Empirical Evidence of the Consequences of Bad Smells in Software

by

Fatma Neda Topuz

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 7, 2022

Keywords: Software Engineering, Code Smell, Bad Smell, Object-oriented design metrics

Approved by

David Umphress, Chair, Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Cheryl Seals, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering
SueAnne Nichole Griffith, Assistant Professor of Electrical and Computer Engineering

Abstract

A “smell” is a property of software source code that indicates a deeper problem. Smells do not necessarily affect the code at the immediate point in time but are thought to cause problems during a later phase of the software’s evolution. Developers tend to downplay the impact of bad smells and, consequently, fail to remove them; however, bad smells introduce complexity and, thus, introduce the possibility of degrading code maintainability over time and causing troublesome side effects later as the software is changed. While the concept of code smells and their potential impact on future software maintainability has logical appeal, the proof of the impact of code smells is largely anecdotal. Our research sought empirical evidence of the effect of code smells on defects by examining five production-level open-source Python projects. Specifically, we analyzed (1) GitHub change logs and (2) defect logs to determine smell-to-defect correlation and an empirical analysis of open-source Python projects. We made the investigation on five different scale projects. Our results indicate that the Long Parameter List code smell has the highest positive correlation, the Long Method, Large Class and Shotgun Surgery, when detected with medium extracted threshold values, smells have a moderate correlation, and the Parallel Inheritance Hierarchy, Data Class, Lazy Class, Refused Bequest, Feature Envy and Shotgun Surgery smells when detected with low and high extracted threshold values proposed in the literature have no significant correlation on the presence of defects.

Acknowledgments

I thank God for this achievement. You always supported me through this long PhD journey until my graduation. There were many ups and down during my life at Auburn, but with the grace of God every time I was able to choose the right way and be who I am. Thank you for letting me through all the difficulties. I am grateful to you for all the blessings. I will keep on trusting you and pray for you guidance for my future.

I will forever be thankful to my Professor Dr. David Umphress for his valuable feedback and guidance. He is someone you will never forget once you meet. He is the funniest and warmest advisor you could have ever met. He is one of the smartest and nicest people I have known. He has dedicated his time and energy guiding me towards at every step of my dissertation. I hope that I could live as enthusiastic and energetic as Dr. Umphress and become an excellent Professor as he is. I could not have accomplished without your help, motivation and support. Thank you so much for everything you have done for me!!

I would like to thank to my committee members: Dr. Cross, Dr. Seals and Dr. Hendrix for your valuable feedback on my work, and Dr. SuaAnne Griffith for taking time to serve as the University Reader for my dissertation.

A warmest thank you to my mom for all hard work, determination, and supporting me in everything that I need and her never ending patience. Thank you for encouraging me to be the best that I could be. To my Dad, thank you for being so supportive of my decision to pursue a PhD degree in the US. My hard-working parents have sacrificed their lives to build me and my sisters a better future. I would not have made it this far if I would not have had them. My sisters: Busra and Burcin have always been my best friends. I also thank them for all their advises and support. Thank you my wonderful family: my mom, dad and little sisters. I appreciate all the things you have done for me. I love you so so much!! I also would like to thank to my uncle, Hamit, for encouraging me in all of my pursuits. I always knew that you believed in me and wanted the best for me.

I would like to thank all my friends at Auburn; especially Ilknur, Menekse, Fehmi, Nihan and Shubbhi, for your friendship and emotional support through all these years. I learned so many things from you. To Ilknur my childhood and dearest friend, I could not have survived without you! To Menekse, and Fehmi who I met during first years of my Auburn journey, I could not have lived this much happy at Auburn if I would not have had you. Thank you for always providing support and friendship that I needed throughout my time in the US. Thank you for all fun times, and beautiful memories. I will keep them forever.

I thank Dr. Umphress's PhD students (too many to list here) and Haneen thank you for being supportive and always helping me. I also thank all of my Professors here at Auburn who guided and helped to shape my life on the right path. I also thank Kecia Hill and the wonderful staff in the Computer Science and Software Engineering Department and in the other departments at Auburn University for all the administrative help. Kecia, you were always helpful and so friendly. I will not forget your support and kindness.

