Mutation Testing assignment

Luigi Coniglio (Group 4)

March 12, 2018

1 Introduction

In the context of this assignment I examined seven projects written in Java, measured the branch coverage of their test suites as well as their mutation score. I then analyzed the collected data in order to investigate the correlation between coverage and size of a test suite and the correlation between coverage and effectiveness.

In this report I present the results of my experiment and compare them to previous works on the same topic. The repository at `https://github.com/werew/Mutation-testing-experiment` contains the complete coverage and mutation coverage reports generated by the tools presented in section 2.1 as well as the instructions and all the material required to easily repeat the measurements, including a script that can be used to automatically run all tests and generate the respective reports.

2 Methodology

For the goal of this experiment the following measurements have been taken for each project: project size in terms of source lines of code (SLOC), number of test cases, branch coverage, mutation score and further details about the mutants generated (number of mutants generated, mutants not covered, mutants killed and mutant survived).

The remainder of this section illustrates the subject project of the experiment and the tools and procedures used to perform the measurements.

2.1 Tools

The size of each project in terms of SLOC[1] has been measured using the tool Cloc [1]. I decided to use JaCoCo [2] for measuring branch coverage since it offers easy integration with all the dominant Java build automation systems (e.g. Maven) and is still under active development (in contrast with other tools used in older works such as CodeCover [3]). For collecting data regarding the mutation score I used PIT [4] a state of the art mutation testing system.

2.2 Subject projects

This assignment focuses on projects written in Java and having one or more JUnit test suites. Table 1 gives an overview on the projects used and provides a short description for each of them. All projects are freely available on GitHub.

The project selected have all a certain degree of popularity (in terms of starts on GitHub), with the least popular one having 1,327 stars on GitHub at the time of writing [2]. The most popular project, Retrofit [12], with its 26,708 stars is the fourth most popular Java project on GitHub at the time of writing, the measurement taken on its test suite are for this reason particularly interesting.

The projects come from different domains. Four of them are libraries while the others are applications. I decided to include, mostly due to my own curiosity, the library JavaHamcrest which coverage and mutation score has already been measured last year by Liam Clark and Jean de Leeuw in occasion of the same course [13], this in order to see if the effectiveness of the test suite has degraded or improved after one year of development (and apparently the results are very similar).

As it will be illustrated in section 3, the project picked show different characteristics in terms of number of tests cases (in relation to the SLOC) and coverage: the sample include projects with few tests and low coverage (e.g. Vectalign), few tests and high coverage (e.g. Dagger 1) and a large number of tests and high coverage (e.g. JavaHamcrest).

[1] Source lines of codes, excluding comments and whitespace

[2] I believe popular projects are in this particular case a better target since they are more likely to adopt modern development practices recognized by the programming community, and therefore likely to be more representative

Table 1: Overview on the projects used

Project	Stars on GitHub	Description
JavaHamcrest [6]	1,327	A library of matchers for testing pourposes
Vectalign [7]	1,957	Tool for create complex morphing animations
Moshi [8]	3,621	A modern JSON library for Android and Java
Spring PetClinic [9]	3,812	A sample application based on the Spring framework [5]
JavaPoet [10]	4,898	A Java API for generating .java source files
Dagger 1 [11]	6,664	A fast dependency injector for Android and Java
Retrofit [12]	26,708	Type-safe HTTP client for Android and Java

Table 2: Overview of the measurements

Project	SLOC	Test cases	Branch coverage	Mutation score	Normalized mutation score
JavaHamcrest	3333	433	95%	73%	98%
Vectalign	2416	9	48%	21%	60%
Moshi	12745	668	82%	81%	88%
Spring PetClinic	1441	41	74%	89%	96%
JavaPoet	8297	320	86%	82%	86%
Dagger 1	8894	130	74%	64%	92%
Retrofit	19184	595	80%	81%	94%

In order to effectively run the analysis, small modifications have been necessary in a couple of occasions, as for example when the use of reflection[3] in a part of a test suite conflicted with the synthetic methods added by the analysis tools (e.g. JaCoCo). However the modifications are minimal and do not impact the outcome of the analysis, furthermore it is given to the reader to opportunity to verify their nature by the mean of the provided source code available on https://github.com/werew/Mutation-testing-experiment.

3 Results

Table 2 summarizes the measurements for each project. It is important to notice that the SLOC measurement does not include the code of test suites, therefore the number of test cases and the size of the program in terms of SLOC are virtually independent (e.g. a program could be very small and have a very large number of test cases at the same time). This is particularly relevant for the analysis illustrated in section 3.1.

Given that each project used JUnit 4, the number of test cases has been simply obtained counting the occurrences of the `@Test` annotation. Branch coverage and mutation score are the ones respectively reported by JaCoCo and PIT. The normalized mutation score has been obtained dividing the number of killed mutants by the total number of mutants covered by the test suite: $\frac{\text{Mutants detected}}{\text{Mutants generated} - \text{Mutants not covered}}$ this measure let us compare the effectiveness of different test suites only in relation to the portion of code actually covered since, without normalization, the comparison between a test suite performing low in coverage and another providing higher coverage would be biased by the number of mutants not covered.

