Politecnico di Torino

Master's Degree Course in Computer Engineering

Master's Degree Thesis


# Interaction among software quality metrics, bug prediction and test fragility: a machine-learning aided analysis

**Supervisor:**
Prof. Luca Ardito

**Candidate:**
Vito Damaso

**Co-supervisor:**
Prof. Maurizio Morisio
PhD. Riccardo Coppola

Academic Year 2020/21
Torino

**Abstract**

***Context*:** Software Maintainability is an important and at the same time challenging task, due to its cost and time-consuming factor. One branch of Software Maintainability is the bug prediction which in the last decade has attracted many interest in the research community.

***Goal*:** The aim of this thesis' work is to understand if the *'Bug Prediction'* can be used as predictor for the *'Test Fragility'*, in other words if there is any sort of correlation between the two.

***Method*:** A script has been created that calculates for each project the code quality metrics, uses them to predict the bug-proneness of the classes, and finally calculates the linear regression between the results of the bug prediction and the code fragility metrics.

***Results*:** Through linear regression, it was possible to compare the Bug Prediction and the code fragility metrics, it emerged that there is no correlation between the two except in some rare cases.

***Conclusions*:** The study demonstrates there is no correlation between Bug Prediction and code fragility metrics except in rare cases. However, it must be considered that the analyzed sample was composed only of 30 Android projects. Hence, it would be helpful to repeat the analysis, first of all, on a larger set of projects and then to try on other software families, as well.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Background and related works

## 1.1 Introduction

Software maintainability is a significant software quality attribute that defines the degree by which software is understood, fixed, or improved (Nayrolles & Hamou-Lhadj, 2018) and this process has been estimated to cost between 40% and 90% of the total build cost (Coleman, 2018).
Lientz and Swanson (1980) categorized maintenance activities into four classes:

1. **Perfective:** Improve code quality without affecting the global software behavior or when the end-user changes the requirements.

2. **Adaptive:** Manage changes due to changes in the environment, ex. new versions of the operating system or the system are ported to new hardware.

3. **Preventive:** Increase the maintainability of the software in order to avoid any problems in the future and make subsequent maintenance easier. An example can be the management decay of software parts due to code smells (a symptom of poor maintainability carried out).

4. **Corrective:** Fix/Remove any bugs identified.

These activities can be carried out independently or, as shown in Figure 1.1, can be one the consequence of the other.

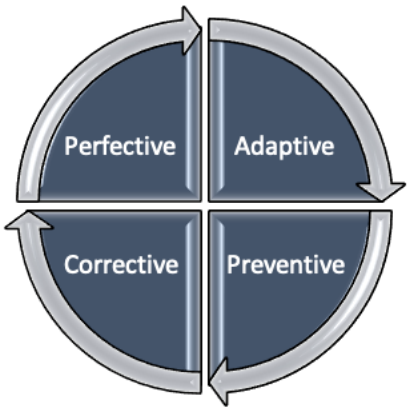


Figure 1.1. Maintenance activities Cycle

In the past, software maintainability was considered as the final phase of the software lifecycle, which could be managed separately from development. An example of this practice could be when an organization hands over its system to an external party who takes overall responsibility for maintenance following an initial investigatory period (Bennett, 1990). However, it came to light how this activity was inconsistent and lead to inefficient outcomes. As expected, nowadays, the software is developed and maintained by the in-house development team and maintenance group, respectively.

Furthermore, maintaining software throughout the software life cycle is crucial but also challenging for development teams because, in the real-world scenario, software systems change every day. Usually, the software changes so that it can be adapted to new requirements or fixed due to discovered bugs (Lehman, 1984). Even what coding practices and techniques used may influence the maintainability process.

Unfortunately, sometimes companies prefer to perform less maintenance to develop features whose business value is more relevant in the short term, nevertheless, it remains probably the most attractive, observed, and studied characteristic of the software products (Hegedus, 2013).

During corrective maintenance, the developer has to understand the software enough to analyze the problem, locate the bug, and determine how should fix it without breaking anything (Vans et al., 1999). Later, everything must undergo a testing process. There can be two types of testing: Confirmation Testing and

Regression Testing. Confirmation Testing is a re-test, i.e. after fixing bugs an additional test execution is required to confirm that those bugs do not exist anymore. Regression Testing, instead, is needed to ensure that all the new changes have not undermined the global software behavior (Yusop & Ibrahim, 2011).

## 1.2 Testing

Over the years, Object-Oriented software systems have become larger and more complex. Hence, try to efficiently and equally test all the classes belonging to a system has become extremely difficult both from an economic point of view and from a time point of view(energy and time spent by developers). Therefore, the developers had to focus on classes that, possibly, required more time and effort (human effort) at the expense of the other ones.

A possible methodology is to group classes into critical levels (in terms of time, labor, and consequently costs), to focus first on the classes with a high level of criticality, and then move on to the remaining ones. In the literature, there are numerous OO[1] metrics that can predict and quantify in terms of time and resources, which portion of code needs the most maintenance. During the software development process, unit testing is the most significant part of the overall testing process. It consists of writing test code that tests the single class (Toure et al., 2018), to identify the presence of bugs inside the source code, but also to verify if the source code already written still works correctly following the inevitable evolution of the entire software system (Vahabzadeh et al., 2015). Each class needs this work (Toure et al., 2018). In addition, when the class under test is improved or more simple changes, the testing unit must be adapted to recognize the new changes and test them. Thus, it can remain efficient throughout the project lifecycle, and this process is known as test coevolution (Zaidman et al., 2011).

Software testing can be either manual or automated. In the manual test, human testers take over the role of end-user and interact with the software under test by verifying its behavior and detecting any malfunctions (Amannejad et al., 2014). In the case of the automated test, is required no human interaction then, obviously, this leads to benefits (in terms of efficiency, repeatability, and reliability). On the

[1]Object Oriented

other hand, if it is not implemented correctly, it can lead to higher costs and efforts and be even less efficient than human testing (Amannejad et al., 2014).

Over the years, various guidelines have been provided on how to write high-quality tests which are not always followed, for this reason, there are problems such as bad smells, which in the case of the test are called test smells, and are an indication of a design error and bad implementation choices, thus if the developer does not immediately remove them, they can lead to problems in the future, for instance, low-maintainability of the test code, tests are not able to identify defects in the source code (Garousi & Küçük, 2018).

Writing testing scripts is not an easy task because they can contain errors (e.g. bugs, we'll talk about them later). Thus, it could take a considerable investment of time and money, after all, as every product in software engineering needs quality assessment and maintainability, as well. Moreover, a test script must coevolve with the source code that is under test. Automated test means using software or a framework, which is something isolated from the code under test and aims to create test-case, test code (scripts), manage the various test cases, and verify that the tests have been executed. The scripts are code written in a common programming language to test (exercise) the source code and monitor its behavior. Generally, a script simulates a use-case, and some of these tasks can be long and require both time and effort. Is preferred the automated approach to a manual one because the former can be performed quickly and repeated multiple times, on the contrary, the latter is not always efficient.

As stated in literature ((Ammann & Offutt, 2016); (Binder, 2000); (Mathur, 2013)), the testing phase can be summarized in 5 sub-phases:

1. *Test-case design:* Design and list all the test-cases, or the constraints that must be respected or other goals to be achieved;

2. *Test scripting:* Develop the test-case in manual tests or in automated tests;

3. *Test execution:* Run the tests on the code to be tested and save the results;

4. *Test evaluation:* Evaluate the results collected and give a verdict (passed or fail);

5. *Test-result reporting:* Report the test result and any problems encountered to the developers.

The main part of the testing phase is, of course, the script and it must be executed and must be structured according to these steps:

1. *Setup:* Set the initial phase, in which everything that must be positioned so that the source code can show its common behavior and that the test is able to monitor the results that will come;

2. *Exercise:* The source code is tested, i.e. it interacts with the test code;

3. *Verify:* Evaluate if the result obtained is the expected one and express the verdict (passed or fail);

4. *Teardown:* Bring everything back to its initial state, i.e. before the test was performed, so that everything is ready to be retested if necessary.

Like any object made up of code, also the test needs a quality assessment and maintainability, indeed its life cycle begins when it is designed, just like a software and it must coevolve with the code under test. The test code is usually modified for two reasons:

1. when the source code is modified and therefore the test must be updated - in the literature this process is called "repair of the test code" ((Daniel et al., 2010); (Mirzaaghaei et al., 2010); (Choudhary et al., 2011); (Pinto et al., 2012));

2. the test code is modified to improve its quality.

The test code should last throughout the source code lifecycle, but it may happen that developers decide not to use that test anymore, and in this case, it is said that the test has been "retired/archived". A retired test can occur when it is no longer possible, from an economic point of view, to make the test coevolve with the source code. Similar to the types of software maintainability activities, shown in 1st paragraph, these definitions can also be applied to test code (Yusifoğlu et al., 2015):

1. Perfective maintenance of test code: Improve test code quality, e.g. refactoring.

2. Adaptive maintenance of test code: Co-maintenance as the production code is maintained.

3. Preventive maintenance of test code: Quality Assessment of test code, e.g. detection of test smells in test code.

4. Corrective maintenance of test code: Finding and fixing bugs in test code.

## 1.3 Limits of Software Testing

One of the problems that plague the test code is the test smells because, is demonstrated that, they harm its maintainability. If the complexity of the test code makes it difficult to understand or to modify, on the other hand, the presence of smells affects its ability to be reused, to be isolated, and on being stable (Van Rompaey et al., 2007). The test codes that suffer from smell tests are more likely to be modified (47% of cases) and contain defects (81% of situations) than test codes without smells. Hence, the production code tested by smelly tests is more likely to have bugs (71% of cases). In the study conducted by Tufano et al. (2016), it emerged that test smells appear in the first commit of the test code and that they are never removed in almost 80% of times, simply because the developers do not know they created them (Spadini et al., 2018).

The test code, of course, being itself a code written by a human being, is not immune to having bugs just like the code it is testing and, as listed by Cunningham (2006), this could lead to:

- a lack of detection of bugs in the production code (*silent horrors*);

- to a test fail even if the production code is correct (*false alarm*).

Lu et al. (2008) first and then Luo et al. (2014) focused in detail on studying real-world concurrency bugs and discovered that they are the principal cause of non-deterministic test failures, and this condition is known as the flaky test. Although much has been done for bugs in production code, the research community has paid very little attention to test bugs (Vahabzadeh et al., 2015).

### 1.3.1 Test Smells

A “bad smell” is an indication of inadequate design and implementation choices. The first to classify the code smells by giving a definition was Fowler et al. (1999). Fowler identified almost 22 smells giving, for each of them, a description used to verify their presence and provided some helpful techniques, called “Refactoring”, to remove them once found. According to Fowler, “refactoring” means modify a portion of software without altering its general behavior and, at the same time, improves its inner structure.

However, the concept of "bad smell" emerged with the introduction of Object-Oriented languages, indeed it is very straightforward to establish that some code smells are interrelated (a direct consequence of a wrong application) to the principles of Object-Oriented programming (Yamashita, 2014).

The evolution of a bad smell within a code that includes its introduction up to its removal, if this is possible, is schematized in Figure 1.2. As shown in the figure, once that a smell is located the developer has three options:

1. No action is taken because the developer considers the smell just identified as not harmful for the correct execution of his program.

2. The developer decides to use the appropriate refactoring techniques on the target smell to remove it.

3. The developer could decide to ignore the smell because:

   (a) even with the use of the refactoring techniques, it was impossible to remove it;

   (b) the detection was a false positive.

Recently, academics stated that bad smells were not exclusive of the production code, on the contrary, they could affect test code, as well. Indeed, like for code smells there is a set of bad smells that involve the test code and are called test smells. They concern about how test cases are organized, how they are implemented, and how they interact with each other. Thus, smell tests are the result of inadequate design and implementation choices.

The first to talk about them in a study was Beck (2003) (even if he never talked explicitly of test smells but about test fixtures). He claimed the developers that refactor the source code must ensure that the tests continue to work correctly, and

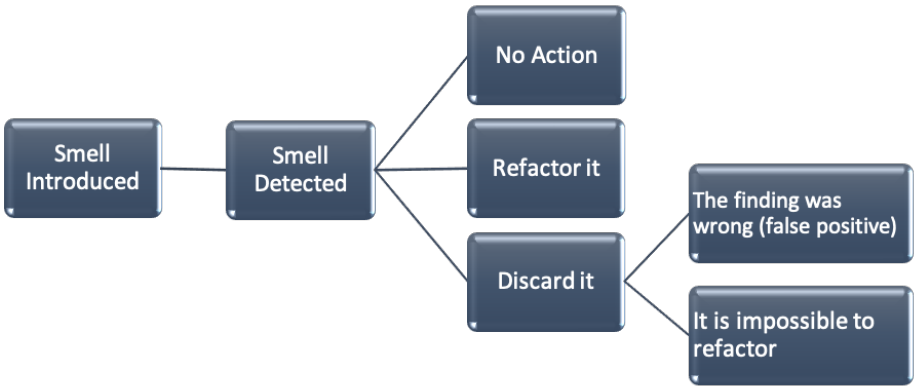


Figure 1.2. Smell life cycle.

therefore also the test code requires refactoring. The refactoring of the test code must be a natural consequence of the refactoring of the source code (Van Deursen & Moonen, 2002).

The first, however, to give a formal definition of test smells and explain the various refactoring techniques to remove them were Van Deursen et al. (2001). Later, Meszaros (2007) treated them in a broader context than the former and explained the reasons why both smells test and their side effects exist, how higher-level smells are the consequence of the combination of two or more lower-level smells.

Understanding both the code that must be tested and the test code is essential for the review, maintainability, reuse, and extensibility of the code itself. Bavota et al. (2012) demonstrate how smells tests, in the context of test cases manually written by developers, are widespread and how they negatively impact the understanding and maintainability of the program.

Afterward, (Palomba et al., 2016) conducted an empirical investigation on the diffusion of test smells in the JUnit test classes automatically generated by EvoSuite[2], differently from the former study. Results indicate that test smells are largely diffused also in automatically generated tests, i.e., 83% of JUnit classes are affected by at least one test smell.

[2]https://www.evosuite.org/

To avoid running into test smells there are well-defined good programming practices because the quality of the test code depends a lot on the developers who have implemented it (Qusef et al., 2011). However, for many reasons, such as tight deadlines or inexperience of the developers themselves, these guides are not always followed and, inevitably, this leads to the presence of smells and consequently to an increase in maintenance costs (Bavota et al., 2012).

Many approaches have been provided in the literature to detect code smells, and some of them can also be adapted to identify test smells, on the contrary, the academic community has focused very little on detecting test smells. An example can be the approaches introduced by Marinescu (2004) and Moha et al. (2009) which have as main aim to identify code smells, and when they have been applied to test smells the outcomes were quite remarkable.

Marinescu (2004) provided "detection strategies" for obtaining metric-based rules that measure deviations from good-design principles. The detection strategies are divided into different phases. First, the characteristics that describe a specific smell are defined. Second, is designed a suitable set of metrics quantifying these characteristics. The next step is to determine thresholds to mark the class as affected (or not) by the defined properties. Finally, are used AND/OR operators to link the characteristics to achieve the final rule for detecting the smells.

Moha et al. (2009) presented DECOR, a method for specifying and identifying code and design smells. DECOR uses a Domain-Specific Language (DSL) to specify smells using high-level abstractions. DECOR detects four design-smells, namely Blob, Swiss Army Knife, Functional Decomposition, and Spaghetti Code.

Few scholars have targeted test smells, such as Van Rompaey et al. (2007), Reichhart et al. (2007), Breugelmans and Van Rompaey (2008), Greiler et al. (2013) and lastly Palomba, Zaidman, and De Lucia (2018).

Van Rompaey et al. (2007) provided a heuristic metric-based approach to detect the General Fixture and Eager Test. General Fixture occurs when the test setup method creates fixtures, and a portion of the tests use only a subset of the fixtures. Eager Test occurs when a test checks more than one method of the class under test, making it difficult to understand the actual test target) bad smells.

Reichhart et al. (2007) developed TestLint, a rule-based tool to locate static and dynamic test smells in Smalltalk SUnit code.

Breugelmans and Van Rompaey (2008) presented a reverse engineering tool called TestQ able to detect test smells using static analysis.

Greiler et al. (2013) proposed a test analysis technique, implemented in the TestHound tool, which provides reports on test smells and gives refactoring recommendations to remove them from the code. Their method focuses on several types of smells that can arise in the test fixture.

