Universiteit Antwerpen

ANALYZING MUTANT DENSITY AS A COMPLEMENTARY METRIC TO CYCLOMATIC COMPLEXITY FOR FINDING REFACTORING OPPORTUNITIES

Senne Rosaer

Principal Advisor: Serge Demeyer
Assistant Advisor: Ali Parsai

June 4, 2023

AnSyMo
Antwerp Systems & Software Modelling
University of Antwerp

Contents

List of Figures

List of Tables

Abstract

The complexity of code is a research field that has existed for multiple decades and has a large amount of existing work. Despite the large amount of existing work, it still remains a very active field due to its lasting importance in the current DevOps era. Within the context of DevOps, maintainability is an extremely important term that is supported by testability for example. Certain complexity metrics have the ability to be related to testing effort needed which is why it remains relevant. One of the most researched metrics for estimating this effort is cyclomatic complexity. It has a long history of usage, even in the current DevOps tooling, despite there being critique on the metric on different aspects. This work tries to contribute to finding refactoring opportunities related to testing effort by working with a complementary metric for cyclomatic complexity to provide additional opportunities and a failsafe for false positives. This is done by positioning mutant density as a complementary metric, which measures code complexity based on the principles of mutation testing. Mutant density utilizes a flexible set of mutation operators to measure the amount of mutants which is used to define complexity. This different approach, which is still related to testing effort due to the principles it is based upon, provides a potential complementary metric which can expand on the need for finding refactoring opportunities. The contributions could be divided in these aspects: 1) Defining the correlation between cyclomatic complexity and mutant density, 2) Analyzing different metrics utilizing mutant density and cyclomatic complexity that provide deeper insight in code complexity and refactoring opportunities , and 3) Investigating the influence of the modular aspect of mutant density by selecting different mutation operators.

Nederlandstalige Samenvatting

De complexiteit van code is een onderzoeksveld dat al meerdere decennia bestaat en dat ook een grote hoeveelheid gepubliceerde werken rond het onderwerp heeft. Ondanks de lange geschiedenis van dit onderzoeksveld, is er nog steeds zeer actief onderzoek naar dit onderwerp vanwege het blijvende belang in het huidige DevOps tijdperk. Binnen de context van DevOps is maintainability een zeer belangrijke term that onder andere ondersteund wordt door testbaarheid. Bepaalde complexiteit metrieken hebben de de capaciteiten om gerelateerd te zijn aan de moeite die in testen gestoken moet worden, dit is een van de redenen voor de blijvende relevantie. Een van de meest onderzochte metrieken voor deze moeite te meten is cyclomatische complexiteit. Het heeft een lange geschiedenis van gebruik, zelfs in de huidige DevOps tooling ondanks het feit dat er op sommige vlakken kritiek is tegen deze metriek. Dit werk probeert een bijdrage te leveren aan het vinden refactoring opportuniteiten die gerelateerd zijn aan de moeite voor het testen door te werken met een complementaire metriek. Deze complementaire metriek gaat cyclomatische complexiteit ondersteunen in het vinden van nieuwe refactoring opportuniteiten en het vermijden van false positives. Dit gebeurt door mutatie densiteit te positioneren als een complementair metriek, deze meet code complexiteit aan de hand van principes die origineel uit mutatie testen komen. Mutatie densiteit gebruikt een flexibele verzameling van mutatie operatoren om te meten hoeveel mogelijke mutaties er aanwezig zijn in code, dit aantal wordt gebruikt om de complexiteit te meten. Deze aanpak verschilt zeer hard van cyclomatische complexiteit maar is nog steeds gerelateerd aan de moeite om te testen vanwege de oorsprong uit mutatie testen. Dit geeft een potentieel complementair metriek die voor uitbreiding kan zorgen aan de nood voor refactoring opportuniteiten. De bijdragen van dit werk kunnen onderverdeeld worden in deze aspecten: 1) Definiëren van de correlatie tussen cyclomatische complexiteit en mutatie densiteit, 2) Analyze van verschillende metrieken die gebruik maken van mutatie densiteit en cyclomatische complexiteit om dieper inzicht te krijgen in refactoring opportuniteiten, en 3) Onderzoek naar de invloed van het modulaire aspect van mutatie densiteit door de selectie van verschillende mutatie operatoren.

Acknowledgments

Student's personal acknowledgements.

1   Introduction

Software development tries to create software that is not only functional but also includes a factor of maintainability and adaptability. The landscape of software development on its own is ever-evolving, causing applications to have a constant growth in complexity. This increases the difficulty surrounding maintenance in the face of changing requirements, technological advancements and the constant pursuit of improvement. Software maintainability therefore emerges as a critical factor in achieving long-term success. It encompasses the ease with which a software system can be changed, updated and expanded upon without causing unwanted ripples through the existing code or influencing the stability.

At the core of maintaining software lies the practice of refactoring, a specific approach to improve the internal structure and design of code without changing the external behavior. This is a key factor to supporting the concept of maintainable code [1], enabling developers to correct flaws, reduce technical debt and optimizing performance. Being able to identify refactoring opportunities is a fundamental functionality needed to support this.

Finding refactoring opportunities is widely supported in the current DevOps era where there is much more emphasis on automation of quality assurance [2]. Popular industrial tooling such as CodeScene [3] and Sonarqube [4] all support different aspects of automated quality assurance. The given examples each utilize code complexity metrics to determine refactoring opportunities, providing simpler ways of increasing and managing maintainability.

The actionability of these tools lies within their own specific recommendations for code improvements. SonarQube, for example, identifies code smells, potential bugs, refactoring targets and many others. Codescene similarly offers insights by analyzing code complexity, coupling, change patterns and also refactoring targets. Despite these tools being adapted in many different industries, acknowledging the potential false positives that can occur in such tooling is very important [5, 6]. This could occur by putting too much weight on a singular complexity metric to define refactoring targets, causing suggested improvements that may not be relevant. The relevance of suggestions is however also dependent on the context of the project itself. Complete avoidance of false positives is difficult but combining complementary metrics can reduce the probability.

Within the DevOps context, testing is a critical component in ensuring the quality, reliability and the evolvability of a software product [7]. There are many rapid and frequent deployments with DevOps which makes testing even more crucial. It functions as a safety net that aids in capturing bugs and issues as they emerge during the development process. By integrating testing into the DevOps pipelines, it is possible to automate various testing processes such as unit testing, integration testing, performance testing and security testing. This can also be seen by the testing coverage inclusion in tools such as the previously mentioned Sonarqube. On top of that, testing also provides a certain sense of confidence in the stability and robustness of the software. Based on these factors and within the context of DevOps, it is very interesting to study refactoring opportunities from the perspective of complexity metrics which estimate testing efforts. Using complexity metrics to refactor methods with the goal of simplifying the testing process, while the testing process has so much importance, provides a very interesting field of study.

One of the more known metrics which is able to estimate testing effort is cyclomatic complexity [8]. This metric is closely linked to the testing effort since the cyclomatic complexity can be seen as the minimum amount of tests needed to create full coverage, therefore directly representing the difficulty of testing [9]. Cyclomatic complexity does have a relatively large history of critique on a few different aspects, however most emphasis for this work is placed upon how well cyclomatic complexity measures maintainability and not the direct critique. Current measures for calculating maintainability still frequently incorporate cyclomatic complexity [10, 11].

Placing too much emphasis on cyclomatic complexity alone could lead back to the principle of false positives or obtaining incomplete information. A possible method to avoid this is utilizing a complementary metric. This work will investigate how well mutant density [12] can be used as the complementary metric to provide additional insight into refactoring opportunities. Mutant density is a complexity metric which also estimates testing effort based on an approach derived from mutation testing. It counts the amount of potential mutations in a method as the complexity. Doing this also allows some modularity by choosing the mutation operators used to create mutants. Mutant density will not only be positioned as a failsafe for false positives but also as a metric capable of finding refactoring opportunities which are unavailable when only using cyclomatic complexity.

The corresponding research questions behind this general claim can be formulated as:

- RQ1. Correlation - what is the correlation between cyclomatic complexity and mutant density?

  Investigating the correlation has its importance on a few different levels. An extremely strong correlation between the two metrics would cause redundancy for one of them. Since mutant density has a higher computational complexity, this metric would be more easily seen as redundant. Claiming that these metrics can be used in a complementary manner also needs a certain amount of correlation. A very low correlation would

make this impossible since they would not be able to verify each other while also providing new refactoring opportunities.

- RQ2. Complementary - how can we utilize mutant density to provide additional insight into refactoring opportunities? Is there anecdotal evidence supporting this?

    This research question essentially has two main reasons why it's necessary. Firstly, it needs to be defined how these two metrics are to be used to obtain new insights. There are different ways possible of combining these metrics so finding the difference between these combinations is important. Secondly, it needs to be defined how the complementary claim can be investigated and how the results can be obtained surrounding this.

- RQ3. Fine Tuning - what is the impact of fine-tuning mutant density on correlation and complementary?

    Since mutant density has an aspect of modularity based on choosing different mutation operators, creating variations based on this can influence the results. It is therefore interesting to investigate whether this could positively impact the ability of finding refactoring opportunities or support the complementary aspect.

The rest of this work will be structured as follows. In Section 2, there will be some background information on the most relevant metrics used in the experiments and about the tooling for finding refactoring opportunities. The main methods used in this work will be described in Section 3. This is followed by a showcase of the results combined with extensive discussion in Section 4. After this the threats to validity are discussed in Section 5. An overview of the related work is shown in Section 6. Finally the conclusions and potential work is given in Section 7.

2 Background

Due to the emphasis mostly being placed on code complexity metrics which also relate to the testing effort, this section will dedicate itself to covering the most important metrics in question for this work. There will also be some coverage of Sonarqube and Codescene on the aspects that are utilized within this work. Since this work emphasises the detection of refactoring opportunities, some fixed code examples will be used. This also allows for clear visualisation and contextualization of the issues this work addresses. These two Java functions will be used for the examples, the exact reasoning behind choosing these will become clear in further subsections. Both of these examples are obtained from the Spring library through methods used in this work.

```java
private static int nextPowerOf2(int val) {
    val--;
    val = (val >> 1) | val;
    val = (val >> 2) | val;
    val = (val >> 4) | val;
    val = (val >> 8) | val;
    val = (val >> 16) | val;
    val++;
    return val;
}
```

```java
private boolean isSockJsSpecialChar(char ch) {
    return (ch <= '\u001F') || (ch >= '\u200C' && ch <= '\u200F') ||
            (ch >= '\u2028' && ch <= '\u202F') ||
            (ch >= '\u2060' && ch <= '\u206F') ||
            (ch >= '\uFFF0') || (ch >= '\uD800' && ch <= '\uDFFF');
    }
```

2.1 Source lines of code

Source lines of code is potentially one of the oldest complexity metrics. Originating from the time when FORTRAN and assembly where the most commonly used languages. At that time the development was also done through the usage of punched cards, where a single card often represented a single line of code. Therefore, measuring the complexity based on this was an obvious conclusion.

This metric can be divided in two general types: physical SLOC (LOC) and logical SLOC (LLOC) [13]. The exact definitions might vary but the core concept remains the same. Physical SLOC counts all the lines in the source code without including the commented lines. Logical SLOC tries to measure based on the amount of executable statements, their specific definitions are however coupled to the programming languages.

In essence, this metric is very simple. The counting can be easily automated and the metric is very intuitive because it can be seen directly. Being one of the oldest metrics, it also has a lot of data available. Within this work this metric will be used alongside cyclomatic complexity and mutant density to capture the influence of function sizing. The influence of this metric on testing effort is not directly the most emphasised aspect. Still, functions with larger amount of lines tend to be more difficult to test. This aspect will be discussed further when discussing correlation between the metrics. One of the downsides to source lines of codes is effect of different programming languages on the same functionality.
  The difference over languages can be clearly seen in Fig. 1, where the same functionality over different languages drastically changes the lines of code. Although it will be argued later on that the effect of this is limited in this specific use-case, it remains relevant to keep in mind.

BASIC	C	COBOL
PRINT "hello, world"	```#include <stdio.h>

int main() {
    printf("hello, world\n");
}``` | ```identification division.
program-id. hello .
procedure division.
display "hello, world"
goback .
end program hello .``` |
| Lines of code: 1 (no whitespace) | Lines of code: 3 (excluding whitespace) | Lines of code: 6 (excluding whitespace) |

Figure 1: The effect of different languages on source lines of code computation when implementing the same functionality

2.2 Cyclomatic complexity

Cyclomatic complexity is one of the most known complexity metrics and it will have a very important role throughout this whole work. Originally introduced in 1976 by Thomas J. McCabe [8] it has since seen quite some usage. Even industrial tooling frequently relies on this metric for complexity measurement [3, 4, 14].
The way cyclomatic complexity works is based on the measurement of the control flow of a program. This means the metric counts the amount of unique and independent paths through a program's source code. The formula for the calculation is as follows:

$$V(G) = E - N + 2$$

where V(G) is the cyclomatic complexity, E the amount of edges in the control flow graph, and N the number of nodes in the graph. Since each program has a so called "entry" and "exit" points, the formula includes a "+2" factor.

```java
private boolean example1(int b, int c) {
    int a = 10;
    if ( b > c ) {
        a = b;
    } else {
        a = c
    }
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
}
```

If we visualize this *example1* function as a flow graph we obtain this:
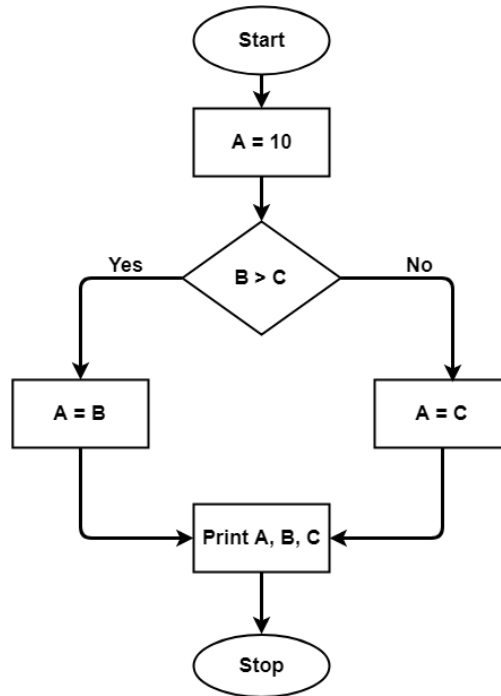
Figure 2: Flow graph representing the independent paths through function1

If we apply the previous formula we would obtain a cyclomatic complexity of two for that function.

In general there are multiple different statements that can increase the cyclomatic complexity.

- Conditional statements: **if**, **else if**, **switch**
- Loops: **for**, **while**, **do-while**
- Boolean operators: **&&**, ||, **!**
- Exception handling: **try**, **catch**, **except**

From this list, there are a few key takeaways. Not every single one of these statements or operators are available for every single programming language. This might cause small variation in cyclomatic complexity for the same functionality in different languages. Still, compared to how this influences source lines of code, the difference in this case is a lot smaller.

One of the advantages that something like cyclomatic complexity has over source lines of codes is the closer relation with the exact functionality. Since it does so by using statements that provide clear division in methods, it looks and feels very intuitive. This is also one of the reasons it is still used so frequently.

When looking back at the relationship between complexity and testing, the premise is that you need a single test for each independent path through a method. Therefore, cyclomatic complexity would dictate the amount of tests needed. This is not wrong, but it ignores one part of testing. The difficulty of writing tests is not always the amount of tests needed, but also how difficult it is to test all of the functionality. This is something that cyclomatic complexity does not directly cover.

There is also categorisation or threshold as to how cyclomatic complexity should be interpreted.

- 1 - 10 Simple procedure, little risk
- 11 - 20 More complex, moderate risk
- 21 - 50 Complex, high risk
- > 50 Untestable code, very high risk

This categorisation is not fixed however. In NIST [9] it is mentioned that there is sufficient support for this categorisation for general application but that variation might still be possible and is not necessarily bad depending on the provided context. They do however, never specify any real indication on why this categorisation works and which studies back these findings. Despite this, the threshold of 10 is often used as point where refactoring or splitting the method is appropriate. Since thresholding on its own is a core aspect of defining refactoring opportunities, this remains highly relevant.

2.3   Mutant Density

Originally complexity metrics are not tied to different defect types. Mutant density is introduced as a complexity metric that pinpoints statements more likely to be at risk of a fault. As such, creating a metric for increasing the maintainability of the software. The base concept of this already exists within the field of mutation testing.

Mutation testing injects intentional faults in the code and runs these mistakes through the tests to verify if the tests detect the introduced mistakes. Every injected mistake creates another mutant. The exact mutations created depends on the types of faults that are injected, so there is a certain amount of variation within this. These types of faults are called the mutation operators and allow for fine tuning the usage of mutant density.

The general workflow of mutation testing can be seen in Fig. 3
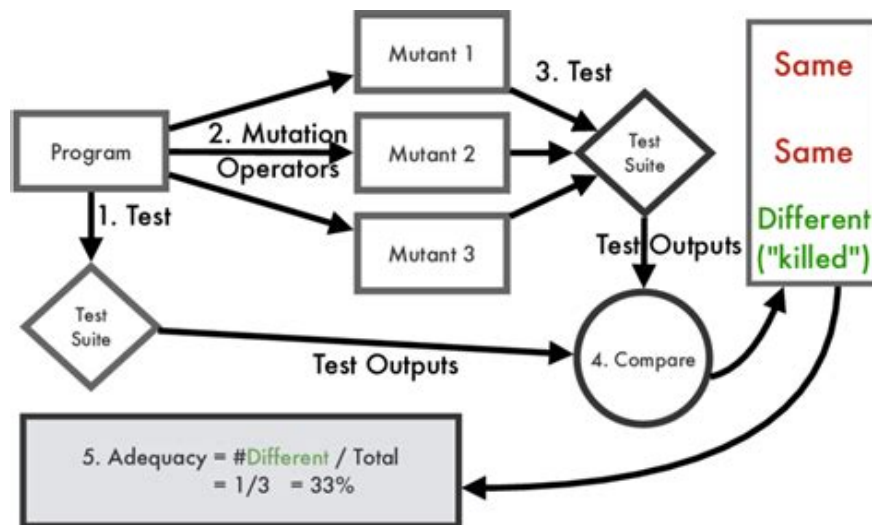
Figure 3: General workflow of mutation testing

Mutation testing was originally introduced in 1975 in the work "The design of a prototype mutation system for program testing" [15]. Its a technique provided for the evaluation of the effectiveness of a software testing suite. This is, as previously explained, done by measuring the ability to detect bugs and faults in the program. The work argues that mutation testing could provide a more accurate measurement of the effectiveness of tests. Especially in comparison to code coverage analysis because this metric is can easily provide wrong insights. Mutation testing functions very differently and also directly measured the ability of the test suite to detect faults.
This is the concept that is used as a foundation for mutant density, which is the amount of mutants that can be generated for each line of code. The average mutant density for a file is thus defined by the amount of mutants divided by the number of relevant lines. Comments and whitespaces are ignored, just like source lines of code.