3.1 Correlation between size and effectiveness

Considering our sample, is the size of the test suite correlated to its effectiveness ? In order to answer this question we must first define a meaningful measure of size which would also take into account the dimension of a project. I decided to express the size of a test suite in terms of lines of code (SLOC) per test case: the higher is this value the smaller the size of the test suite.

Figure 1 highlights the relation between size of a test suite and effectiveness suggesting that they are indeed correlated. Computing the Pearson correlation coefficient confirms our hypotheses showing a strong negative correlation of -0.952[4].

We can conclude that the size of a test suite seems to contribute positively to its effectiveness.

3.2 Correlation between coverage and effectiveness

Is the coverage of the test suites a good measure for their effectiveness? Or in other words, are those two value correlated? Figure 2 suggests a low correlation in the case of raw (non normalized) mutation score

[3]http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html

[4]Note that the negative coefficient is given by the fact that our measure of size is inversely proportional to the size of the suite itself.

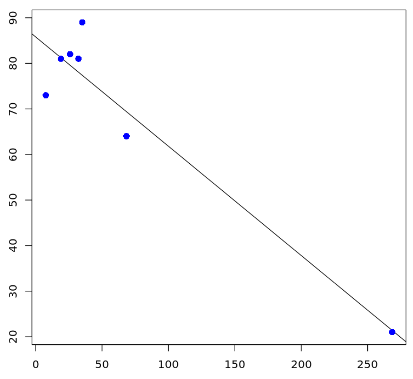
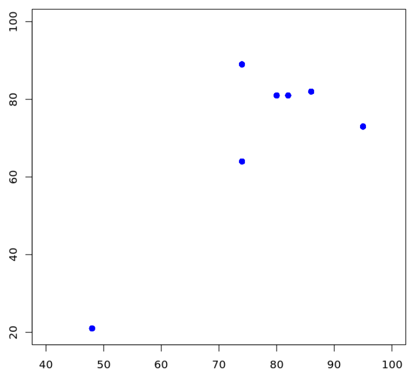
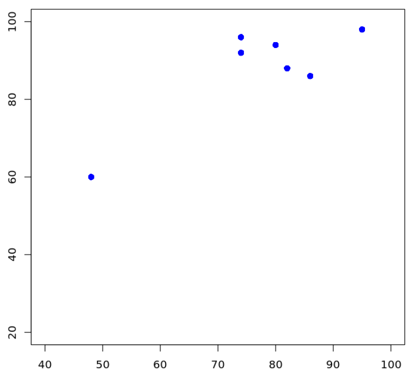


Figure 1: Plot showing the relation between lines of code (SLOC) per test case (bottom axis) and mutation score (left axis)



(a) Mutation score



(b) Normalized mutation score

Figure 2: Plow showing the relation between coverage (bottom axis) and effectiveness (left axis), this last is expressed as mutation score in plot (a) and normalized mutation score in plot (b)

and an even lower correlation when using the normalized mutation score. When comparing the two plots we can clearly see that the dots are moved upwards in the case of normalized mutation score (which is an expected behavior since the removal of the non covered mutants leads to a general improvement of the effectiveness) and their distribution gets slightly flattened. Computing Kendall τ[5] we obtain a correlation coefficient of 0.25 when using the non normalized mutation score and a coefficient of 0.19 when the mutation score is normalized. This seems to confirm the previous considerations.

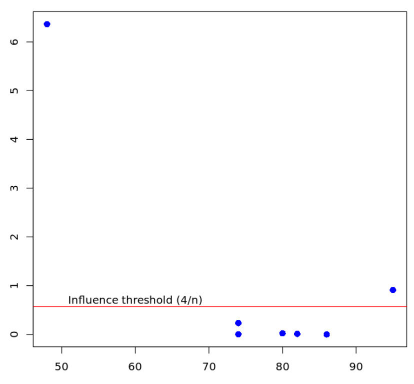
However it is difficult to draw meaningful conclusions when working on such small set of observations. Not only our sample risks to lack of statistical significance due to its size, but one outlier could be enough to produce a high or low correlation coefficient. One way to attempt to remove eventual outliers is to calculate Cook's distance, which provide us with an estimate of the influence of each point on the correlation coefficient, and subsequently remove the observations which exhibit a degree of influence above a certain threshold.

As suggested by [14] we define a threshold of $4/n$, where n is the size of our sample, and proceed to remove the points exceeding that threshold (figure 3). Computing Kendall τ we obtain now a coefficient of 0.22 for the raw mutation score and -0.14 for the normalized score. Since we can safely exclude the possibility that a higher coverage could lead to a loss of effectiveness, this last measure seems to suggest that there is a considerable drop in correlation between coverage and effectiveness when using the normalized mutation score.