(Palomba, Zaidman, & De Lucia, 2018) developed TASTE (Textual AnalySis for Test smEll detection), which can detect General Fixtures, Eager Tests, and Lack of Cohesion of Methods using Information Retrieval techniques, therefore avoiding the need to parse the entire test code. Their tool shows a better precision and recall than the AST-based tools TestQ and TestHound.

### 1.3.2 Bugs in Test Scripts

The testing activity is largely performed to detect bugs in production code. However, sometimes it happens that code under test results be in error, and only after several attempts by running tests, turns out that the test itself is in error because the latter, just as the production code, is written by a person thus, can contain some bugs (McConnell, 1993).

Cunningham (2006) defined two main categories of test bugs:

1. *Silent Horrors* - is when a test passes even though there are bugs in the production code because the test code is affected by bugs. This type of bugs is hard to find and can remain hidden in the test code even for a long time;
2. *False Alarm* - is when a test says there are bugs in the production code when it is not true. Though this kind of bug can be easily located, unlike the previous ones, they may still require a lot of time and considerable effort, simply because it may take a while for the developers to understand that the bug is in the test code and not in the production code.

The reasons for the presence of bugs in test code are easy to find. For instance, developers have to work to tight deadlines, maybe misunderstand the requirements, or as in the case of a test case which tends to be created on the fly instead through a planned design because they are seen as a one-time test (McConnell, 1993).

In literature, there is a lack of research and analysis on test bugs compared to what has been done about production-code bugs. Vahabzadeh et al. (2015) are the first to present a quantitative and qualitative study of test bugs. They

analyzed in their work the prevalence of bugs in test codes and their root cause. The results obtained show how bugs are frequent in the test codes, and the majority belong to the category of False Alarm, only the 3% to the type of Silent Horror (mainly assertion-related fault). Specifically, False Alarm is mostly consequent to semantic bugs (25%), flaky test (21%), environment-related (18%) followed by resource-related faults and obsolete tests both (14%), finally other (8%).

Furthermore, the study of Vahabzadeh et al. (2015) shows that, although there are similarities between bugs existing in the test codes and those present in the production codes, the patterns of the former differ considerably from the patterns of the latter, and this leads to the ineffectiveness of the current bug detection tools. For instance, FindBugs[3] has only six bug prediction patterns to detect test bugs compared to its 424 bug patterns.

### 1.3.3 Flaky Tests

The regression testing, as said before, ensures that all the changes made have not undermined the global software behavior. At the base of the regression testing, there is the concept that test outcomes are deterministic, which means that not modified test always passes or always fails when executed on the same portion of code under test. In practice, this is not always true, some tests have non-deterministic outcomes, and they are called flaky tests.

If all the tests pass, it means the latest changes did not corrupt the overall behavior of the production code. Instead, if any test fails the developers have to think about the cause of failure and understand whether the latest changes introduced a fault in the production code or whether the test code itself needs to be changed. So, when a test does not pass, the developer has to assume that the recent changes introduced a problem in the production code or the test code.

The main drawbacks of flaky tests are:

- Tests affected by flakiness are hard to fix, because it is difficult to reproduce a certain result due to their non-deterministic behavior.
- There can be a great waste of time because these flaky tests may not even be caused by the recent changes but, instead caused by changes made in the past (Lacoste, 2009).

[3] http://findbugs.sourceforge.net/

- Flaky tests may also hide real bugs, in fact if a flaky test fails with some frequency, developers are tempted to ignore its failures and, thus, could miss real bugs (Luo et al., 2014).
- From a psychological point of view flaky tests can reduce a developer's confidence in the tests, possibly leading to ignoring actual test failures (Melski, 2013).

The most common approach to overcome flaky tests is to execute a test multiple times, if it passes any execution, label it as passing, even if it has failed in several other executions. Another approach is to remove them from the test suite or not consider their results most of the time. However, these approaches, just mentioned, are more like "workarounds" rather than solutions because they do not solve the root causes of flaky tests. Therefore, they can waste a lot of machine resources or limit the effectiveness of the test suite.

Luo et al. (2014) in their research tried to dig into the main origins of flaky tests and they have identified ten categories as main causes:

1. Async Wait: when the test makes an asynchronous call and does not properly wait for the result of the call.
2. Concurrency: when the non-deterministic outcome of the test is due to thread's synchronization issues, e.g. deadlocks, starvation.
3. Test Order Dependency: the test outcome depends on the order in which the tests are executed.
4. Resource Leak: the test does not properly acquire/release one or more of its resources.
5. Network: because the network is a resource that is hard to control, ex. remote connection failure or bad sockets management.
6. Time system: tests that use time system can have non-deterministic failures, e.g. a test can fail when the midnight changes in the UTC time zone.
7. I/O operations: dealing with external resource can lead to intermittent test failures.
8. Randomness: the use of random numbers generator can lead to flaky tests.

9. Floating Point Operations: dealing with floating point operations is known to lead to tricky non-deterministic cases.

10. Unordered Collections: in general, when iterating over unordered collections.

## 1.4 Software Metrics

Software metrics are adopted to quantify several aspects of computer software, such as cost estimation, resources management, programming methods efficiency, and system reliability. The principal use of these measurements is as defect detector (Cook, 1982). There are two families of metrics: static and dynamic - static metrics are executed directly on the source code without running the latter, dynamic metrics get computed at runtime.

Bug prediction is a very studied topic in the literature that continues to this day, because the cost of fixing bugs varies enormously, and it depends, of course, on when the bug is detected. To visually have an idea of how important an early bugs detection is, it is sufficient to observe the Boehm's cost of change curve (Figure 1.3) which is an exponential curve, implying that the cost of fixing a bug at a given stage will always be greater than the cost of fixing it at an earlier stage (Boehm et al., 1976).

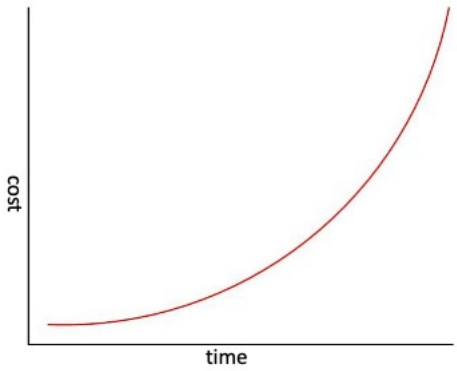


Figure 1.3. Cost of change.

However, the waterfall model of software development suggests testing for flaws after integrating all of the components in the system. Testing each unit or component after it has been developed increases the probability of finding a flaw.

The iterative model for each smaller iteration of the whole software system introduces a testing phase. This leads to a greater chance of finding the bugs earlier in the development cycle.

The V-model is characterized by intense testing and validation phases. Though, functional defects are very difficult to fix in this model, as it is hard to roll back once a component is in the testing phase. The agile model also uses smaller iterations and a testing phase in each iteration.

As we can see, the testing phase is always done later in all the development cycles described above. This will inevitably lead to larger costs of fixing the defect. Consequently, modern research has shifted focus from "detecting" to "predicting" bugs in the code. So, by establishing the presence or absence of a bug in a software release, developers can foresee the success of a software release even before it is released, based on a few characteristics of the release version. If this estimation is performed at a stage earlier than the production phase (Figure 1.4) in the software development cycle (the prediction is "fed back"), it will drastically reduce the cost of fixing cost by helping software engineers to allocate the limited resources strictly to those modules of the system that are most certainly affected by defects (Subbiah et al., 2019).

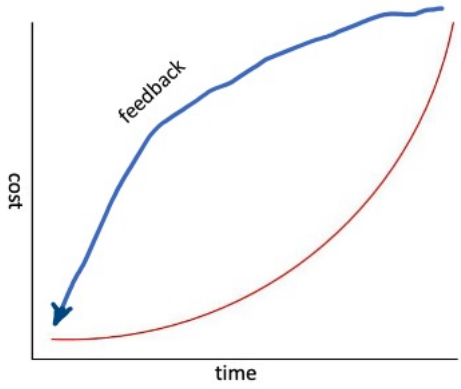


Figure 1.4. Cost of change with feedback.

Catal and Diri (2009) made a Systematic Literature Review (SLR) of all the existing metrics and divided the various papers about bug prediction metrics into six different categories according to metrics' granularity level. The categories are: method-level, class-level, component-level, file-level, process-level and quantitative-level (Table 1.1).

Method-level metrics, such as Halstead (1977) and McCabe (1976), are the most used for software fault prediction. These metrics can be applied both to programs developed with structured programming and object-oriented programming since the latter have method concepts in them. When these metrics are used, the outcome defect-prone modules are the methods that have more probability be affected by defects. (Menzies et al., 2006) and (Tosun et al., 2010) introduced the first investigations exploring a finer granularity: function-level. Successively, Giger et al. (2012) and Hata et al. (2012), independently and almost contemporaneously investigate the method-level bug prediction. Giger et al. (2012) found that product and process metrics contribute to the identification of buggy methods and their combination achieves promising performance. Hata et al. (2012) found that method-level bug prediction saves more effort than both file-level and package-level prediction.

Class-level metrics are only used for object-oriented programs because the class concept belongs to the object-oriented paradigm. The most popular metrics of this category are Chidamber-Kemerer, better known as CK, metrics suite (Chidamber & Kemerer, 1994) which have been used by many software tool vendors and researchers that are working on fault prediction. Furthermore, there are other metrics such as MOOD, which was presented by Abreu and Carapuça (1994) then validated by Abreu and Melo (1996); L&K metrics suite which collects eleven metrics formulated by Lorenz and Kidd (1994); and QMOOD which was introduced by Bansiya and Davis (2002).

Method-level and Class-level metrics cannot be used on systems developed with a component-based approach. For example, the beforementioned metrics do not take into account the interface complexities (which is one of the main concerns in component-based development), or some are based on the number of code lines but, the component's size is not known in advance, and so on (Gill & Grover, 2003). Researchers are not yet in complete agreement on the meaning to give to the term component, but in a nutshell, a component is what can be used as a feedback cost black-box. Component-level metrics were presented in 2001, but researchers are still working on them.

Belong to the File-level category all the metrics that can be used for source files, e.g., the number of times the source file was analyzed before the system test release, the number of lines of code in source files before the coding phase (auto-generated code), the number of lines of code in sources files before the system test release, the number of commented lines belonging to the auto- generated code per source

files, the number of commented lines in source files before the system test release (Khoshgoftaar et al., 2001).

As in the Method-level case, class-level, component-level, and file-level are all product metrics, and it has been shown how predictive models that use only product metrics can be improved with process metrics. An example is the work of Jiang et al. (2007), in which they proved that code metrics (lines of code, complexity) combined with requirement metrics lead to a performance improvement of the prediction models that used only code metrics. Beyond requirement metrics, belong to process metrics also: programmer experience level, the number of bugs found in reviews, the amount of time spent by a module in review, the number of test cases and unique test case execution that interested the module (Kaszycki, 1999). Furthermore, process metrics can also be divided into delta metrics and code churn metrics. Delta metrics are computed as the difference of metrics values between two versions of software, that is how the value of the metrics has changed between two versions, but it did not show how much change has occurred. For instance, if several lines of code have been added, this change will be reported as a changed delta value, but if the same number of lines have been added and then removed, there will be no change in the delta value. Otherwise, with code churn metrics this problem would not occur, as it is reported the global change of the software between two versions. The concept behind delta and the code churn metrics can be applied to any metric (G. A. Hall & Munson, 2000).

Quantitative-level metrics use quantitative values such as CPU usage, disk usage to predict software bugs. Bibi et al. (2006) in their study used regression via classification (RvC) to predict software defect adopting document quality, CPU usage, disk space used, number of users, and average transactions metrics.

## 1.5 Network Metrics

Besides software metrics, for predicting bugs, have been used also network metrics and entropy changes metrics. First, Zimmermann and Nagappan (2008) presented network metrics as defect predictors, they studied for Windows Server 2003 the correlation between binary file dependencies and bugs, and how the former can be used for predicting the latter. They found that network measures have a higher recall rate compared to software complexity metrics and they were also better predictors of critical bugs.

Table 1.1. Software metrics.

| Group | Metrics | Description |
|---|---|---|
| Method-level | Halstead (1977), McCabe (1976) | Both metrics can be applied to programs developed with structured programming and object-oriented programming. |
| Class-level | C&K (1994), MOOD (1994), L&K (1994), QMOOD (2002). | These metrics are only used for object-oriented programs. |
| Component-level | - | Component-level metrics have been presented in 2001 but researchers are still working on them. |
| File-level | Khoshgoftaar et al. (2001) | The number of times the source file was analyzed before the system test release, the LOC in source files before the coding phase (auto-generated code), the LOC in sources files before the system test release, the commented LOC belonging to the auto-generated code per source files, the commented LOC in source files before the system test release. |
| Process-level | Kaszycki (1999), Jiang et al. (2007). | Requirement metrics, programmer experience level, the number of bugs found in reviews, the amount of time spent by a module in reviews, the number of test cases and unique test case execution that interested the module. |
| Quantitative-level | Bibi et al. (2006) | These metrics use quantitative values i.e. CPU usage, disk usage to predict software bug. |

Tosun et al. (2009) proved that network metrics work better in large and complex projects than in smaller projects. Conversely, the results achieved by Premraj and Herzig (2011) in their study demonstrated that network metrics performed better for small projects rather than for larger projects.

Hassan (2009) proposed the concept of complexity of code change using information principles. He said that the process of code change could be considered a

system that sends data, where data were the "feature introducing changes" (FIC) to source files. Thus, he could use Shannon's information entropy theory[4] to measure the complexity of code change. Moreover, he presented a new entropy-based complexity metric named history complexity metric (HCM), which has been used either as an independent or predictor variable for the prediction of bugs. He concluded that history complexity metrics (HCM) predicted bugs more accurately than the code churn metrics and prior faults.

Although so many varieties of metrics have been proposed by the academic world, on the other hand, one of the most widespread criticisms made by practitioners is and will be regarding the granularity, i.e., at what level bugs are found because most metrics predict bugs at a coarse level, such as file or component level. This degree of granularity is not revelatory enough since files and components can be arbitrarily large thus, a significant number of files needs to be examined (Giger et al., 2012).

## 1.6 Bug Prediction Models

One of the most efficient techniques aimed at dealing with the testing-resource allocation is the creation of bug prediction models (Malhotra, 2015), which allow predicting the software components that are more likely to contain bugs and need to be tested more extensively.

Bug prediction models are classifiers (supervised methods) trained to detect bug-prone software modules using statistical or machine learning classification techniques. The classifiers usually have a set of independent and configurable variables (the predictors) used to predict the value of a dependent variable (the bug-proneness of a class). Since the optimal settings cannot be known in advance, the predictors are often left at their default values. However, not all the classifiers require parameter settings, depends on classification techniques used, such as logistic regression that does not have any parameters (Tantithamthavorn et al., 2016).

Studies have demonstrated how the assignment of non-optimal values may have a negative impact on performance and outcomes. Therefore the choice of these parameters must be done very carefully ((Koru & Liu, 2005); (Mende, 2010); (Mende

[4]Entropy (Information Theory)

& Koschke, 2009)). Though, it is not workable to examine all the possible settings in the parameter space of a classification technique.

The first use of machine learning against bugs was presented by Murphy and Cubranic (2004) and they achieved to build a completely automated method, based on text classification, to make decisions on what action to perform when bugs are reported. This method works correctly on 30% of the bugs reported to developers.

Afterward, Zimmermann et al. (2007) analyses bug reports at the file and package level using logistic regression models.

Shivaji et al. (2009) weigh the gain ratio of each feature and select the best features from the dataset to predict bugs in file-level changes.

The use of linear regression to compute a bug proneness index is explored by Puranik et al. (2016). They perform both linear and multiple regression to find a globally well-fitting curve for the dataset. This approach of using a regression for bug prediction did not lead to convincing results.

The key concept is that results of bug prediction models built using machine learning depend on historical data, that is, are trained on the data of historical release of the project under analysis and predict bugs in the future releases (He et al., 2012). This strategy is named Within- Project Defect Prediction (WPDP), Zimmermann et al. (2009) pointed out that more amount of available data to train any models and better will be the bugs prediction of the latter. It is evident how for new projects it is almost impossible to collect a sufficient amount of historical data, therefore in some cases, a high accuracy bugs prediction cannot be reached using the within-project approach.

Another approach is to use data across projects to build prediction models, and this strategy is named Cross-Project Defect Prediction (CPDP). CPDP models are trained using historical data of other projects similar to the one that is under analysis. Although the scientific community has shown much interest in these predictors due to their potential, studies have highlighted their low performance in practice since the presence of non-homogeneous data (Rahman et al., 2012).

