveness? New Metrics and Their Empirical Evaluation on Projects Assessed for Industrial Relevance

Bartosz Boczar, Michał Pytka, Lech Madeyski

Abstract One of corner stones of software development are test cases, which help in assessment of created production code. As long as they are properly designed, they have a capacity to capture faults. In order to check whether tests are well made, different procedures have been established, like statement coverage or mutation testing, to evaluate their performance. This has an obvious downside of being computationally expensive and as such is not employed on a wide enough scale. Finding solutions to increase efficiency of assessing test cases, could lead to a more widespread adoption and for that reason we investigate one such approach. We tested possibility of predicting test case effectiveness, strictly on a basis of static code metrics of production and test classes. To solve this task we employed three different learning classifiers, to check feasibility of the process and compare their performance. We created our own set of metrics all of which were later assessed for their impact on prediction. Out of seven most impactful predictors, four of them were proposed by us: Number Of test Cases used in Test class (NOCT), Number Of Defined Variables in a class (NODV), Number of New Objects created in a class (NONO), Number Of Assertions used In Test class (NOAIT). Created models yield a promising result, with best of them achieving over 85% for both *F-Measure* and *Precision* along with 73% for *Matthews Correlation Coefficient*. With the fact of well balanced data used in creation of model, it is safe to assume, that they hold some merit. All steps taken to achieve this result are explained in detail.

Bartosz Boczar
Wroclaw University of Science and Technology, e-mail: 238067@student.pwr.edu.pl

Michał Pytka
Wroclaw University of Science and Technology, e-mail: 233146@student.pwr.edu.pl

Lech Madeyski
Wroclaw University of Science and Technology, e-mail: Lech.Madeyski@pwr.edu.pl

1 Introduction

In order to implement reliable software, the programmers need to prepare set of tests that could indicate the errors present during the execution of the code. For that reason tests implementation is an important process. One of the best methods of estimating whether the test is good is mutation testing. Unfortunately this is a time consuming process. The studies performed by Grano et al. (2019) prove that this process can be simulated using *Machine Learning* methods, which costs significantly less time.
In our paper we present our attempts at reproducing study byGrano et al. (2019). Furthermore, we describe the process of using similar *Machine Learning* methods to create a model able to estimate effectiveness of the tests. However, our solution does not use coverage, a measure which requires much greater computational power in order to be computed, compared to other measures. Instead we present a solution which uses metrics provided by JavaMetrics tool (Ziobrowski, 2020), earlier used by Grodzicka et al. (2020), and our new set of proposed metrics.
Our studies can be reproduced with the package provided by us. All steps needed to recreate this study are included in the appendix of this paper and appropriate scripts are available in our repository[1].

Our contributions in this paper are as follows:

- An attempt to reproduce study by Grano et al. (2019) including a detailed list of issues we encountered.
- Completely new replication package addressing the issues in the original package by Software Evolution and Architecture Lab (2018).
- Extending the study by Grano et al. (2019) to different (class and function level) samples and projects assessed for industrial relevance Madeyski and Lewowski (2020).
- Extension of the data sets proposed by Grano et al. (2019), by new metrics related to source code, code coverage, and Java features related to testing.
- Empirical evaluation of the new metrics as predictors of mutation score indicator.

2 Literature review

Our literature review focused on finding work related to the subject of mutation testing and finding bad code smells. Our main goal was extending the MLCQ data set Madeyski and Lewowski (2020) by new metrics. We wanted to find simple metrics that would identify the usage of new java features in code. That way we could also verify whether those features increase or decrease the chances of bad smells.