Lets take the previously described *nextPowerOf2*, this function has a larger mutant density of 12 due to its many operations. Based on cyclomatic complexity alone, it would be placed below the threshold of 10 and not detected as something to worry about. The discrepancy between those two and simply looking at the functions show clear signs of potential improvement.

2.3.1 Mutation operators

Before wrapping up this section, a recap will be given on potential mutant operators. This is necessary since it is a part of the modularity of mutant density. It should be noted that there is also variation possible between the types of mutations for different languages. Currently the focus will not be on addressing this but more on the available types of mutation and what they represent.

- Relational Operator Replacement (ROR): replaces a single operand with another operand, these include $<, <=, >, >=, ==, ! =, true, false$. Depending on the implementation this could mean $x < y$ can be mutated to: $x <= y, x! = y, false$. Although more mutations are possible, **Improving logic-based testing** proved that these three cover the others.
- Arithmetic Operator Replacement (AOR): replaces a single arithmetic operator with another operand. these include $+, -, *, /, \%$
- Logical Connector Replacement (LCR): replaces logical operand with the inverse, these include $||, \&\&$
- Constant Replacement (CR): replaces literal values with other constants.

This only showcases the most basic examples, it should also be clear that not every mutation can be applied to every language in the same way. Take python for example, when looking at LCR, python also has the option to replace this and/or. The differences can also be larger than this, depending on the software, mutations may also be applied to more specific statements such as $try, except$ or $switchcases$. Not every language includes this thus the mutations varies quite a lot based on this. By using different mutation operators, mutant density can go extremely in depth in its modularity. Some portions will be covered in later sections but a full analysis on this is outside the scope of this work.

2.4 DevOps tools

This section will provide an overview of the tooling used such as: CodeScene and Sonarqube.

2.4.1 CodeScene

Within the interface of CodeScene, the first step is connecting through the chosen git platform where your project is located. A project can be selected and there will be automatic analysis done after the selection. The functionality provided by CodeScene is broad so the scope is limited to functionality relevant to this specific work. On the main
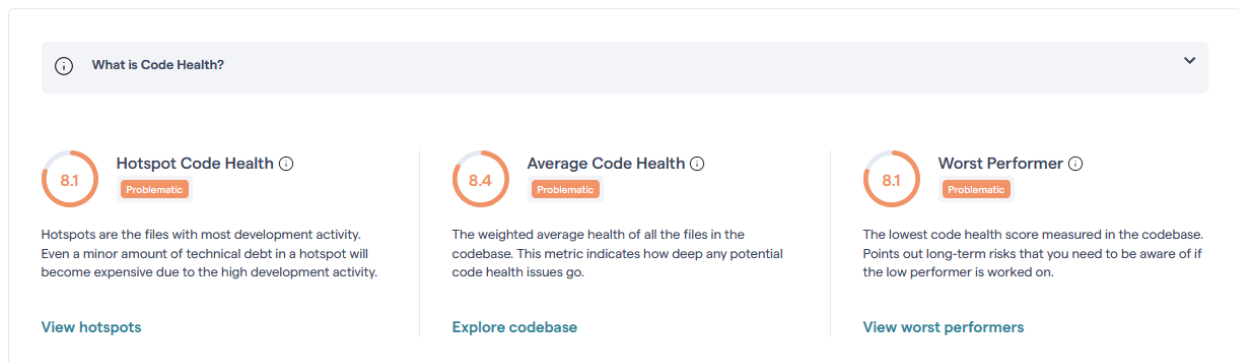
Figure 4: CodeScene main screen

screen, the first important part of information is given in the form of Fig. 4. Of this code health is the most important aspect since it directly correlates to the maintenance costs. This cost is determined by a combination of both properties of the code as well as organizational factors. These include low cohesion, presence of god classes, lines of code complex methods, nested complexity and many others. One of the complexity metrics used in the case of code complexity is cyclomatic complexity.

When further inspecting code health, code scene provides an overview of code health hotspots as seen in Fig. 5. This allows finer details of the different files which cause the most impact on the current code health. It also provides a few actions that can be executed on the most important files, for this use-case the emphasis will be on the X-Ray action. The results on a very small example project are given in Fig.6. It clearly highlights specific functions in

16
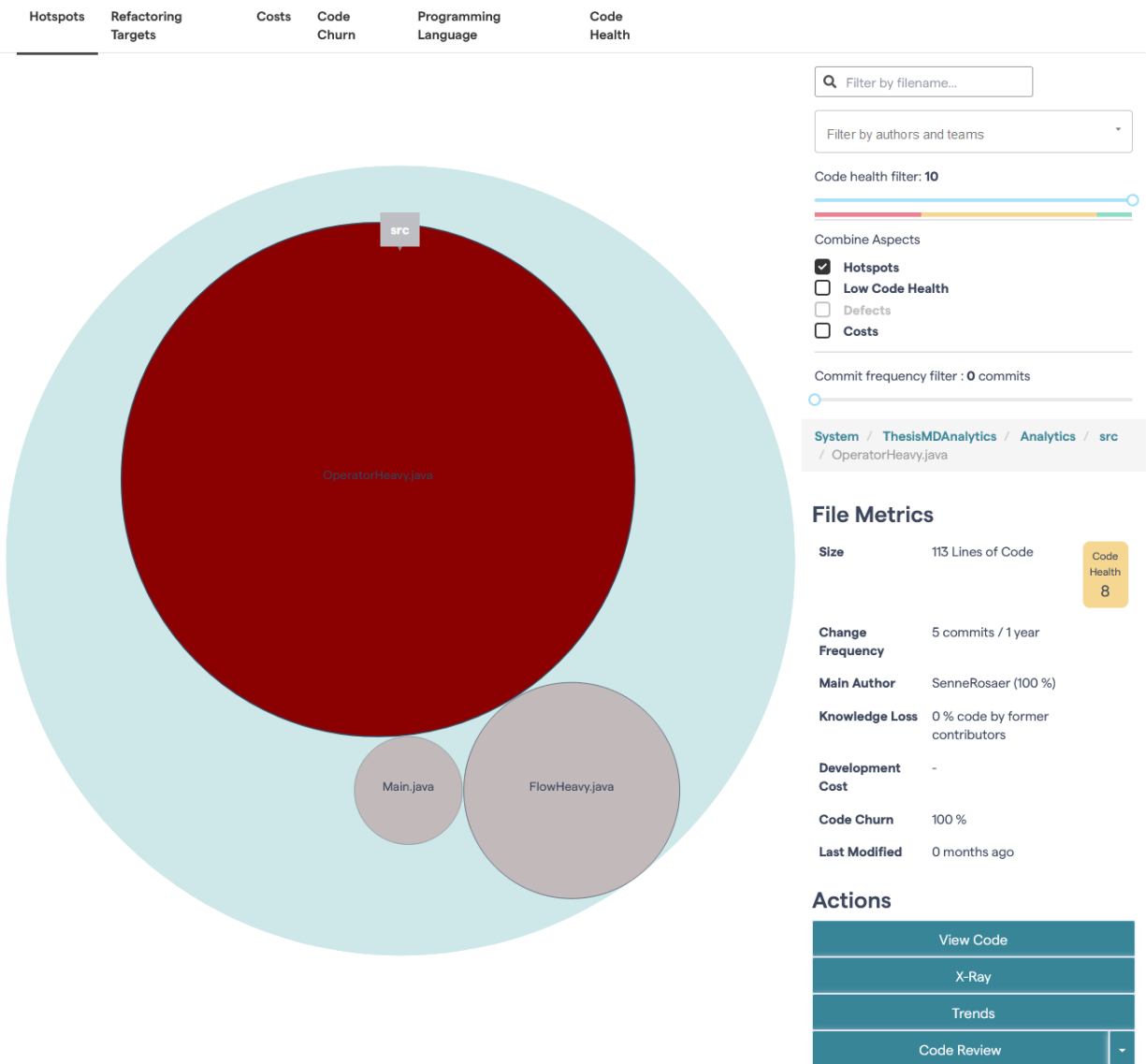
Figure 5: CodeScene hotspot overview

different colors based on a few different factors. Certain functions such as f1, are also flagged as refactoring targets. It chooses this based on a complex conditional present instead of only on relying on cyclomatic complexity. Still, the *areBoxingCompatible* is indicated as the most complex function which is based on cyclomatic complexity, meaning this indication makes less usage of complementary metrics.

2.4.2 SonarQube

Just like CodeScene, Sonarqube provides a large amount of functionality so the scope will again be limited to the necessary parts. The main setup of SonarQube is a bit more involved and differs between programming languages, pipeline usage and other factors so it will not be discussed. The primary overview given by Sonarqube is shown in Fig. 7. It mostly provides information about technical dept, bugs, code smells, testing coverage and duplication. More specific information can however be found when moving over to the measures tab. In Fig. 8 this is shown and the most important parts can be seen. Within SonarQube there is a clear separation between maintainability and complexity. Testing effort is more closely linked to the complexity metrics for this work which is a part of the maintainability. Sonarqube calculates maintainability only based the time needed to fix the code smells. As for complexity, this is separated into cyclomatic complexity and cognitive complexity. The definitions of cyclomatic complexity remains the

Function name			Change Frequency		Lines of Code		Code Review		Function Complexity		Actions
f1 ⊚			1		16		• Complex Conditional		4		⋮
areBoxingCompatible			0		40		• Complex Method • Bumpy Road Ahead		14		⋮
f1_refactored			0		15		• Complex Conditional		4		⋮
nextPowerOf2_refactored2			0		10				2		⋮
nextPowerOf2_refactored			0		10				1		⋮
nextPowerOf2			0		10				1		⋮
f2			0		5		• Complex Method		10		⋮
val_operation			0		3				1		⋮
f1_operation			0		3				1		⋮

Figure 6: CodeScene precise file information

same and it is mostly used to indicate the minimum amount of tests needed to have full test coverage. The cognitive complexity indicates how hard it is to understand the code's control flow. These metrics are however not calculated for each method but for each file, therefore refactoring targets are not directly defined but rather indicated through the code smells.
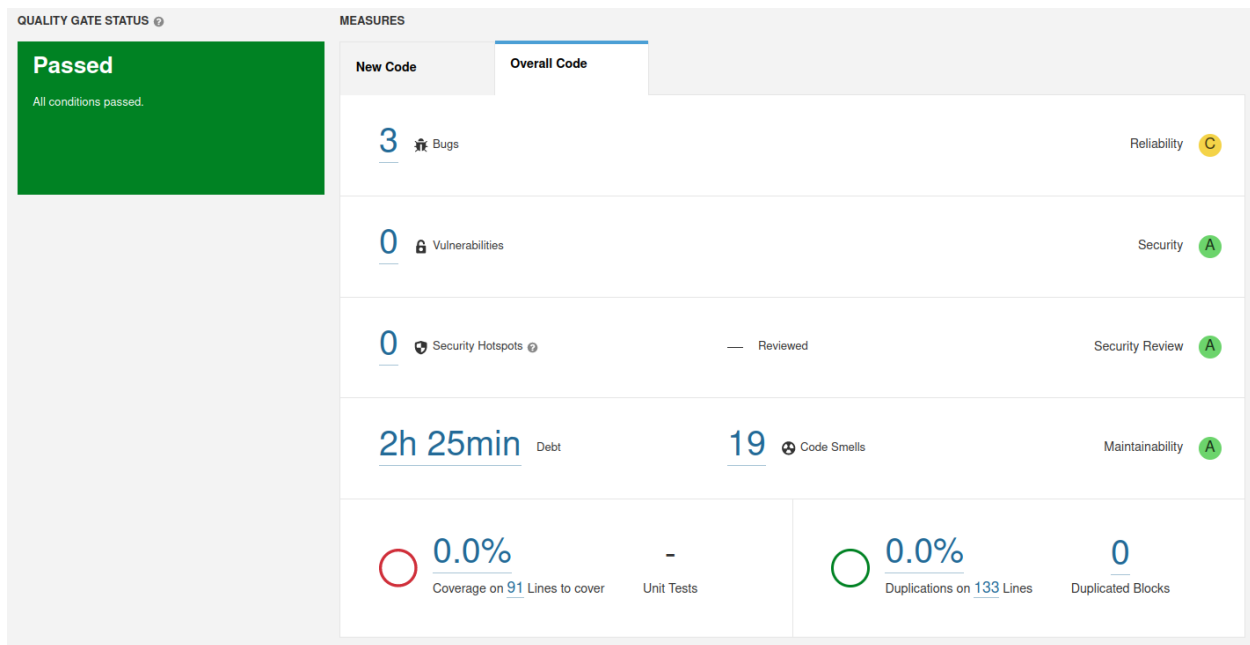
Figure 7: Sonarqube overview
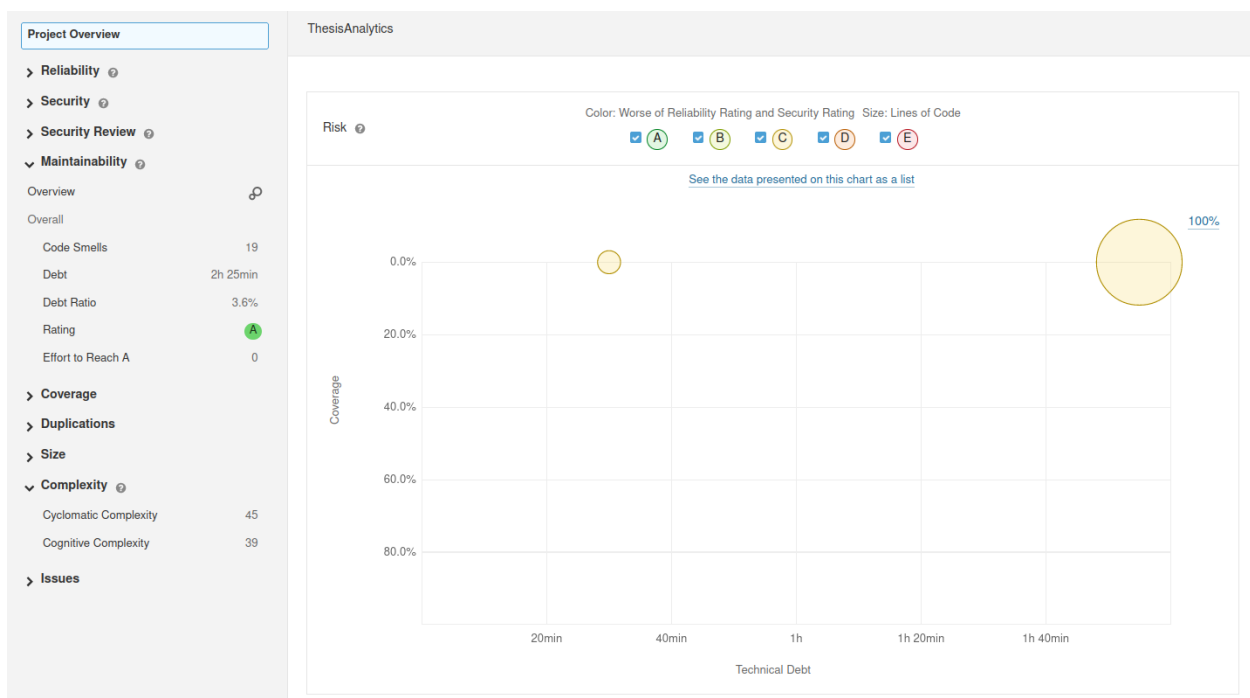
Figure 8: Sonarqube precise information

3 Method / Experimental Design

The experimental design takes a very step-wise approach. As most of the experimentation is done with a combination of source lines of code, cyclomatic complexity and mutant density there is an initial need for this data. The gathering of this is not as one-sided due to difficult tooling and dataset genera.tion. With the data in place there are different types of analysis that are done with each their own use-cases. Correlation analysis between the metrics is the first step in this process. Further experiments rely on finding patterns in how outliers for specific metrics correspond to testing effort and potential refactoring opportunities. This is done on a few different levels which are explained within the following sections. Despite the usage of SonarQube and CodeScene in earlier sections, this is not used during most of this experimentation. Outlier detection is at the core of finding refactoring opportunities and support for mutant density as a complementary metric. DevOps tooling will mostly be used alongside other information to highlight the applicability and provide anecdotal information.

3.1 Data gathering

Table 1: Metrics for programming languages

	Java	**Python**	**C/C++**
Cyclomatic complexity	Chaosmeter	Mccabe	Metrix++
Mutant density	Chaosmeter	Mutpy	Dextool mutate
Source lines of code	Chaosmeter	Custom	Metrix++

The process of gathering data contained quite some hurdles due to difficulty with tooling. Since the goal is to obtain source lines of code, cyclomatic complexity and mutant density there is the need for a tool which supports all of these. For java this is not an issue since Chaosmeter [16] covers all of these metrics. Other languages do not have this luxury, since some experiments compare between languages this cannot be avoided.

For C and C++ the same tooling can be used. The calculation of the amount of mutations is done through Dextool mutate [17]. This tool was not created with the intention of just calculating mutant density but has more emphasis on mutation generation and execution. Due to this emphasis, Dextool is also not an out of the box solution. To gather the amount of mutations, an additional parser was written to obtain it from the output it gives. Dextool provides an html output with the mutations visualized, it would also be possible to alter the source code to provide this functionality, but writing the parser was an easier solution. The source lines of code and cyclomatic complexity are generated by Metrix++ [18]. Metrtix++ keeps functions by the start and end line, therefor the parser needs to be able to read this information from the html files.

Python also has a few different tools to extract the data. Cyclomatic complexity is calculated by Mccabe [19], the source lines of code are calculated by a custom tool and the mutations are calculated by mutpy [20]. Existing tooling for source lines of code mostly focuses on calculating it for a single method. This work does analysis for each method separately, therefor something additional had to be provided.

Again, the tooling used for mutations have a pure emphasis on the testing which is the same with mutpy. Therefore, mutpy had to be altered since the base version was not able to just generate and count mutants for each function. The mutations where originally coupled to the written tests, meaning any functionality which wasn't tested did not get mutated. This functionality was overwritten such that the presence of tests was no longer a necessity. The code was also altered in a way to provide the mutant, their corresponding function and file in structured manner.

Although this provided us with all of the data, there are still a few flaws that should be discussed. Since the tooling is not obtained from a single provider, there might be variations in implementation which could deliver small deviations in results. As was mentioned in the background information on mutation operators, there are differences between languages which is reflected in differences in the tooling. How source lines of code are counted could also vary across the tooling due to the lack of standardization. This issue is only shortly covered during the experimentation itself since it would cover much broader aspects than goal of this work intends to.