Almost all the classifiers are built on the combinations of software metrics, with which they can reach a remarkable accuracy but still lead to a time-consuming prediction process.

Moreover, in addition to classifiers built using software metrics, have been studied typical classifiers, such as Naïve Bayes, Support Vector Machine (SVM), Logistic Regression and Random Tree, which are simpler and theoretically could be adopted

in various projects ((Catal & Diri, 2009);(Jin & Liu, 2010);(Song et al., 2006)).

Complex predictors reach high prediction accuracy with loss of generality but increase the cost of data acquisition and processing. On the contrary, simple predictors are more versatile, and they reduce the total effort and cost at the expense of accuracy. To build a suitable and practical prediction model should be considered the accuracy, generality, and cost according to specific requirements.

The classifier used to predict bug-prone components is a factor that distinctly influences the precision of bug prediction models. Specifically, Ghotra et al. (2015) demonstrated that the precision of a bug prediction model varies about 30% depending on the type of classification chosen. Also, Panichella et al. (2014) proved that the outcomes of different classifiers are complementary, despite sharing a similar prediction precision.

Consequently, the focus is shifted to the development of prediction models which are able to combine multiple classifiers (a.k.a., ensemble techniques (Rokach, 2010)) and their use to bug prediction (Panichella et al., 2014),(Liu et al., 2010); (Menzies et al., 2012); (Petrić et al., 2016)). For instance, Liu et al. (2010) presented the Validation and Voting (VV) strategy, an approach where the result, thus the prediction of the bug-proneness of a class, is obtained by considering the output of the majority of the classifiers. Panichella et al. (2014) came up with CODEP, a strategy in which the outputs of 6 classifiers are the predictors of a new prediction model, which is trained using Logistic Regression (LOG).

However, Bowes et al. (2018) pointed out that traditional ensemble approaches did not predict the majority of bugs that are correctly detected by a single classifier and, therefore, *"ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of bugs"* (Bowes et al., 2018).

An alternative approach to deal with the problem of non-homogeneous data of cross-project bug prediction is the local bug prediction (Menzies et al., 2012). This technique consists of two steps: first, are created clusters of homogeneous data and then builds, for each of them, a different model using the same classifier. Unfortunately, local bug prediction is very hard to apply in practice because it tends to create too small clusters that cannot be used as training sets.

Di Nucci et al. (2017), propose a novel adaptive prediction model, named ASCI (Adaptive Selection of Classifiers in bug prediction), which dynamically suggests the classifier to use to better predict the bug-proneness of a class, based on the

structural characteristics of the class (i.e., product metrics). In particular, first train a set of classifiers using the structural characteristics of the classes, then build a decision tree where the internal nodes represent the structural characteristics of the classes contained in the training set, and the leaves represent the classifiers able to detect the bug-proneness of instances having such structural characteristics.

To sum up, the empirical evaluation of all these approaches shows that there is no machine learning classifier providing the best accuracy in any context, highlighting interesting complementary among them. For these reasons' ensemble methods have been proposed to estimate the bug-proneness of a class by combining the predictions of different classifiers.

Figure 1.5 shows the schematic behavior of the process of bug prediction using machine learning. The bug reports from development environments along with various software metrics are stored in a bug database. This database is used to train a suitable machine learning model. Furthermore, deploying the machine learning model on the cloud, bug prediction can be provided as a cloud-based service (MLaaS – Machine Learning as a Service) to software development companies across the world (Subbiah et al., 2019).

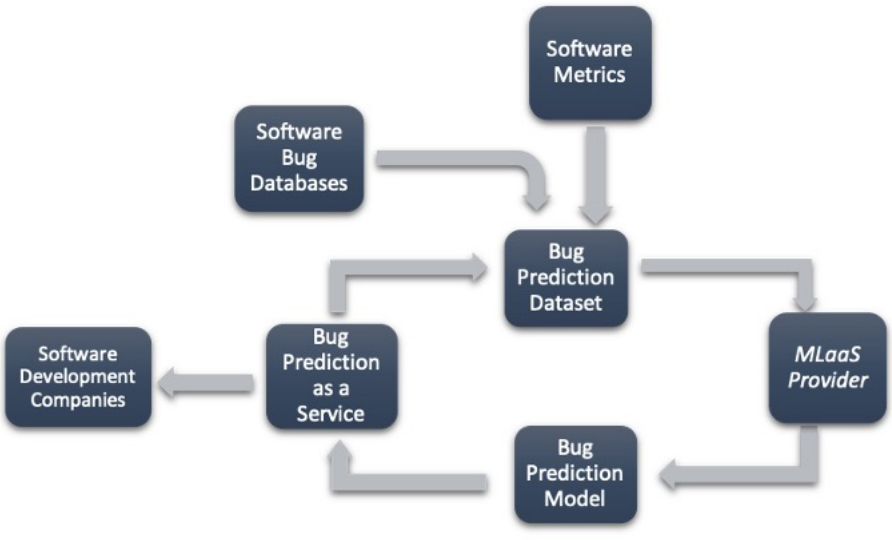


Figure 1.5. Bug Prediction Life Cycle.

With the progress of technology and above all with the spread of cloud platforms, also about bug prediction, the research community has focused on the possibility of providing prediction models as a Service. The advantage of using MLaaS is that users can easily complete an ML task through a web page interface, thus the cloud-based context simplifies and make ML even accessible to non-experts. Another benefit is the affordability and scalability because these services inherit the strengths of the beneath cloud infrastructure (Yao et al., 2017). The study of Yao et al. (2017) empirically analyzes the performance of MLaaS platforms, focusing on how the user control impacts both the performance and performance variance of classification in common ML tasks and the study shows that used correctly, MLaaS systems can provide results comparable to standalone ML classifiers.

## Bug Prediction with Antipatterns

Although a code, even if affected by antipatterns (or smells) can still work correctly, however, their presence signals the weaknesses of the design choices and this may lead to bugs in the future.

First Khomh et al. (2012) and then (Palomba, Bavota, et al., 2018) demonstrated that classes with antipatterns are more likely to have bugs than other classes. As we know, antipatterns can be removed using the appropriate refactoring techniques, it is obvious that if we manage to predict which portion of code will probably contain bugs by simply using antipatterns, developers will be able to use refactoring to reduce the risk for bugs in the code.

Taba et al. (2013) were the first to propose a bug prediction model which takes into account code smells information. Their research focus on the relationship between antipatterns and bugs density and they have tested their study on multiple versions of two open-source systems: Eclipse[5] and ArgoUML[6]. In particular, they presented 4 antipattern-based metrics: Average Number of Antipatterns (ANA), Antipattern Complexity Metric (ACM), Antipattern Recurrence Length (ARL), and Antipattern Cumulative Pairwise Differences (ACPD). The outcomes of the study have shown that files with antipatterns have a greater predisposition to contain a high bug density than others without antipatterns. The proposed metrics

[5]https://www.eclipse.org/

[6]https://en.wikipedia.org/wiki/ArgoUML

provide an increase of performance up to 12.5 percent. Among the four metrics, ARL has shown significant results, and it improves the bugs prediction both if it is used in a within-project context and if it is used cross-projects context.

Palomba et al. (2016) have conducted a study to investigate whether and how the severity of code smells impacts the bugs density. They used an intensity index which is an estimation of the severity of a code smell and its values are in the range [ 1 , 10 ]. The index computed using a code smell detector called JCodeOdor[7], and they have focused on six different types of code smells:

1. God Class: It occurs when a class is too large, it controls most of the processing and takes most of the decisions.

2. Data Class: It is when the only purpose of a class holding data.

3. Brain Method: A large method that implements more than one function.

4. Shotgun Surgery: It is when every change made in a class triggers many little changes to several other classes.

5. Dispersed Coupling: It occurs when inside a class there are too many relationships with other classes.

6. Message Chains: A method containing a long chain of method calls.

For each code smell instance is computed its intensity taking into account the code smell detection strategy, the metric thresholds used in the detection strategy, the statistical distribution of the metric values computed on a large dataset represented as a quantile function, and the actual values of the metrics used in the detection strategies. The task of the code smell detector is to classify the training data set as smelly and non-smelly classes to build a bug prediction model. Classes that do not have any code smells are getting an intensity value of 0 and so on. The results of the study show how the use of intensity always positively contributes to identifying bug-prone code components. This work has been performed only in the context of within-project bug prediction, therefore, we do not know specifically how the model behaves in a cross-project context.

[7]https://essere.disco.unimib.it/jcodeodor/

Ubayawardana and Karunaratna (2018) provided empirical evidence that code smells-based metrics can be very helpful in bug prediction. They presented a bug prediction model which uses different source code metrics and code smell-based metrics, regarding ML algorithms they used Naive Bayes, Random Forest, and Logistic Regression to build the model that was trained against multiple versions of 13 different open-source projects. They concluded by stating that Cross-projects bug prediction can be accurately done with the help of code smell-based metrics, and is possible to obtain higher accurate results when more than 30% of no buggy instances were in the training set.

## 1.7 Thesis Work Goals

This thesis work aims to find a possible correlation between the Bug Prediction and the test suites fragility.

Firstly, have been selected the most used machine learning algorithms in the literature and have been described all their features. Secondly, has been performed an analysis, trying different techniques, of the performance of the algorithms to choose the best one. Thirdly, was conducted the quality analysis using static metrics on 30 Android projects, furthermore, was built a shell script to automate the various tools and collect the outcomes.

The last part of this thesis includes the discussion of the final results obtained through Linear Regression and will illustrate any further suggestions for future works.

Specifically, the remainder of this thesis is structured as follows:

- Chapter 2 describes the distinctive features of the evaluated ML techniques;
- Chapter 3 covers the development process of the experiment, together with the description of the used metrics and tools;
- Chapter 4 illustrates the Bug Prediction's results and will be commented correlation results;
- Chapter 5 shows the conclusions and suggestions for the future works.

# Chapter 2

# Machine Learning Algorithms

Over the last ten years, there have been many studies in the literature about the prediction of the presence of bugs within the production code. In these studies, the effectiveness of existing simple techniques (such as Naïve Bayes (2.2), Support Vector Machine (SVM) (2.3), Logistic Regression (2.1) and Random Tree (2.4)) and new proposals were analyzed. The latter usually consists of a combination of software metrics, that until then, had only been used for locating bugs, and subsequently, their effectiveness was evaluated.

At the moment, net of my research, similar work has not yet been done regarding the test code, so the first step will be to start identifying among these studies which one is most suitable for being re-adjusted to our case.

## 2.1 Logistic Regression

Logistic regression is a supervised classification algorithm used to predict the probability that a given input data belongs to a specific class. Then, these probabilities must be transformed into binary values (they can assume values in the range $[\,0,\,1\,]$) in order to actually make a prediction and this is the task of the logistic function[1], also called the sigmoid function.

Particularly, logistic regression is used to measure the relationship between the dependent variable (i.e., what we want to predict) and one or more independent

[1]Logistic Function

variables (i.e., our features), estimating probabilities by means of the logistic function.

Logistic regression is represented by the equation:

$$y = \frac{e^{(\beta_0 + \beta_1 * x)}}{1 + e^{(\beta_0 + \beta_1 * x)}} \tag{2.1}$$

with:

- $x$ is the input value;
- $\beta_0, \beta_1$ are coefficients of the input values (constant real numbers). The Beta coefficients are estimated using the maximum likelihood method[2] and without going into details, this estimation can be reached by using an efficient numerical optimization algorithm (e.g., Quasi-Newton method[3]);
- $y$ is the output prediction.

In order to get the logistic regression equation expressed in probabilistic terms, we have to introduce the probabilities in equation (2.1). Thus, it is possible to model the probability of an input ($X$) that belongs to the default class ($Y = 1$). We can formally write that:

$$P(X) = P(Y = 1 \mid X) \tag{2.2}$$

Therefore, assuming that $y$, which is the result, belongs to class 1 (2.2), the (2.1) can be write as:

$$P(X) = \frac{e^{(\beta_0 + \beta_1 * X)}}{1 + e^{(\beta_0 + \beta_1 * X)}} \tag{2.3}$$

**Pros**:

- It is easier to implement, interpret, and very efficient to train so much so that usually is the first algorithm to be tried and sometimes it is used to evaluate the precision of other algorithms.
- It does not need the input features to be scaled.

[2]Maximum Likelihood Estimation

[3]Quasi-Newton

- It does not require any tuning and it is easy to regularize.
- It can easily be extend to multiple classes(multinomial regression).
- The outcomes are well-calibrated predicted probabilities.

**Cons**:

- It cannot solve non-linear problems since its decision surface is linear.
- It can only be used to on discrete functions. Thus, the dependent variable of Logistic Regression is bound to the discrete number set.
- If the number of observations is lesser than the number of features, it may lead to overfitting[4].

## 2.2 Naïve Bayes

It is a probabilistic classifier made by a set of methods based on *Bayes' Theorem*[5] with a naive assumption of independence among predictors. In other words, a Naïve Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Naïve Bayes model is easy to build and very useful for large datasets and it is known to outperform even highly sophisticated classification methods.

Bayes' Theorem allows to calculate the posterior probability $P(c|\mathbf{X})$ from $P(c)$, $P(\mathbf{X})$ and $P(\mathbf{X}|c)$:

$$P(c|\mathbf{X}) = \frac{P(\mathbf{X}|c)P(c)}{P(\mathbf{X})} \tag{2.4}$$

where:

[4]Overfitting

[5]Bayes' Theorem

- $P(c|\mathbf{X})$ is the posterior probability of class 'c' (target) given the dependent feature vector '$\mathbf{X}$' (of size n):

$$\mathbf{X} = (x_1, x_2, x_3, \ldots, x_n) \tag{2.5}$$

- $P(c)$ is the prior probability of the class.

- $P(\mathbf{X}|c)$ is the likelihood which is the probability of predictor given class.

- $P(\mathbf{X})$ is the prior probability of predictor.

Considering the naive assumption which is the independence among the features, we know that if any two events A and B are independent, then:

$$P(A|B) = P(A)P(B) \tag{2.6}$$

Applying (2.5) and (2.6) to the (2.4), it becomes:

$$\begin{aligned} P(c|\mathbf{X}) &= \frac{P(c)P(x_1|c)P(x_2|c)P(x_3|c)\ldots P(x_n|c)}{P(x_1)P(x_2)P(x_3)\ldots P(x_n)} = \\ &= \frac{P(c)\prod_{i=1}^{n} P(x_i|c)}{\prod_{i=1}^{n} P(x_i)} \end{aligned} \tag{2.7}$$

**Pros**:

- It is easy and fast to predict class of test dataset. It also performs well in multi-class prediction.

- When the assumption of independence is verified, a Naive Bayes classifier performs better compared to other models, such as logistic regression, and it needs less training data.

- It performs well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

**Cons**:

- If the categorical variable has a category (in test dataset), which was not observed in training dataset, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is known as "Zero Frequency" and to solve this problem, it is possible to use the Laplace estimation.
- Another drawback of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent.

## 2.3 Support Vector Machine

*Support Vector Machine*[6] or SVM is a supervised machine learning algorithm which can be used for both classification and regression. In the SVM algorithm, the data item is seen as a point in n-dimensional space (where n is number of features) and its value represents a particular coordinate.

The goal of the SVM algorithm is to find the hyperplane that best divides the n-dimensional space into two classes so that a new data item can be easily inserted in the correct category in the future. Theoretically, it is possible to draw infinite hyperplanes that divide the data, but there is only one that is optimal, the one that minimizes the error when happen a new classification.

In order to select the best hyperplane, the SVM chooses extreme points from both the categories which are called support vectors and selects the hyperplane that is farthest from all support vectors. Mathematically speaking, SVM chooses the hyperplane that maximizes the minimum distance between that hyperplane and all the support vectors. During testing, the class label $y$ of a class pattern $\mathbf{x}$ is determined by:

$$y = \begin{cases} n, & \text{if } d_n(\mathbf{x}) + t_l > 0 \\ 0, & \text{if } d_n(\mathbf{x}) + t_l \leq 0 \end{cases} \tag{2.8}$$

[6]Support Vector Machine

where $d_n(\mathbf{x}) = \max\{d_i(\mathbf{x})\}_{i=1}^{N_l}$, $d_i(\mathbf{x})$ is the distance from $\mathbf{x}$ to SVM hyper-plane corresponding to class $i$, and $t_l$ is the classification threshold.

If such hyperplane does not exist, SVM uses a non-linear mapping, called the *kernel trick*[7], to transform the training data into a higher dimension (e.g., the data in two dimensions will be evaluate in three dimensions and so on), in this way the data of two classes can always be separated by a hyperplane. It is mostly useful in non-linear separation problem.

