How to Calculate Code Coverage Without Running the Tests? An Heuristic Using Statical Code Analysis

Mauricio Aniche
Institute of Mathematics and Statistics
University of São Paulo
aniche@ime.usp.br

Marco Aurelio Gerosa
Institute of Mathematics and Statistics
University of São Paulo
gerosa@ime.usp.br

ABSTRACT

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

1. INTRODUCTION

One advantage of studies that make use of mining software repository techniques is that it can be done on a large quantity of projects and data. This enormous quantity of information allows researchers to validate a phenomenon with more property.

Researchers can extract data from many different sources, such as issue trackers, source code repositories, mailing lists, and so on. Most of them are text-based, which means that the researcher should only find a way to extract and then process it. However, source code is tricky: some metrics need the code to be compiled first.

Compiling code is a complicated task. Each project has its own build process and required libraries. It is common that many of them are not available at the time the researcher has the source code.

An example of code metric that usually requires compiled code is code coverage. The metric calculates the percentage of the production code that is covered by at least one unit test. Popular tools, such as Emma[1] or Cobertura[2], instrument the code, execute the unit test suite, and then extract the information. As said before, when dealing with many different repositories, compiling and executing unit tests may not be an alternative.

To sort this problem out, we present an heuristic to calculate code coverage without compiling or even running the unit tests. In Section X, we discuss bla bla bla, bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla , bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla[1].

[1]http://emma.sourceforge.net/
[2]http://cobertura.github.io/cobertura/

2. CODE COVERAGE

Code coverage measures the quantity of production code that is being tested by the test suite. When a unit test exercises a piece of code, developers say that the piece of code is covered. The common output of the tools is a percentage, which is calculated by the quantity of production code lines that are covered by at least one unit test divided by the total number of production code lines.

There are many different ways to calculate code coverage. The tool can calculate the number of lines (as mentioned before), the number of instructions, the number of conditions, and even the number of executed branches.

As mentioned before, Emma and Cobertura are popular tools. They both use dynamic analysis to calculate the metric. They instrument the code, execute the unit test suite, and then discover which instructions were executed. Besides taking a long time to calculate (as they need to execute the entire test suite), they require compiled code.

3. THE HEURISTIC

As we wanted to avoid dynamic analysis, the only option was to apply statical analysis to the source code. The first step was to estimate the maximum number of tests that a class (or a method) needed. To do that, we decided to use McCabe's cyclomatic complexity [?]. McCabe's number basically shows the different number of execution paths per method. The listing 1 below exemplifies it. In that case, McCabe's number would be 2. As we are calculating code coverage in class method, we calculate the class' cyclomatic complexity by summing up McCabe's number of all methods.

```
public void doSomething(int a) {
  int total = 0;
  if(a>10) total += 1;
  return total;
}
```

Listing 1: McCabe's number to this method is 2.

The next step was to count the number of unit tests that invoke a specific production method. To do that, in a statically, we count all the invocations that happens in the unit tests. As an example, in Listing 2, we consider the invocation to *calculateTaxes()* as a unit test to it. We then repeat this procedure to all unit tests.

```
class InvoiceTest {
  @Test
  public void shouldCalculateTaxes() {
    Invoice inv = new Invoice("Customer",
        5000.0);

    double tax = inv.calculateTaxes();
    double taxAgain = inv.calculateTaxes();

    assertEquals(5000 * 0.06, tax);
  }
}
```

Listing 2: An example of a unit test

We then calculate the ratio between the number of tests and McCabe's number. In the Figure 1, we show the formula. Suppose that the class *Invoice* has McCabe's number equals to 10, and there are 6 unit tests that invoke methods that belong to it. In this case, **6 / 10 = 0.6**, which means **60%** of code coverage.

$$\frac{\sum(Number of testspermethod)}{\sum(McCabe'snumberpermethod)}$$

Figure 1: Formula used to calculate code coverage in our heuristic

There are a few exception cases that are worth mentioning. As we are interested in the number of distinct unit tests per production method, even if a test invokes the same method twice, it counts as one. In the example above, *calculateTaxes()* is invoked twice, but we only mark that it is being tested by *shouldCalculateTaxes()*.

4. EXPERIMENT DESIGN

To validate the heuristic, we decided to compare the code coverage calculated by it with the ones calculated by a dynamic analysis tool. We chose Emma as we had a previous experience with it. We had 3 projects as subjects, all of them developed by the same company, in Java. Two of them are web applications and one of them is a console application. In Table 1, we describe the size of each one.

Table 1: Quantity of classes and methods per project

	Classes	Production Methods	Unit Tests
Gnarus	769	1823	826
MetricMiner	225	1009	341
Tubaina	261	372	298

With both code coverage numbers (Emma's and ours), we then compared both to see how different they were. In Section below, we present the results found.

5. FINDINGS

To do the analysis, we decided to use the number calculated by our heuristic minus the number calculated by Emma. It means that every time we find a zero, it meant that both numbers were equal, and the heuristic was perfect. A negative number indicates that the heuristic calculated a smaller number than the real; a positive number indicates that the heuristic calculated a higher number.