Across the languages there are also issues regarding the projects used. Although dataset generation will be covered in a later section, there are some issues that are coupled with the metric extraction. For Java, the metrics are

obtained in a static manner, this implies that there is no building of any sort included. Python allows for this to happen but the implementation for Mutpy does not completely support this. For example, loading a project that includes usage of a GPU, forces you to have one working on your machine. This forces certain workarounds which makes the tooling more flawed then what would be optimal.

C and C++ have similar issues, Dextool needs to build the project before the mutant generation can be applied. This not only consumes quite some time, it also forces you to have a system that can build the project. With older C or C++ projects this is not always trivial.

For these reasons, and for the general time that large datasets take to parse, the datasets were kept relatively small.

3.2 Dataset composition

Due to the previously discussed difficulties, the composition of the dataset is very small. Table 2 gives an overview of all the projects used, their methods and the size. Note that for the method count and the nature of this paper, only methods which include at least a single possible mutation are included. The goal was to have a certain amount of variation withing the projects sizes and also have some difference in functionality. Python for example, contains some projects for API's and frameworks while also having some projects focusing more on mathematical problems. Java does have Spring as it's largest project which is much larger than the largest projects for other languages. This is not ideal since it might provide certain imbalance. Equally large projects for other languages are also difficult to obtain due to more errors occurring during the generation of this data. Dextool often outputs that it skips certain methods or files due to internal issues so this also impacts the size of the resulting data.

3.3 Correlation analysis

Within the comparison of complexity metrics there have been multiple works on the correlation between cyclomatic complexity and other metrics [21, 22]. The main purpose is often to verify the necessity of different types of metrics. If there would be an extremely strong correlation between source lines of code and cyclomatic complexity, it would trivialise cyclomatic complexity. It is a metric that is more difficult to compute and if the additional information it provides is not sufficient then there is less need for such a metric. A strong negative correlation on the other hand is also no perfect occurrence. The two metrics might be very different but not having a basis of similarities also indicates underlying issues. Code complexity can be measured and looked at through different perspectives. Even though the complexity of a method can be viewed differently, a complete lack of overlap shows that the metrics do not cover a similar basis of complexity.

This creates the necessity to verify the correlation in combination with mutant density. Lack of correlation with other metrics would indicate that its vision on complexity is vastly different or potentially wrong. A too strong correlation would make mutant density completely redundant. Cyclomatic complexity might not be the least computationally intense metric to calculate, but it is still relatively simple compared mutant density. As mentioned during the discussion of the tooling. It has quite a few complexities attached to the calculation, a very strong correlation would therefor be a large issue for the core of this work. This is also needed to verify the ability of mutant density to exist as a complementary metric to cyclomatic complexity.

The analysis of correlation includes a variety of distribution plots and other visualizations through boxplots and scatterplots. These compare all the possible pairs between cyclomatic complexity, source lines of code and mutant density. The visualization provides some insight, but most of the concrete information will be obtained through the calculation of pearson correlation between the same pairs.

3.4 Outlier/Refactoring opportunities detection

After the correlation analysis most emphasis is placed upon the detection of outliers or refactoring opportunities. This section is the largest part of the experimentation.

The first few questions that should be answered are what do we calculate outliers of and why? The main essence of using complexity metrics is to find certain methods that are overly complex or require more testing effort. These could be also be called potential refactoring targets. Cyclomatic complexity often defines these based on the defined thresholds, specifically above 10. Due to a lack of foundation and backing for this aspect as mentioned earlier, this work tries to work around this by using outlier detection. This only works when there are enough methods to use this calculation, the finer details of this issue will be discussed later.

Outlier detection will thus be the main method for finding refactoring targets. Finding these refactoring tar-

Table 2: Programming language, project name, method count, and SLOC

Programming Language	Project Name	Method Count	SLOC
Java	EventBus	157	1813
Java	FastJson	2030	48479
Java	Mockito	741	9805
Java	Rebound	148	2228
Java	Socket	67	977
Java	Spring	8701	110078
Python	BioPython	687	12984
Python	Ansible	1536	34892
Python	FastApi	28	752
Python	Google Images	20	703
Python	Photon	44	453
Python	Scrapy	700	7993
Python	Qbittorrent	65	356
Python	Flask	131	2248
C++	Duckdb	1420	35366
C++	OpenCv	1678	40100
C++	PrusaSlicer	226	7310
C++	Transmission	521	11399
C	Curl	438	19464
C	LibrdKafka	283	7977
C	Redis	1501	38523
C	Mbedtls	842	34625
C	SoftEthervpn	604	15084

gets is of utmost important for the goal of this work. Finding similarities between the targets of mutant density and cyclomatic complexity while also showing the unique targets it can find.

To find these outliers or target functions, outliers are calculated for a few different metrics. The most basic metrics are average cyclomatic complexity and average mutant density.

$$AverageMutantDensity = (MD/SLOC)$$

$$AverageCyclomaticComplexity = (MD/SLOC)$$

These metrics are calculated for each method within a dataset, out of these a mean and standard deviation is derived. Outliers are defined based on a z-score or the amount of standard deviations from the mean a method is located. The average mutant density and cyclomatic complexity are simply the previously described metrics divided by the source lines of code of the method. This is introduced to take the sizing of the functions into account.

The outliers based on these metrics do not give information on the relation between mutant density and cyclomatic complexity but it gives an initial feeling of their behaviour. Looking in depth for relations and deeper patterns without covering the fundamentals might leave certain information untouched. These basic form of outliers also serve their role later on to support other claims. As mentioned earlier, metrics with completely different outliers on their own are not necessarily good. Metrics should provide certain similar results and these outliers serve that purpose.

The most important aspect of these outliers, and not only these based on basic metrics, is that they still show problematic methods. Using this type of outlier detection, it is possible to show problematic methods without relying on thresholds. This is especially necessary for mutant density since a threshold is not present for this metric yet.

3.4.1 Scaled outliers

While looking for outliers and refactoring targets, using something such as average mutant density or average cyclomatic complexity leaves some potential issues.

Two methods might have the exact same average mutant density due to the relation of mutants to the source lines of code. However, if one of these functions is much longer this indicates a larger issue than the smaller function. A small function with a bad composition might be a choice to encapsulate the problematic code. Although it does not excuse the design of the function, the same bad composition for a larger method would have more direct influence on the maintainability of the code. Therefore, they should be more easily filtered or have a score that represents this better.

$$ScaledAverageMutantDensity = (MD/SLOC) * (MD + SLOC)/100$$
$$ScaledAverageCyclomaticComplexity = (CC/SLOC) * (CC + SLOC)/100$$

This applies a multiplication to the previous formula that takes both the size of the source lines of code and mutant density into account. This divided by a factor of 100 to limit the influence of this multiplication. The chosen factor of 100 is not based on specific empirical evidence and would need more verification, for the current use cases it is sufficient however. The same is done to obtain a scaled version of the average cyclomatic complexity.

3.4.2 Cross metric

The previous ways of handling outlier detection, only focused on cyclomatic complexity and mutant density separately. They provide some initial insight but a large part of this work is investigating how cyclomatic complexity and mutant density can complement each other.

There are a few ways to look at relations between cyclomatic complexity and mutant density. The first one is using the previous outliers and applying set theory to look at the intersection and dissections. This can be visualized as Fig. 9. The similarities between metrics has been a recurring topic until now, this will also remain so for the rest of this work. The intersection on it's own will be more important later on, but for the moment there is mostly an emphasis on the dissection of those two. These cases contain useful information due to the fact that only a single metric captures it as an outlier. Depending on the code contained in those outliers, it could show methods that are not noticed through a singular metric but still contain refactoring opportunities. This part is of utmost importance in terms of showing the additional functionality of mutant density. There needs to be sufficient functions to showcase that the additional outliers are also complex but that will be covered in the results.

The second way is by using a metric that uses the distance between the cyclomatic complexity and mutant density of a method.
$$AverageCC\&MDdistance = (CC - MD)/SLOC$$

If the outliers are calculated based on this metric it shows which functions have an abnormal difference between the cyclomatic complexity and the mutant density. Just like the previous metrics, there could be the introduction of some sort of ratio or scaling to take the sizing into account.
$$ScaledAverageCC\&MDdistance = (CC - MD)/SLOC * (MD + CC + SLOC)/100$$
The interesting aspect about this metric is that it takes a more direct approach to measure the relation between the two metrics. It might show functions that would otherwise not appear in any of the previously listed method and also emphasises that complexity can occur in more than a single way. Using this metric only functions to a certain extent if both cyclomatic complexity and mutant density have similarities. If there is completely no correlation than the difference between these two metrics would provide very random data. Therefore, the correlation analysis has its importance on supporting this experiment.
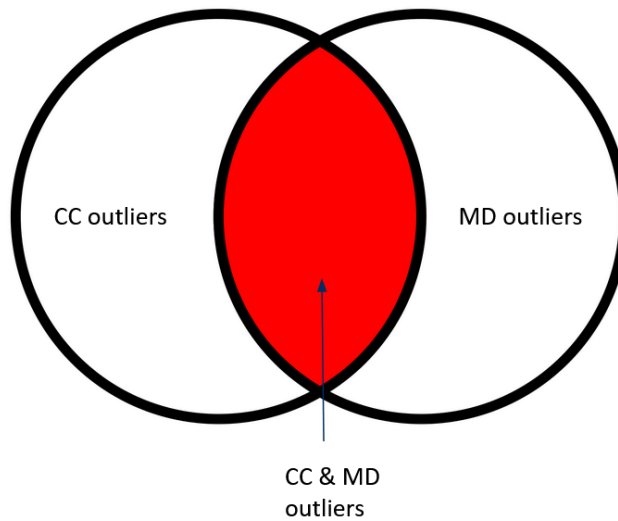
Figure 9: Caption

3.5 Mutation operator specific

An additional advantage of using mutant density is the fact that it is more akin to a modular metric. Choosing the different mutation operators is what will heavily define the results. There are too many possible combinations of mutation operators to consider them all.

However, if the same outliers from experiments are calculated for every single type of mutation operator separately there are still some interesting factors. This means that that instead of calculating mutant density with a set of mutation operators, there is a separate run for each mutatation operator.

There are two ways of looking at this new data. The first one is looking at outliers that occur for each mutation operator separately. This implies that there are methods which have a very large variety of possible mutations spread out over different mutation operators, on top of that they are outliers in each case. Normally cases like this should not occur frequently. If they occur, it is assumed to be more problematic to cases that are only outliers on a few mutation operators instead of a large set of operators.

The second interesting possibility are outliers which are only outliers for a single mutation operator. They might not contain enough mutations to be a general outlier, but they would have enough when tweaking the mutation operators. Cases like this might not be prime refactoring targets, but they do show the essence of different mutations.

This also reintroduces the modularity of using mutant density. Although most emphasis will be on the usage of a general set of mutation operators, this still allows for more in depth analysis of which metrics cause the complexity.

3.6 Dataset formulation

Although dataset formulation will mostly be described in another section, there are still some decisions that align closer to the methods. For each of the different programming languages used in this project a dataset is formulated, it is compromised of a set of different sized and type of projects.

For the outlier detection, not every experiment was done on the complete dataset for a specific language. Certain experiments were executed on only a single project instead of a set of projects. This opens up a different type of comparison in terms of generalization.

The assumption behind this decision, is that using a set of projects might cause for generalisation of outliers. Each project has its own unique set of features. This could be a different type of problem they try to solve, which could lead to more algorithmic code for example. Aside from the type of problem they solve, there are also different visions on styling and what code is assumed good and healthy within a project. Therefor, using outlier detection over a set of project could generalise the uniqueness of a specific project within that dataset. Because of this, experimentation is done in both directions to ensure that the correct dataset formulation is chosen as not to lose valuable information.

24

3.7 Overview of formulas

Recap of formulas and adding a name by which they will be referred to from now on.

Metric	Formula
Average Mutant Density	(MD/SLOC)
Average Cyclomatic Complexity	(CC/SLOC)
Scaled Average Mutant Density	(MD/SLOC) * (MD+SLOC)/100
Scaled Average Cyclomatic Complexity	(CC/SLOC) * (CC+SLOC)/100
Average CC & MD distance	(CC-MD)/SLOC
Scaled Average CC & MD distance	(CC-MD)/SLOC * (MD+CC+SLOC)/100

Table 3: Recap of the most important formulas which are used for outlier detection

4   Results / Evaluation

4.1   Correlation analysis

Correlation can be calculated between the three provided metrics as to answer RQ1. The correlation between cyclomatic complexity and source lines of code already has some existing work such as [21], this is mostly to measure potential redundancy between these methods. This work applies similar tactics but also includes mutant density. The correlation can be visualized in a few different ways but this work will limit itself to boxplots and scatterplots. The data that the graphs are created upon will be specified in the captions of the images since the following parts will contain quite some variation. Although most experiments focus on the Java dataset, some overlap will be shown over languages to address issues that might occur. Due to the difference between using a single project or multiple projects discussed earlier, a singular dataset will be used more frequently, especially from the outlier section onward.

Starting with the visualization of the correlation between cyclomtic complexity and source lines of code, as seen in Fig.10 and Fig.11. A certain correlation is clearly visible due to the gradual increase in both directions. The dataset used for these two graphs is the complete Java dataset which is still not particularly large. In terms of the boxplot, the tail of the boxplot is clearly more susceptible to variation. Looking at the tail end of this boxplot from the perspective of code can also give more insight in the variation. A larger function has more lines to incorporate statements which increase cyclomatic complexity, this does not imply that larger functions always have a higher cyclomatic complexity but the probability is higher. Therefore, larger functions have a more prominent variation in this visualization. The same reasoning can be applied to the results of the scatterplot. Although the dots are very cluttered, the regression line gives a good view on potential correlation. The regression line also clearly visualizes an increasing variation towards higher source lines of codes and cyclomatic complexity. Although a very obvious statement, it is clear that the cyclomatic complexity increases with a much slower rate than the source lines of code. Taking the same graphs for the relation
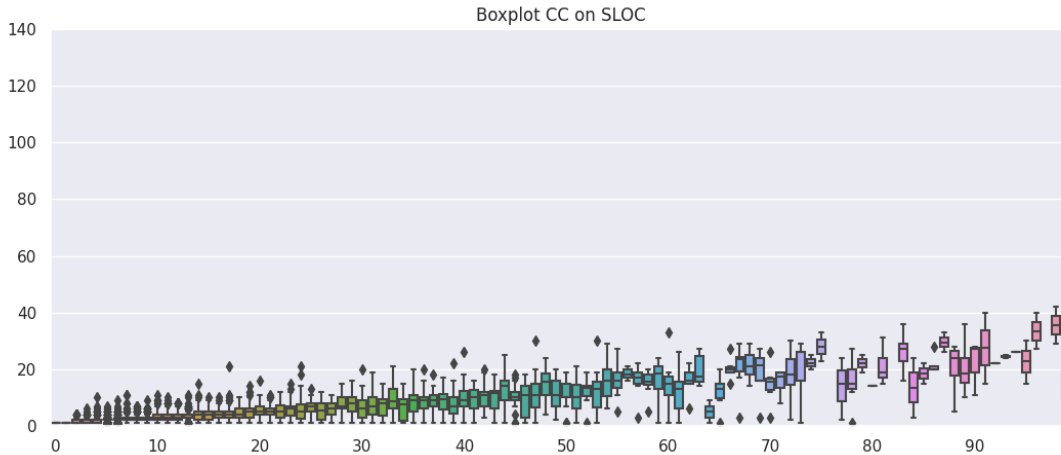
Figure 10: Boxplot on the correlation between cyclomatic complexity (y-axis) and source lines of code (x-axis).

between mutant density and source lines of code can be seen in Fig. 12 and Fig. 13. At certain levels, this shows similar patterns but there are a few things that stand out in comparison to the previous graphs. The largest difference is the clear increase in variation towards the tail of the boxplot. The scatterplot also has visible increase in variation with more points spread out. This increasing variation is however, only slightly visible in the regression line. At its core, this follows the same principle, where larger functions simply have more possible variation. This is amplified due to the nature of mutant density. The amount of cyclomatic complexity you can cause in a single line is not necessarily limited. It is always possible to increase it by adding more conditional operators. In reality, there is more of a limitation on how much cyclomatic complexity you can increase for every line of code. For example, there are also the lines of code within the bodies of for loops and if statements. Unless they contain nested statements, it does not contain statements increasing cyclomatic complexity any further. Having a higher mutant density than cyclomatic complexity for the same amount of lines is therefore much easier. Most statements that increase cyclomatic complexity, also increase mutant density. On top of that, every single line that does not increase cyclomatic complexity, can still increase the mutant density. For example the bodies of for loops, they are more likely to include mutable operators. This means that with a lower source lines of codes, a higher mutant density is very much a possibility.   In Fig.14 the scatterplot is shown for the correlation between cyclomatic complexity and mutant density. There are some immediate difference visible compared to the previous visualizations. While cyclomatic complexity and mutant density both have a slower increase

Figure 11: Scatterplot with a regression line on the correlation between cyclomatic complexity (y-axis) and source lines of code (x-axis)
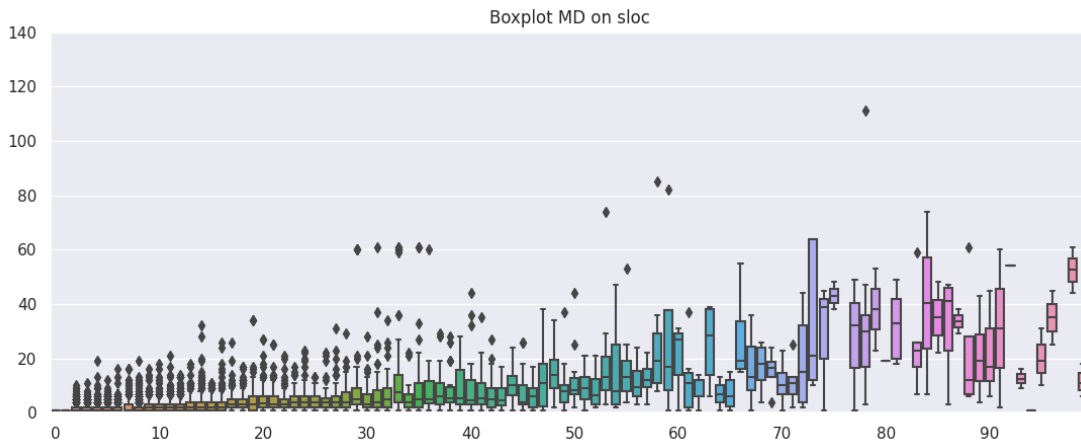
Figure 12: Boxplot on the correlation between mutant density (y-axis) and source lines of code (x-axis).

compared to a relative increase in source lines of code. This is not present when looking at the direct relation between those two. In essence, this result is only slightly unexpected. Both mutant density and cyclomatic complexity differ vastly with source lines of code on a specific aspect. This is the inclusion of some form of functionality when viewing at complexity. Either through for statements or possible operators present in the code. Therefore, a more even increase in both of these metrics could be expected. Another potential reason for this more even increase is the fact that most statements that increase cyclomatic complexity, include operators that can be mutated. Despite this, it was mentioned that for the same amount of lines, it is more likely to expect more possible mutations than statements that increase cyclomatic complexity. With the limited data available it is hard to determine if this is specifically due to the chosen Java projects or due to the programming language itself. To verify if it is caused by the selection of projects, a larger dataset should be formed. Verification if it is based on the programming language is done in a following part.