[1] `https://github.com/pwr-pbrwio/PBR20M2`

2.1 Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators

Study by Grano et al. (2019) is a highly relevant paper that we tried to reproduce. The main goal of Grano et al. (2019) was to create a model which uses code quality metrics to estimate the effectiveness of the tests without creating and running mutation tests. The model was trained using good and bad tests of 18 projects. They divided tests into good and bad ones using their mutation score. To compute mutation score, they used *PIT*, which provides 13 mutation operations. They considered 67 code quality factors and 5 dimensions: Code Coverage, Test Smells, Code Metrics, Code Smells and Readability. In total, Grano et al. (2019) used 18 open source Java projects.
Their first task was to determine whether there is a relationship between the chosen quality factors and the effectiveness of the tests. The effects of their work indicate that the greater the number of statements executed by test cases tend to improve their effectiveness. Production code metrics also seem to be linked to test effectiveness, which indicates that the code with higher quality is better at finding faults in the production code. Similarly the results show that code smells can cause test cases to be less effective, although test smells do not appear to have an impact on test effectiveness.
Then they proceeded to check to what extent their tool can be used to estimate the effectiveness of test cases as compared to mutation score. To answer this question they used different Machine Learning classification algorithms: *Random Forest*, *K-Neighbours* and *Support Vector Machines*. This lead to the creation and comparison of 6 different models as each algorithm was used to create dynamic and static one. A dynamic model uses the same data set as static model extended with coverage because this is dynamic feature. Their experiments indicate that *Random Forest* model achieves better results when it comes to evaluation metrics than *K-Neighbours* and *Support Vector Machines*. Their study shows that *Random Forest* dynamic model could achieve results of 95% in terms of metrics like *F-Measure* and *AUC-ROC*, which confirms that Machine Learning models can be used for effective estimation of test cases. It also appears that static model performs a little worse than dynamic one, which has decreased performance by about 9%.
Finally they discussed how the created model can be used. One of the most important possible usage would be integrating the model within the code analysis software to let the developers diagnose their code. That would give them desired information about potential effectiveness of test cases. Those information might be used to discard non-effective tests or to study them to understand which operations cause test cases to be non-effective. The model can also be used simply as an alternative to standard mutation testing. This solution could save developers their time as mutation testing is a time consuming method.

2.2 Predictive Mutation Testing

Zhang et al. (2019) concerns mutation testing approaches. As that technique involves a lot of computationally expensive operations, they proposed a novel classification model, which would be used to predict, whether a mutant is to be killed or not. This would be the main factor increasing the performance of tests, obtaining results for mutants without executing them. Features used in that prediction were gathered with the dynamic technique called PIE (propagation, infection, and execution) in mind (Voas, 1992).

According to PIE, program's computational behaviour can be estimated based on three characteristics: probability of particular code section execution, probability of that section influencing data state and probability of produced data state influencing the output. With this in mind, selected features for *PMT* would indicate either the execution of mutated statement, infection of the program state after the mutated statement was executed or propagation of the infected program state leading to different output. With features in place, several algorithms for the model were tested, leading to selection of the best performing one, namely *Random Forest*.
The pipeline of creating a prediction is as follows: Selecting a project for which testing is to be done, taking its previous release as training set, extracting the features from both of gathered versions, teaching the model and finally predicting results. Initial testing comprised of 9 projects, to make initial judgements and later expanded to 154 projects. Obtained results were promising. A test was conducted, in order to check performance of this approach under imbalanced data. This was done by, selecting two groups of projects, one with mutation score lower than 0.2 and second with score higher than 0.8. This yield a result, that the process is feasible despite the selecting inherently imbalanced data. Result analysis with research questions, shows that effectiveness of this method rivals the traditional execution of mutants and it is significantly more efficient time wise. Additional data balancing steps were not impacting results negatively nor positively, which was attributed to the *Random Forest* and its ability to handle imbalanced data.
Features themselves were analysed in the context of their merit in prediction. Execution and propagation features showed a visible impact on performance, while the infection ones made no difference. This was attributed to the fact, that object oriented design limits the propagation of infections, between test outcomes and such features would be more impactful in procedural languages. Finally a consideration on predictability was presented. In most projects, mutants with high predictability were the most numerous group. Additionally mutants with higher amount of executions are more predictable along side mutants with high killability.

2.3 Comparison of Lightweight Assessment and Predictive Mutation Testing

As Grano et al. (2019) and Zhang et al. (2019) are studies from which we base our work, it is important to compare and contrast those two works.