**Pros**:

- It is effective in high dimensional spaces.
- It is effective in cases where the number of dimensions is greater than the number of samples.
- It uses a subset of training points in the decision function (support vectors), so it is memory efficient.

**Cons**:

- It does not perform very well when the data set has more noise (i.e. target classes are overlapping).
- It does not perform well when the data set is large due to the higher required training time.
- SVM does not directly provide probability estimates, they are calculated using an expensive five-fold cross-validation.

[7]Kernel Method

## 2.4 Decision Tree

Decision tree[8] (DT) is a supervised learning algorithm that is used for both classification and regression problems. The algorithm used to predict the class or value of the target variable consists to splitting the sample into two or more homogeneous sets by means of decision rules extracted from training data.

In Decision Trees, the algorithm starts from the root of the tree, it compares the values of the root attribute with the record's attribute. According to the outcome of comparison, DT follows the branch corresponding to that value and jump to the next node until the stopping criteria are reached.

There are two types of decision trees, they are based on the type of the target variable:

1. **Categorical Variable Decision Tree**: The target variable is categorical;
2. **Continuous Variable Decision Tree**: The target variable is continuous;

**Pros**:

- It is easy to understand thanks to its graphical representation which is very intuitive and users can easily relate their hypothesis.
- It can be used as the fastest way to identify most significant variables and relation between two or more variables in data exploration stage.
- It requires less data cleaning because it is not influenced by outliers and missing values.
- It can handle both numerical and categorical variables.
- It is a non-parametric method, it means that DT has no assumptions about the space distribution and the classifier structure.

**Cons**:

- It is not immune to overfitting, but it can be avoided by applying Pruning method (2.4.1) or using Random Forest (2.4.2).

[8]Decision Tree

- While working with continuous numerical variables, decision tree looses information when it categorizes variables in different categories.

### 2.4.1 Pruning

Pruning reduces the size of decision trees by removing branches of the tree that do not provide power to classify instances. Decision trees are very susceptible to overfitting and effective pruning can reduce it.

In practice, overfitting means that the final subsets (leaves of the tree) each enclose only one or a few data points. Although, the tree learned the data exactly, a new data point might not be predicted well.

There are three pruning strategies:

1. **Minimum error.** The tree is pruned back until is reached the point where the cross-validated error is a minimum. The cross-validation consists building a tree with most of the data and then using the remaining part to test the accuracy of the decision tree.

2. **Smallest tree.** The tree is pruned back less beyond the minimum error.

3. **None.**

An alternative method to prevent overfitting is to try to stop the tree-building process early, before the leaves with very small samples are produced. This technique is known as early stopping or as pre-pruning decision trees. This technique at each stage of splitting the tree checks the cross-validation error, if the error does not decrease significantly then it stops.

Early stopping and pruning can be applied together, separately, or neither. Post pruning decision trees is more mathematically rigorous, while early stopping is a quick fix heuristic.

### 2.4.2 Random Forest

Random Forest[9] (RF) is a versatile machine learning method capable of performing both regression and classification tasks. Random forest, as suggested by the name, is made by a large number of single decision trees that work as an ensemble[10]. Each tree produces a class prediction and the class which obtains the most votes becomes the prediction model.

The hallmark of RF algorithm is the low correlation between models, because uncorrelated models can make ensemble predictions that are stronger and more accurate than any of the single predictions. The reason behind the RF's strength is that the trees protect each other from their individual errors, indeed while some trees may be wrong many other trees will be right, therefore the group of trees is able to move in the correct direction.

Hence, random forest in order to perform well must have these prerequisites:

- The input features must have some signal (predictive power) so that predictions made by the models built using those features are better than random guessing.

- The predictions (and then the errors) made by the single tree need to have low correlation with each other.

Random Forest in order to ensures that the behavior of each single tree is not correlated with the behavior of any of the other trees, it uses the following methods.

**Bagging.** The bagging technique[11], also know as Bootstrap Aggregating, is used with decision trees, it remarkably raises the stability of models in the reduction of variance and improving accuracy and as result it eliminates the problem of overfitting. This technique takes several weak models and aggregates the predictions to select the best prediction.

**Feature Randomness.** In a classical decision tree, when it is time to split a node, it is considered every possible feature and it is selected the one that produces

[9]Random Forest

[10]Ensemble Methods

[11]Bagging Technique

the most separation between the observations in the left node and those in the right node. On the other hand, in random forest the single tree can pick only from a random subset of features and as consequence there will be even more variation amongst the trees in the model that will lead to lower correlation across trees and more diversification.

**Pros**:

- Random Forest can handle large dataset with higher dimensionality and identify most significant variables, for this reason it belongs to dimensionality reduction methods. Furthermore, the model prints out the *Importance of variable*[12], which can be a very handy feature.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- It has methods for balancing errors in datasets where classes are imbalanced.

**Cons**:

- It works better for classification but not as good as for regression problems since it does not give precise continuous nature predictions. Particularly in case of regression, it does not predict beyond the bounds in the training data, and that may overfit datasets that are significantly noisy.
- Random Forest works as a black box, i.e. you have very little control on what the model does.

[12]Importance of variable

## 2.5 Artificial Neural Networks

Artificial Neural Networks[13] (ANNs) belongs to statistical learning algorithms used in machine learning and cognitive science domains. They are based on the concept of biological neural networks found in the central nervous system of animals. They are useful when it is required to approximating a function that depends on some inputs that are often unknown.

A feed forward ANN has an input tier, output tier, and one or more hidden tiers. Each of them corresponds to a number of processing units. Each unit consists in the model of an artificial neuron. The units belonging to one layer are linked to the units in the next layer and each connection between two units is associated with a weight which indicate connection strength. The weights are obtained by training data, which are an ordered pair made of input data and expected result. All the connection weights are called weights vector.

During the learning phase, ANN propagates the error backward (i.e., output units to the input units) by adjusting the weights using a back-propagation algorithm. The algorithm consists of two phases: forward- and back- propagation. In the forward-propagation phase, a pair from the training set is given to the input tier which will generate an output using the current weight vector. Then, the error between the actual output and the desired output is computed and, of course, the goal of training is to minimize this error. In the other hand, during the back-propagation phase, the weights are adjusted in order to minimize the error. These steps are repeated for all the pairs in the training data.

**Pros**:

- ANNs are flexible and can be used for both regression and classification problems. Any data which can be expressed as numeric variable can be used in the model because ANN is a mathematical model with approximation functions.
- ANNs are suitable for model with nonlinear data with large number of inputs.
- Once trained, the predictions are very fast.

[13] Artificial Neural Network

- ANNs can be trained with any number of inputs and layers.
- ANNs work better with more data points.

**Cons**:

- ANNs are black boxes, meaning no one can know how much the independent variables are influencing the dependent variables.
- It is computationally very expensive and time consuming to train with traditional CPUs.
- ANNs are very training data dependent, this can lead to over-fitting and generalization problems.

### 2.5.1 Bayesian Network

A Bayesian network[14] is a directed acyclic graph (DAG), composed of **E** edges and **V** vertices which are the joint probability distribution of a set of variables. Each vertex represents a variable and each edge represents the causal or associational influence of one variable to its successor in the network.

Let $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ be the variables that can assume either continuous or discrete values. The probability distribution of $x_i$ is expressed as $P(x_i \mid a_x)$ where $(a_x)$s represent the parents of $x_i$, if any. When do not exist any parents of $x_i$, the probability distribution can be shown as $P(x_i)$.

The joint probability distribution of $\mathbf{X}$ can be calculated using *chain rule*[15], as for Bayes' Theorem:

$$P(\mathbf{X}) = \prod_{i=1}^{n} P(x_i \mid x_{i+1}, \ldots, x_n) \tag{2.9}$$

Since other variables are independent from $x_i$ and given the parents $(a_x)$ s,

[14] Bayesian Networks

[15] Chain Rule

the (2.9) can be written as:

$$P(\mathbf{X}) = \prod_{i=1}^{n} P(x_i \mid a_{x_i}) \tag{2.10}$$

By using the Bayes' Theorem (2.4) it is possible to compute the probabilities either from causes to effects $P\,x_i \mid E)$ or from effects to causes $P(E \mid x_i)$.

In Bayesian network the search space is composed of all of the possible combination of directed acyclic graphs based on the given variables (nodes). Thus, it is very difficult to compute all of these possible DAGs without a heuristic method, such as *K2 Algorithm* (Cooper & Herskovits, 1992).

**Pros**:

- It can incorporate both direct evidence and indirect evidence into a single analysis.
- It can produce results for all comparisons of interest within a connected network.
- It can incorporate meta-regression to assess heterogeneity.

**Cons**:

- It requires greater statistical expertise than some other methods.
- It might produce results not accurate for one closed loop networks.
- It is sensitive to the prior probabilities chosen.

### 2.5.2 Radial Basis Function Network

The Radial Basis Function[16] (RBF) Network belongs to ANNs which uses radial basis function as activation functions (Buhmann, 2000). RBF is made of three tiers:

1. input tier which corresponds to the predictors (software metrics).
2. the output tier which maps the outcomes to predict (the defect-proneness of entities).
3. the hidden tier which is needed to connect the input tier with the output tier.

The hidden tier is activated using a radial basis function, i.e. a Gaussian function. The prediction model is defined as follows:

$$P(c_i) = \sum_{k=1}^{n} \alpha_k e^{\frac{-\|x-\gamma_k\|}{\beta}} \tag{2.11}$$

where $P(c_i)$ is the defect proneness prediction of the entity $c_i$, $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the set of linear weights, and $\gamma_k$ are the centers of the radial basis function.

Finally, the whole function $\left(e^{\|x-\gamma_k\|-\frac{1}{\beta}}\right)$ is the radial basis function which is the center of the RBF network.

### 2.5.3 Multi-layer Perceptron

The Multi-layer Perceptron[17] (MLP) belongs to ANNs and it is trained using a back-propagation algorithm. Usually, the multi-layer perceptron is composed by multiple layers of nodes in a directed graph: an input layer, one or more hidden layers, and an output layer. The output from a layer is used as input to nodes in the subsequent layer. The model can be formally defined using the following mathematical formulation:

$$P(c_i) = \sum_{k=1}^{n} w_k \frac{1}{1 + e^{\alpha + \beta_1 \dot{x}_1 + \beta_2 \dot{x}_2 + \cdots + \beta_n \dot{x}_n}} \tag{2.12}$$

[16]Radial Basis Function

[17]Multi-layer Perceptron

where $P(c_i)$ is the defect-proneness prediction of the entity $c_i$, $\alpha$, $\beta_1$, $\beta_2, \ldots, \beta_n$ are the linear combination coefficients while $w_k$ are the weights of the each layers.

# Chapter 3

# Experiment design

To achieve our goal, i.e. understand if the *'Bug Prediction'* can be applied to the test code and, especially, if the former can be used as a predictor for the *'Test Fragility'*, in other words, if there is any sort of correlation between *'Bug Prediction'* and *'Test Fragility'*.

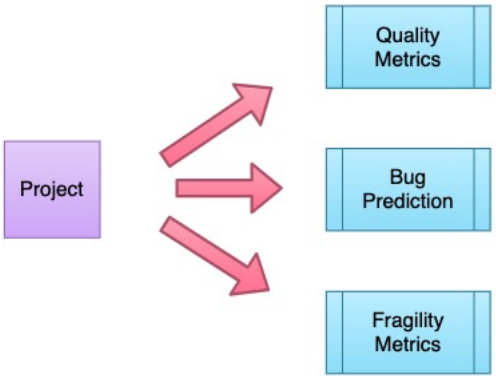


Figure 3.1. Experiment Data Flow.

The experiment is divided into two phases: the first phase is focused on the Bug Prediction, whereas the second one on the correlation between the Bug Prediction and fragility metrics.

The first phase can be outlined as: firstly go through the literature and select the most used quality metrics for Java, secondly find which public bug dataset is more suitable for our scope and finally test which MLAs[1], thoroughly described in

[1]Machine Learning Algorithms

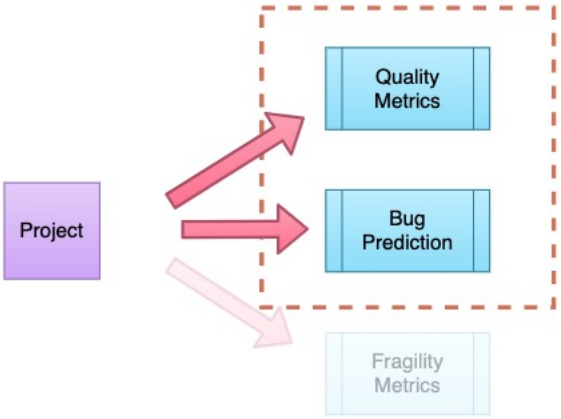


Figure 3.2. First phase of the Experiment.

the Chapter2, perform well when trained on the dataset found during the previous step.

The second phase can be summarized as firstly cherrypick at least 30 projects to be analyzed, secondly examine the obtained results and by means of a *Linear Regression* determine a possible correlation between bug prediction and fragility metrics (Chapter4).

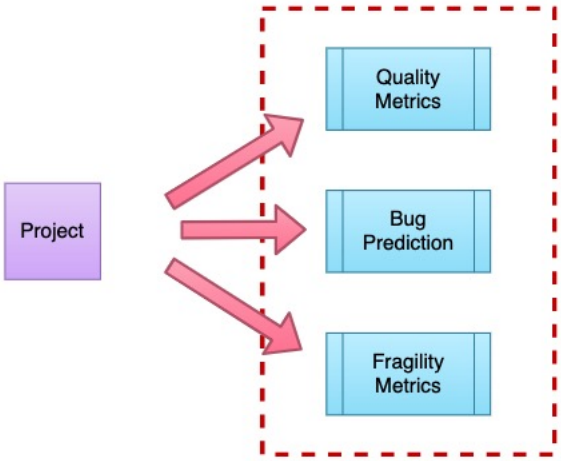


Figure 3.3. Second phase of the Experiment.

## 3.1 Metrics and tools

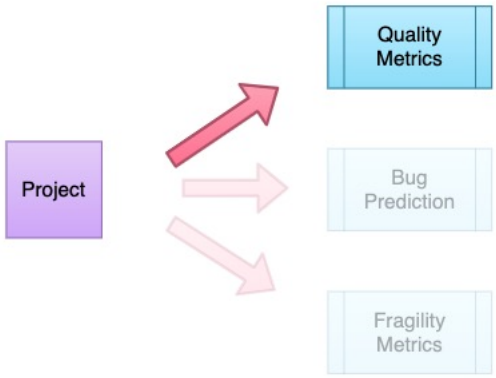


Figure 3.4. Focus on quality metrics.

From the scientific literature, it emerged that the following nine metrics were the most used for the Java language, and they are a combination of Class-level and method-level metrics:

- **WMC:** *Weighted Method per Class*. This metric estimates the complexity of a class as the sum of the complexity of its methods.
- **CBO:** *Coupling Between Objects*. This metric counts the number of classes the given class depends on (coupling). Two classes are defined as coupled when methods belonging to one class use methods or instance variables defined inside the other class.
- **Ca:** *Afferent Coupling*. This metric measures how many classes depend on a given class. Classes with high afferent will affect other classes when changes are made.
- **RFC:** *Response For Class*. This metric counts the number of methods that are executed after they received a message by an object of that class.
- **DIT:** *Depth of Inheritance Tree*. This metric expresses the length of the maximal path from the leaf node to the root of the inheritance tree of the classes of the software under analysis.

- **NOC:** *Number Of Children.* This metric counts the number of direct subclasses of the class under analysis. When the value of NOC increases, maintainability of the code increases.
- **LOC:** *Lines of Code.* This metric expresses an immediate measure of the size of the source code and even though it should be the most unambiguous metric, there are several definitions of it in the literature.
- **NPM:** *Number of Public Methods.* This metric returns the number of all the public methods in a class.
- **CC:** *McCabe's Cyclomatic Complexity.* This metric estimates the complexity of the code by examining the control flow graph of the program.

In order to calculate these metrics, 3 open source tools were used taken from the various studies that dealt with the quality of the software:

**ckjm** calculates C&K object-oriented metrics by processing the bytecode of compiled Java files (Spinellis, 2005).

**cloc** counts blank lines, comment lines, and physical lines of source code.[2]

**checkstyle** is an open source tool, highly configurable, that can be used by developers to write Java code respecting the coding standard (best practice).[3] In this work, it was used to compute McCabe's Cyclomatic Complexity.