Despite the low variation at the start of the regression line, there is a significant increase in variation visible on the regression line. At higher cyclomatic complexity and mutant density this is expected. Previously when comparing with source lines of code, it was mentioned that with larger source lines of code there is more potential variation. When only using these two metrics which have high possible variation at the tail end of the regression line, this effect is amplified. It indirectly also means that those cases with higher variation have a higher source lines of code so this metric is slightly incorporated without direct usage.

Before continuing with the exact numbers for Pearson correlation, there are a couple of issues that should be discussed when covering multiple programming languages.
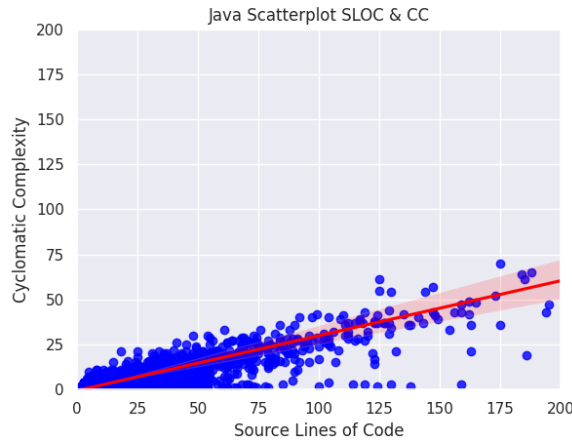
Figure 13: Scatterplot with a regression line on the correlation between mutant complexity (y-axis) and source lines of code (x-axis)
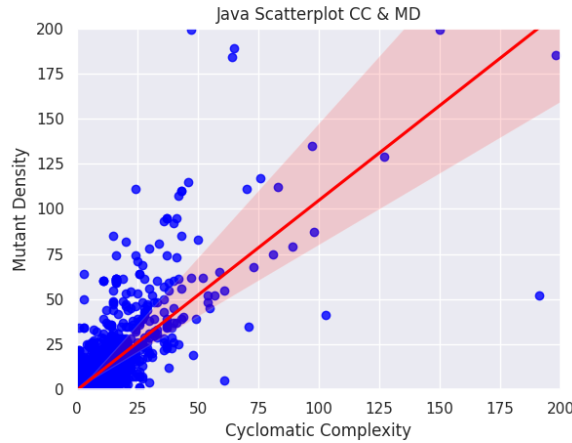
Figure 14: Scatterplot with a regression line on the correlation between cyclomatic complexity (y-axis) and mutant density (x-axis)

In Fig. 15 the boxplot showing the relation between cyclomatic complexity and source lines of code can be seen for C++. Instead of using the complete java dataset, this is created based on the C++ dataset. So quite different languages in terms of applications and general complexity. This is clearly extremely similar to the boxplot comparing cyclomatic complexity and source lines of code for Java. The scatterplot from Fig.16 also clearly visualizes this similarity.

However, when looking at the boxplot and scatterplot for the correlation between source lines of code and mutant density in C++ in Fig.17 and Fig.18, there is a large difference. Although the sizing of the datasets used in these experiments are nearly not large enough to have a concrete conclusion on this issue, there is still an assumed cause. Languages such as C and C++ have a vastly different complexity in comparison to other languages such as Python, this also results differences in implementation time for the same functionality [23]. Languages with this higher complexity also contain more potential targets for mutations based on the build in operators. An example of this is writing a for loop to iterator over a list. C++, Python and Java all contain simplified ways of creating this for loop. C does not have this luxury, the same for loop to iterate over list will contain an initialization, a condition and a update statement. Each of these parts can be mutated while a simplified version from Python has less possible mutations. There are other such difference between programming languages which could imply the presence of generally more targets for mutation. This is merely an assumption since there are also other factors that could influence the amount of mutations.

28

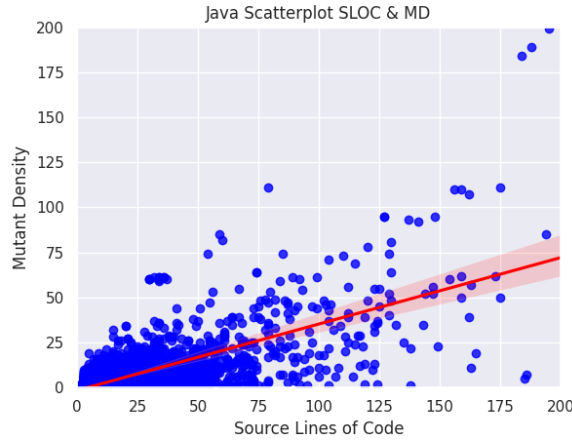Figure 15: Boxplot on the correlation between cyclomatic complexity (y-axis) and source lines of code (x-axis).

Figure 16: Scatterplot with a regression line on the correlation between cyclomatic complexity (y-axis) and source lines of code (x-axis)

The composition of the dataset is the biggest potential influence. If the selected projects contain more mathematical focused solutions, it's obvious this would create more influence for mutant density. Reducing the influence of the selection of projects would need a much larger dataset for each of the languages or a more thorough process for selecting the projects. The scatterplot shows some support for this statement. In previous scatterplots, the mutant density or cyclomatic complexity increased at a lower rate than the source lines of code. In C++ this remains the same for cyclomatic complexity, mutant density however seemingly increases at an extremely similar rate when looking at the regression line. Although the issue of the small dataset size remains, this indicates the general higher amount of mutations due to applications or programming language. Another potential cause could be the difference in tooling and computation method as mentioned earlier. This issue is not easy to solve without the introduction of generalized tooling across languages. In the Java dataset, there was a very similar increase when comparing mutant density and cyclomatic complexity. The same scatterplot for C++ can be found in Fig.19. In the case of C++, the mutant density increases at a higher rate than the cyclomatic complexity. This falls more within the expectations but the difference between the languages shows some discrepancy on that aspect. With the available data, it remains hard to pinpoint this issue. A much larger scale dataset should be used to determine if this is purely based on project selection or on the programming language. It has been argued that programming language does influence these results but a larger scale investigation should still be done to more correctly measure the exact influence. All of the previous graphs are purely a visualisation and they need to be supplemented by additional numerical data to support it. This is done by using the Pearson correlation for these experiments. As mentioned in the methods, this is due to the continuation of the approach of [21]. In Table4 the pearson correlation between cyclomatic complexity and source lines of code is shown for the Java dataset. It is calculated for different coverages of the dataset based on a minimum source lines of codes. This also
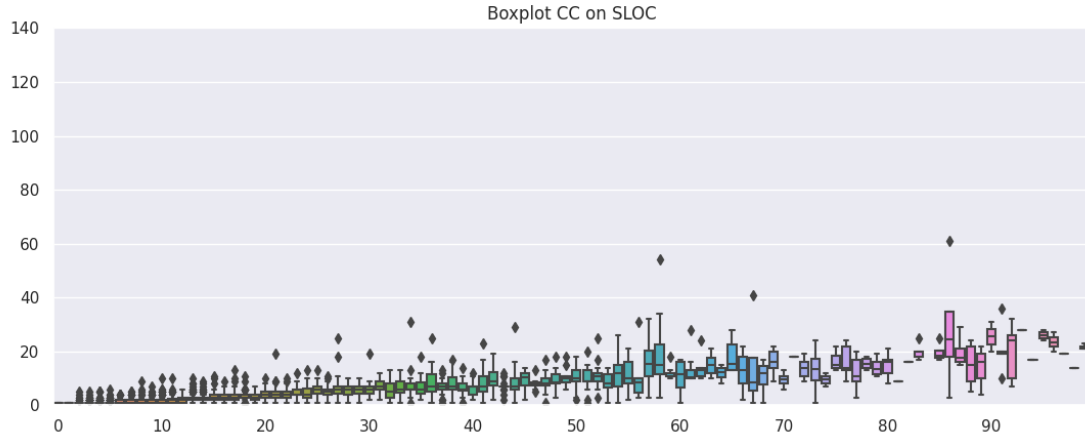
29
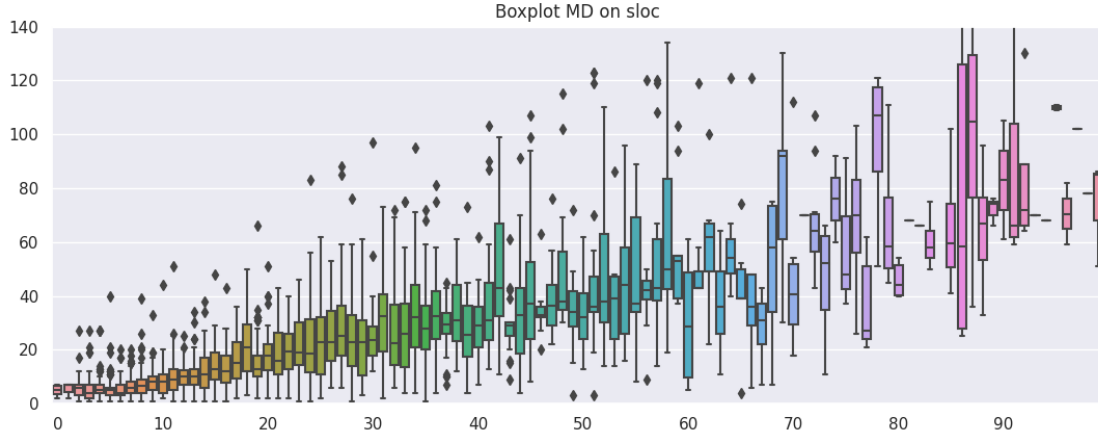
Figure 17: Boxplot on the correlation between mutant density (y-axis) and source lines of code (x-axis).
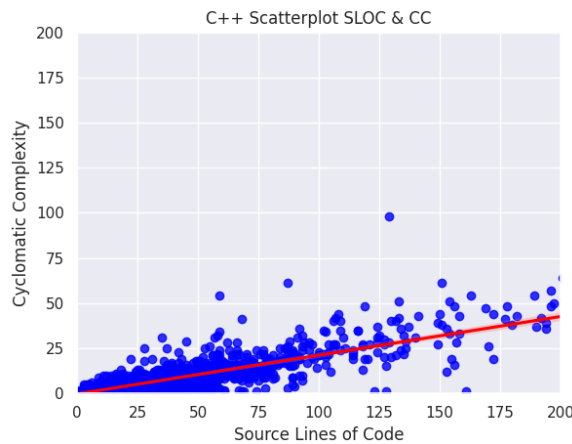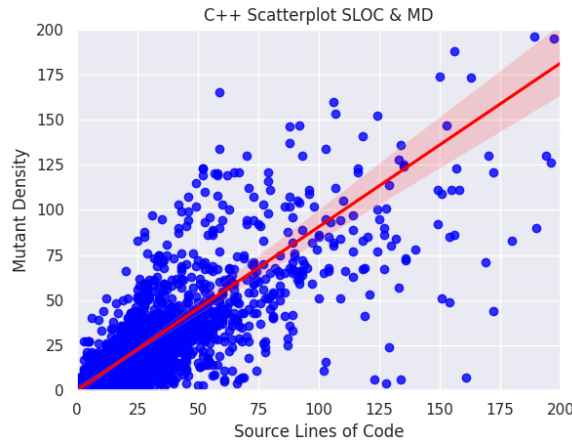

Figure 18: Scatterplot with a regression line on the correlation between mutant density (y-axis) and source lines of code (x-axis)

shows a significant drop-off of functions between a source lines of code of 5 and 9. Although the coverage drops, the correlation barely changes between those. The correlation is in itself fairly positive, but based on the work of [21] this correlation is not enough to make one of the metrics redundant. Changing this correlation analysis to mutant density and source lines of codes can be seen in Table5. A drop in correlation is to be expected after looking at the changes in the boxplot and scatterplots. Despite this, the drop is still fairly significant. From a R value of 0.88 with full coverage to 0.76 is a significant drop but not large enough to discard the positive correlation that still exists. The lower correlation causes the need for a more careful approach surrounding this but it is not necessarily an issue. This lower correlation

Table 4: Pearson correlation between source lines of code and cyclomatic complexity for various code coverage's based on lines of code.

Minimum SLOC	Coverage	Pearson CC
1	1.0	0.8888096205091884
3	0.9994934143870314	0.8888080266538486
5	0.8317291455589328	0.8893752681541656
9	0.5546268152651131	0.8901921044245265
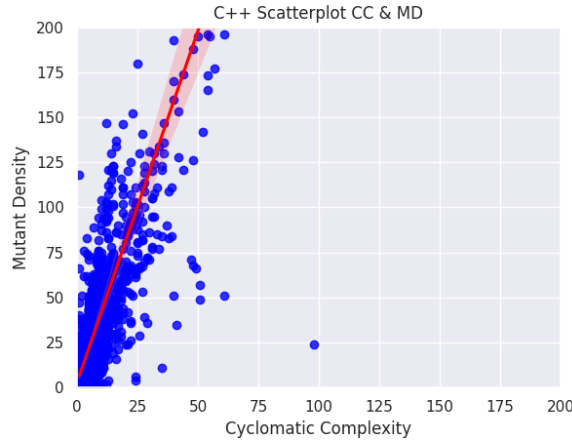
Figure 19: Scatterplot with a regression line on the correlation between cyclomatic complexity (y-axis) and mutant density (x-axis)

Table 5: Pearson correlation between source lines of code and mutant density for various code coverage's based on lines of code.

Minimum SLOC	Coverage	Pearson MD
1	1.0	0.7606250350314062
3	0.9994934143870314	0.760628300049117
5	0.8317291455589328	0.762462470681415
9	0.5546268152651131	0.7632538309621288

completely eliminates the thought that one metric could replace the other. Although it was never the goal of this work, it allows more emphasis on the differences between the metrics. It could also be argued that this drop in correlation is caused by the higher variation with higher source lines of code. However, rerunning the calculations with an upper limit of 150 on the source lines of codes suggests that this is not the case. It even causes a slight drop in correlation of about 0.05.

Due to the previous discrepancy between Java and C++ in the visualization, it is also important to look

Table 6: Pearson correlation between source lines of code and cyclomatic complexity for various code coverage's based on lines of code.

Minimum SLOC	Coverage	Pearson CC
1	1.0	0.9032900606076069
3	0.9976970317297851	0.9032585658099703
5	0.9498464687819856	0.9026921936104859
9	0.7024053224155579	0.8966620279058849

how this relates to the pearson correlation. The correlation between cyclomatic complexity and source lines of code is given in Table 6. There is a very slight increase in correlation but nothing notable outside of this. The biggest difference is visible in Table7, especially since there was a small increase in correlation between cyclomatic complexity and source lines of code. The drop in correlation with mutant density is present, which was to be expected. The obtained 0.7 would still qualify as positive correlation even though it is a drop of 0.2 compared to the correlation with cyclomatic complexity.

31

Table 7: Pearson correlation between source lines of code and mutant density for various code coverage's based on lines of code.

Minimum SLOC	Coverage	Pearson MD
1	1.0	0.6993161787635052
3	0.9976970317297851	0.6992248977192665
5	0.9498464687819856	0.6972006136354084
9	0.7024053224155579	0.680870897225745

These results all focus on correlation that includes source lines of code which is essentially a very different metric. Source lines of code does not contain any grasp on the functionality while both mutant density and cyclomatic complexity do. In that sense, it could be assumed that a correlation between the cyclomatic complexity and mutant density will be higher. In the visualisations graphs from Java, both metrics increased at a similar rate but there was still a fair amount of variation. The same results for C++ showed a very different rate of increase for both metrics but still in relatively correlated manner. Note that since the correlation between mutant density and cyclomatic complexity lacks the inclusion of source lines of code, there is no possibility to view this for different types of coverages. All of the results from Table 8 are thus based on full coverage.

For Java, the correlation between mutant density and cyclomatic complexity is even a drop compared to

Table 8: Code Metrics

	Pearson MD CC	Pearson CC SLOC	Pearson MD SLOC
Java	0.7427374837402358	0.8888096205091884	0.7606250350314062
C++	0.7142645801978144	0.9032900606076069	0.6993161787635052

the other correlation. For C++, this correlation is in between the other metrics but still a lot closer to the lower correlations. This implies that the correlation is not stronger despite both of them having some incorporation of functionality. The more similar rate of increase does not say anything about the correlation based on these results. Although it might have been expected, when looking more closely at the scatterplots, it is clear that for Java the points are still a big clutter. The same spread of points for C++ does look a bit less clutter which results in slightly higher correlation but not an extreme difference.

These results answer RQ1. **Correlation - what is the correlation between cyclomatic complexity and mutant density?**. There is a positive correlation cyclomatic complexity and mutant density which is not high enough to cause redundancy and not low enough to discredit complementary usage. There are some issues surrounding consistency over different programming languages but despite the differences, the correlation does not change enough to alter the conclusion. These results all provide a solid foundation to look further into the placement of mutant density as a complementary metric to cyclomatic complexity.

4.2 Outlier & refactoring opportunities detection

This section will focus mostly on the results surrounding experiments for RQ2. **Complementary - how can we utilize mutant density to provide additional insight into refactoring opportunities? Is there anecdotal evidence supporting this?**. Before diving directly into defining outliers for the previously defined metrics, there are a few visualizations in place to better contextualized some of the upcoming information. Again most emphasis will be on the Java dataset with the usage of examples from other languages in case there are any unexpected results.

The first step is looking at the distribution of differences between two metrics for the complete Java dataset. This gives additional information about the relationship two specific metrics hold to each other without directly looking at the correlation. An example of this can be found at Fig.20. Note that this is purely a distribution of the cyclomatic complexity subtracted from the source lines of code for each single method. This also implies that there is no division by the source lines of code for this experiment. This graph does not provide new groundbreaking information but it does convey certain interesting aspects. Since it is cyclomatic complexity subtracted from source lines of code, it is

Figure 20: Distribution plot of cyclomatic complexity subtracted from source lines of code where the x-axis is the difference and the y-axis the count

to be expected that the distribution is heavily shifted towards the right (more source lines of code than cyclomatic complexity). Due to this heavy shift in the distribution, if it were to occur that cyclomatic complexity is higher than the source lines of codes from the same function, it could be more easily defined as an issue. Defining refactoring opportunities on the right side of the graph is however more difficult with this type of graph without using real outlier detection methods If the same graph is used for mutant density subtracted from source lines of code, this results in Fig.21. From an intuitive perspective, in contrast to cyclomatic complexity, having more mutations than source lines of code is a possibility. Since a single line of code can have multiple different mutations depending on the chosen operators, these cases are assumed to occur more than a higher cyclomatic complexity compared to the lines of code.