- Outcome of classification: Grano et al. (2019) classifies tests to one of two classes: good or bad. Zhang et al. (2019) classifies whether the mutant will be killed or not and uses this knowledge to calculate mutation score.
- Focus of work: Grano et al. (2019) focuses on static code-quality features as predictors of test quality. Additionally coverage of test cases is considered as such predictor. Zhang et al. (2019) focuses on predicting outcomes of mutants
- Data used in their prediction: Grano et al. (2019) analyses code smells, test smells, code coverage and readability. Zhang et al. (2019) considers features of generated mutants, along with code quality.
- Research questions: Grano et al. (2019) enquire about relationship between code quality and test-case effectiveness, see to what extent test case effectiveness can be estimated and distinguish the most important factors in predicting effectiveness. Zhang et al. (2019) enquires about how effective is this approach in predicting whether mutants survive or get killed, how different application scenarios (cross-project and cross-version) influence the prediction, applicability of the model, impact of different features on outcome.
- Goal of work: Grano et al. (2019) focuses on characterising effective test cases based on static features of code, finding impactful factors and informing about them and estimating quality of code based on said factors. Zhang et al. (2019) focuses on testing performance of *PMT*, evaluating the effect that different classification parameters had on it, checking effectiveness under two different application scenarios, checking viability of the proposed approach to evaluating mutants, identifying features that are the most impactful, finding characteristics of mutants that are hard/easy to predict.
- Effect of work: Grano et al. (2019) aims to give feedback to developers based on easily accessible measures of static code, inform them about problematic parts of code. The model created in their study might be integrated with already existing software analysis tools. Zhang et al. (2019) aims to speed up tremendously the process of mutation testing, clarify the connection between mutant features and code quality. They present the new methodology which decreases the time necessary for the process.

3 Methods and Materials

Our first task was reproducing the package provided by Grano et al. (2019) to verify whether we can use their model in our research and extend it by our own metrics. Then we wanted to prepare our own set of metrics which would be implemented in a

tool *Java Metrics* by Ziobrowski (2020). The last step was to create a model with those metrics, check the model's effectiveness and find the most important metrics in this model.

3.1 Study reproduction

Before we reproduced the package by Grano et al. (2019), we had to solve all the issues encountered while building all the projects. We had to build each of the projects separately and fix the ones that failed to build. Then we proceeded to generate mutation tests and create the model.
Mutation tests using *PIT* and appropriate generated scripts were completed successfully without significant issues. This step ended with scores for each test case, that had to be aggregated.
Scripts responsible for aggregating data, had some bugs, that needed changing in order to complete their described tasks. All of the mentioned bugs were technical in their nature, detailed description of needed changes will be provided in appendix.
As we were analysing the code responsible for creating the model, we noticed that the input data used for training was included in the git repository Software Evolution and Architecture Lab (2018). The mutation tests generated during previous steps of our reproduction were not used which means that the whole process had no effect on the final outcome. The training data was pre-calculated using tools that have not been published yet. Therefore we are not able to correctly prepare data of model training and our reproduction of study by Grano et al. (2019) is not successful.

3.2 New metrics propositions

We decided to propose our own metrics which could be used to estimate tests effectiveness. These metrics are supposed to replace the coverage, because calculating coverage is a time consuming process. We would like to continue Grano et al. (2019)'s idea to create a lightweight solution. For that reason we performed further studies on the same set of tests used by Grano et al. (2019). The list of metrics we proposed is as follows:

- *ALU* - Assertion Library usage, the metric shows if the class uses Assertion Library. The list of considered libraries: Hamcrest, Assertj, Atrium, Truth, Valid4j, Datasource-Assert;
- *NOAIT* - Number of assertions used in test class;
- *NOASIT* - Number of assumptions used in test class (Junit 5.0 feature);
- *NOAU* - Number of @After annotations used in test class;
- *NOBU* - Number of @Before annotations used in test class;
- *NOCT* - Number of test cases used in test class;
- *NODT* - Number of dynamic tests in a test class (Junit 5.0 feature);