I would like to thank Turkish Government for financial support essential to successfully complete my PhD degree.

Finally, I would like to thank everyone who have helped me out throughout this journey.

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures ix

List of Tables xv

1 Introduction 1

1.1 Motivations 2

2 Literature Review 4

2.1 Bad Smells 4

2.2 Empirical Evidence on Tracing Changes on the Issue Tracking System 8

2.3 Analysis of the Consequences of Bad Smell in Open Source Python System . . 9

3 Research Methodology 11

3.1 Object Oriented Design Metrics 11

3.2 Smells Definition and Detection Strategies 13

3.2.1 Bad Smells Used for Investigation 15

3.2.2 Bad Smells that are not investigated due to metrics model unavailability 19

3.3 Research Approach 22

3.4 Research Methodology 22

4 Validation 28

4.1 Accuracy of Smell Detection . . . 32
4.2 Bayes Theorem Calculation on Defect-Smell Analysis . . . 33
4.3 Measuring Association Between Code Smells and Defects . . . 34
4.4 Code Smell-Defect Relation Findings on Python Open-Source Projects . . . 34
4.4.1 Large Class . . . 34
4.4.2 Long Parameter List . . . 43
4.4.3 Parallel Inheritance Hierarchy . . . 50
4.4.4 Lazy Class . . . 56
4.4.5 Data Class . . . 62
4.4.6 Long Method . . . 68
4.4.7 Refused Bequest . . . 75
4.4.8 Feature Envy . . . 81
4.4.9 Shotgun Surgery . . . 87

5 Conclusion . . . 92
5.1 Summary . . . 92
5.2 Observations . . . 95
5.3 Contributions . . . 96
5.4 Future Work . . . 97

References . . . 99

Appendices . . . 104

A Shotgun Surgery Smell Detection with Low Threshold Values . . . 105
A.1 Shotgun Surgery Smell Analysis with Low Threshold Values . . . 105
A.2 Shotgun Surgery Smell Findings with Low Threshold Values . . . 107

B Shotgun Surgery Smell Detection with Medium Threshold Values . . . 109

B.1 Shotgun Surgery Smell Analysis with Medium Threshold Values 109
B.2 Shotgun Surgery Smell Findings with Medium Threshold Values 111

C Shotgun Surgery Smell Detection with High Threshold Values 113
C.1 Shotgun Surgery Smell Analysis with High Threshold Values 113
C.2 Shotgun Surgery Smell Findings with Large Threshold Values 115

D Shotgun Surgery Dependent Analysis Results 117
D.1 Keras-team 117
D.2 Tensorflow 117
D.3 scikit-learn 117
D.4 Zulip 118
D.5 NumPy 118

E Code Smell and Defect Density Analysis 119
E.1 Correlation Coefficient Analysis of Code Smells 119
E.2 Code Smell Density Analysis Results 120
E.3 Defect Density Traced to Code Smell Analysis Results 120

List of Figures

3.1 The steps of identifying bad smells in the research plan 24

3.2 The flow of the research plan 27

4.1 Main components of the Dissertation Project 29

4.2 Example of commits obtained from NumPy project 30

4.3 Example of Filtered Issue List 30

4.4 Shows a screenshot of the spreadsheet after both the analysis are done 31

4.5 Shows an example of the process about how the project works to find the relation between bad smells and defects 32

4.6 The relation between the probability of a defect due to large class and number of bug fixed commits 42

4.7 The relation between the probability of a defect due to Large Class Smell and Number of Large Classes 43

4.8 The relation between the Number of Bug-fixed Commits and methods with long parameter lists 49

4.9 The relation between the probability of a defect and Number of Long Parameter List 50

4.10 The relation between number of Bug-fixed commits and classes with parallel inheritance hierarchy smell 55

4.11 The relation between classes with parallel inheritance hierarchy and the probability of defects occurring 56

4.12 The relation between the number of bug-fixed commits and lazy classes 61

4.13 The relation between lazy class smell and the probability of defects occurring . 62

4.14 The relation between classes with data class smell and number of bug-fixed commits 67

4.15 The relation between number of classes with data class smell and the probability of a defect occurring 68

4.16 The relation between long method code smell and probability of a defect occurring 74

4.17 The relation between number of bug-fixed commits and probability of a defect occurring for long method code smell 74