Looking at the graph however, this is still a very infrequent presence. It occurs, but not as frequently as it might be possible. This is not that much of an issue since it might indicate the use good practices throughout the projects. The infrequent presence of these cases is not unexpected after the visualisations of the correlation. In the Java dataset, the mutant density and cyclomatic complexity were closely aligned. This means that despite it being possible of having methods with a much higher mutant density than source lines of code, the selected projects for this dataset do not contain it in large quantities.

Finally, it is also possible to visualize the difference between mutant density and cyclomatic complexity. This can be seen in Fig.22, which visualizes the mutant density subtracted from the cyclomatic complexity. For this dataset, this provides interesting results. The graphs are fairly balanced with most of its methods having no, or a very small difference between cyclomatic complexity and mutant density.

There is also no shift to a specific direction, there is a seemingly balanced distribution of cases with a higher mutant density and a higher cyclomatic complexity. This is particularly interesting due to the correlation seemingly not being
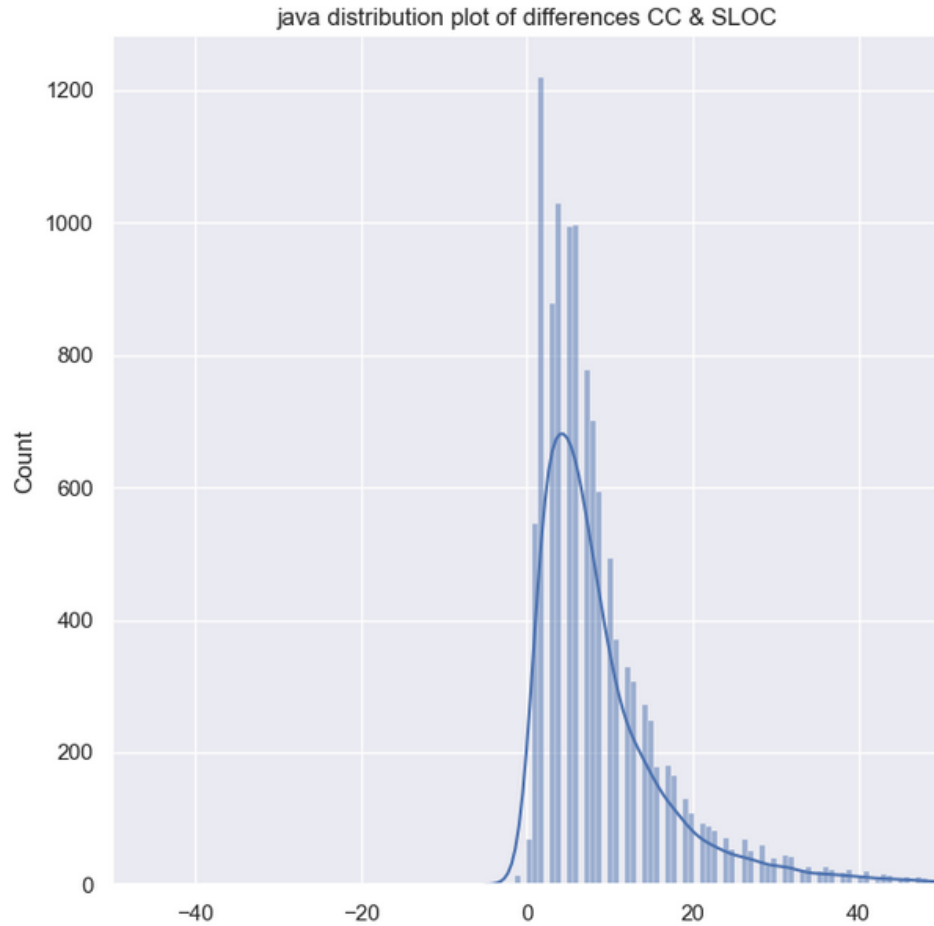
Figure 21: Distribution plot of mutant density subtracted from source lines of code where the x-axis is the difference and the y-axis the count

high enough to result in this visualization. This makes the comparison between those metrics all the more useful, especially when looking at the unique information each of these metrics provide. The same results over the datasets of different languages provide some differences yet again. The differences between cyclomatic complexity and source lines of code does not change over the different languages but any comparison with mutant density does. A first example of this occurs when looking at the difference between mutant density and source lines of codes for the C++ dataset, this can be seen in Fig.23. Instead of having a few cases where the mutant density is higher, there now is an almost even balance between the differences. This could be cause by the three causes that were discussed earlier, the differences between the tooling, general differences between implementations for languages and variation within the selection of the projects. It is fairly hard to differentiate between those issues and pinpoint the cause within the scope of this work.

This large variation in mutant density for C++ is also visible when plotting the difference between the cyclomatic complexity and mutant density. It causes an extreme shift where most cases have a higher mutant density when compared to its corresponding cyclomatic complexity. This also changes the perspective on what an outlier could possible be. Having different distributions over the languages is not optimal. It makes it very hard to generalize results and be certain about what the exact influences are on these metrics. Although it makes generalization very hard, it does not make the metrics and upcoming outlier detection bad. In the examples shown and the data resulting these, it all functions well within expectations. The difference in languages and the behaviour simply prompts careful consideration when looking at the possible generalization of singular results.

Figure 22: Distribution plot of mutant density subtracted from cyclomatic complexity where the x-axis is the difference and the y-axis the count

Now how does this transfer to effective outliers. In Fig.**??**, the distribution of outliers is represented based on how many standard deviations it differs from the mean. As mentioned in the methods, this is going to be the main method for outlier detection. Since there is no effective threshold, the outliers are based on the furthest distance from the mean. When looking at this distribution it is very clear that there are only very few cases that are at least four standard deviations away. Three standard deviations does also not occur that frequently so in the case of the difference between cyclomatic complexity and source lines of code, three times the standard deviation is a good cutoff.

Another interesting aspect is that there are more methods that are at least -1 time the standard deviation away from the mean. This is not something that falls within the original expectations. However, there is a fairly logical conclusion for this. Remember the distribution plot from Fig.20, this is a right shifted distribution due to the fact that it is based on cyclomatic complexity subtracted from source lines of code. Since there is a shift towards the right, it also shifts the mean in this direction. This happens despite the fact that there are still quite a lot of functions where the difference between the metrics is low. Add the variation that generally happens in coding different methods and these small "outliers" are not that unexpected anymore. Therefore, using the standard deviation a small amount of times is not sufficient to avoid these cases.

The same is applied for the outliers based on the difference between mutant density and source lines of code, as seen in Fig.26. This provides almost exactly the same patterns, except for the fact that there are now more cases which are a negative distance away from the mean. These are the cases where there is a higher mutant density
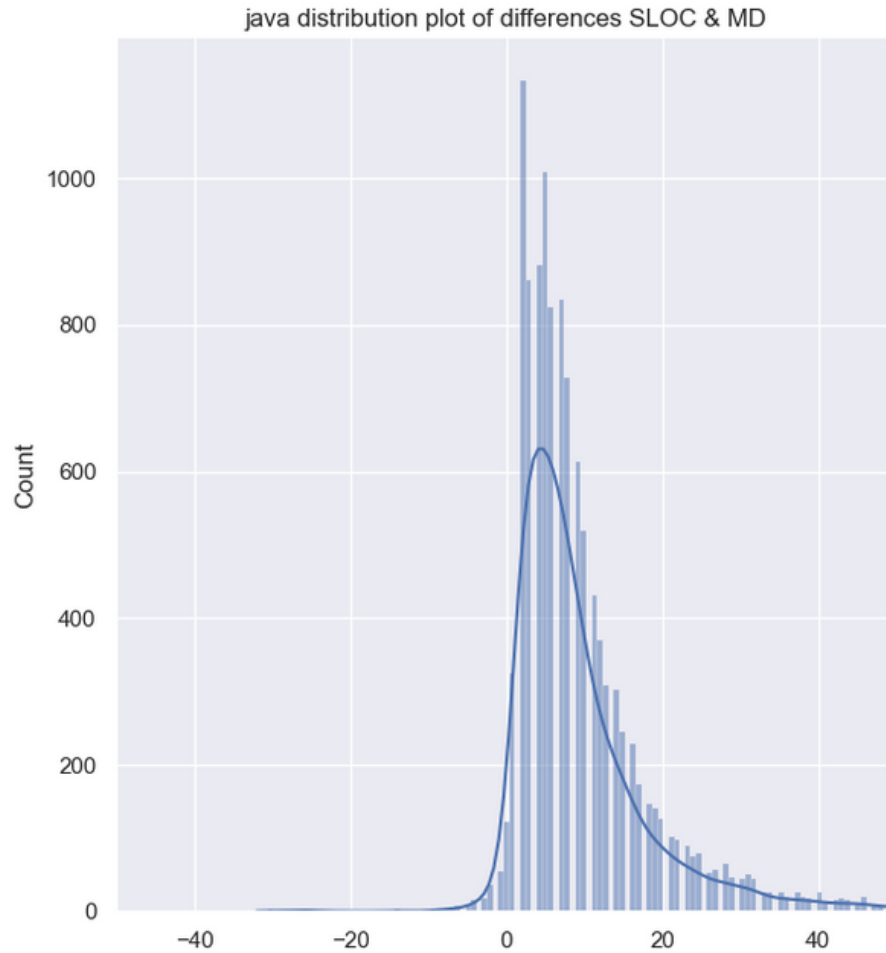
Figure 23: Distribution plot of mutant density subtracted from source lines of code where the x-axis is the difference and the y-axis the count

compared to the source lines of code. As mentioned earlier, these are possible but should occur more infrequently which is clearly the case. They are still worth looking at as outliers based on the distribution.

No specific new information comes to light when the same principles are then applied to the difference between cyclomatic complexity and mutant density. Also when looking across the different languages, there are slight differences but they all fall in line with what was previously discussed.

Using the simple difference between the two metrics is good for baseline indications but not for general usage. Therefore, the next step is looking at the average mutant density and average cyclomatic complexity. The results on more advanced metrics on the relation between cyclomatic complexity and mutant density will be discussed a bit later.

Using the results of average mutant density and average cyclomatic complexity in a distribution which shows the distances to mean that, can be seen in Fig.28 and Fig.29. These are very similar to the previous results with a few exceptions. The general shape of the graphs are extremely similar, the largest change is that there is a better spread of methods over different z-scores. Take for example the graph based on the average cyclomatic complexity, the highest outliers are now 10 standard deviations from the mean. Previously when only using the difference, the highest outlier was four times the standard deviation away from the mean.

All of this data is still generated over datasets comprised of multiple different projects even though it was mentioned in the methods that there might be certain issues surrounding this. Up until now, this is not something blocking since the previous information was all fairly general. Still, before diving into further metrics and code example, there needs to be some verification surrounding this.

4.2.1 Difference complete dataset vs single

The largest potential issue surrounding the usage of a single project or a set of projects is the potential generalisation that occurs when making a larger dataset. The most simplistic way of verifying if generalisation takes place over a
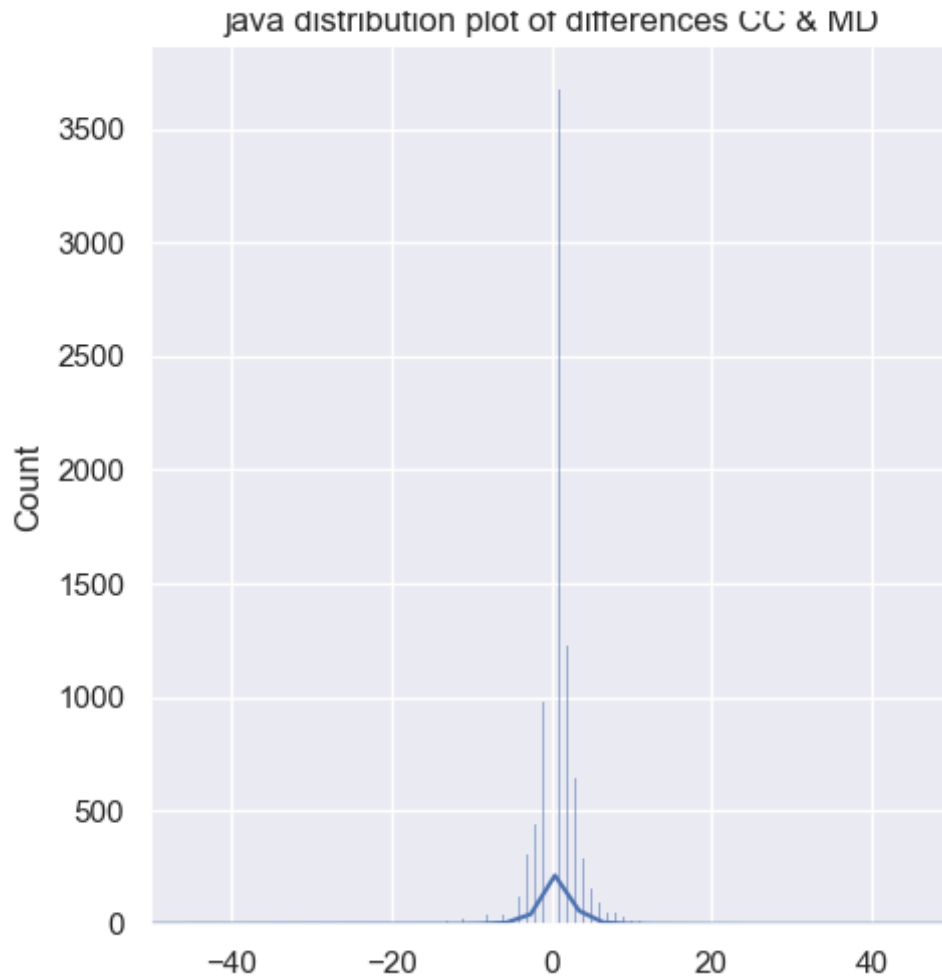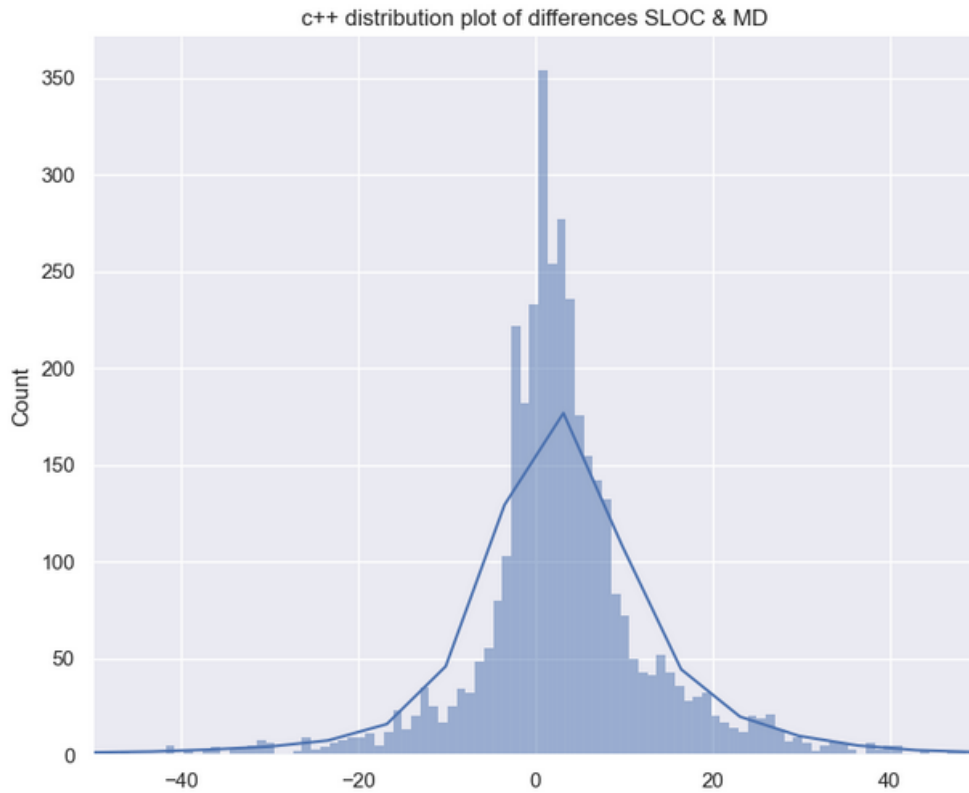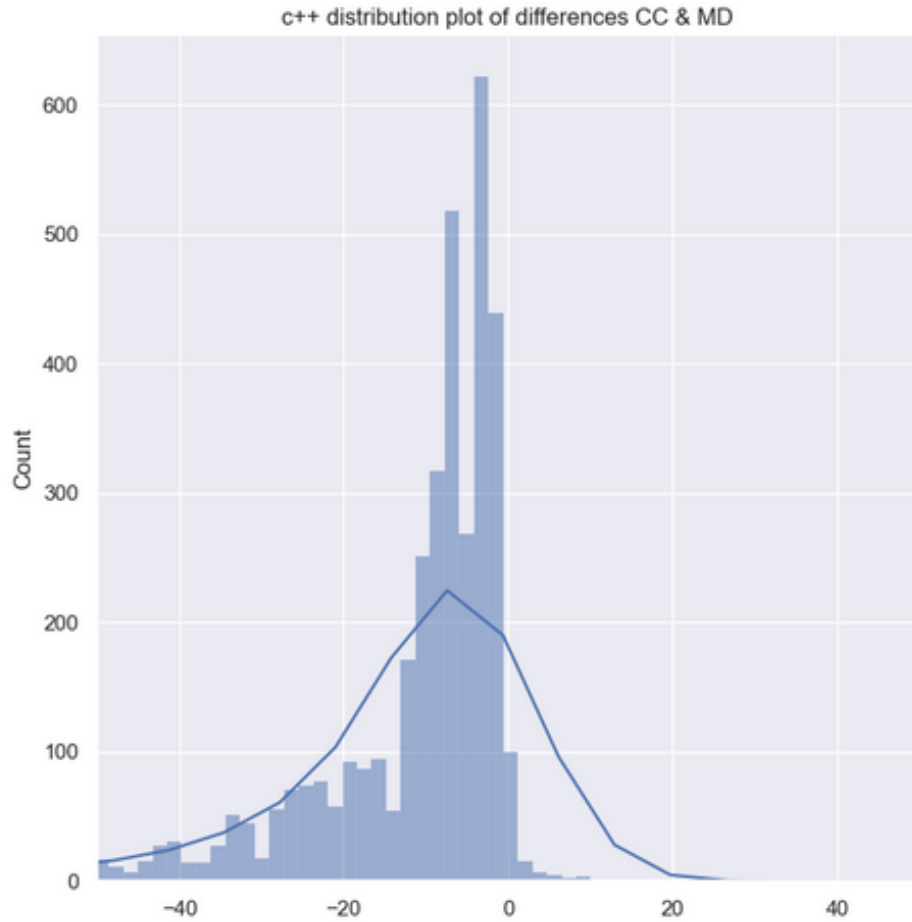
Figure 24: Distribution plot of mutant density subtracted from cyclomatic complexity where the x-axis is the difference and the y-axis the count

complete dataset is using basic outlier detection and verifying the differences between the two datasets.