## 3.2 Bug Prediction

The goal of this step is to build a model which predicts bugs within projects, we are handling a classification problem where we need to predict the value of a discrete variable (label) that can assume a specific value belonging to a limited set. Specifically, the label which is named ***'bug'***, is used to mark as buggy if the file contains at least one bug and as not buggy, if there are no bugs inside, therefore it can assume only two values that are true or false, and for that reason, we are facing a binary classification problem.

[2] cloc

[3] Checkstyle

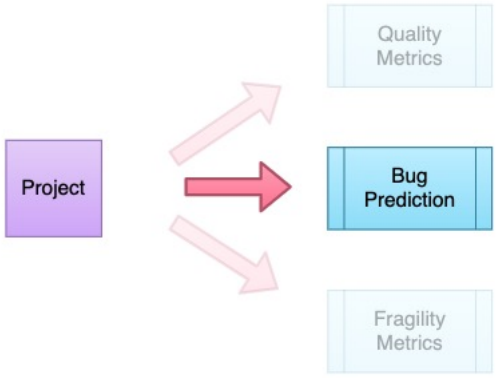


Figure 3.5. Focus on Bug Prediction.

In general, a classifier before being able to be used for prediction on real data must go through a training phase using a dataset of defined examples with which the model learns the optimal combination of variables that will generate a good predictive model. Subsequently, the trained model is applied to a second dataset called the test dataset which must be independent of the training dataset, but at the same time must have the same probability distribution as the training dataset. These two constraints are necessary to obtain an unbiased evaluation of the trained model, to obtain the performance characteristics such as accuracy, sensitivity, specificity, F-measure, etc. The tool used is **weka**[4] which is an open-source machine learning software that can be used through a graphical user interface, standard terminal applications, or a Java API.

The dataset used in this work was created by Ferenc et al. Ferenc et al. (2020) and is the result of the union of 5 different public datasets: PROMISE dataset (Sayyad Shirabad & Menzies, 2005), ECLIPSE dataset (Zimmermann et al., 2007), Bug Prediction dataset (D'Ambros et al., 2010), Bugcatchers Bug dataset (T. Hall et al., 2014) and GitHub Bug dataset (Tóth et al., 2016).

Since there is a single dataset available, and it is very large, instead of creating a new one for the test phase, has been applied first the percentage split technique to obtain two datasets from the starting dataset. A larger one is used for training and a much smaller one for testing. Then, has been used on the original dataset

[4]weka

the k-fold cross-validation, which consists of dividing the dataset into $k$ equal parts and using $k-1$ for the training phase and 1 for the test phase, this operation is repeated $k$ times choosing $k-1$ different batches each time to obtain $k$ different models. Finally, we also assigned a cost matrix (3.1) to the classifiers to improve the results by minimizing the final cost. The main diagonal is all *zeros* as a correct prediction costs zero, whereas a false negative prediction costs 38838 and a false positive prediction costs 8780. These numbers are not random but they are chosen both to meet the constraint that a false negative prediction costs more than a false positive prediction and at the same time balance the dataset.

After computed the performances of the models and identified the model or models with the best results (Chap.4), the prediction can be made on new data, the so-called unseen data, or unpredicted data.

Table 3.1. Cost Matrix

| a | b | |
|---|---|---|
| 0 | 38838 | a = True |
| 8780 | 0 | b = False |

## 3.3 Project Selection

During the first part has been developed a shell script[5] to automate all the steps described so far, the Fig. 3.6 shows the script structure. As shown by the figure, the script takes as input a project's location, then it will isolate the test classes on which will be computed the before mentioned metrics by running the respective tools. Each tool has its own input/output file format and generates a file containing the metrics results that must be merged and converted in the correct format before running the last tool.

Previously, at the Polytechnic of Turin was made another thesis, whose aim was to find a possible correlation between the maintainability of the source code and the fragility of the tests by analyzing the code of a set of 101 Android projects with their test suites, considering all versions (Pirrigheddu, 2021). Hence, for the

[5]The script can be download here.

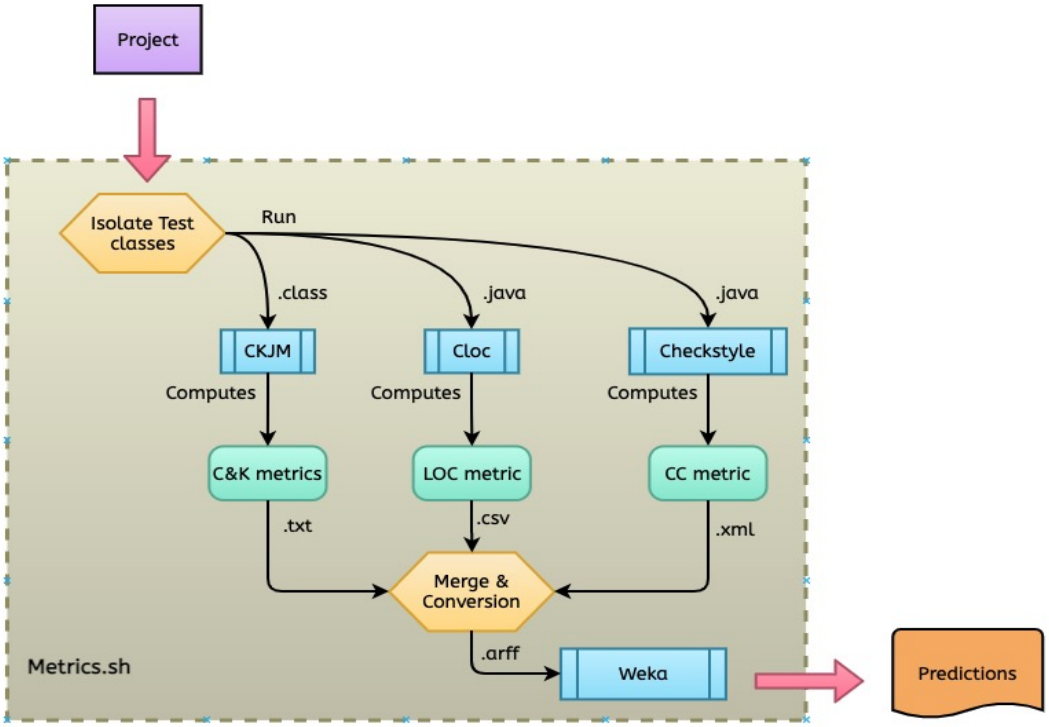


Figure 3.6. Modular representation of the shell script.

second part of this thesis was decided to draw from the Android projects already analyzed in the before mentioned thesis since the fragility metrics have already been calculated for these projects.

However, there were many difficulties because the tools used in the bug prediction, which have been mainly used on the Desktop application, now have to be run on Android projects. One clear example is the latest version of the tool used to compute the main metrics dates back to 2011 and the Java language has changed since then, hence we were not sure whether it worked for Android projects a priori. Then, even though many Android projects are still written mainly in Java, the use of Kotlin is quite widespread, therefore not all the 101 projects were suitable to be used but it was necessary to narrow down the selection by picking only those projects that have a very long release history to find versions written entirely in Java.

Subsequently, we had to analyze this restricted set of projects, and discard all the corrupted projects and select only those that compiled without errors as the 'ckjm' tool works on the binary files (.class) of the single classes. With this further restriction from the initial 101 projects, we managed to get 30 working projects which, in addition to ensuring the feasibility of this approach, also guarantees a good number of projects to have statistical significance. Later, to work on these 30 processes it was necessary to understand how the Android Studio generates the binary files because the Android operating system is based on the 'Dalvik' virtual machine, so the executables are not classic java binary files but specific binary files (.dex) which can run on the Android virtual machine (Fig. 3.7).

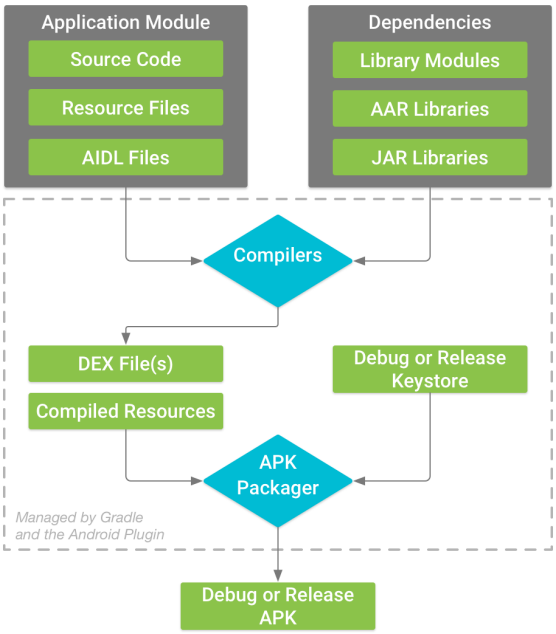


Figure 3.7. Android Build Process (Google).

Ckjm computes C&K metrics using as input the java binary files and by means the 'Apache Bcel' libraries every time it encounters a method call can rebuild the dependency to the class, to which that specific method refers to, and therefore all the dependencies must be set in command-line option 'classpath'. For this reason, the main difficulty was to find where the IDE generated the java binary files, where it downloaded the binaries of all the Android and third-party libraries, which differ for each project and it depends on the API version.

After a close study of the IDE compilation process, it was possible to locate the java binary files of the classes which are generated before the corresponding Dalvik binary files and to find out that the binary files of the libraries downloaded in temporary directories that varied from project to project within the cache directory used by the IDE, therefore was used a shell script (Fig.3.8) to build the dependency path of each project and pass it to the tool 'ckjm'. For the other tools used in this thesis it was not necessary any type of extra work, as in the case of 'ckjm', because they run on java files.

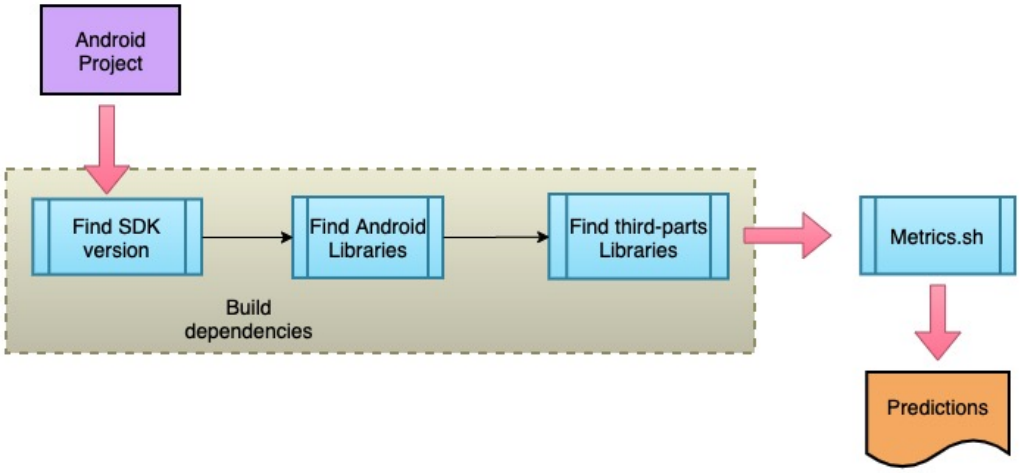


Figure 3.8. Final representation of the shell script.

## 3.4 Test Fragility

The first to give a general definition were Garousi and Felderer (2016) which cites: *'a test is defined as fragile when it does not compile or fails following changes on the System Under Test (SUT) that do not affect the part that the test is looking into.'*

Subsequently, a further definition was given by Coppola et al. (2019) which, although was given in the context of mobile development, gives a deeper insight about test fragility: *a GUI test case is fragile if it requires one or more interventions when the application evolves into subsequent versions due to any modification made on the Application Under Test (AUT).*

From both the before mentioned definitions it is clear that the presence of fragile tests within a project is very expensive both in economic terms as the software maintenance costs increase and in terms of time as it cots a lot of time for developers to modify fragile test codes in general, even more so if this happens every time there is a change in the functionality of the program (Garousi & Felderer, 2016).

## 3.5 Test Fragility Quantification

The tests' fragility is quantified by analyzing how applications and their tests have changed over their lifespan, comparing file by file. Coppola et al. (2019) defined 12 metrics that can be used as indicators of changes, and they are divided into two groups:

- Diffusion and Size metrics: used to estimate how widespread the test tools are and the amount of test suites that use them;
- Test evolution metrics: they describe the evolution of an Android project and the related test suite.

Table 3.2. Test Fragility Metrics.

| Group | Symbol | Name |
|---|---|---|
| Diffusion and Size metrics | TD | Tool Diffusion |
| | NTR | Number of Tagged Releases |
| | NTC | Number of Test Classes |
| | TTL | Total Test LOCs |
| | TLR | Test LOCs Ratio |
| Test evolution metrics | MTLR | Modified Test LOCs Ratio |
| | MRTL | Modified Relative Test LOCs |
| | MRR | Modified Releases Ratio |
| | TSV | Test Suite Volatility |
| | MCR | Modified Test Classes Ratio |
| | MMR | Modified Test Methods Ratio |
| | MCMMR | Modified Classes with Modified Methods Ratio |

### 3.5.1 Fragility Metrics

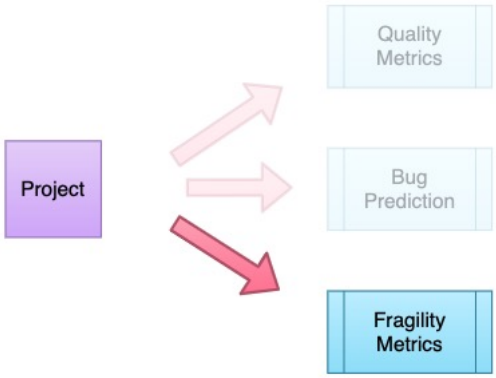


Figure 3.9. Focus on fragility metrics.

#### Diffusion and Size metrics

- TD: Percentage of projects that use a specific tool;
- NTR: Number of released versions of an Android project that are obtained with the 'git tag' command. This metric is used to understand the nature of the most tested application;
- NTC: Number of test classes belonging to a single version of an Android project;
- TTL: Total lines of code belonging to the test classes in a single version of an Android project;
- TLR: defined as:

$$TLR_i = \frac{TTL_i}{P_{locs_i}}, \tag{3.1}$$

where $P_{locs_i}$ are the total lines of code present within the version *'i'* of the project *'P'*.

**Test evolution metrics**

- MTLR: defined as

$$MTLR_i = \frac{Tdiff_i}{TTL_{i-1}}, \tag{3.2}$$

  where $Tdiff_i$ is the quantity of LOCs modified, added or deleted in the test classes, during the transition between versions $i-1$ and $i$, and $TTL_{i-1}$ is the total quantity of LOCs concerning the test classes in version $i-1$. This metric is defined only when $TTL_{i-1} > 0$, i.e. only when the previous version is provided with test code. It also quantifies the amount of changes made to the LOCs of existing test code for a specific project version.

- MRTL: defined as

$$MRTL_i = \frac{Tdiff_i}{Pdiff_i}, \tag{3.3}$$

  where $Tdiff_i$ is the quantity of LOCs modified, added or deleted associated with the test classes, in the transition between versions $i-1$ and $i$ and $Pdiff_i$ is the quantity of LOCs modified, added or deleted concerning the entire project, including test classes, in the transition between versions $i-1$ and $i$. This metric is defined when $Pdiff_i > 0$ and is calculated only in case $TRL_i > 0$. It returns a value between $[0-1]$ and values close to 1 imply that a significant part of the total work needed for the application evolution, was necessary to keep the tests up to date over the various versions.

- MRR: defined as the ratio between the number of released versions in which at least one test class has been modified and the number of released versions of the application (NTR). This metric returns a value in the range between $[0-1]$ and higher MRR values indicate a lower value of the adaptability of the test suite relating to changes in the Android application.

- TSV: defined for each project as the ratio between the number of test classes that are changed at least once in their lifespan and the total number of test classes in the history of the project.

- MCR: defined as

$$MCR_i = \frac{MC_i}{NTC_{i-1}}, \tag{3.4}$$

where $MC_i$ is the number of test classes that are modified in the transition between version $i-1$ and i $NTC_{i-1}$ is the number of test classes relative to version $i-1$. This metric is not defined when $NTC_{i-1} = 0$ and returns a value in the range between $[0-1]$. Furthermore, the higher MCR values are, the less stable the test classes are during the evolution of the Android application.

- MMR: defined as

$$MMR_i = \frac{MM_i}{TM_{i-1}}, \tag{3.5}$$

where $MM_i$ is the number of test class methods that are changed between versions $i-1$ and $i$. $TM_{i-1}$ is the total number of test class methods associated with version $i-1$. This metric is not defined when $TM_{i-1} = 0$ and returns a value in the range between $[0-1]$. Finally, the higher MMR values are, the less stable the methods of the test classes are during the evolution of the application they test.