4.18 The relation between Number of Bug-fixed commits and classes with Refused Bequest Smell 80

4.19 The relation between the Probability of a Defect and number of classes with Refused Bequest Smell 80

4.20 The relation between classes with feature envy smell and probability of a defect occurring 86

4.21 The relation between probability of a defect and number of bug fixed commits for feature envy smell 86

4.22 Shotgun Surgery Smell Detection Strategy 88

4.23 Shotgun Surgery Dependent Analysis 91

D.1 Keras-team Shotgun Surgery Dependent Analysis 117

D.2 Tensorflow Shotgun Surgery Dependent Analysis 117

D.3 scikit-learn Shotgun Surgery Dependent Analysis 117

D.4 Zulip Shotgun Surgery Dependent Analysis 118

D.5 NumPy Shotgun Surgery Dependent Analysis 118

List of Tables

3.1 Metrics for detecting Python smells 14

3.2 The availability of metrics model for code smell and the appropriate threshold allocation for software metrics 16

4.1 Python Open-Source Project Details 29

4.2 Correlation Coefficient Description 34

4.3 Large Class Frequency table of 'NumPy' 36

4.4 Large Class Likelihood table of 'NumPy' 36

4.5 Large Class Frequency table of 'Keras-Team' 37

4.6 Large Class Likelihood table of 'Keras-Team' 37

4.7 Large Class Frequency table of 'scikit-learn' 38

4.8 Large Class Likelihood table of 'scikit-learn' 38

4.9 Large Class Frequency table of 'Zulip' 39

4.10 Large Class Likelihood table of 'Zulip' 39

4.11 Large Class Frequency table of 'Tensorflow' 40

4.12 Large Class Likelihood table of 'Tensorflow' 40

4.13 Results - Probability of a defect due to a large class bad smell of all projects . . 41

4.14 Long Parameter List Frequency table of 'NumPy' 44

4.15 Long Parameter List Likelihood table of 'NumPy' 44

4.16 Long Parameter List Frequency table of 'Keras-team' 45

4.17 Long Parameter List Likelihood table of 'Keras-team' 45

4.18 Long Parameter List Frequency table of 'scikit-learn' 45

4.19 Long Parameter List Likelihood table of 'scikit-learn' 46

4.20 Long Parameter List Frequency table of 'Zulip' 46
4.21 Long Parameter List Likelihood table of 'Zulip' 46
4.22 Long Parameter List Frequency table of 'Tensorflow' 47
4.23 Long Parameter List Likelihood table of 'Tensorflow' 47
4.24 Results - Probability of a defect due to a long parameter list bad smell of all projects 48
4.25 Parallel Inheritance Hierarchy Frequency table of 'NumPy' 51
4.26 Parallel Inheritance Hierarchy Likelihood table of 'NumPy' 51
4.27 Parallel Inheritance Hierarchy Frequency table of 'Keras-team' 51
4.28 Parallel Inheritance Hierarchy Likelihood table of 'Keras-team' 52
4.29 Parallel Inheritance Hierarchy Frequency table of 'scikit-learn' 52
4.30 Parallel Inheritance Hierarchy Likelihood table of 'scikit-learn' 52
4.31 Parallel Inheritance Hierarchy Frequency table of 'Zulip' 53
4.32 Parallel Inheritance Hierarchy Likelihood table of 'Zulip' 53
4.33 Parallel Inheritance Hierarchy Frequency table of 'Tensorflow' 53
4.34 Parallel Inheritance Hierarchy Likelihood table of 'Tensorflow' 54
4.35 Results - Probability of a defect due to a parallel inheritance hierarchy bad smell of all projects 54
4.36 Lazy Class Frequency table of 'NumPy' 57
4.37 Lazy Class Likelihood table of 'NumPy' 57
4.38 Lazy Class Frequency table of 'Keras-team' 57
4.39 Lazy Class Likelihood table of 'Keras-team' 58
4.40 Lazy Class Frequency table of 'scikit-learn' 58
4.41 Lazy Class Likelihood table of 'scikit-learn' 58
4.42 Lazy Class Frequency table of 'Zulip' 59
4.43 Lazy Class Likelihood table of 'Zulip' 59
4.44 Lazy Class Frequency table of 'Tensorflow' 59