- *NODV* - Number of defined variables in a class;
- *NOECU* - Usage of ErrorCollector in class (JUnit 4.0 feature);
- *NOEET* - Number of expected exception tests;
- *NOET* - Number of extended tests (Junit 5.0 feature);
- *NOFS* - Number of for statements in a class;
- *NOIS* - Number of if statements in a class;
- *NONO* - Number of new objects created in a class;
- *NOPT* - Number of parametrized tests (Junit 5.0 feature);
- *NORT* - Number of repeated tests (Junit 5.0 feature);
- *NOST* - Number of nested tests (Junit 5.0 feature).

To calculate metrics we decided to use a tool *Java Metrics* by Ziobrowski (2020), which we extended by implementing our metrics. The metrics that were already present in *Java Metrics* were also used to train the model.

3.3 Model creation

Our process of creating a model started with selecting 18 projects, which were also used in Grano et al. (2019) and gathering them from their GitHub repositories. After that manual building of those projects had to be done, as this process proved to be too unfeasible to be completed automatically. With projects prepared, their static code metrics were computed using *Java-Metrics* and outputs stored for later use. Next mutation testing has to be done, in order to compute mutation score, which was done with use of scripts from Software Evolution and Architecture Lab (2018) reproduction package. Generated mutation values have to be then gathered. With both metrics and mutation scores ready, preprocessing of data could be done, where metrics of production classes get merged with metrics of their test classes and appropriate mutation score. Additionally *MIN*, *MAX*, *MEAN* of McCabe Complexity are calculated for each of classes. Final step in preparing a data frame is assigning a classification for all of the computed rows, using first quartile of mutation scores as *Bad Tests* and forth as *Good Tests*. This yield *1342* rows of data, with exactly even split of *671* for both of classifications.
As classifier selection has a strong influence on outcome of classification, we chose to test three different ones: *Random Forest*, *K-Neighbours* and *Support Vector Machines*. Our method of training and testing was nested cross-validation. This choice was made, to ensure best possible parameter selection for each of the models. Following are set of tuned parameters:

- Random Forest
  - *ntree*: (300:500)
  - *mtry*: (10:40)
  - *nodesize*: (5:20)
- K-Neighbours

  - *k*: (1:20)
  - *distance*: (0.001:2)
- Support Vector Machines
  - *cachesize*: (20:150)
  - *cost*: (30:80)
  - *gamma*: (0.00001:0.01)

4 Results

The following are model performance parameters obtained after their generation:

Table 1 Table showing the performance results of classification algorithms. Fbeta parameter is the name of F-Measure in mlr3 package

	Mcc	Ce	Precision	Fbeta
Random Forest	0.729	0.135	0.880	0.853
KNN	0.703	0.148	0.876	0.837
SVM	0.631	0.184	0.802	0.810

Interestingly judging their performance based on *Matthews Correlation Coefficient* different hierarchy of classifiers was obtained, as compared to Zhang et al. (2019) and Grano et al. (2019). *Random Forest* still performed the best, but *K-Neighbours* scored higher than *Support Vector Machines*.
Additionally we looked at the *Importance* of parameters, that was generated alongside our *Random Forest* model, to later evaluate how impactful are certain predictors.

5 Discussion

Some of the key results for our study is the performance of obtained models, as ultimately, that was one of the goals. With precision of 0.880 and a balanced data set, it can be assumed that it has some merit and could be valuable when put into a real life scenario. When judging classifier performance, we mostly tried to maximise for *Matthews Correlation Coefficient*, as it takes into account all parts of a confusion matrix.
To compare our results, we looked at final performance values obtained in Grano et al. (2019), to see how effective our process is, when compared to other contemporary work. Scenario we looked at, was the one without *Code Coverage*, as it resembles our study the most. With values of precision equal to *0.880* and F-measure equal to *0.853*, and values of both metrics achieved by Grano et al. (2019) equal to *0.864*, it

Table 2 Table showing the importance of the metrics. Positions with *.x suffix reference production class metrics and *.y suffix reference test class metrics.