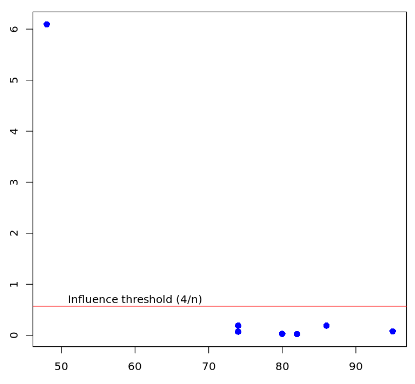
3.3 Other considerations

It is interesting to observe that libraries tends to have a better test suite effectiveness if compared to applications. The weighted average mutation score per lines of code is 80% for the libraries and 36%

[5]We compute Kendall's τ in order to be able to capture non-linear correlations



(a) Mutation score



(b) Normalized mutation score

Figure 3: The left axis shows Cook's distance estimate, while the bottom axis shows the coverage. The estimate is calculated using raw mutation score in plot (a) and normalized mutation score in plot (b)

for the applications, suggesting that libraries are tested twice as better as applications. However this measure is probably biased by the fact that the projects with lower coverage in our sample are mostly applications. We can obtain a better estimate if we consider the weighted average of normalized mutation score according to the portion of code actually covered:

$$\frac{\sum\limits_{projects} \text{Normalized mutation score} \times (\text{SLOC} \times \text{Coverage})}{\sum\limits_{projects} \text{SLOC} \times \text{Coverage}}$$

In this way we obtain an effectiveness average of 91% and 73% respectively, showing a smaller and but still significant difference. A difference in terms of test effectiveness between libraries and application could be due to the difficulty in testing the interdependence between different components of an application if compared with libraries which tends to be naturally more suited for unit testing.

Surprisingly there doesn't seem to be a significant correlation between the popularity of a project and other measures.

4 Discussion

The results obtained in this experiment seem to reflect the findings of [15] and [16], showing that coverage is not a good measure for test effectiveness, or in other words *"it is not generally safe to assume that effectiveness is strongly correlated with coverage"* [15]. In this assignment this was done using branch coverage in contrast with other coverage methods used in [15, 16]. We also concluded that the size of a test suite, expressed as number of test methods over SLOC, is positively related to its effectiveness, this connection was also highlighted by [15].

Since mutants can be safely considered (regardless of some minor limitations) as valid substitutes for real faults in software testing [17], this results hold in relation to the effectiveness of a test suite in detecting "real life bugs".

While in the case of [15] the measurement were taken on the same project multiple times using test suites of different size (this for five different projects), in the context of this assignment we compared the measures taken for seven different projects using their originals test suites. Therefore we didn't capture the effectiveness trend in connection to the coverage of the same code base but rather focus on the relation between test performances and coverage among different projects. I fear however that the comparison between projects of different nature could lack of significance and pose a threat to the validity of the results[6]. In addition, given the relatively small amount of projects taken into account for this experiment one should be careful when generalizing the result.

[6]Furthermore the same threats to validity illustrated in [15] hold for this experiment.

References

[1] Cloc: counts blank lines, comment lines, and physical lines of source code in many programming languages, https://github.com/AlDanial/cloc. Last access: 10-03-2018

[2] JaCoCo Java Code Coverage Library, http://www.eclemma.org/jacoco/. Last access: 10-03-2018

[3] CodeCover - an open-source glass-box testing tool, http://codecover.org/. Last access: 09-03-2018

[4] PIT Mutation Testing, http://pitest.org/. Last access: 10-03-2018

[5] Spring Framework, http://spring.io/. Last access: 09-03-2018

[6] JavaHamcrest: Java version of Hamcrest, https://github.com/hamcrest/JavaHamcrest. Last access: 09-03-2018

[7] Vectalign: Tool for create complex morphing animations using VectorDrawables, https://github.com/bonnyfone/vectalign. Last access: 09-03-2018

[8] Moshi: A modern JSON library for Android and Java, https://github.com/square/moshi. Last access: 09-03-2018

[9] Spring PetClinic: A sample Spring-based application , https://github.com/spring-projects/spring-petclinic. Last access: 09-03-2018

[10] JavaPoet: A Java API for generating .java source files, https://github.com/square/javapoet. Last access: 09-03-2018

[11] Dagger: A fast dependency injector for Android and Java, https://github.com/square/dagger. Last access: 09-03-2018

[12] Retrofit: Type-safe HTTP client for Android and Java by Square, Inc., https://github.com/square/retrofit. Last access: 09-03-2018

[13] Mutation Testing using PIT, https://youtu.be/10vJ31WgE3I. Last access: 09-03-2018

[14] Bollen, Kenneth A.; Jackman, Robert W. (1990). Fox, John; Long, J. Scott, eds. *Regression Diagnostics: An Expository Treatment of Outliers and Influential Cases. Modern Methods of Data Analysis.* Newbury Park, CA: Sage. pp. 257–91.

[15] Laura Inozemtseva and Reid Holmes. 2014. *Coverage is not strongly correlated with test suite effectiveness.* In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 435-445. DOI: https://doi.org/10.1145/2568225.2568271

[16] A. S. Namin and J. H. Andrews. *The influence of size and coverage on test suite effectiveness.* In Proc. of the Int'l Symposium on Software Testing and Analysis, 2009.

[17] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. *Are mutants a valid substitute for real faults in software testing?.* In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 654-665. DOI: https://doi.org/10.1145/2635868.2635929