4.45 Lazy Class Likelihood table of 'Tensorflow' 60
4.46 Results - Probability of a defect due to a lazy class bad smell of all projects . . 60
4.47 Data Class Frequency table of 'NumPy' 63
4.48 Data Class Likelihood table of 'NumPy' 63
4.49 Data Class Frequency table of 'Keras-team' 64
4.50 Data Class Likelihood table of 'Keras-team' 64
4.51 Data Class Frequency table of 'scikit-learn' 64
4.52 Data Class Likelihood table of 'scikit-learn' 65
4.53 Data Class Frequency table of 'Zulip' 65
4.54 Data Class Likelihood table of 'Zulip' 65
4.55 Data Class Frequency table of 'Tensorflow' 66
4.56 Data Class Likelihood table of 'Tensorflow' 66
4.57 Results - Probability of a defect due to a data class bad smell of all projects . . 67
4.58 Long Method Frequency table of 'NumPy' 69
4.59 Long Method Likelihood table of 'NumPy' 69
4.60 Long Method Frequency table of 'Keras-team' 70
4.61 Long Method Likelihood table of 'Keras-team' 70
4.62 Long Method Frequency table of 'scikit-learn' 70
4.63 Long Method Likelihood table of 'scikit-learn' 71
4.64 Long Method Frequency table of 'Zulip' 71
4.65 Long Method Likelihood table of 'Zulip' 71
4.66 Long Method Frequency table of 'Tensorflow' 72
4.67 Long Method Likelihood table of 'Tensorflow' 72
4.68 Results - Probability of a defect due to a Long Method bad smell of all projects 73
4.69 Refused Bequest Frequency table of 'NumPy' 76
4.70 Refused Bequest Likelihood table of 'NumPy' 76

4.71 Refused Bequest Frequency table of 'Keras-team' 76
4.72 Refused Bequest Likelihood table of 'Keras-team' 77
4.73 Refused Bequest Frequency table of 'scikit-learn' 77
4.74 Refused Bequest Likelihood table of 'scikit-learn' 77
4.75 Refused Bequest Frequency table of 'Zulip' 78
4.76 Refused Bequest Likelihood table of 'Zulip' 78
4.77 Refused Bequest Frequency table of 'Tensorflow' 78
4.78 Refused Bequest Likelihood table of 'Tensorflow' 79
4.79 Results - Probability of a defect due to Refused Bequest bad smell of all projects 79
4.80 Feature Envy Frequency table of 'NumPy' 82
4.81 Feature Envy Likelihood table of 'NumPy' 82
4.82 Feature Envy Frequency table of 'Keras-team' 82
4.83 Feature Envy Likelihood table of 'Keras-team' 83
4.84 Feature Envy Frequency table of 'scikit-learn' 83
4.85 Feature Envy Likelihood table of 'scikit-learn' 83
4.86 Feature Envy Frequency table of 'Zulip' 84
4.87 Feature Envy Likelihood table of 'Zulip' 84
4.88 Feature Envy Frequency table of 'Tensorflow' 84
4.89 Feature Envy Likelihood table of 'Tensorflow' 85
4.90 Results - Probability of a defect due to Feature Envy bad smell of all projects . 85
4.91 The extracted threshold values for shotgun surgery detection strategy 88
4.92 The correlation analysis of shotgun surgery smell with difference threshold values 89
4.93 The result of shotgun surgery method dependent analysis 90
A.1 Shotgun Surgery Frequency table of 'NumPy' 105
A.2 Shotgun Surgery Likelihood table of 'NumPy' 105
A.3 Shotgun Surgery Frequency table of 'Keras-team' 105