Metric name	Importance	Metric name	Importance
MAX_CYCLO.x	0.0493	NOMR_C.x	0.0009
MEAN_CYCLO.x	0.0453	MRD.x	0.0008
NOCT.y	0.0429	LD.x	0.0006
NODV_C.y	0.0355	NOPA.y	0.0005
NOPA.x	0.0336	NOL_C.x	0.0003
Nono_C.x	0.0310	MIN_CYCLO.y	0.0002
NOAIT_C.y	0.0293	LD.y	0.0002
NPM.x	0.0267	NOAU.y	0.0002
NOAM.x	0.0257	NOL_C.y	0.0001
LOC_C.y	0.0243	ALU.y	0.0001
WMCNAMM.y	0.0220	NOAIT_C.x	0.0001
WOC.y	0.0191	ALU.x	0.0000
LOC_C.x	0.0191	NOMM.y	0.0000
WMC.y	0.0170	MIN_CYCLO.x	0.0000
WMCNAMM.x	0.0163	NOASIT_C.x	0.0000
WMC.x	0.0155	NOAU.x	0.0000
NOM.x	0.0154	NOBU.x	0.0000
NOM.y	0.0151	NOCT.x	0.0000
NPM.y	0.0146	NODT_C.x	0.0000
WOC.x	0.0143	NODT_C.y	0.0000
NODV_C.x	0.0137	NOECU_C.x	0.0000
NOIS_C.x	0.0136	NOECU_C.y	0.0000
Nono_C.y	0.0128	NOEET.x	0.0000
NOPV.x	0.0117	NOET.x	0.0000
NOAM.y	0.0102	NOET.y	0.0000
NOFS_C.y	0.0076	NONT.x	0.0000
MEAN_CYCLO.y	0.0067	NONT.y	0.0000
NOMM.x	0.0056	NOPT.x	0.0000
NOFS_C.x	0.0047	NOPT.y	0.0000
NOBU.y	0.0041	NORT.x	0.0000
NOEET.y	0.0035	NORT.y	0.0000
MAX_CYCLO.y	0.0029	NOMR_C.y	0.0000
NOPV.y	0.0020	MRD.y	0.0000
NOIS_C.y	0.0012	NOASIT_C.y	0.0000

can be said that both solutions are similarly capable.

We also would like to remark on performance of some of the predictors used in the classification. As shown by Grano et al. (2019), *Code Coverage* was the most important information in all of the predictions and because of this, metrics which are the most closely connected to it have highest importance values, namely maximal cyclomatic complexity for production class and mean cyclomatic complexity for production class. This could be lead to an interpretation, that a single complex method of a class, decreases its quality. Perhaps what is surprising, is the fact that out of seven most impactful predictors, four of them were proposed by us (NOCT, NODV and NONO for test classes and NOAIT for production classes). Among those *NODV*, number of defined variables for test class, scores highly. This was an unexpected

finding, seeing such correlation between this metric and test case effectiveness. A significant portion of predictors, have an importance factor of zero, meaning they have no bearing on the outcome. Given further exploration of this topic, those parameters would have to be omitted, due to being of no relevance.

Table 3 Table showing the importance of metrics proposed by us. Positions with *.x suffix reference production class metrics and *.y suffix reference test class metrics.

Metric name	Importance	Metric name	Importance
MAX_CYCLO.x	0.0493	NOAIT_C.x	0.0001
MEAN_CYCLO.x	0.0453	ALU.x	0.0000
NOCT.y	0.0429	MIN_CYCLO.x	0.0000
NODV_C.y	0.0355	NOASIT_C.x	0.0000
Nono_C.x	0.0310	NOAU.x	0.0000
NOAIT_C.y	0.0293	NOBU.x	0.0000
NODV_C.x	0.0137	NOCT.x	0.0000
NOIS_C.x	0.0136	NODT_C.x	0.0000
Nono_C.y	0.0128	NODT_C.y	0.0000
MEAN_CYCLO.y	0.0067	NOECU_C.x	0.0000
NOFS_C.x	0.0047	NOECU_C.y	0.0000
NOFS_C.y	0.0076	NOEET.x	0.0000
NOBU.y	0.0041	NOET.x	0.0000
NOEET.y	0.0035	NOET.y	0.0000
MAX_CYCLO.y	0.0029	NOPT.x	0.0000
NOIS_C.y	0.0012	NOPT.y	0.0000
MIN_CYCLO.y	0.0002	NORT.x	0.0000
NOAU.y	0.0002	NORT.y	0.0000
ALU.y	0.0001	NOASIT_C.y	0.0000