- MCMMR: defined as

$$MCMMRi = \frac{MCMM_i}{NTC_{i-1}}, \tag{3.6}$$

where $MCMM_i$ is the number of test classes that are modified and that include at least one modified method between versions $i-1$ and $i$. $NTC_{i-1}$ is the number of test classes relative to the version $i-1$. This metric is not defined when $NTC_{i-1} = 0$ and is bounded above by MCR, since by definition $MCR_i = MC_i/TC_i$, and consequently $MCMM_i \leq MC_i$.

## 3.6 Metrics Normalization

The Bug Prediction process generates two text files for each project, the first (Fig. 3.10) contains for each class belonging to the project the final prediction (true/false) and the second (Fig. 3.11) a summary of how many classes were positive and how many were negative. The bug prediction is performed on the code quality metrics calculated for every single class belonging to the single project. While in the thesis Pirrigheddu (2021) for each project were obtained two files as a final result, one (Fig. 3.12) contains the fragility metrics that depend on the version of the project, and the second (Fig. 3.13) contains the fragility metrics calculated for the single project. Therefore, it was necessary to normalize the metrics so that we could merge the results of the bug prediction and the values of the metrics into a single table and then see if there is any sort of relationship between them.

For the fragility metrics depending on the project, the release was calculated the mathematical mean in order to obtain a single value for each metric that referred to the single project and no longer just to the single version of the project. To normalize the quality metrics were first isolated the test classes and then was computed a mathematical average so that the values refer to each project and not to a single class. Finally, only the percentages of the total number of true and false for each project were taken (Fig. 3.11).

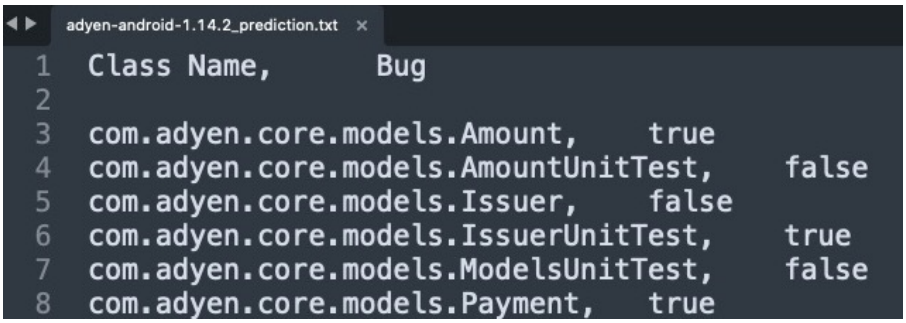


Figure 3.10. Prediction output example.

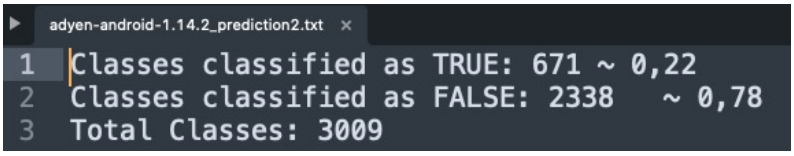


Figure 3.11. Prediction summary example.

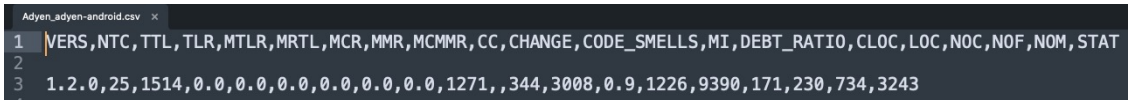


Figure 3.12. Fragility output 1 example.

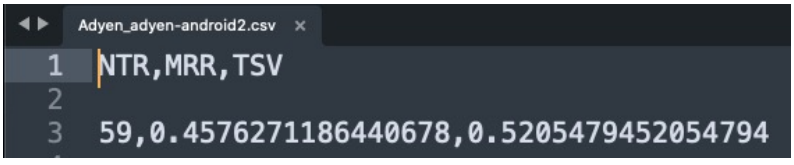


Figure 3.13. Fragility output 2 example.

# Chapter 4

# Results

The dataset of Ferenc, R. et al. Ferenc et al. (2020) is unbalanced as in some of the cases of reality, such as the prediction of a disease or a bad payer when talking about a loan from a bank. These problems are called imbalanced classifications, and the examples from the majority class are considered the negative class with a class label 0 (false). Those examples from the minority class are considered the positive class with a class label 1 (true). The reason behind the negative/positive naming convention is because the examples from the majority class typically represent a normal or no-event case, whereas examples from the minority class represent the exceptional or event case.

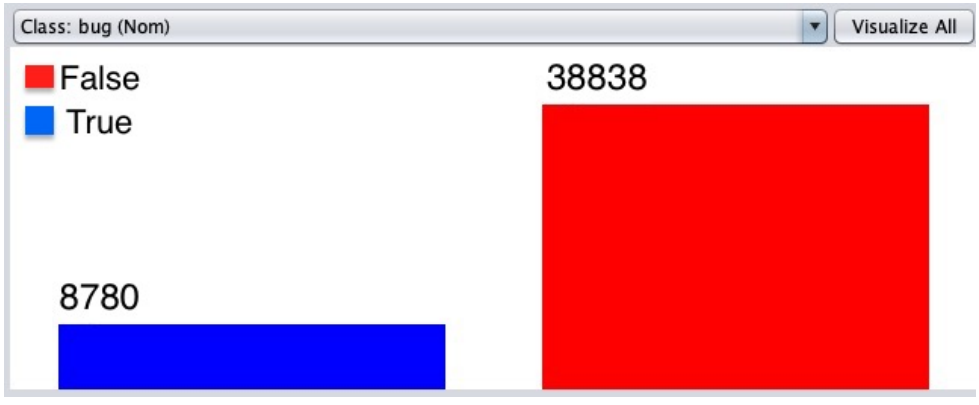


Figure 4.1. Classes Distribution

When dealing with unbalanced datasets, it is necessary to wonder:

- Are both classes equally important or is one more important than the other?

- A wrong negative prediction has the same cost as a wrong positive prediction or not?

Depending on the answers to the previous questions, there is a specific characteristic of the model that can be taken into consideration to quantify its performance and compare it with that of other models. Example: There is a model which can correctly predict 98% of instances because the majority of them belonged to a bigger class if we are interested in the bigger class and we are not concerned how many instances of the other class are correctly predicted or incorrectly predicted, then we can conclude that the performance of this model is very good. Instead, if we are interested in the smaller class we cannot rely on the correctly predicted percentage but we have to opt for a better metric in order to evaluate our model.

The assumption made in this work is that the positive class is the more important since the false-negative predictions are more costly (i.e., a buggy code classified as not buggy can provoke much more damage than a not buggy code classified as buggy), hence among the classic properties the one to focus on to evaluate the performance of the classifiers is F2-measure (4.2). In binary classification, the F-measure function is the Harmonic Mean of the precision and sensitivity values also known as F1-measure, moreover, there is the F $(\beta)$-measure (4.1) that according to the values of $\beta$ can highlights more the precision or the sensitivity. F1-measure is F $(\beta)$-measure with $\beta$ equal to 1. Optimal values of the F-measure are greater than 0.5 (e.g. 0.5 is random guessing).

$$F(\beta) = (1 + \beta^2) * \frac{precision * sensitivity}{(\beta^2 * precision) + sensitivity} \tag{4.1}$$

with $\beta = 2$, it becomes

$$\begin{aligned} F(2) &= (1 + 2^2) * \frac{precision * sensitivity}{(2^2 * precision) + sensitivity} = \\ &= 5 * \frac{precision * sensitivity}{(4 * precision) + sensitivity} \end{aligned} \tag{4.2}$$

# 4.1 Bug Prediction Performances

## 4.1.1 Logistic Model

The "Logistic Regression" was used in conjunction with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly approximately 82.4% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.1. Summary Logistic Classifier.

| | **Precision** | **Sensitivity** | **Specificity** | **F2 - measure** | **ROC - AUC** |
|---|---|---|---|---|---|
| **Values** | 0.609 | 0.109 | 0.984 | 0.130 | 0.724 |

The Logistic model is characterized by low sensitivity and a high precision (tab. 4.1), this means that the model cannot detect the positive class well but is very trustable when it does, indeed a high value of specificity involves the 'rule in' concept (i.e. a positive classification implies the presence of at least one bug, but a negative classification cannot exclude the presence of bugs). The final confirmation is given by the F2 measure which is very low, therefore the prediction is more biased towards the majority class.

In order to compensate for the dataset unbalance, we tried to apply a cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false-positive predictions. As consequence, we did have an improvement in the performances as shown in the table (4.2), but the influence of the majority class is still present.

Table 4.2. Summary Logistic with costs.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.364 | 0.515 | 0.798 | 0.475 | 0.657 |

### 4.1.2 Naive Bayes

The "Naive Bayes" was used along with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly roughly 81.3% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.3. Summary Naive Bayes Classifier.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.477 | 0.225 | 0.945 | 0.252 | 0.674 |

The Naive Bayes model is characterized by low sensitivity and a low precision (tab. 4.3), this means that the positive class is poorly handled by the model. On the other hand, a high value of specificity confirms that the model whenever detects a true value we can be very confident about it (rule in). A low F2 measure suggests that the Naive model is more biased on detecting the negative class.

Also in this case we applied the cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false-positive predictions. We did not have any improvement in the performances as shown in the table (4.4) due to the predominant influence towards the majority class.

Table 4.4. Summary Naive with costs.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.455 | 0.255 | 0.932 | 0.28 | 0.593 |

### 4.1.3 Decision Tree Model

The "Decision Tree" was used together with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly nearly 82.7% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.5. Summary Decision Tree Classifier.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.560 | 0.265 | 0.953 | 0.296 | 0.747 |

The Decision Tree model is characterized by low sensitivity and a low precision (tab. 4.5), this means that the positive class is poorly handled by the model. The F2 measure suggests that this model is biased towards the majority class. On the other hand, a high specificity value implies that the Decision Tree model when classifies as true an instance, is very reliable.

In order to tackle the strong influence of the majority class, we applied the cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false-positive predictions. We did have an improvement in the performances as shown in the table (4.6), in particular a significant improvement in the value of sensitivity which together with a good value of specificity means that the model can distinguish between the two classes.

Table 4.6. Summary Decision Tree with costs.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.362 | 0.6 | 0.763 | 0.53 | 0.682 |

### 4.1.4 Random Forest Model

The "Random Forest" was used combined with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly around 83.1% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.7. Summary Random Forest Classifier.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.568 | 0.328 | 0.944 | 0.358 | 0.783 |

The Random Forest model is characterized by low sensitivity and a low precision (tab. 4.7), this means that the positive class is poorly handled by the model. A high-value specificity confirms this model is very reliable when the prediction's outcome is true. The F2 measure suggests that with this model we cannot rule out the presence of bugs.

We applied the cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false positive predictions in order to balance the dataset. We did have an improvement in the performances as shown in the table (4.8), in particular a significant improvement in the value of sensitivity which together with a good value of specificity means that the model can distinguish between the two classes.

Table 4.8. Summary Random Forest with costs.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.355 | 0.715 | 0.708 | 0.594 | 0.712 |

### 4.1.5 SVM Model

The "Support Vector Machine" was used in conjunction with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly roughly 81.7% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.9. Summary SVM Classifier.

| | Precision | Sensitivity | Specificity | F2 - measure | ROC - AUC |
|---|---|---|---|---|---|
| **Values** | 0.643 | 0.007 | 0.999 | 0.009 | 0.503 |

The SVM model is characterized by very low sensitivity and a high precision (tab. 4.9), this means that the model cannot detect the class well but is highly trustable when it does. This is confirmed by the specificity value which is almost 1. The F2 measure suggests that this model is more biased towards the negative class therefore we cannot rule out the presence of bugs.

We also applied the cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false positive predictions in order to balance the dataset without having any improvement in the performances.

### 4.1.6 ANN Model

The "Artificial Neural Network" was used together with both 70% percentage split and k-fold cross-validation (k = 10), during the training phase and test phase. In both cases, the model predicted correctly approximately 82.6% of instances, therefore as the results obtained are very similar and for the sake of readability are reported only the percentage split results.

Table 4.10. Summary ANN Classifier.

| | **Precision** | **Sensitivity** | **Specificity** | **F2 - measure** | **ROC - AUC** |
|---|---|---|---|---|---|
| **Values** | 0.666 | 0.104 | 0.988 | 0.125 | 0.740 |

ANN model is characterized by very low sensitivity and a high precision (tab. 4.10), this means that the model cannot detect the class well but is highly trustable when it does. The F2-measure suggests that this model is more biased towards negative class thus it can rule out the presence of bugs, but on the other hand, the ANN model is reliable when detects the positive class.

We applied the cost-matrix (3.1) in which we gave a higher cost to false-negative predictions than to false positive predictions in order to balance the dataset. We did have an improvement in the performances as shown in the table (4.11), in particular a significant improvement in the value of sensitivity which together with a good value of specificity means that the model can distinguish between the two classes.

Table 4.11. Summary ANN Classifier with costs.

| | **Precision** | **Sensitivity** | **Specificity** | **F2 - measure** | **ROC - AUC** |
|---|---|---|---|---|---|
| **Values** | 0.352 | 0.611 | 0.747 | 0.533 | 0.679 |

## 4.2 Bug Prediction Summary:

In this work was used dataset Ferenc et al. Ferenc et al. (2020) which is composed of the union of 5 public datasets used mostly in literature for bug prediction. This dataset has a moderate unbalance between the two classes (true class <20%). Six classifiers were tested on this dataset and the results obtained (tab. 4.12) showed how all six were affected by the bias towards the majority class and even if they have a high specificity which generally implies that when a true value is classified, it is possible to rule in the presence of bugs but not vice-versa, their use must be excluded to classify new data. To confirm this, the F2-measure which is the chosen metric to evaluate the models is very low for all the classifiers, therefore the amount of false negative predictions is too high and the classification is compromised.

Table 4.12. Results Table

| | Precision | Sensitivity | Specificity | F2 | ROC |
|---|---|---|---|---|---|
| Logistic | 0.609 | 0.109 | 0.984 | 0.130 | 0.724 |
| Naive Bayes | 0.477 | 0.225 | 0.945 | 0.252 | 0.674 |
| Decision Tree | 0.560 | 0.265 | 0.953 | 0.296 | 0.747 |
| Random Forest | 0.568 | 0.328 | 0.944 | 0.358 | 0.783 |
| SVM | 0.643 | 0.007 | 0.999 | 0.009 | 0.503 |
| ANN | 0.666 | 0.104 | 0.988 | 0.125 | 0.740 |

In order to improve the results, other techniques have been exploited to balance the dataset, and the one which showed improvements was to apply a cost matrix (tab. 3.1) to the classifier so that we could give a cost to the misclassification, taking into account the assumptions made at the beginning of this chapter that is a cost higher to false negatives rather than false positives. We have seen improvements in the performance of some models (tab. 4.13), in particular, both sensitivity and F2-measures values have improved suggesting a better distinction between the two classes and a decrease of the false positives predictions. Nonetheless, these results are not optimal, and we choose the Random Forest model whose results have shown more improvements.

Table 4.13. Results after matrix-cost

| | Precision | Sensitivity | Specificity | F2 | ROC |
|---|---|---|---|---|---|
| Logistic | 0.364 | 0.515 | 0.798 | 0.475 | 0.657 |
| Naive Bayes | 0.455 | 0.255 | 0.932 | 0.28 | 0.593 |
| Decision Tree | 0.362 | 0.6 | 0.763 | 0.53 | 0.682 |
| **Random Forest** | 0.355 | 0.715 | 0.708 | **0.594** | 0.712 |
| SVM | 0.643 | 0.007 | 0.999 | 0.009 | 0.503 |
| ANN | 0.352 | 0.611 | 0.747 | 0.533 | 0.679 |

## 4.3 Bug Prediction working Example

Toggle[1] (Coppola et al., 2021) is a tool developed within the Politecnico of Torino University whose purpose is to reduce the costs and efforts related to the maintainability of the transition from second-generation tests, i.e. tests that work directly on the visual components and their state, to those of third-generation, which are tests that use image recognition algorithms and have the highest level of abstraction of the application under test (AUT). In particular, Toggle takes as input second-generation tests, such as tests generated by the "Espresso"[2] framework, and translates them both into "SikuliX"[3] and "EyeAutomate"[4] scripts in order to generate third-generation tests.