A.4 Shotgun Surgery Likelihood table of 'Keras-team' 106

A.5 Shotgun Surgery Frequency table of 'scikit-learn' 106

A.6 Shotgun Surgery Likelihood table of 'scikit-learn' 106

A.7 Shotgun Surgery Frequency table of 'Zulip' 106

A.8 Shotgun Surgery Likelihood table of 'Zulip' 107

A.9 Shotgun Surgery Frequency table of 'Tensorflow' 107

A.10 Shotgun Surgery Likelihood table of 'Tensorflow' 107

A.11 Projects Results - Probability of a defect due to Shotgun Surgery bad smell with low threshold values 108

B.1 Shotgun Surgery Frequency table of 'NumPy' 109

B.2 Shotgun Surgery Likelihood table of 'NumPy' 109

B.3 Shotgun Surgery Frequency table of 'Keras-team' 109

B.4 Shotgun Surgery Likelihood table of 'Keras-team' 110

B.5 Shotgun Surgery Frequency table of 'scikit-learn' 110

B.6 Shotgun Surgery Likelihood table of 'scikit-learn' 110

B.7 Shotgun Surgery Frequency table of 'Zulip' 110

B.8 Shotgun Surgery Likelihood table of 'Zulip' 111

B.9 Shotgun Surgery Frequency table of 'Tensorflow' 111

B.10 Shotgun Surgery Likelihood table of 'Tensorflow' 111

B.11 Projects Results - Probability of a defect due to Shotgun Surgery bad smell with medium threshold values 112

C.1 Shotgun Surgery Frequency table of 'NumPy' 113

C.2 Shotgun Surgery Likelihood table of 'NumPy' 113

C.3 Shotgun Surgery Frequency table of 'Keras-team' 113

C.4 Shotgun Surgery Likelihood table of 'Keras-team' 114

C.5 Shotgun Surgery Frequency table of 'scikit-learn' 114

C.6 Shotgun Surgery Likelihood table of 'scikit-learn' 114

C.7 Shotgun Surgery Frequency table of 'Zulip' 114

C.8 Shotgun Surgery Likelihood table of 'Zulip' 115

C.9 Shotgun Surgery Frequency table of 'Tensorflow' 115

C.10 Shotgun Surgery Likelihood table of 'Tensorflow' 115

C.11 Projects Results - Probability of a defect due to Shotgun Surgery bad smell with large threshold values 116

E.1 The result of correlation coefficient for each code smell 119

E.2 Code Smell Density of Python projects 120

E.3 Code Smell and Defect Density Findings 121

Chapter 1

Introduction

At upward of 75% of total lifecycle costs [13], software maintenance is an expensive activity. The cost is high due to human nature: tight deadlines reward developers for quickly fixing defects and adding new functionality without regard to improving code maintainability [13]. This leads to the software complexity to increase and software quality to decrease. Refactoring was proposed as means to help with this problem [16]:

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its external structure."

Fowler and Beck [16] describe 22 general patterns of common, but unnecessary, code complexity and how to refactor the code to remove the complexity. These patterns are called bad smells, code smells, or code bad smells. In this study, these terms are used interchangeably. Bad smells are not problematic on their own, but they indicate complexity that could complicate future modifications. Bad smells can be found in both production or test code. Fowler notes that a bad smell is a surface indication that usually corresponds to a deeper problem in the system and calling something as bad smell is not an attack, but it is simply a sign that a closer look is needed [16].