6 Conclusions

In our studies we proposed new metrics that could be used to estimate test effectiveness. We tried recreating studies done in Grano et al. (2019), but were unable to finalise the process, which in turn lead to creation of our proposed solution. Despite using separate process, albeit similar to tested reproduction package, we created a model, which performed similarly well to the one created by Software Evolution and Architecture Lab (2018). After analysing the importance computed by the *Random Forest* model we discovered that some of those metrics might be useful features in test effectiveness estimation.

We have documented the whole process of our studies and provided reproduction package that can be used by anyone who would like to reproduce our studies.

In the future we could expand our studies and examine model with only relevant metrics. We could also propose and implement more metrics that might be useful features for test effectiveness estimation.

7 Appendix: Reproducibility of the presented research

This section presents materials and steps required to reproduce the presented research. Details are available at `https://github.com/pwr-pbrwio/PBR20M2/blob/master/README.md`.

7.1 Study reproduction

In the process of recreating the Software Evolution and Architecture Lab (2018) we encountered several issues, which needed to be solved in order to yield a valid result. We documented them, to transparently show what was changed.

7.1.1 Fixing the process of building projects

In order to reproduce study by Grano et al. (2019), we cloned the *git repository*[2] and attempted to follow instructions presented in the `README.md` file. The reproduction was performed on Ubuntu machine with the following software versions:

- Maven 3.6.0 [3]
- Python 3.6 [4]
- Java 1.8 [5]

The main problem was encountered during the execution of the script `get_projects.sh`. The script clones each project, that was used in the study, from its git repository and performs maven installation. Unfortunately a few projects failed to build successfully. The first project that could not be built was *gson*. The project uses flags supported since *JDK* 1.9. We managed to resolve the issue by using *JDK* 14 to build this project. We also had to correct the source and target version in the `pom` file to 1.9.
The next project that had to be fixed was *cat*. While installing the project maven did not manage to download maven-source-plugin specified in `pom` file. The issue is caused because The Central Repository does not support HTTP communication since January 15, 2020. The repository can be accessed only with HTTPS. We resolved the issue by correcting urls associated with maven repository in *pom* file to use HTTPS communication.
Another problematic project was *RXJava*. The project uses *Gradle*[6] and the script uses maven to build it. The problem could be resolved by chaining the script to use *gradle install* instead of *mvn install* for *gradle* projects. However this would cause

[2] `https://github.com/sealuzh/lightweight-effectiveness`

[3] Miller et al. (2010)

[4] Van Rossum and Drake (2009)

[5] Arnold et al. (2005)

[6] `https://gradle.org/`

problems during the process of creating mutation tests. For that reason we converted the project to *maven* by generating a new `pom` file. In the generated `pom` file we specified target and source version to be 1.9.
The last project that caused us problems was Opengrok. We resolved the issue by installing necessary software present on the git repository. Furthermore, the building process had trouble with generating *javadoc*. The *javadoc* comments were not necessary so we changed the `pom` settings to skip the generation process.