Since Toggle was conceived as an academic tool, despite having demonstrated the feasibility of a transition between the second-generation and the third-generation tests, it was affected by defections and weaknesses that did not make it suitable in an industrial context. For this reason, it has undergone a re-engineering process addressing the before mentioned problems, specifically, the tool was improved by reducing as much as possible the manual interaction requested to an end-user, the architecture was re-designed so that the end-user would get a clear, flexible, and easy-to-use interface and lastly were added new features that were missing in the first version of Toggle.

[1]TOGGLE = Translation Of Generations of GUI testing at Low Effort

[2]https://developer.android.com/training/testing/espresso/

[3]http://sikulix.com/

[4]https://eyeautomate.com/eyeautomate/

In order to verify the performance of the Bug Prediction script, it was decided to run it on the first version of Toggle, before were applied any improvements and then on the second one, i.e. after the changes, and to compare the results obtained. From the comparison it emerged what was expected, i.e. the 70% of the first version is affected by bugs (tab 4.14) while the second is only 53% affected by bugs (tab 4.15). Therefore, there is a decrease of about 20% confirming an actual improvement between the first and the second version, and in addition, the "Random Forest" algorithm in conjunction with weights (costs) produced acceptable results.

Table 4.14. Toggle v1.0 Bug Prediction

| TOGGLE v1.0 | | |
|---|---|---|
| True | 27 | $\sim$ **70%** |
| False | 12 | $\sim$ 30% |
| Total | 39 | |

Table 4.15. Toggle v2.0 Bug Prediction

| TOGGLE v2.0 | | |
|---|---|---|
| True | 29 | $\sim$ **53%** |
| False | 25 | $\sim$ 47% |
| Total | 54 | |

## 4.4 Correlation among metrics

The last step of this thesis is to find a possible correlation among the fragility metrics, the quality metrics, and the bug prediction of the test suites, thus it was decided to use the statistical technique of linear regression, applied multiple times for all metrics that were analyzed.

Linear regression is a linear approach useful for seeking a relationship between two variables by adapting a linear equation to the observed data. The first variable is considered to be of the independent type, while the other is of the dependent type.

When a simple linear regression is performed, several types of coefficients are computed, some of which will be useful in answering the correlation question of this thesis study:

- Estimates for the model parameters which indicate how much the value of the dependent variable increases when the independent variable increases by 1;
- The standard error (*Std. Error*) of the estimated values;
- The statistical test (*t value*);
- The *p value*, also known as $Pr(> |t|)$, i.e. the probability of finding the observed results when the *null hypothesis* of a study question is true;

In our case, the most important coefficient among the before mentioned coefficients is the *p value*, because if its value is less than 0.05 this means that the model fits well to the data and also that there is a linear correlation among metrics.

Tables below show the p values of all simple linear regressions applied to the test fragility metrics (tab. 4.16), the code quality metrics (tab. 4.17) and between fragility metrics and quality metrics (tab. 4.18). Furthermore, all the values below the threshold (0.05) have been highlighted in bold as they indicate correlations between metrics.

Table 4.16. Pr(>|t|) table of Fragility Metrics

| | NTC | TTL | TLR | MTLR | MRTL | MCR | MMR | MCMMR | NTR | MRR | TSV |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NTC | - | **4.2e-06** | **3.4e-05** | 0.27 | **8.8e-06** | 0.18 | 0.98 | 0.34 | 0.99 | **0.00073** | 0.29 |
| TTL | - | - | **0.0048** | 0.93 | **0.014** | 0.46 | 0.41 | **0.022** | 0.26 | **0.028** | 0.071 |
| TLR | - | - | - | **0.017** | **1.9e-10** | **0.005** | 0.91 | 0.26 | 0.80 | **1.2e-05** | 0.83 |
| MTLR | - | - | - | - | **0.028** | **1.5e-07** | 0.70 | 0.69 | 0.83 | 0.13 | 0.56 |
| MRTL | - | - | - | - | - | **0.0012** | 0.35 | **0.029** | 0.74 | **1e-08** | 0.84 |
| MCR | - | - | - | - | - | - | 0.32 | **0.021** | 0.72 | **0.00044** | 0.22 |
| MMR | - | - | - | - | - | - | - | **3.3e-06** | 0.92 | 0.14 | 0.59 |
| MCMMR | - | - | - | - | - | - | - | - | 0.59 | **0.0039** | 0.93 |
| NTR | - | - | - | - | - | - | - | - | - | 0.35 | 0.85 |
| MRR | - | - | - | - | - | - | - | - | - | - | 0.35 |
| TSV | - | - | - | - | - | - | - | - | - | - | - |

Table 4.17. Pr(>|t|) table of Quality Metrics

| | CC | WMC | CBO | Ca | RFC | DIT | NOC | LOC | NPM | TRUE | FALSE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CC | - | **0.00027** | 0.13 | 0.073 | **0.0021** | 0.69 | 0.2 | **6.5e-09** | **0.0053** | 0.5 | 0.5 |
| WMC | - | - | **0.0031** | 0.32 | **1.7e-05** | 0.74 | 0.58 | **3.1e-05** | **7.2e-11** | 0.61 | 0.61 |
| CBO | - | - | - | 0.18 | **2.1e-08** | 0.21 | 0.42 | **0.012** | **0.02** | 0.15 | 0.15 |
| Ca | - | - | - | - | 0.21 | 0.66 | **5e-14** | 0.19 | 0.64 | 0.43 | 0.43 |
| RFC | - | - | - | - | - | 0.33 | 0.54 | **1.1e-05** | **0.0017** | 0.24 | 0.24 |
| DIT | - | - | - | - | - | - | 0.75 | 0.48 | 0.91 | 0.81 | 0.81 |
| NOC | - | - | - | - | - | - | - | 0.41 | 0.49 | 0.19 | 0.19 |
| LOC | - | - | - | - | - | - | - | - | **2e-04** | 0.83 | 0.83 |
| NPM | - | - | - | - | - | - | - | - | - | 0.26 | 0.26 |
| TRUE | - | - | - | - | - | - | - | - | - | - | 0 |
| FALSE | - | - | - | - | - | - | - | - | - | - | - |

Table 4.18. Pr(>|t|) table of Fragility Metrics & Quality Metrics

| | CC | WMC | CBO | Ca | RFC | DIT | NOC | LOC | NPM | TRUE | FALSE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NTC | 0.99 | 0.071 | 0.30 | 0.92 | 0.14 | 0.29 | 0.88 | 0.19 | 0.19 | 0.36 | 0.36 |
| TTL | 0.68 | 0.16 | 0.31 | 0.95 | 0.47 | 0.36 | 0.90 | 0.36 | 0.34 | **0.049** | **0.049** |
| TLR | 1 | 0.5 | 0.35 | 0.94 | 0.47 | 0.25 | 0.92 | 0.71 | 0.71 | 0.91 | 0.91 |
| MTLR | 0.53 | 0.45 | 0.27 | 0.96 | 0.17 | 0.52 | 0.97 | 0.32 | 0.43 | 0.56 | 0.56 |
| MRTL | 0.34 | 0.46 | 0.25 | 0.24 | 0.36 | 0.2 | 0.37 | 0.9 | 0.39 | 0.96 | 0.96 |
| MCR | 0.2 | 0.87 | 0.63 | 0.091 | 0.67 | 0.14 | 0.12 | 0.52 | 0.50 | 0.4 | 0.4 |
| MMR | 0.41 | 0.56 | 0.80 | **0.05** | 0.81 | 0.71 | 0.051 | 0.76 | 0.22 | 0.18 | 0.18 |
| MCMMR | **0.0041** | 0.89 | 0.98 | **0.00061** | 0.55 | 0.57 | **0.0011** | 0.17 | 0.38 | **0.024** | **0.024** |
| NTR | 0.10 | 0.49 | 0.24 | 0.97 | 0.5 | 0.7 | 0.83 | **0.042** | 0.29 | 0.94 | 0.94 |
| MRR | 0.071 | 0.63 | 0.4 | 0.16 | 0.62 | 0.21 | 0.18 | 0.25 | 0.65 | 0.44 | 0.44 |
| TSV | 0.28 | 0.058 | 0.092 | 0.42 | **0.013** | 0.18 | 0.72 | **0.0087** | 0.076 | 0.65 | 0.65 |

## 4.5 Main Correlations

The table 4.16 shows the correlations among the fragility metrics whose values are in bold. Overall, there is a noticeable diffusion of correlations as was expected due to the nature of the metrics and their scope, in other words, each of these metrics describes a fragility of a test and some of them are dependent on the others. One clear example is the *'MRTL'* metric that depends on *'TLR'*, which in turn depends on *'TTL'*, and on *'NTC'* metrics.

The table 4.17 shows the correlations among the quality metrics whose values are in bold. Same as in the case of fragility metrics, also in this table, there is a significant presence of correlations because of the family of these metrics since they all describe the same phenomenon. For example, it is quite obvious that when the *'LOC'* metric varies also the *'CC'* does, i.e., the whole program complexity is

affected, and so on.

The table 4.18 shows how quality metrics and fragility metrics are very unrelated except for the presence of few correlations whose values are in bold. Those which were found relevant are set out below.

Firstly, the relation between the fragility metric *'TTL'* and the percentage of the total number of positive classes affected by bugs (*'TRUE'*), as shown in Fig. 4.2, is given by the straight line $y = -1307 + 11220x$ which means when the total test lines increases also the percentage of bugged classes increases. This is a reasonable result since test classes with many lines of code are more likely to have bugs.

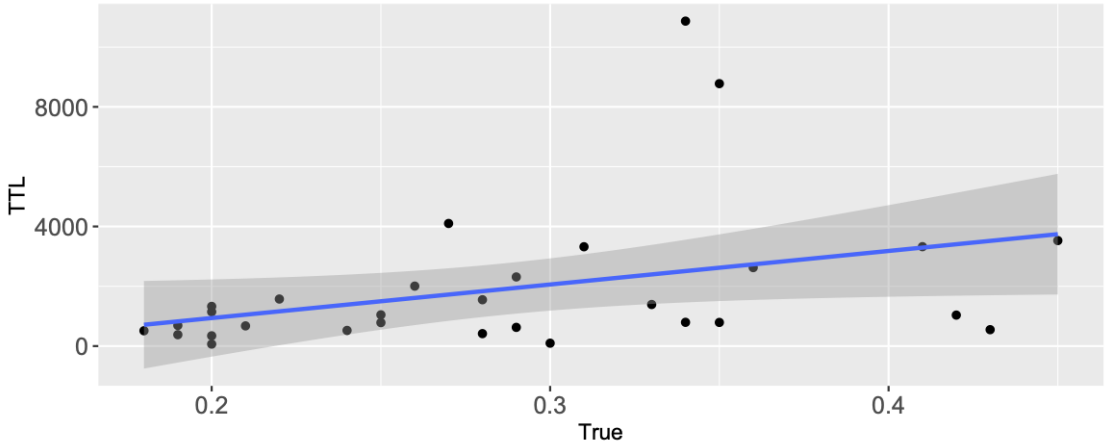


Figure 4.2. Scatter Plot linear regression between TTL and TRUE.

Secondly, the fragility metric which has shown more correlation with quality metrics is *'MCMMR'*, for example, the relation between it and the percentage of the total number of positive classes affected by bugs *'TRUE'* shown in Fig. 4.3, is represented by the straight line $y = -0.013 + 0.245x$ and it means when test classes and their methods need more changes those classes are more likely to be affected by bugs.

Furthermore, the correlation among *'MCMMR'* and *'CC'* (Fig. 4.4) is represented by the straight line $y = 0.032 + 0.004x$ and it can be explained as: classes with high cyclomatic complexity perform lots of method calls, hence have inside a lot of methods, and for this reason are more likely to be modified. All of this results in many class changes, and there are huge chances that many of those changed classes are tests, then a high *'MCMMR'* value.

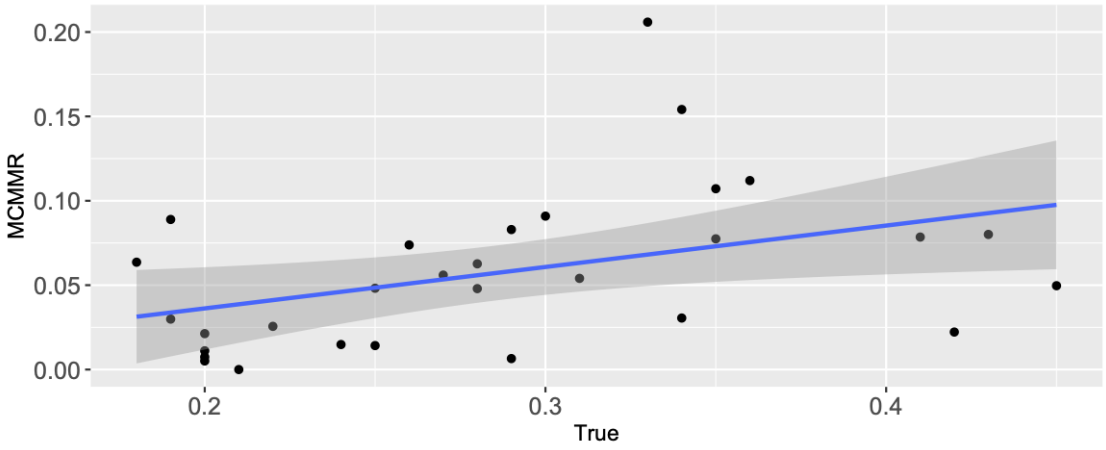


Figure 4.3. Scatter Plot linear regression between MCMMR and TRUE.

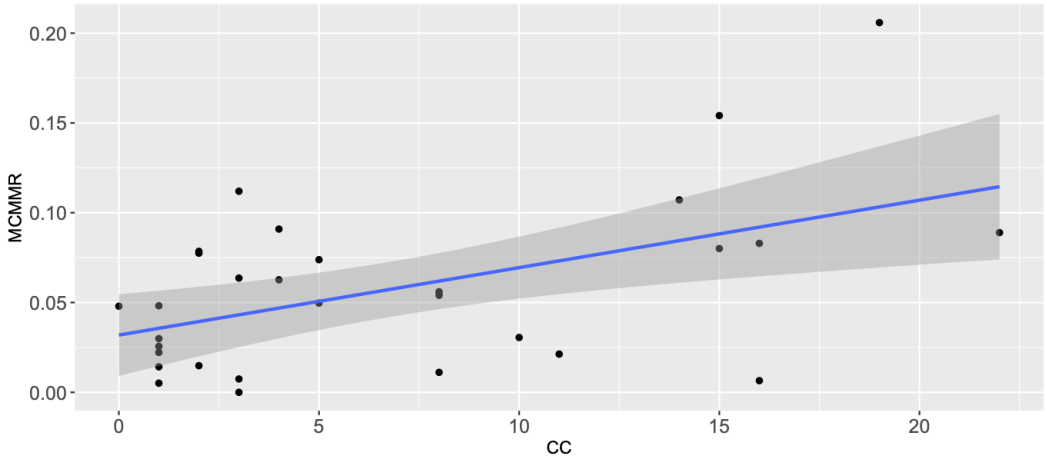


Figure 4.4. Scatter Plot linear regression between MCMMR and CC.

Finally, the correlations among *'MCMMR'*, *'Ca'* and *'NOC'*, can be easily explained, also considering the above explanation, a code that has a high number of couplings or sub-classes is classified as poor quality code and, for this reason, more changes-prone than a good quality code.

# Chapter 5

# Conclusions

Software maintainability is a significant software quality attribute that has always found a lot of interest in the academic world and, subsequently, also in the industrial world. Nevertheless, sometimes companies prefer to perform less maintenance to develop features whose business value is more relevant in the short term.

Software testing is a relevant activity belonging to Software maintainability because OO software systems have become larger and more complex. Hence, test efficiently and equally in all the classes has become extremely difficult, therefore, to simplify the test phase, was preferred the use of automated tests (scripts).

Test code, of course, is itself code written by a human being, is not immune to having bugs just like the code it is testing, and as a result, it requires interventions that bring us to the test fragility concept.

Thus, this thesis work aimed to find a possible correlation between the Bug Prediction and the test suites fragility. The experiment consisted of two phases: the first phase focused on the Bug Prediction, while inside the second one is discussed the correlation between the Bug Prediction and fragility metrics.

## 5.1 Results Summary

The results achieved were extensively covered in Chapter 4. In the first part, were analyzed 6 ML algorithms by applying some techniques in the training phase, such as percentage-split, k folds cross-validation, and the assignment of two different weights to the two classes (True/False). Percentage-split and k folds cross-validation returned almost the same results, hence, for readability reasons were

reported only the results of one technique. However, as shown in Table 4.12, the classifiers cannot acceptably distinguish the two classes, predicting the negative class more since the significant imbalance of the dataset (Fig. 4.1).