These examples are based on the Java dataset, with Spring as the project that will be singled out. Generating outliers on average mutant density for the complete dataset gives 45 outliers that belong to the Spring project. Everything that is four standard deviations from the mean are considered outliers in this case. There are more outliers when doing this on the complete dataset but only 45 that belong to Spring. If the outliers are calculated based on Spring alone, this number increased to 76.

An important detail is that the increase of outliers does not cause a decrease of information. Every single one of those 45 outliers also occur in the 76 of only using the Spring project. This pattern is reoccurring, implying that using only a single project has no downsides. Staying in line with the original assumptions on this experiment, it is even safer to say that using more projects generalises and causes potential loss of information.

Looking more in depth on the cases that were added by only looking at Spring gives more insight on the situation. All the outliers that are added are relatively close to the edge of what was considered an outlier for this experiment. Every outlier that is close to five or even six times the standard deviation is caught in both scenarios. This further strengthens the idea that using more projects to use the mean and standard deviation from generalises. The cases visualized like the previous outliers can be seen in Fig.30. The orange cases are all the newly added outliers. As mentioned earlier, these are all very close to the edge of what can be considered an outliers so the usefulness is still a bit up to debate.

Since thresholding remains a difficult issue to address, it is also unsure whether these additional outliers con-

```java
private boolean isSockJsSpecialChar(char ch) {
    return (ch <= '\u001F') || (ch >= '\u200C' && ch <= '\u200F') ||
           (ch >= '\u2028' && ch <= '\u202F') || (ch >= '\u2060' && ch <= '\u206F') ||
           (ch >= '\uFFF0') || (ch >= '\uD800' && ch <= '\uDFFF');
}
```

```java
public int readInt(final int offset) {
    byte[] classBuffer = classFileBuffer;
    return ((classBuffer[offset] & 0xFF) << 24)
         | ((classBuffer[offset + 1] & 0xFF) << 16)
         | ((classBuffer[offset + 2] & 0xFF) << 8)
         | (classBuffer[offset + 3] & 0xFF);
}
```

Figure 25: Code example
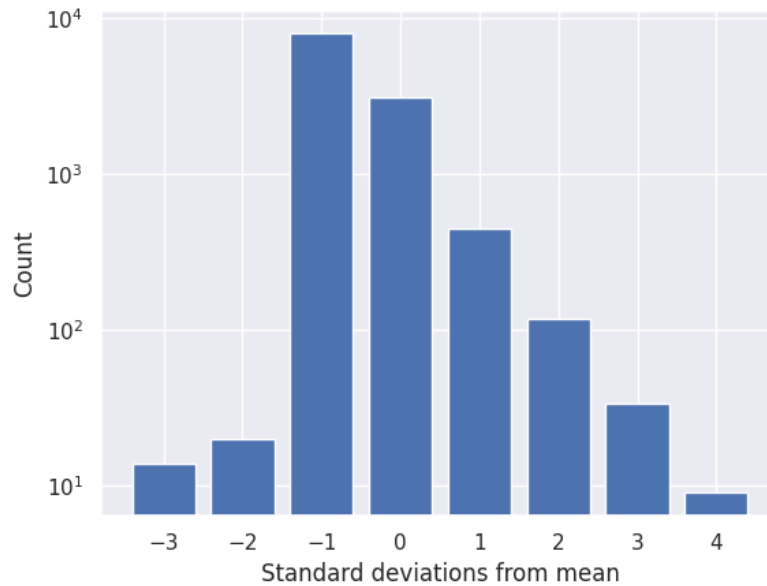
Figure 26: Distribution of outliers based on the difference between mutant density and source lines of code

tain the most useful information. Despite this, due to there not being any decrease in information, and using only a singular project reducing computational time, this is the approach from here on out.

One downside to this approach is that outliers are also more easily created due to the lack of more general data. If a project only has a few functions with a source lines of code above 50, these are much more prone to be outliers. Using a division by source lines of code and keeping the sizing of functions in mind reduces the impact of this. Still, even by using methods to reduce the influence of the sizing of methods, functions with higher source lines of code are more prone to be outliers. This is not a problematic occurrence. Functions that are too large in source lines of code compared to other functions within the same project might not be outliers in terms of cyclomatic complexity or mutant density, but the source lines of code still remains a complexity metric. Therefore, outliers based on source lines of code are more easily caught without specifically using this metric.
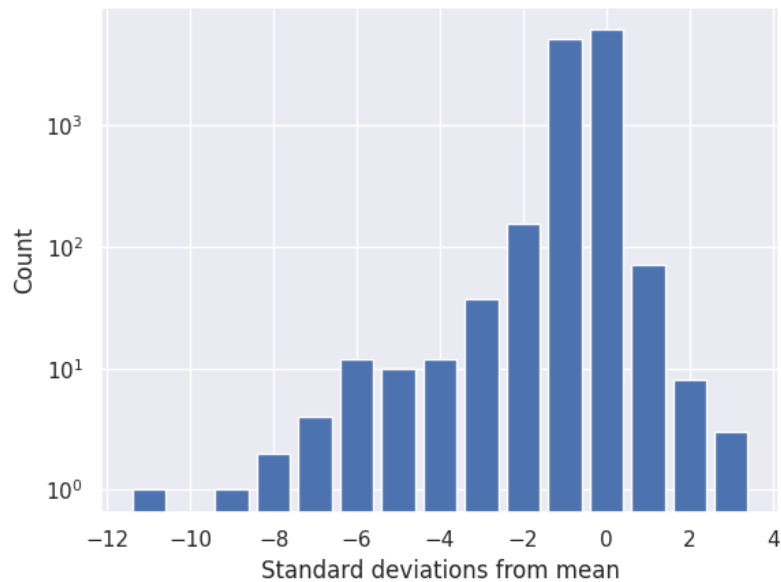
Figure 27: Distribution of outliers based on the difference between mutant density and cyclomatic complexity
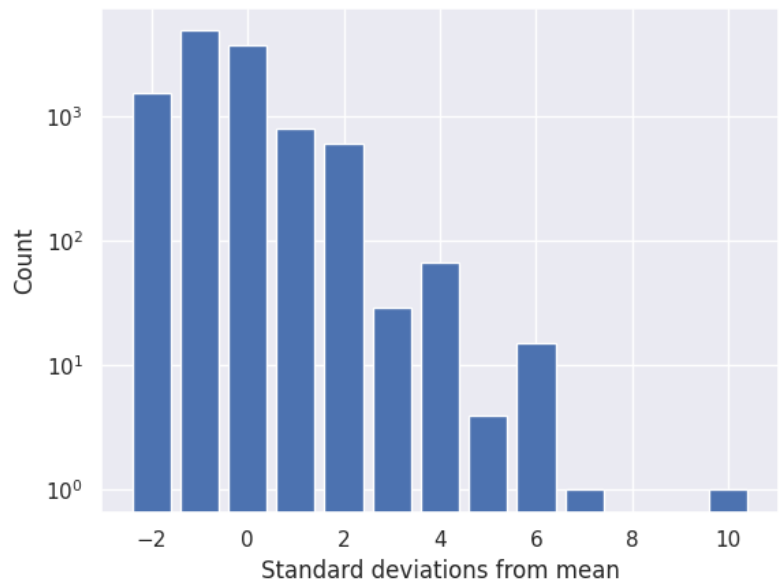
Figure 28: Distribution of outliers based on the average cyclomatic complexity

4.2.2 Basic Metric code examples

With the previous experiments finished, it is now more appropriate to start diving into code examples on the previously discussed average mutant density and average cyclomatic complexity. The following results are all based on the Spring project, the reasoning behind this choice is that it's a large project with a large amount of variation in the domains it covers. The first example is the function used in the background information. This *isSockJsSpecialChar*1 is a severe outlier for both average mutant density and average cyclomatic complexity. The z-score for both of these metrics is above 10. This outliers stands out the most, especially compared to other outliers for average cyclomatic complexity.
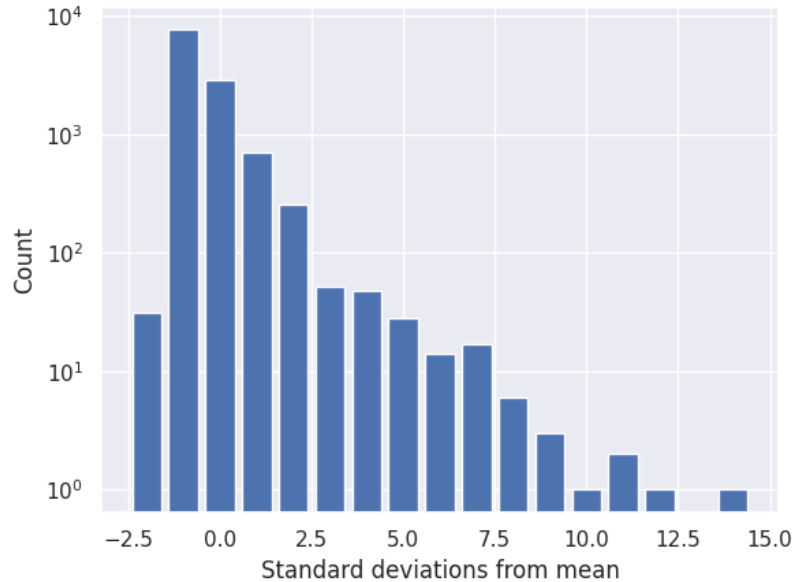
Figure 29: Distribution of outliers based on the average mutant density

```java
private boolean isSockJsSpecialChar(char ch) {
    return (ch <= '\u001F') ||
            (ch >= '\u200C' && ch <= '\u200F') ||
            (ch >= '\u2028' && ch <= '\u202F') ||
            (ch >= '\u2060' && ch <= '\u206F') ||
            (ch >= '\uFFF0') || (ch >= '\uD800' && ch <= '\uDFFF');
}
```

Listing 1: Highest outlier for average mutant density

The second highest outlier for this metric has a z-score of 7.2, this is quite a large gap with the z-score of 10 but with the examples this gap is quite obvious.

The clear issue with the *isSockJsSpecialChar* is immediately visible. There are too many possible operations, too many conditional operators causing a high complexity on multiple aspects. In earlier sections it was mentioned that small functions like these, are not as big of an issue as it seems. It is written too complex and could certainly be refactored and simplified, but it still remains a small function separated from other functionality. This encapsulation of extremely problematic functionality might simply be caused by the circumstances of usage, for example this calculation could be called at multiple locations. It could also be a conscious decision to create this encapsulation but despite the cause, it remains a fairly reasonable refactoring opportunity.

The *matches*2 function is the second highest outlier for average mutant density. This is for very similar reasons as the previous function, a long statement on a single lined function. There are less operators increasing cyclomatic complexity so it is not an outlier for that metric. Again, this is a very small function and most other cases with a high z-score based on average mutant density and cyclomatic complexity are fairly small.

Another issue that is also visible through these metrics is that there is no filtering for test functions. This results in a few test functions appearing as outliers. Test functions are also slightly different in structure compared to normal methods, they often do not follow the same clean code aspects. This also means that the test functions are used for the calculation of the mean and the standard deviation. This is a relatively small issue but a detail to be kept in mind for further improvements and fine-tuning. These functions might influence the mean and standard deviation calculation but in a very limited sense. Especially since testing functions are in most projects much less present compared to normal functions.
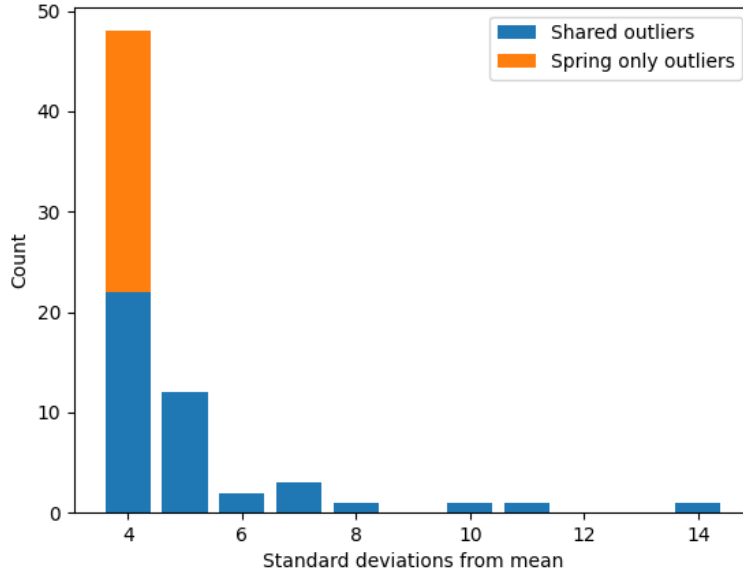
Figure 30: Distribution of outliers based on the difference between mutant density which shows the unique cases over a using a single project or a dataset of projects

```
public boolean matches(Method m, @Nullable Class<?> targetClass) {
    return m.getName().equals("overload") &&
            m.getParameterCount() == 1 &&
            m.getParameterTypes()[0].equals(int.class);
}
```

Listing 2: Second highest outlier for average mutant density

4.2.3 Scaled outliers

With the scaled metric, it is expected that functions which contain problematic patterns combined with larger source lines of codes are shown. This means that the worst rated methods might not be seen as the worst code, but they contain patterns that combined with size cause a problematic result. This does however drastically reduce the amount of outliers that are found with a z-score higher than 4. For the scaled average cyclomatic complexity this gives only 2 outliers and on scaled average mutant density this decrease is less drastic. This change in distribution is clearly visible in Fig.**??** where the distribution is much more limited than the previous Fig.28. A first example of how this influences the obtained functions can be seen in Fig.31. For the Spring project, this is the worst rated function after using a scaled version of the average mutant density. At first sight it does not look like a terribly written method, especially compared to the previous shorter examples. However, with more detailed inspection, it is clearly visible that it contains more potential mutations as well as more if statements. What mostly influences the scoring right now is the additional scaling that has been done. Due to the size of the function, and the higher than average cyclomatic complexity and mutant density, it becomes a prime target for refactoring. The smaller functions still have a relatively large z-score since they were such large outliers to begin with but this remains acceptable. The small functions are not expected to be completely dropped through the addition of scaling. A better balance in the given outliers is the main goal which the is clearly demonstrated by the obtained outliers.

Since there are only two outliers for cyclomatic complexity after the scaling with a score higher than 4, choosing one is not that hard. The first function is the *getOpcode* 3 function. It is an extremely large switch case statement that clearly could use some simplification. Large switch case statements like these often exist for some sort of parser. This also means that it is not always extremely problematic or avoidable. These types of outliers are extremely similar to the very small functions in terms of how they should be interpreted. Despite being outliers, not every outlier is directly a

```java
public int getOpcode(final int opcode) {
    if (opcode == Opcodes.IALOAD || opcode == Opcodes.IASTORE) {
        switch (sort) {
            case BOOLEAN:
            case BYTE:
                return opcode + (Opcodes.BALOAD - Opcodes.IALOAD);
            case CHAR:
                return opcode + (Opcodes.CALOAD - Opcodes.IALOAD);
            case SHORT:
                return opcode + (Opcodes.SALOAD - Opcodes.IALOAD);
            case INT:
                return opcode;
            case FLOAT:
                return opcode + (Opcodes.FALOAD - Opcodes.IALOAD);
            case LONG:
                return opcode + (Opcodes.LALOAD - Opcodes.IALOAD);
            case DOUBLE:
                return opcode + (Opcodes.DALOAD - Opcodes.IALOAD);
            case ARRAY:
            case OBJECT:
            case INTERNAL:
                return opcode + (Opcodes.AALOAD - Opcodes.IALOAD);
            case METHOD:
            case VOID:
                throw new UnsupportedOperationException();
            default:
                throw new AssertionError();
        }
    } else {
        switch (sort) {
            case VOID:
                if (opcode != Opcodes.IRETURN) {
                    throw new UnsupportedOperationException();
                }
                return Opcodes.RETURN;
            case BOOLEAN:
            case BYTE:
            case CHAR:
            case SHORT:
            case INT:
                return opcode;
            case FLOAT:
                return opcode + (Opcodes.FRETURN - Opcodes.IRETURN);
            case LONG:
                return opcode + (Opcodes.LRETURN - Opcodes.IRETURN);
            case DOUBLE:
                return opcode + (Opcodes.DRETURN - Opcodes.IRETURN);
            case ARRAY:
            case OBJECT:
            case INTERNAL:
                if (opcode != Opcodes.ILOAD && opcode != Opcodes.ISTORE && opcode != Opcodes.IRETURN) {
                    throw new UnsupportedOperationException();
                }
                return opcode + (Opcodes.ARETURN - Opcodes.IRETURN);
            case METHOD:
                throw new UnsupportedOperationException();
            default:
                throw new AssertionError();
        }
    }
}
```

Listing 3: Average cyclomatic complexity highest outlier

```java
final boolean merge(
    final SymbolTable symbolTable, final Frame dstFrame, final int catchTypeIndex) {
    boolean frameChanged = false;
    int numLocal = inputLocals.length;
    int numStack = inputStack.length;
    if (dstFrame.inputLocals == null) {
    dstFrame.inputLocals = new int[numLocal];
    frameChanged = true;
    }
    for (int i = 0; i < numLocal; ++i) {
    int concreteOutputType;
    if (outputLocals != null && i < outputLocals.length) {
        int abstractOutputType = outputLocals[i];
        if (abstractOutputType == 0) {
        concreteOutputType = inputLocals[i];
        } else {
        concreteOutputType = getConcreteOutputType(abstractOutputType, numStack);
        }
    } else {
        concreteOutputType = inputLocals[i];
    }

    if (initializations != null) {
        concreteOutputType = getInitializedType(symbolTable, concreteOutputType);
    }
    frameChanged |= merge(symbolTable, concreteOutputType, dstFrame.inputLocals, i);
    }

    if (catchTypeIndex > 0) {
    for (int i = 0; i < numLocal; ++i) {
        frameChanged |= merge(symbolTable, inputLocals[i], dstFrame.inputLocals, i);
    }
    if (dstFrame.inputStack == null) {
        dstFrame.inputStack = new int[1];
        frameChanged = true;
    }
    frameChanged |= merge(symbolTable, catchTypeIndex, dstFrame.inputStack, 0);
    return frameChanged;
    }
```

Figure 31: Code example of scaled average mutant density

bad function due to the functionality it implements. Enforcing refactoring on these might lead to solutions that could negatively influence maintainability and readability. Examples like this make it very clear that despite the usage of these metrics, it is always important to keep the context of implementation in mind. Certain types of complexity are simply unavoidable or have bigger upsides than downsides.