7.1.2 Fixing python script problems within the package

In order to complete the reproduction, a handful of python scripts has to be executed in order as presented in the read me in *git repository*[7]. In process of recreating the study, several issues has been found within them.
Script *calculate_results.py*[8] was responsible for gathering all mutation scores of all test cases into a csv file. In function of the same name (that is *calculate_results*), one of operations required a path to a directory with mutation results. String representing the path was not being created in a correct way, leading to finding no scores to gather. This was fixed by changing the string creation.
Script *aggregate_sources.py*[9] was responsible for combining data about mutation scores with data on code metrics into a single file and dividing gathered data into sets of good and bad tests. This script was placed in a different directory, than the one mentioned in the read me. The function *process_results* which combined the data, would in its first step check for the existence of different files and directories. Some of the files (like *test_readability.csv*[10] or *source_readability.csv*[11]) were not present in the expected locations, which would lead to immediate halt of the script. This can be solved in two ways, either modify the structure of the package and place all file accordingly, or modify the code to accommodate for the different location. In the process of recreation the first option was picked. Additionally it is important to note, all files were present in the package, but their locations were different.
Script *plots.py*[12] was responsible for creating plots after classification was complete. Single problem with this script, was connected with a use of external service called *Plotly*. In order to use it, in the script one has to provide credentials for their account. Line of code, which authorises the user with given credentials was left with

---

[7] `https://github.com/sealuzh/lightweight-effectiveness`

[8] `https://github.com/sealuzh/lightweight-effectiveness/blob/v1.0/effectiveness/mutation/calculate_results.py`

[9] `https://github.com/sealuzh/lightweight-effectiveness/blob/v1.0/effectiveness/metrics/aggregate_sources.py`

[10] `https://github.com/sealuzh/lightweight-effectiveness/blob/v1.0/metrics/test_readability.csv`

[11] `https://github.com/sealuzh/lightweight-effectiveness/blob/v1.0/metrics/source_readability.csv`

[12] `https://github.com/sealuzh/lightweight-effectiveness/blob/v1.0/effectiveness/classification/plots.py`

placeholder user information. This would lead to an error and was solved simply by removing the line.

7.1.3 The issue unable to be solved

The process of generating mutation tests resulted in creating 26 *csv* files, each file represented the specific mutation operation. Half of these files contained tests classified as good ones, the other half contained tests classified as bad ones. The code responsible for creating the model imported only the 2 of the 26 files which were already present in the repository (one with good tests, the other one with bad tests). Each of the generated files has the size of around 300 *kB*, while the 2 files used for the training of the model have the size of around 400 *kB* each.
We do now know what causes the difference in the size. The problem with classifier data was two fold. If data provided in the package was using only one mutation operator, then we did not know which one that was and why our data sets had smaller amount of entries. On the other hand, if their data includes all operators, the size of our data set is significantly bigger. For that reason we were forced to stop our attempts at reproducing the study by Grano et al. (2019).

7.2 Chosen Environment

We decided to use *R* programming language to prepare our own classification model, because it provides useful *Machine Learning* libraries. We used *mlr3* library to create models and tune them. This library provides numerous objects that let the user perform learning, re-sampling and analysing of *Machine Learning* models. There are also many additional packages that can further extend functionalities provided by *mlr3*.
The script we implemented creates three, each with different classification algorithm: *Random Forest*, *K-Nearest Neighbours* and *Support Vector Machine*. The models are tuned using *Nested Cross Validation* with 10 folds. We wanted to compare results of our study to the results achieved by Grano et al. (2019). For that reason we chose the same algorithms to create *Machine Learning* models.

7.3 Reproduction instructions

As our attempt at reproduction of aforementioned package was unsuccessful, this meant that we could not improve on it either and created a separate one. Our package was created from ground up, but with some use of scripts from Software Evolution and Architecture Lab (2018). All of used scripts still reference the original author. List of needed tools to recreate the process completely is as follows:

- Ubuntu 18.04
- Maven 3.6.0
- Python 3.6 with *pandas* 1.0.4 package installed
- Java 1.8
- R 3.4.4 with *pacman* package installed