A bad smell causes no harm when it is initially injected in the code; it makes the code difficult to understand and, by extension, difficult to modify. Common wisdom suggests that developers feel pressure to deliver a product by a deadline, causing them to prioritize delivering features over code quality and maintenance activities. This is often suggested as the one of the causes of the bad smells. Fernandes et al. [13] also note that bad smells are one of the important factors that affect the quality and maintainability of a software system and may indicate a deeper quality problem in the system design or code.

1.1 Motivations

Bad smells are, thus, segments of code that require extraordinary effort to comprehend, test or maintain. Smells signify the need to improve readability and understandability. Many companies ask their developers to evaluate their code for logical errors or bugs. The developers may be aware of bad smells in their projects, but smells are not considered to be a high priority [30]. Developers prefer to resolve bad smells to improve performance or readability or to fix bugs. If bad smells do not cause performance or functional issues, the developers typically leave the code as is. However, some logical errors may lead to bugs that might even cause the entire system to crash. Khomh et al. [20] suggest that systems that contain a high number of smells are likely to be more change prone and therefore the cost of ownership of such systems will be higher than for other systems because developers will have to put in more effort. Peters [30] asserts that bad smells have a negative influence on software quality. If no action is taken in a timely manner, then a software system will deteriorate over time. Although bad smells may not affect a system at that moment, they have the potential to affect future modifications. Refactoring code may help to remove bad smells and prevent a larger problem that might occur later on.

Fowler indicates that there is no precise criteria for refactoring a bad smell because bad smell definitions differ from researcher to researcher [8]. Bad smell detections results may differ from tool to tool because there are no standard rule that explains the thresholds by which tools identify bad smells.

Since there are no precise criteria, several tools have been proposed to automate bad smell detection to improve software maintainability. Fernandes et al. [13] compare 84 of bad smell detection tools. They state that tools aim to detect 61 bad smells by using different detection techniques. They also note that these tools target different programming languages such as Java, C, C++ and C#. There is a lack of technique or tool support that targets code smells in

Python. A search of the literature revealed five studies [37], [10], [9], [33] and [8] that detect code smells in Python programs. There is no specific catalogue of Python specific code smells. Due to the lack of study in Python programs, in this study, the bad smells in Python programs have been investigated. Within this project, we intent to seek the answer the following research questions:

1. Do bad smells contribute to later problems?

2. What percentage of defects in open source software can be traced to bad smells?

3. In what order should bad smells be fixed in open source software?

Our findings are expected to identify the bad smells in open source software. The identified bad smell will be linked to an issue tracking system to rank the "badness" of bad smells. The findings will be able to help developers to prioritize which bad smells to fix first if a number of them exist in open source project.

The rest of this paper is organized as follows: Chapter 2 discusses related work on identifying bad smells in open source projects, the empirical studies on tracing changes on the issue tracking system, and analysis of the consequences of bad smell in open source Python project. Chapter 3 describes the methodology that is used to identify the bad smells using change logs and the plan about how to link the bad smells to the defects on the defect logs of open source project. Chapter 4 presents the research methodology, how the results are validated. Chapter 5 provides research conclusions and suggestions for future work.

Chapter 2

Literature Review

This chapter includes three sections. The first section explains the studies that we used to detect the bad smells. The second section explains the studies that have the methods to track the changes on the issue tracking system. The third section describes the studies that focuses on the impact of Python smells on open-source projects. Bad smells of this study will be detected on GitHub change logs using the studies on the first section. Defects will be investigated from GitHub issue tracking system and correlated to the bad smells with the help of studies on the second section of this chapter.

2.1 Bad Smells

Fowler defines a bad code smell as [16]:

"Code smells are indications of poor coding and design choices that can cause problems during the later phase of software developments. Code smells are considered as flags to the developer that some parts of the design may be inappropriate."