If the style of the outliers are closer to the example of mutant density, looking more like a standard implementation but higher complexity due to size inclusion, it would be more interesting. Not every project will necessarily have functions like these. Still, this does not mean this is a bad example, the function still contains an if else statement which could be separated in two different functions. Even though this would most likely create two outliers instead of one, the deviation from the mean would not be as large and it might increase readability.
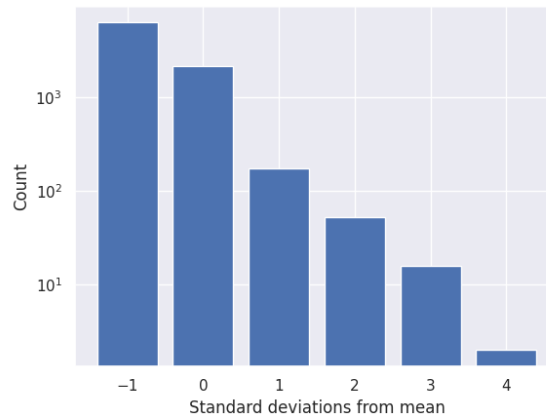
Figure 32: Distribution of outliers based on the scaled average cyclomatic complexity

The second outlier is closer to the mutant density example. It is an very large function that does not contain switch statements, the complexity purely comes from the size combined with a high amount of if else statements. This creates another very good example about the importance of the addition of the scaling.

4.2.4 Cross metric & unique cases

The first steps of looking at the relation between outliers for mutant density and cyclomatic complexity is using set theory as explained earlier. As for the metrics, the previous scaled metrics are going to be used. One small variation is that instead of only taking from four standard deviations as outliers, this will be decreased to two. Four leaves too little outliers to apply set theory and after applying the scaling, two is still a reasonable cutoff. Let's go back to the visualization of how the outliers are distributed after applying the scaling. In Fig.32 it shows that two does not have too much cases but also not too little. Compared to the same graphs when there is no scaling applied, a cutoff of two would be too low and four is a better option. Due to the shifting of the general distribution, two is a reasonable option now. Using this cutoff value provides 168 outliers based on mutant density and 70 based on cyclomatic complexity. This difference in amount of outliers is not new information. The larger possible variation on mutant density with higher source lines of code is the main cause for this. These two sets have an intersection of 46 cases, this is a large part of the cyclomatic complexity set. This means that a good portion is covered through mutant density but that the unique information still remains. Looking at the unique cases based on mutant density there are two general types of functions that are obtained. The first type is shown by Fig.**??**, these functions are relatively average looking and still contain cyclomatic complexity. The cyclomatic complexity is however not enough to raise it as an outlier for that metric. The second type of functions are similar to Fig.34. These are extremely short functions that often include a very small cyclomatic complexity or completely none. The same idea about short functions as mentioned earlier applies. Then there are the type of functions that are only outliers for cyclomatic complexity. The following function might seem extremely similar to the previous large switch case statement but there is a fundamental difference. This *invoke*4 function has a lot less potential mutation operators which causes it to only be an outlier for cyclomatic complexity.

 Now the second method to find the relation between these metrics is by using the scaled average difference between cyclomatic complexity and mutant density. The distribution of outliers can be visualized as Fig.35. With the cutoff still being at -2 or 2, this provides 156 outliers based on this metric. The important question for the results of this metric is how many outliers are found that are not found by the scaled average mutant density and cyclomatic complexity. There are 97 of those 156 outliers that are also found by these two other metrics. If we remove all similarities between the three metrics, only 59 completely new outliers are found. All of these outliers have a relatively low z-score, it still falls outside the cutoff value but they mostly fall between 2 and 4. This means that more extreme cases are already caught by the usage of only a single metric. Does this mean the metric lacks in performance? Not really, it still gives a more precise reasoning on why something is an outlier. The previous metrics had no notion of the direct relation between cyclomatic complexity or mutant density, now it can be shown whether an outlier is caused by a proportionally higher mutant density or cyclomatic complexity.

The functions that are unique with a slightly lower z-score are still interesting. A first example is this *are-*

```java
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    switch (method.getName()) {
        case "equals":
            return (proxy == args[0]);
        case "hashCode":
            return System.identityHashCode(proxy);
        case "getTargetConnection":
            return getTargetConnection(method);
        case "unwrap":
            if (((Class<?>) args[0]).isInstance(proxy)) {
                return proxy;
            }
            break;
        case "isWrapperFor":
            if (((Class<?>) args[0]).isInstance(proxy)) {
                return true;
            }
            break;
    }
    if (!hasTargetConnection()) {
        switch (method.getName()) {
            case "toString":
                return "Lazy Connection proxy for target DataSource [" + getTargetDataSource() + "]";
            case "getAutoCommit":
                if (this.autoCommit != null) {
                    return this.autoCommit;
                }
                break;
            case "setAutoCommit":
                this.autoCommit = (Boolean) args[0];
                return null;
            case "getTransactionIsolation":
                if (this.transactionIsolation != null) {
                    return this.transactionIsolation;
                }
                break;
            case "setTransactionIsolation":
                this.transactionIsolation = (Integer) args[0];
                return null;
            case "isReadOnly":
                return this.readOnly;
            case "setReadOnly":
                this.readOnly = (Boolean) args[0];
                return null;
            case "getHoldability":
                return this.holdability;
            case "setHoldability":
                this.holdability = (Integer) args[0];
                return null;
            case "commit":
            case "rollback":
                return null;
            case "getWarnings":
            case "clearWarnings":
                return null;
            case "close":
                this.closed = true;
                return null;
            case "isClosed":
                return this.closed;
            default:
                if (this.closed) {
                    throw new SQLException("Illegal operation: connection is closed");
                }
        }
    }
    try { return method.invoke(getTargetConnection(method), args); }
    catch (InvocationTargetException ex) {
        throw ex.getTargetException();
    }
}
```

```
public Resource findLocalizedResource(String name, String extension, @Nullable Locale locale) {
    Assert.notNull(name, "Name must not be null");
    Assert.notNull(extension, "Extension must not be null");

    Resource resource = null;

    if (locale != null) {
        String lang = locale.getLanguage();
        String country = locale.getCountry();
        String variant = locale.getVariant();

        if (variant.length() > 0) {
            String location =
                    name + this.separator + lang + this.separator + country + this.separator + variant + extension;
            resource = this.resourceLoader.getResource(location);
        }

        if ((resource == null || !resource.exists()) && country.length() > 0) {
            String location = name + this.separator + lang + this.separator + country + extension;
            resource = this.resourceLoader.getResource(location);
        }

        if ((resource == null || !resource.exists()) && lang.length() > 0) {
            String location = name + this.separator + lang + extension;
            resource = this.resourceLoader.getResource(location);
        }
    }

    if (resource == null || !resource.exists()) {
        String location = name + extension;
        resource = this.resourceLoader.getResource(location);
    }

    return resource;
}
```

Figure 33: Code example of unique information from average mutant density

```
public boolean isOverridable() {
    return !isStatic() && !isFinal() && !isPrivate();
}
```

Figure 34: Code example of unique information from average mutant density

*BoxingCompatible*5 function which is an outlier due to a much higher cyclomatic complexity than mutant density. This is mainly due to a single reason. Often the conditional part of an if statement contains the possibility for quite some mutations. Most conditional statements of this function consist of a single true or false statement. On top of that, the bodies of those if else statements contain very few lines of code which also reduce the possible mutations. Due to these reasons, this is an outlier and also an interesting function for this case. This style of functions is very close to previous examples containing large switch case statements, therefore the same argumentation remains that not all of these cases will correspond to refactoring opportunities. This means that a proportionally higher cyclomatic complexity is not always a direct indication of the need for refactoring. A second example is this *nextPowerOf2*6 function which was an outlier due to the higher mutant density compared to cyclomatic complexity. This function is quite a bit shorter but it is
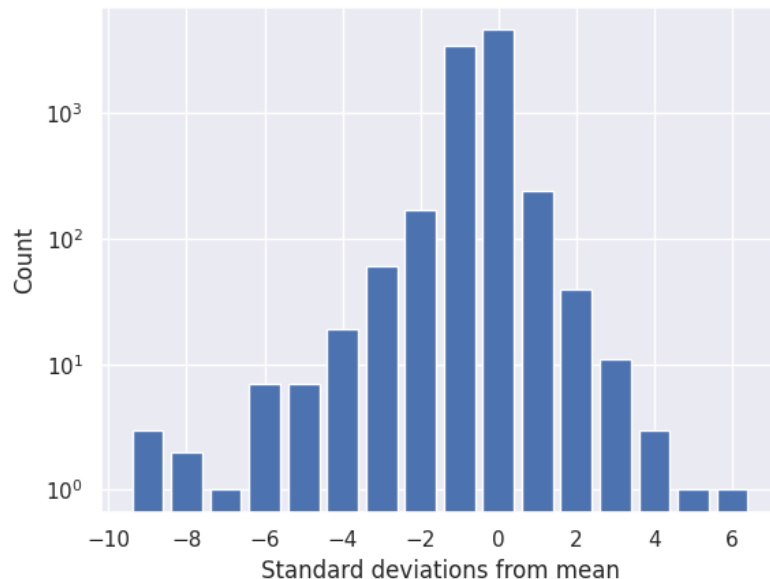
46

Figure 35: Distribution of outliers based on the scaled average CC & MD

still clear why this is an outlier. There is no code that increases the cyclomatic complexity at all. Lacking cyclomatic complexity is not a bad indication, some functions simply need to execute a certain amount of operations. Unlike the previous function, it is easier to argue that a higher mutant density compared to cyclomatic complexity remains a good refactoring target. The example function has clear duplication which could be simplified. If functions like this occur with large source lines of code, it would be even better to provide these as refactoring targets.

This metric does have a downside causing it not be a perfect fit for everything. Since it relies on the difference between cyclomatic complexity and mutant density, if this distance is not present, there won't be an outlier. Even if both of these metrics are extremely high, as long as the difference is smaller than a method will not be detected. Therefor the previous metrics retain their value as a metric since this metric emphasises other aspects. Another flaw was shown by the *areBoxingCompatible* function. It is an outlier and for a clear reason but this does not make it a good target. Filtering these cases is also not that easy, at a certain point, a higher cyclomatic complexity might mean a good refactoring target even with a low mutant density. Deciding when these could be seen as refactoring opportunities is a non trivial issue.

How do these results answer **RQ2. Complementary - how can we utilize mutant density to provide additional insight into refactoring opportunities? Is there anecdotal evidence supporting this?**. The previous results have shown different aspects of the usage of mutant density and how it could possibly be applied. Although combining mutant density and cyclomatic complexity into a singular metric provides new interesting information, it leaves out other aspects and is fairly hard to filter. This does not mean that it is impossible to do, just within the scope of this work this is not further experimented upon. The most interesting results are obtained by utilizing the scaled average mutant density and cyclomatic complexity. They provide a good insight into refactoring opportunities taking factors such as function size into account. The set theory experimentation also proved that outside of the correlation between the metrics, there is also a solid similarity between the two. This strengthens the idea of using mutant density in a complementary manner. The anecdotal evidence is shown by the code examples, these examples also provide the same insight when revisiting them from the perspective of the DevOps tooling. As an example the previous *nextPowerOf2* function will be used, it has an original mutant density of 12 and a cyclomatic complexity of 1. CodeScene does not recommend any changes to this function based on it's metrics or on based on cyclomatic complexity. Refactoring this function as *nextPowerOf2_refactored*heavily reduces the potential mutations by moving the reoccurring operation to a separate function. This changes nothing in terms of cyclomatic complexity but it does improve other aspects which are not covered. This example is very small in size, it might not even be worth looking at functions of this size but the principles remain the same. Even if it would be contained within a larger function, functionality such as *nextPowerOf2* is not viewed by cyclomatic complexity. Codescene also frequently rely on other factors such as large conditionals and

47

```java
public static boolean areBoxingCompatible(String desc1, String desc2) {
    if (desc1.equals(desc2)) {
        return true;
    }
    if (desc1.length() == 1) {
        if (desc1.equals("Z")) {
            return desc2.equals("Ljava/lang/Boolean");
        }
        else if (desc1.equals("D")) {
            return desc2.equals("Ljava/lang/Double");
        }
        else if (desc1.equals("F")) {
            return desc2.equals("Ljava/lang/Float");
        }
        else if (desc1.equals("I")) {
            return desc2.equals("Ljava/lang/Integer");
        }
        else if (desc1.equals("J")) {
            return desc2.equals("Ljava/lang/Long");
        }
    }
    else if (desc2.length() == 1) {
        if (desc2.equals("Z")) {
            return desc1.equals("Ljava/lang/Boolean");
        }
        else if (desc2.equals("D")) {
            return desc1.equals("Ljava/lang/Double");
        }
        else if (desc2.equals("F")) {
            return desc1.equals("Ljava/lang/Float");
        }
        else if (desc2.equals("I")) {
            return desc1.equals("Ljava/lang/Integer");
        }
        else if (desc2.equals("J")) {
            return desc1.equals("Ljava/lang/Long");
        }
    }
    return false;
}
```

Listing 5: Outlier based on a higher than average cyclomatic complexity compared to mutant density

```java
private static int nextPowerOf2(int val) {
    val--;
    val = (val >> 1) | val;
    val = (val >> 2) | val;
    val = (val >> 4) | val;
    val = (val >> 8) | val;
    val = (val >> 16) | val;
    val++;
    return val;
}
```

Listing 6: Outlier based on a higher than average mutant density compared to cyclomatic complexity

```java
private static int nextPowerOf2_refactored(int val) {
    val--;
    val = val_operation(val, 1);
    val = val_operation(val, 2);
    val = val_operation(val, 4);
    val = val_operation(val, 8);
    val = val_operation(val, 16);
    val++;
    return val;
}

private static int val_operation(int val, int shift) {
    return (val >> shift) | val;
}
```

Listing 7: Refactoring of nextPowerOf2

nested blocks to determine refactoring opportunities. Both of these rely on the presence of statements that also increase the cyclomatic complexity. This implies that also viewing complexity through the present operator is a viable way to analyze refactoring opportunities.

4.2.5 Operator specific

The last results that need to be covered are those for operator specific operators which correspond to **RQ3. Fine Tuning - what is the impact of fine-tuning mutant density on correlation and complementary?**. Due to the easier tooling in C and C++ using dextool, the results for this data is not generated on Java like previous cases. This might cause variation but the general takeaway from the results will remain the same.

Looking at the amount of outliers for each single mutation operator in Table 9, there are a few clear differences. Certain specific mutation operators have a much larger presence than others, this can be seen in the amount of total mutations and in the amount of methods that are influenced. Out of this data UOI (Unary operator insertion) has the least presence while DCR (Decision/Condition Requirement) and SDL (Statement Deletion) have the same presence. Despite this difference in presence of mutation operators, Dextool notes that a combination of ABS, UOI, LCR, AOR and ROR reaches a coverage of 99,5%. This means that despite the low presence of UOI, it still covers possible mutations that other metrics don't. Before looking at unique outliers, there are 24 overarching outliers. The following *ReadTagFallback*

Table 9: Mutation Statistics

Mutations	Methods	Total Mutants
rorp	958	5920
aor	628	7906
dcr	2946	20776
cr	1449	5877
lcr	692	6004
aors	641	2051
sdl	2946	20776
ror	958	5920
lcrb	441	4198
uoi	422	829

8 function is an example of those cases.

```
uint32_t CodedInputStream::ReadTagFallback(uint32_t first_byte_or_zero) {
  const int buf_size = BufferSize();
  if (buf_size >= kMaxVarintBytes ||
      (buf_size > 0 && !(buffer_end_[-1] & 0x80))) {
    GOOGLE_DCHECK_EQ(first_byte_or_zero, buffer_[0]);
    if (first_byte_or_zero == 0) {
      ++buffer_;
      return 0;
    }
    uint32_t tag;
    ::std::pair<bool, const uint8_t*> p =
        ReadVarint32FromArray(first_byte_or_zero, buffer_, &tag);
    if (!p.first) {
      return 0;
    }
    buffer_ = p.second;
    return tag;
  } else {
    if ((buf_size == 0) &&
        ((buffer_size_after_limit_ > 0) ||
         (total_bytes_read_ == current_limit_)) &&
        total_bytes_read_ - buffer_size_after_limit_ < total_bytes_limit_) {
      legitimate_message_end_ = true;
      return 0;
    }
    return ReadTagSlow();
  }
}
```

Listing 8: Outlier over every single mutation operator

It is immediately clear that this function contains a high amount of possible mutations due to the presence of many different operators. There is also a good amount of variation visible in the types of operators, thus it being an outlier over every mutation operator. On top of that, these outliers are based on average mutant density, there is no additional scaling for the sizing of the functions. Despite there not being any scaling for this, all of the functions in this category have a size large enough to consider them good refactoring targets. This is caused by the structure of this experiment. Being an outlier over a multitude of different mutation operators requires a fair amount of operators in the code that conform to each of the types. This is less feasible with smaller sized functions, directly causing the functions the increase in size without specifically including this in the metric.

Looking at the outliers that are unique for a single mutation operators is a very different way of working then the previous experiment. In Table 10 it is shown how many unique outliers there are for each mutation operator. Most functions have at least some form of overlap in terms of mutation operators. Going from multiple thousand mutations, only a very few remain when looking at pure unique cases. Most operators do not even have unique cases remaining, this might not be optimal since some operators also have overlap but it is how it will be interpreted for now. What is interesting is how most of the unique cases are for operators that do not have a very large coverage. The mutation operators with less coverage are more likely to be more different to other thus causing more potential unique cases.

There are not that many code example too choose from, the examples can also be generally divided in two groups. One of the examples is the following *opj_jp2_start_copmress* 9 function, obtained as an outliers of UOI.

This function is not extremely small and contains a clear limit on the types of operators it contains. Therefore, functions such as these could be placed in the first category, medium sized functions with a simple limit on the operators. Note that there are seemingly very little operators in this function. UOI has a much smaller reach and potential targets in comparison to other operators as shown earlier. Since the threshold for calculating the outliers are now based on singular metrics, this means that the mean and standard deviation vary a lot between the different operators. This information is given in Table 11, where the difference is clearly visible. A mutation operator with more reach does not directly correlate to higher means or standard deviations. DCR has the highest amount of mutations but a relatively low mean and standard deviation. In the case of UOI the mean is very low, outliers are thus much more easily created.