In order to reproduce our studies you need to clone our git repository. Then you need to clone git repositories of the projects used in our studies into the *projects* folder. The full list of used projects is listed in the *projects.csv* in our repository. However, if you prefer you can use your own projects. In order for the package to work with external projects, their names have to be added to the *projects.csv* as well.
In order to prepare the static code metrics, our fork of Ziobrowski (2020) has to be used. It is available on this *git repository*[13]. Instructions on how to build and use this tool are present on the repository page. After outputs from selected projects are computed, *.csv* files with metrics have to be put into *javametrics_outputs* directory.
The next step is to build all the projects. To build the project you can open terminal in the project's root folder and use the command *mvn clean install -DskipTests*. Keep in mind that all the projects must be build successfully. You need to resolve any issues Yourselves and try to build the project again if the building process ends with failure. Once the project is built successfully you should run unit tests with the command *mvn test -Dmaven.test.failure.ignore=true*, which will also ignore failed tests.
In order to generate mutation tests and execute them, the same scripts were used, as in Software Evolution and Architecture Lab (2018). All of them, are being executed through our *runExternalScipts.R*, to stay in single environment. It is important to note, that this script will take a considerable amount of time to complete. Outcome of this script is *mutationScoresGathered.csv*, which holds mutation scores of all executed tests. Alternatively, if any trouble would be met while executing this script, all of the external scripts could be called individually in the following order:

- *generate_script.py*
- *run_experiment_ALL.sh*
- *gatherMutations.py*

In order to execute python scripts, the *PYTHONPATH* variable has to be set on the root of the project. Additionally the bash script has to have it's mode changed to *777*, as advised previously in Software Evolution and Architecture Lab (2018). Both python scripts reside in *python_scripts* and the bash script is an outcome of the *generate_script.py*.
The final step is to create the model. In order to do that you need to run the *preprocessing.R* script in the *r_scripts* folder. The script will create the file *cleanData.csv* containing all the data prepared for machine learning. Finally you should run *basePipeline.R*, which will train 3 models each with different classification algorithm: *K-Neighbours*, *Support Vector Machines* and *Random Forest*. Created models are saved in the folder *saved_models* and can be loaded into *R* environment.

[13] `https://github.com/michalpytka-pwr/JavaMetrics`

References

Arnold K, Gosling J, Holmes D (2005) The Java programming language. Addison Wesley Professional

Grano G, Palomba F, Gall HC (2019) Lightweight assessment of test-case effectiveness using source-code-quality indicators. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2019.2903057

Grodzicka H, Ziobrowski A, Łakomiak Z, Kawa M, Madeyski L (2020) Code Smell Prediction Employing Machine Learning Meets Emerging Java Language Constructs. In: Poniszewska-Marańda A, Kryvinska N, Jarząbek S, Madeyski L (eds) Data-Centric Business and Applications: Towards Software Development (Volume 4), vol 40 of book series Lecture Notes on Data Engineering and Communications Technologies, Springer International Publishing, Cham, pp 137–167, DOI 10.1007/978-3-030-34706-2_8, URL https://madeyski.e-informatyka.pl/download/GrodzickaEtAl20LNDECT.pdf

Madeyski L, Lewowski T (2020) MLCQ: Industry-relevant code smell data set. In: Evaluation and Assessment in Software Engineering (EASE2020), ACM, New York, NY, USA, DOI 10.1145/3383219.3383264, URL http://madeyski.e-informatyka.pl/download/MadeyskiLewowski20EASE.pdf

Miller FP, Vandome AF, McBrewster J (2010) Apache Maven. Alpha Press

Software Evolution and Architecture Lab (2018) Lightweight effectiveness. URL https://github.com/sealuzh/lightweight-effectiveness

Van Rossum G, Drake FL (2009) Python 3 Reference Manual. CreateSpace, Scotts Valley, CA

Voas JM (1992) Pie: a dynamic failure-based technique. IEEE Transactions on Software Engineering 18(8):717–727, DOI 10.1109/32.153381

Zhang J, Zhang L, Harman M, Hao D, Jia Y, Zhang L (2019) Predictive mutation testing. IEEE Transactions on Software Engineering 45(9):898–918, DOI 10.1109/TSE.2018.2809496

Ziobrowski A (2020) Java metrics. URL https://github.com/LechMadeyski/JavaMetrics