In Figure 2, we show the histogram of the difference between Emma and our approach to the MetricMiner project. In the X axis, we have the difference (from -1 to 1), and in the Y axis, we have the frequence of that difference. By looking to this Figure, one can notice that most of the data are between -0.2 and 0.2. The median of the distribution is 0, and the mean is 0.1149. The first quartile is 0, and the third quartile is 0.2414.

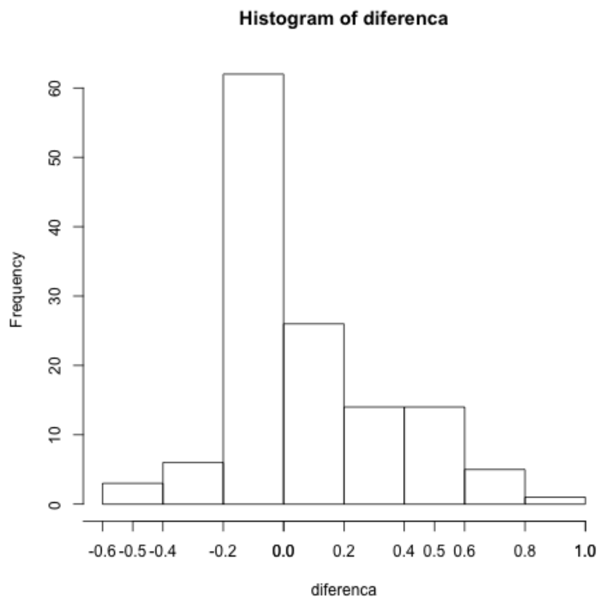

Figure 2: Our approach compared to Emma's in the MetricMiner

In Figure 3, from Tubaina project, the data is between -1 and 0. The median is 0, and the mean is 0.0790. The first quartile is 0, and the third quartile is 0.1667.

In Figure 4, from Gnarus project, we can see that most of data are between -2 and 0. The median is 0, the mean is -0.0152. The first quartile is -0.7273, and the third quartile is 0.

6. DISCUSSION

Based on the numbers presented in the section above, we believe that the heuristic is acceptable. As the maximum mean error was 0.1149 found in these 3 projects, which means 11%, we do not think that it will be a problem when using it in large software repositories.

As Emma needs to execute the tests, and they may take a long time to execute, we also argue that our approach is faster.

7. THREATS TO VALIDITY

o algoritmo pode ter problemas

os projetos foram desenvolvidos pelos devs de uma soh empresa e eles tem a maneira de escrever os testes, entao sao 3 aplicacoes parecidas em termos de arquitetura.

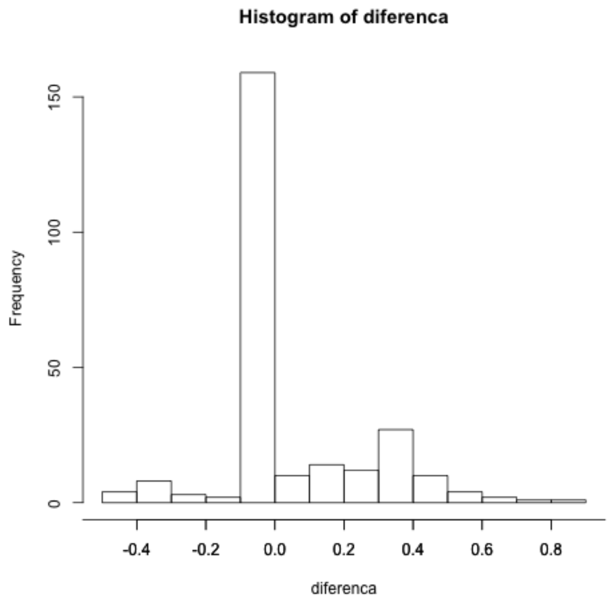

Figure 3: Our approach compared to Emma's in the Tubaina

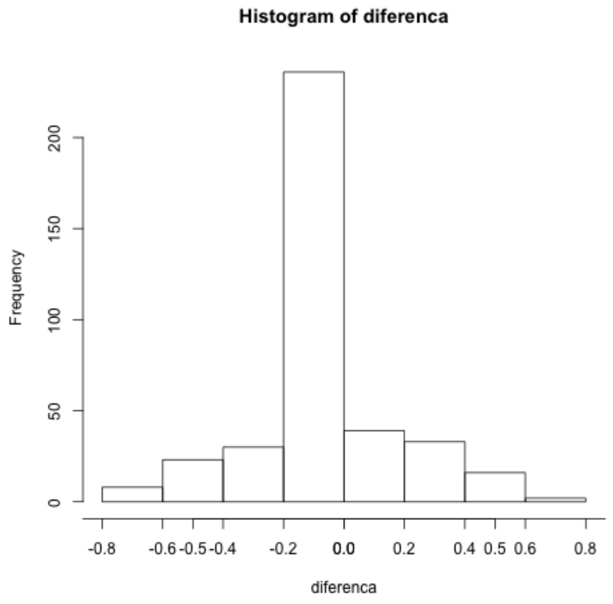

Figure 4: Our approach compared to Emma's in the Gnarus

8. RELATED WORK

9. CONCLUSION AND FUTURE WORK

rodar em mais projetos
melhorar a ferramenta que ainda tem problemas

10. ACKNOWLEDGMENTS

bla bla

11. REFERENCES

[1] M. Bowman, S. K. Debray, and L. L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.