Table 10: Single Mutation Outliers

Mutations	Outliers
rorp	0
aor	0
dcr	0
cr	30
lcr	8
aors	3
sdl	0
ror	0
lcrb	0
uoi	24

Mutations	Mean	Standard Deviation	Total Mutants
rorp	0.433	0.492	5920
aor	0.803	0.833	7906
dcr	0.544	0.410	20776
cr	0.329	1.461	5877
lcr	0.432	0.392	6004
aors	0.204	0.211	2051
sdl	0.544	0.410	20776
ror	0.433	0.491	5920
lcrb	0.380	0.291	4198
uoi	0.087	0.179	829

Table 11: Table of mutations with mean and standard deviation

The second type of functions obtained through this experiment are extremely small functions with a few operators of the corresponding type. When looking at methods that are outliers for each method, this mostly gave larger functions as explained earlier. It is not unexpected that this experiment provides opposite results due to its opposite approach.

How does this answer **RQ3. Fine Tuning - what is the impact of fine-tuning mutant density on correlation and complementary?** The impact of fine-tuning within this work is limited due to the taken approach. Instead of analyzing specific combinations of mutation operators, this work focused on two more defined scenarios. The most interesting part of these experiments is using outliers that occur for each individual mutation operator. It does not directly benefit from the modularity of the metric in a way that was originally expected but it does provide some direction for future attempts at this. Instead of calculating outliers that occur over each mutation operator, it might be possible to create a metric that is used the amount of mutation operators contain an outlier to give a more defined way of finding refactoring opportunities.

```
OPJ_BOOL opj_jp2_start_compress(opj_jp2_t *jp2,
                    opj_stream_private_t *stream,
                    opj_image_t * p_image,
                    opj_event_mgr_t * p_manager
                    )
{
assert(jp2 != 00);
assert(stream != 00);
assert(p_manager != 00);

    if (! opj_jp2_setup_encoding_validation(jp2, p_manager)) {
        return OPJ_FALSE;
    }

    if (! opj_jp2_exec(jp2, jp2->m_validation_list, stream, p_manager)) {
        return OPJ_FALSE;
    }

    if (! opj_jp2_setup_header_writing(jp2, p_manager)) {
        return OPJ_FALSE;
    }

    if (! opj_jp2_exec(jp2, jp2->m_procedure_list, stream, p_manager)) {
        return OPJ_FALSE;
    }

    return opj_j2k_start_compress(jp2->j2k, stream, p_image, p_manager);
}
```

Listing 9: Outlier for only a single mutation operator (Uoi)

5 Threats to Validity

5.1 Conclusion validity

The conclusion validity discusses the degree to which the conclusions that are reached are reasonable. This type of threat is present on few different aspects that have already been discussed throughout the work. The selection of projects used for the experimentation was not done blindly, but also not with an extremely thorough procedure. On top of that, due to the limited sizing of the datasets used, choosing the projects itself is still fairly important. This does not mean that the results from the available data are faulty. Due to the focus on singular projects, the outliers are valid within the context of that singular project. The patterns described are however uncertain to be present over different projects. These concerns have been mentioned by examining the differences over different programming languages.

The choice of outlier detection method might also be a potential threat. This work opted for the usage of a z-score for this purpose. Most of the data used for these experiments do not have a completely standard distribution. It might be generally close but there are still clear differences present. The methods obtained by using the z-score for outliers did align with the expectations in terms of code complexity. Despite this, it was very difficult to concretely determine cut-off values for a z-score. Because of these reasons, it might not be a completely optimal method for detecting outliers and influence the results slightly. Large outliers and the most extreme methods will certainly be found but the issues might occur more close to the cut-off value.

5.2 Internal validity

The internal validity discusses whether it is reasonable to draw a link between the approach and the obtained results. The same issues surrounding the dataset usage apply to internal validity. The similarity between the results from cyclomatic commmplexity and mutant density provided an initial form of internal validity. All of these results also fell in line with initial expectations. Across different languages, there is still a gap between the results.

This could be a threat to the internal validity. The provided differences between Java and C are quite significant. In their own right, these results would be seemingly fine. Compared to each other, the results differ quite a bit causing some form of uncertainty. It is uncertainty whether the chosen approach is faulty and works for Java by chance or if the results for C are closer to the real results. It has been attempted to provide argumentation for these differences. The assumption still remains that this is not based on the approach but that the scale of data used is not enough for complete verification on the issues.

5.3 Construct validity

The construct validity discusses whether the operational measures reflect the goal and intentions of this work. In this work, there a few different aspects which deviated from the original expectations and intentions. Most of these aspects occurred in what was expected to be the preparational phase. More specifically, this concerns the creation of the dataset and getting functional tooling to obtain all the metrics needed for the experiments. Issues surrounding the tooling also directly influenced the creation of the dataset. Mutant density is not a widely adapted metric, on top of that, the calculation of this metric is not nearly as simple as source lines of code and cyclomatic complexity. As mentioned in the tooling section, the only real tool available for the direct calculation of mutant density is Chaosmeter. There are no other tools that provide this functionality for other languages, causing the need for new or altered tools.

New tools falls completely out of scope, although improved tooling would be at the core of potential future work, this is not the intention of this work. Altered tooling was thus the only other option, this is however not without any challenges. Tools for mutation testing exist for different languages since this is a slightly more common procedure. However, not every tool is open source, decently functioning or written in a way that it can be altered for different purposes. Trying different tools that were able to run on the machines available took a lot more time than originally anticipated. The tools described which were used for all the experiments function mostly at this point but there are still remaining flaws that were not covered due to time constraints.

Creating datasets was also very time consuming due to issues with the tooling. Mutpy, after the alterations, still had some functionality in place which caused the need for all the requirements for the project to be in place. This functionality is needed when running the mutation tests but not for the general calculation of the amount of mutations. This is an issue that was not solved since it took more time than intended which needed to be spent on generating and analyzing the results. A similar issue is present for Dextool, the projects that need to be analyzed for this tool need to be able to be build on your machine. In the case of python, is this more easy to work around but for C and C++ this is not always an easy task. Looking through github for opensource C and C++ projects that are sufficient in size and easily

compiled on the machine used was an extremely time consuming task. Larger C and C++ projects also take quite some time to build, this implies that issues with a chosen project might only arise later in the building process after already spending quite some time on this. At a certain point the choice was made to continue with the data that was available at the time instead of continuing to finetune everything surrounding the tooling.

5.4   External validity

The external validity discusses the extent to which the results of this work can be generalized. For the research that is done in this work, this applies mostly to generalizing across different languages and different types of projects. The aspect of applying the results over different languages and projects has been covered from the beginning of the project. Especially at the start, there is coverage over Java, C, C++ and Python. Within those languages there is also some variation within the selected projects. Although throughout the work more emphasis is placed on Java, results from other languages that differ from this are discussed.

This work tries to provide a foundation for working with mutant density alongside cyclomatic complexity. Based on the results, there is additional information that can be obtained through the approaches that are mentioned. There are however multiple aspects that hinder or influence the possible generalization of this work.

Choosing the dataset within a language is the first aspect which influences this. In earlier sections, there has been experimentation on how working with a single project or a set of projects influences the results. If working with a set of projects were the best option, this would prompt additional research in the selection of those projects. It was shown however that for the data provided, there was no loss of information when only working with singular projects. This approach was therefore used in further experiments. Using only a singular project removes the influence from selecting the projects partially. Complete removal of this influence is however not possible by only using a singular project. Applying metrics such as cyclomatic complexity and mutant density which are directly based on specific parts of the code is always effected heavily by the project in question. Projects containing a more mathematical solution to a problem have higher potential mutant density compared to those that provide a completely different type of solution. This is however, not necessarily something that hinders generalization. Since the outlier detection is based on the same project, more mathematical functions in a project will also influence the mean and standard deviation. Therefore, the influence is limited on this aspect.

What could be more influential is the composition of the projects themselves. What is the balance of mathematical methods compared to those that have only function calls for example? An imbalanced distribution between types of methods might hide a specific outliers. If there is an abnormally large presence of mathematical methods, the mean would accommodate this. The presence of functions with a very low amount of mathematical operations are thus much more likely to be considered outliers based on the current approach. Although the distribution of methods in projects are not likely to behave this way, it remains an influential aspect. The distribution of methods within a project has been shown in the previous section and the patterns in these distributions remains the same. The sizing of the dataset is still too small to be absolutely certain about this aspect. Again, this does not directly hinder the generalization, it is more of a reminder of the potential variation that might happen. This specific issue could be addressed in two general ways. The first one would be to investigate some form of thresholding such as those applied for cyclomatic complexity. The second method is more in depth analysis of internal project structures and how it influences the results of applying specific metrics.

The largest potential threat against generalization is choosing the programming language on which the results are applied. In earlier sections, it was already mentioned that there is a difference in the distribution of mutant density over different languages. The same goes for other metrics, especially something like source lines of code. Average mutant density uses both of these metrics and the variation of languages could have a potentially large influence. Over all the languages that are covered in this work the difference is very noticeable. Languages that are not covered in this work might have even more variation. Older languages such as Fortran could give completely different results. The following statements are purely assumptions since the data generated from this work is not sufficient and broad enough to be completely certain.

It is expected that this issue is partially countered by the nature of mutant density itself. The increase in mutant density variation when moving from Java to C or C++, was argued to be caused by the complexity of the languages themselves. Languages such as Java have more built in operators that might not always exist in older or lower level languages. The need for more operations to reach the same functionality is however the same over the complete project. Since the outlier detection works based on the mean and standard deviation, changes throughout the whole project are taken into account this way. Working with average mutant density could also slightly alleviate this issue. This is based

on the same reasoning, more operations needed could also correspond to more source lines of code implementing this. Giving a potentially better balance between these two metrics.

These statements do need proper verification but investigating these aspects is beyond the scope of this project itself. This would cause the need for more in depth analysis of internal project structure and of the differences between languages in terms of specific code patterns. Researching proper thresholding for mutant density or any of the other metrics utilizing this concept could also provide a solution. This is a very complex issue which would be sufficient for a work on its own. Having this can circumvent the most prominent problem for generalization at this time. Currently problematic methods or refactoring targets are defined based on outlier detection within a singular project. When starting to work on a new project, there are simply not enough methods to generate outliers upon. This means this method of detecting refactoring targets is currently only suited for projects that already have a significant code base. Growing a project from the start while using this metric is thus not a realistic approach. Mutant density could still be used to outline the amount of mutations, this would allow developers to use this information on their own accord. A threshold would allow this to be used for the start while giving some clear indications. Due to code variation and general complexity, using a single threshold to define when a function might be problematic, is extremely difficult.

6   Related Work

7 Conclusion

Cyclomatic complexity has a long standing reputation, although this depends on who you ask, this is not always a positive reputation. Despite this, it has been used for a long time in multiple different industrial tools. As not to pile onto the critique this metric has already received throughout the years, the emphasis was on acknowledgement and supplementing the gaps in this metric. The chosen approach was based on mutant density, a metric that takes principles from mutation testing to determine the code complexity. Calculating the amount of possible mutations gives information on the difficulty of testing and based on how complexity is often described, the code. The goal was to show that mutant density is able to provide additional information on which function could be considered problematic or refactoring targets in other words. Not only would this allow more detailed reasoning behind complex functions, it would also allow to show that complex functions can occur in different ways and that mutant density is able to cover others that cyclomatic complexity does not.

The approach throughout this work was based on a few different steps. Outside of gathering all the data, the initial experiments were based on correlation analysis between source lines of code, cyclomatic complexity and mutant density. This provides an initial basis to work around, too high correlation devalues the use of mutant density. Too low would also devalue the results since complexity metrics should have a certain amount of overlap in most cases. After this there were multiple different experiments to identify problematic functions across the different metrics and compare the information obtained from this. This includes experiments indicating the similarities between the metrics and the differences. Lastly, there were additional parts to the experimentation to include more in-depth analysis in the dataset composition and the modularity of mutant density.

The results provided by the correlation analysis fall in line with the expectations, there is a positive correlation between mutant density and the other metrics but not enough to argue for the redundancy of either of the methods. The correlation is still positive enough to acknowledge a strong sense of similarity between the different metrics. This is reflected in the ability to more directly compare cyclomatic complexity and mutant density. A higher correlation between these two allow the argument of a strong similar basis while providing individual unique results to function within the context of this work. These results allow a confident answer on the first research question:

Is there a fundamentally high correlation causing redundancy between cyclomatic complexity, source lines of code and mutant density?

There is a high and positive correlation but not nearly enough to cause redundancy.

Using the fundamental ideas provided by the correlation analysis, outliers are calculated based on a variety of different metrics. Out of these, the most important individual metrics were the scaled average mutant density and cyclomatic complexity. They provided an initial overview of the individual performance of these metrics in detecting outliers. Combined with the set theory approach, these results also directly show that mutant density views a type of complexity that cyclomatic complexity does not. There are sufficiently complex functions that are only found by a single metric that convey the message of their unique capabilities.

Applying a more direct metric to measure the relation between cyclomatic complexity and mutant density also gave some interesting perspectives. It purely emphasises the direct relation between those two metrics, it does however skip outlier based on a single metric. The metric does cover a similar basis while also adding a few unique scenarios. It gives functions that do not occur for a singular metric but do occur when they are put in relation to each other. Based on these results an answer can be provided for the second research question:

Which metrics can be defined to give precise insight in utilizing mutant density alongside cyclomatic complexity?

These metrics have been summarized in Table 1 but due to multiple potential threaths, this answer cannot be given with full confidence. In specific contexts the metrics do show usable results but due to the lack of generalization it remains hard to pinpoint the exact performance.

Modularity was also mentioned as an important aspect of mutant density as a complexity metric. Fine-tuning the possible mutation operators would allow more precise complexity definitions. This fine-tuning allows for vast variety in combinations, most emphasis was thus placed on more simple applications of this modularity. Running the same outlier detection for each possible mutation operator separately is a much more time consuming method which should be kept in mind. Looking at outliers that only occurred for a single mutation operator did not provide interesting results. It mostly showed the difference in presence of certain operators within a projects.

The most interesting results were obtained by looking at methods that were outliers for each single mutation operator. These methods were easily seen as complex due to the presence of too many of each single operator type. Not only that, due to the inclusion of operators of different types, it also affected the size of the functions found without directly

incorporating a scaling factor. This opens up a lot more potential surrounding this approach. These results provided an answer for the last research question:

Does altering the selection of mutation operators change the complex methods found by mutant density and can this be utilized for better results?

This does remain a fairly difficult question to answer since it is extremely hard to measure better results.The methods that were outliers over every single mutation operator did give insight in a very interesting direction. Although it would make for a computationally heavy metric, creating a metric based on the amount of operators a method is an outlier for, has potential.

In the end, these results provide the foundation for mutant density as a provider for additional information alongside cyclomatic complexity. It has been shown that due to a different approach to complexity, mutant density allows the finding of new unique information without being redundant due too high similarity. Different base metrics have been given that use a combination of cyclomatic complexity, mutant density and source lines of code. These combinations have their own different advantages and disadvantages that have been given in earlier sections. Despite the disadvantages, the advantages show to possibilities surrounding mutant density. This results in varied metrics to view complexity in a non singular way.

Although, the possibilities and an original foundation has been shown to exist, there are still a lot of possible improvements. Some improvements are even necessities before larger scale applications of these metrics can exist. These issues have been partially discussed throughout the whole work but with all of the results finished, it is easier to obtain on a complete picture on the state of this topic.

The first and probably the most prominent part of future work is surrounding the tooling that is available. There is almost no tooling available specifically for calculating mutant density. Meaning most of the tools that are applied, are derived from tooling for mutation testing. This is for C, C++ and partially for python, not done in a static manner. In the context of mutation testing this is perfectly fine, for mutant density this is not the most efficient solution. A large first step for this topic would be the creation of a general tooling solution that applies static mutant density calculation to different languages. This would also allow for the streamlining of, source lines of code calculation and the variation in mutation operators.

A second direction that needs further investigation is the influence of these results under a more precisely defined dataset. Right now the dataset is too small and has too many uncertainties to correctly evaluate the generalization of the results. Creating a larger dataset with either a more defined construction, or a size large enough for general results, is quite a difficult task. It is however necessary for more appropriate evaluation of the results presented in this work. Easier and more efficient static tooling would also heavily simplify this issue and would allow for the usage of repository mining and the automation of this process.

Then, there remains the issue of scaling these metrics onto a threshold to allow the usage on smaller sized projects. The usage of outlier detection to determine refactoring targets only works on larger projects as described in this work. Cyclomatic complexity has threshold for determining which functions should be seen as complex. This has however, very little analytical reasoning. It is thus very hard to apply the same or similar thresholding without proper investigation and analysis in this issue. This is also an extremely difficult issue due to all the differences between different projects and programming languages as shown in the results. A singular threshold for across all possible applications is therefor not a really feasible solution.

Another possible approach for utilizing these metrics on smaller scale projects is moving back to calculating the mean and standard deviations across a large set of projects. This would allow the usage of a very general mean that does not need a large project to function upon since it can be used directly. Using a complete dataset would however reintroduce the issue of generalization. It also falls back to the need for a much larger and thought out dataset creation.

Lastly, with the results from the previous improvements, it becomes easier to further investigate the exact performance of the metrics provided. It also might show certain aspects that could still use slight tweaking. Using the potential improvements for the modular aspect also becomes more feasible after the improvement of the tooling. This metric is very dependant on the implementation and efficiency of the tool so this would be a fundamental step before continuing in this direction.

8 Relation to previous works

Appendix

References

[1] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 493–501. IEEE, 2016. pages 10

[2] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016. pages 10

[3] Empear AB. Codescene. pages 10, 13

[4] SonarSource SA. Sonarqube. pages 10, 13

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. pages 10

[6] Ping Yu, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 569–580. IEEE, 2023. pages 10

[7] Frank Faber. Testing in devops. *The Future of Software Quality Assurance*, pages 27–38, 2020. pages 10

[8] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976. pages 10, 13

[9] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of . . . , 1996. pages 10, 15

[10] Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications*, 7(2):67–82, 2013. pages 10

[11] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001. pages 10

[12] Ali Parsai and Serge Demeyer. Mutant density: A measure of fault-sensitive complexity. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 742–745, 2020. pages 10

[13] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007. pages 12

[14] Inc. Synopsys. Coverity. pages 13

[15] Timothy A Budd, Richard J Lipton, Richard DeMillo, and Frederick Sayward. The design of a prototype mutation system for program testing. In *Managing Requirements Knowledge, International Workshop on*, pages 623–623. IEEE Computer Society, 1978. pages 15

[16] Ali Parsai. Chaosmeter. pages 20

[17] Dextool mutate. pages 20

[18] Metrix++. pages 20

[19] Mccabe. pages 20

[20] Mutpy. pages 20

[21] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016. pages 21, 26, 29, 30

[22] Jay Graylin, Joanne E Hale, Randy K Smith, Hale David, Nicholas A Kraft, WARD Charles, et al. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, 2(03):137, 2009. pages 21

[23] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. pages 28