To overcome the imbalance, was used a cost matrix (table 3.1) minimizing the final cost of the classification (see Table 4.13), and there is a general improvement of the results, in particular, the Random Forest classifier is the one that most of all showed a clear improvement. In confirmation of these outcomes, was conduct a practical experiment in which were analyzed two versions of legit software, and the results highlighted how there is a decrease of about 20% of bugs in the second version.

In the second part, were compared the fragility metrics and the Bug Prediction results through Linear regression. The results (Tab 4.18) show no correlation between the two parts except for few cases, and only the most interesting ones are reported.

The correlation between the fragility metric 'TTL' and the percentage of the total number of positive classes affected by bugs ('TRUE'), as shown in Fig. 4.2, this is reasonable since test classes with many lines of code are more likely to have bugs.

The fragility metric that has more correlations is 'MCMMR', for instance, the correlation with the TRUE percentage, shown in Fig. 4.3, which means that when test classes need more changes, all the changes can cause the introduction of bugs. Finally, the correlations among 'MCMMR', 'CC', 'Ca', and 'NOC' can be easily explained as a code that has a high complexity or number of couplings or sub-classes is classified as poor quality code and, for this reason, more changes-prone than a good quality code.

## 5.2 Limitations

During the realization of this study, it has been necessary to make some choices that led to a resizing of the amount of work to make it more suitable for a thesis. Therefore, this has led to some limitations that can be summarized:

- *Generalization*: This study involved thirty Android projects belonging to a small set of projects already used in previous work, and all the projects come from GitHub. In conclusion, it is not possible to establish if can be obtained

better results by using a larger number of projects, or if these results are still valid if we consider no longer Android applications but applications belonging to other software families.

- *Structural*: To measure the code quality has been used only nine static metrics, i.e., the ones that turned out to be the most widespread in the field. The fragility metrics considered in this work have been presented in literature only recently, hence, it is not possible to deem in advance that if the experiment is repeated in the future, if anything, with more metrics or different ones we will get different results.

## 5.3 Future work

In this paragraph, we list some improvements to remove or at least reduce the limitations expressed in the previous section to refine this study and generalize it:

- Consider extending this work to multiple projects to see if there is an improvement in the results, afterward, try applying it to applications from different families, and verify whether the results are more satisfactory than those obtained so far.
- It can be interesting to use more quality metrics to see if the results improve and conduct an in-depth analysis on the fragility metrics to understand how they interact with quality metrics to exploit them proficiently.

# References

Abreu, F. B., & Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality* (Vol. 186).

Abreu, F. B., & Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd international software metrics symposium* (pp. 90–99).

Amannejad, Y., Garousi, V., Irving, R., & Sahaf, Z. (2014). A search-based approach for cost-effective software test automation decision support and an industrial case study. In *2014 ieee seventh international conference on software testing, verification and validation workshops* (pp. 302–311).

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, *28*(1), 4–17.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 56–65).

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Bennett, K. (1990). The software maintenance of large software systems: management, methods and tools. In *Software engineering for large software systems* (pp. 1–26). Springer.

Bibi, S., Tsoumakas, G., Stamelos, I., & Vlahavas, I. P. (2006). Software defect prediction using regression via classification. In *Aiccsa* (pp. 330–336).

Binder, R. (2000). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on software engineering* (pp. 592–605).

Bowes, D., Hall, T., & Petrić, J. (2018). Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, *26*(2), 525–552.

Breugelmans, M., & Van Rompaey, B. (2008). Testq: Exploring structural and maintenance characteristics of unit test suites. In *Wasdett-1: 1st international workshop on advanced software development tools and techniques.*

Buhmann, M. D. (2000). Radial basis functions. *Acta numerica*, *9*, 1–38.

Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, *179*(8), 1040–1058.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Choudhary, S. R., Zhao, D., Versee, H., & Orso, A. (2011). Water: Web application test repair. In *Proceedings of the first international workshop on end-to-end test script engineering* (pp. 24–29).

Coleman, R. (2018). Beauty and maintainability of code. In *2018 international conference on computational science and computational intelligence (csci)* (pp. 825–828).

Cook, M. L. (1982). Software metrics: an introduction and annotated bibliography. *ACM SIGSOFT Software Engineering Notes*, *7*(2), 41–60.

Cooper, G. F., & Herskovits, E. (1992). A bayesian method for the induction of probabilistic networks from data. *Machine learning*, *9*(4), 309–347.

Coppola, R., Ardito, L., Torchiano, M., & Alégroth, E. (2021). Translation from layout-based to visual android test scripts: An empirical evaluation. *Journal of Systems and Software*, *171*, 110845.

Coppola, R., Morisio, M., Torchiano, M., & Ardito, L. (2019). Scripted gui testing of android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering*, *24*(5), 3205–3248.

Cunningham, W. (2006, 07). *Bugs In The Tests.* Retrieved from `http://wiki.c2.com/?BugsInTheTests`

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *2010 7th ieee working conference on mining software repositories (msr 2010)* (pp. 31–41).

Daniel, B., Gvero, T., & Marinov, D. (2010). On test repair using symbolic execution. In *Proceedings of the 19th international symposium on software testing and analysis* (pp. 207–218).

Di Nucci, D., Palomba, F., Oliveto, R., & De Lucia, A. (2017). Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *1*(3), 202–212.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., & Gyimóthy, T. (2020). A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 1–60.

Fowler, M., Beck, K., Brant, J., & Opdyke, W. (1999). *Refactoring: improving the design of existing code. (1st ed.)*. Addison-Wesley Professional.

Garousi, V., & Felderer, M. (2016). Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, *33*(3), 68–75.

Garousi, V., & Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, *138*, 52–81.

Ghotra, B., McIntosh, S., & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 1, pp. 789–800).

Giger, E., D'Ambros, M., Pinzger, M., & Gall, H. C. (2012). Method-level bug prediction. In *Proceedings of the 2012 acm-ieee international symposium on empirical software engineering and measurement* (pp. 171–180).

Gill, N. S., & Grover, P. (2003). Component-based measurement: few useful guidelines. *ACM SIGSOFT Software Engineering Notes*, *28*(6), 4–4.

Google. (n.d.). *The build process of a typical Android app module.* Retrieved from `https://developer.android.com/images/tools/studio/build-process_2x.png` ([Online; accessed June 04, 2021])

Greiler, M., Van Deursen, A., & Storey, M.-A. (2013). Automated detection of test fixture strategies and smells. In *2013 ieee sixth international conference on software testing, verification and validation* (pp. 322–331).

Hall, G. A., & Munson, J. C. (2000). Software evolution: code delta and code churn. *Journal of Systems and Software*, *54*(2), 111–118.

Hall, T., Zhang, M., Bowes, D., & Sun, Y. (2014). Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(4), 1–39.

Halstead, M. H. (1977). *Elements of software science* (Vol. 7). Elsevier New York.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In

*2009 ieee 31st international conference on software engineering* (pp. 78–88).

Hata, H., Mizuno, O., & Kikuno, T. (2012). Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (icse)* (pp. 200–210).

He, Z., Shu, F., Yang, Y., Li, M., & Wang, Q. (2012). An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, *19*(2), 167–199.

Hegedus, P. (2013). Revealing the effect of coding practices on software maintainability. In *2013 ieee international conference on software maintenance* (pp. 578–581).

Jiang, Y., Cukic, B., & Menzies, T. (2007). Fault prediction using early lifecycle data. In *The 18th ieee international symposium on software reliability (issre'07)* (pp. 237–246).

Jin, C., & Liu, J.-A. (2010). Applications of support vector mathine and unsupervised learning for predicting maintainability using object-oriented metrics. In *2010 second international conference on multimedia and information technology* (Vol. 1, pp. 24–27).

Kaszycki, G. (1999). Using process metrics to enhance software fault prediction models. In *Tenth international symposium on software reliability engineering, boca raton, florida.*

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, *17*(3), 243–275.

Khoshgoftaar, T. M., Gao, K., & Szabo, R. M. (2001). An application of zero-inflated poisson regression for software fault prediction. In *Proceedings 12th international symposium on software reliability engineering* (pp. 66–73).

Koru, A. G., & Liu, H. (2005). An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the 2005 workshop on predictor models in software engineering* (pp. 1–5).

Lacoste, F. J. (2009). Killing the gatekeeper: Introducing a continuous integration system. In *2009 agile conference* (pp. 387–392).

Lehman, M. M. (1984). Program evolution. *Information Processing & Management*, *20*(1-2), 19–36.

Lientz, B. P., & Swanson, E. B. (1980). *Software maintenance management.* Addison-Wesley Longman Publishing Co., Inc.

Liu, Y., Khoshgoftaar, T. M., & Seliya, N. (2010). Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Transactions on Software Engineering*, *36*(6), 852–864.

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

Lu, S., Park, S., Seo, E., & Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on architectural support for programming languages and operating systems* (pp. 329–339).

Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 643–653).

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, *27*, 504–518.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *20th ieee international conference on software maintenance, 2004. proceedings.* (pp. 350–359).

Mathur, A. P. (2013). *Foundations of software testing, 2/e*. Pearson Education India.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*(4), 308–320.

McConnell, S. (1993). Code complete microsoft press. *Redmond, Washington, USA*.

Melski, E. (2013, Jan). *unit tests.* Retrieved from `https://blog.melski.net/tag/unit-tests/`

Mende, T. (2010). Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th international conference on predictive models in software engineering* (pp. 1–10).

Mende, T., & Koschke, R. (2009). Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th international conference on predictor models in software engineering* (pp. 1–10).

Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., ... Zimmermann, T. (2012). Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, *39*(6), 822–834.

Menzies, T., Greenwald, J., & Frank, A. (2006). Data mining static code attributes

to learn defect predictors. *IEEE transactions on software engineering*, *33*(1), 2–13.

Meszaros, G. (2007). *xunit test patterns: Refactoring test code*. Pearson Education.

Mirzaaghaei, M., Pastore, F., & Pezze, M. (2010). Automatically repairing test cases for evolving method declarations. In *2010 ieee international conference on software maintenance* (pp. 1–5).

Moha, N., Gueheneuc, Y.-G., Duchien, L., & Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*(1), 20–36.

Murphy, G., & Cubranic, D. (2004). Automatic bug triage using text categorization. In *Proceedings of the sixteenth international conference on software engineering & knowledge engineering* (pp. 1–6).

Nayrolles, M., & Hamou-Lhadj, A. (2018). Towards a classification of bugs to facilitate software maintainability tasks. In *Proceedings of the 1st international workshop on software qualities and their dependencies* (pp. 25–32).

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, *23*(3), 1188–1221.

Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R., & De Lucia, A. (2016). On the diffusion of test smells in automatically generated test code: An empirical study. In *2016 ieee/acm 9th international workshop on search-based software testing (sbst)* (pp. 5–14).

Palomba, F., Zaidman, A., & De Lucia, A. (2018). Automatic test smell detection using information retrieval techniques. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 311–322).

Panichella, A., Oliveto, R., & De Lucia, A. (2014). Cross-project defect prediction models: L'union fait la force. In *2014 software evolution week-ieee conference on software maintenance, reengineering, and reverse engineering (csmr-wcre)* (pp. 164–173).

Petrić, J., Bowes, D., Hall, T., Christianson, B., & Baddoo, N. (2016). Building an ensemble for software defect prediction based on diversity selection. In *Proceedings of the 10th acm/ieee international symposium on empirical software engineering and measurement* (pp. 1–10).

Pinto, L. S., Sinha, S., & Orso, A. (2012). Understanding myths and realities

of test-suite evolution. In *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering* (pp. 1–11).

Pirrigheddu, S. (2021). *Analisi della correlazione tra la qualità del codice e la fragilità del test = analysis of the correlation between code quality and test fragility* (Unpublished doctoral dissertation). Politecnico di Torino.

Premraj, R., & Herzig, K. (2011). Network versus code metrics to predict defects: A replication study. In *2011 international symposium on empirical software engineering and measurement* (pp. 215–224).

Puranik, S., Deshpande, P., & Chandrasekaran, K. (2016). A novel machine learning approach for bug prediction. *Procedia Computer Science*, *93*, 924–930.

Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., & Binkley, D. (2011). Scotch: Test-to-code traceability using slicing and conceptual coupling. In *2011 27th ieee international conference on software maintenance (icsm)* (pp. 63–72).

Rahman, F., Posnett, D., & Devanbu, P. (2012). Recalling the" imprecision" of cross-project defect prediction. In *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering* (pp. 1–11).

Reichhart, S., Gîrba, T., & Ducasse, S. (2007). Rule-based assessment of test quality. *J. Object Technol.*, *6*(9), 231–251.

Rokach, L. (2010). Ensemble-based classifiers. *Artificial intelligence review*, *33*(1-2), 1–39.

Sayyad Shirabad, J., & Menzies, T. (2005). *The PROMISE Repository of Software Engineering Databases.* School of Information Technology and Engineering, University of Ottawa, Canada. Retrieved from `http://promise.site.uottawa.ca/SERepository`

Shivaji, S., Whitehead Jr, E. J., Akella, R., & Kim, S. (2009). Reducing features to improve bug prediction. In *2009 ieee/acm international conference on automated software engineering* (pp. 600–604).

Song, Q., Shepperd, M., Cartwright, M., & Mair, C. (2006). Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, *32*(2), 69–82.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 1–12).

Spinellis, D. (2005). Tool writing: a forgotten art?(software tools). *IEEE Software*,

*22*(4), 9–11.

Subbiah, U., Ramachandran, M., & Mahmood, Z. (2019). Software engineering approach to bug prediction models using machine learning as a service (mlaas). In *Icsoft 2018-proceedings of the 13th international conference on software technologies* (pp. 879–887).

Taba, S. E. S., Khomh, F., Zou, Y., Hassan, A. E., & Nagappan, M. (2013). Predicting bugs using antipatterns. In *2013 ieee international conference on software maintenance* (pp. 270–279).

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016). Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th international conference on software engineering* (pp. 321–332).

Tosun, A., Bener, A., Turhan, B., & Menzies, T. (2010). Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology*, *52*(11), 1242–1257.

Tosun, A., Turhan, B., & Bener, A. (2009). Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th international conference on predictor models in software engineering* (pp. 1–9).

Tóth, Z., Gyimesi, P., & Ferenc, R. (2016). A public bug database of github projects and its application in bug prediction. In *International conference on computational science and its applications* (pp. 625–638).

Toure, F., Badri, M., & Lamontagne, L. (2018). Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software. *Innovations in Systems and Software Engineering*, *14*(1), 15–46.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st ieee/acm international conference on automated software engineering* (pp. 4–15).

Ubayawardana, G. M., & Karunaratna, D. D. (2018). Bug prediction model using code smells. In *2018 18th international conference on advances in ict for emerging regions (icter)* (pp. 70–77).

Vahabzadeh, A., Fard, A. M., & Mesbah, A. (2015). An empirical study of bugs

in test code. In *2015 ieee international conference on software maintenance and evolution (icsme)* (pp. 101–110).

Van Deursen, A., & Moonen, L. (2002). The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd int'l conf. extreme programming and flexible processes in software engineering* (pp. 71–76).

Van Deursen, A., Moonen, L., Van Den Bergh, A., & Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (xp)* (pp. 92–95).

Van Rompaey, B., Du Bois, B., Demeyer, S., & Rieger, M. (2007). On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, *33*(12), 800–817.

Vans, A. M., von Mayrhauser, A., & Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, *51*(1), 31–70.

Yamashita, A. (2014). Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, *19*(4), 1111–1143.

Yao, Y., Xiao, Z., Wang, B., Viswanath, B., Zheng, H., & Zhao, B. Y. (2017). Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 internet measurement conference* (pp. 384–397).

Yusifoğlu, V. G., Amannejad, Y., & Can, A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, *58*, 123–147.

Yusop, O. M., & Ibrahim, S. (2011). Evaluating software maintenance testing approaches to support test case evolution. *International Journal on New Computer Architectures and Their Applications (IJNCAA)*, *1*(1), 74–83.

Zaidman, A., Van Rompaey, B., van Deursen, A., & Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, *16*(3), 325–364.

Zimmermann, T., & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on software engineering* (pp. 531–540).

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction: a large scale experiment on data vs. domain vs.

process. In *Proceedings of the 7th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering* (pp. 91–100).

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting defects for eclipse. In *Third international workshop on predictor models in software engineering (promise'07: Icse workshops 2007)* (pp. 9–9).