The premise behind smells is that they are code segments that are most likely to become defective as software is revised. In small software systems, code smells can be detected by manual inspection and cleaned. However, in large software systems, manual inspection is time consuming and, frequently, inaccurate. To detect bad smells in large systems, there has been several works offered to detect code smells. Sharma et al. [34] classify smell detection

works into five categories. These categories are metric-based detections [9], [25] which identify smells based on measuring the code for a threshold number of properties; rules/heuristic-based smell detection in which code is matched to patterns that correspond to smells; history-based smell detection by using source code evolution information [34]; machine learning-based smell detection [34] which starts with a mathematical model to represent the smell detection problem, existing problem and source code module to be able to use to instantiate a concrete populated model which results in a set of detected smells; and optimization-based smell detection where optimization algorithms are applied to detect smells in the source code. Inspection techniques [36] can also be added to these smell detection categories.

Sharma et al. note that most of the existing tools support detecting a subset of known smells [34]. The duplicated code smell, for example, has been one of the smells that received the most attention. The duplicated code, also known as clones, code clones or code duplication, is present if identical similar code exists in several places. Rahman et al. [31] describe the relationship between the duplicated code and defect proneness based on findings from four different medium-to-large open source projects. They took an empirical approach to evaluate if duplicated code contributes to bugs or not. Their study revealed a low correlation between bugs and duplicated code. They also showed duplicated code contains less buggy code, suggesting that bad smells do not have a significant effect on bugs. On the other hand, Zhang et al. [39] investigate the relationship between six bad smells including duplicated code and software faults. Their objectives were to identify whether code bad smells are potential indicators of software problems and, if so, how to prioritize refactoring of those smells. The result of their study shows that source code containing duplicated code should be refactored first because it is likely to be associated with more faults than source code containing other bad smells. Based on the findings of these two papers, we could not decide whether duplicated codes contain less or more buggy code. That is why we decided to have more research on this smell to show if it leads to a bug or not.

The second most studied smell in the literature is the so-called god class, also known as large class and long methods [34]. The large class smell is a class that contains many fields, methods, and lines of code. The long method smell is similar to large class, it occurs when a method has large number of executable statements. McConnell [27] states that the theoretical maximum limit is the number of lines of code that can fit one screen. Martin [26] analyzes this idea with a different point of view. He thinks that "If small is good, then smaller must be better.". There are several tools that check for the lines per method. PMD [5] is a static analysis tool that helps find unused variables, empty catch blocks, etc. PMD indicates that the maximum number of lines should be 100 per method, 1000 lines for each class, and there should be no more than 10 methods in a class. Checkstyle [7], a static analysis tool that helps programmers to write code according to a coding standard, defaults the maximum number of lines per method and lines per class as 50 and 1500 respectively. These two bad smells might be the most studied smells not because they are relatively easy to identify in the code when compared with the other bad smells, but because they are important to study and most commonly occurring bad smells in the code [17]. Chen et al. [9] implement a detection tool that is used to identify the long method and large class in the Python programs. In addition to long method and large class, the long parameter list, the message chain and seven other bad smells are also detected using their tool. According to the survey made in 2018 [34], there are 3.42% of 117 studies that detect the long parameter list smell. Having too many parameters in a method call is a sign of a long parameter list smell in the code. The message chain occurs when a client asks one object for another object, that object asks for yet another object, and so on. Zhang et al. [40] identify 5 studies on message chains. In our study, to detect the large class, long method, long parameter list and message chains, code metrics and metric threshold values used by Chen et al. [9] was used. The reason to use their study is because the study is on Python programs and their detection tool can detect 285 instances having bad smells in total with the average precision of 97.7% using metrics and the threshold values that they propose.

In contrast to tracing smells to problems, researchers have flipped the perspective by examining the relationship between problematic classes and bad smells. Li and Shatnawi [23] investigate the relationship between bad smells and class error probability at three error severity levels: high being for Blocker and Critical, medium for Major, low for Normal and Minor. They checked for 6 smells: data class, shotgun surgery, god class, god method, refused bequest, and feature envy. The data class smell is a class that has data fields and operations that are only getters and setters. The shotgun surgery smell occurs when a small change in a class causes many changes to many classes. The presence of a method in a given class that is relative to other methods in the same class is an indication to god method smell. The refused bequest smell is a class that does not support its inherited method or data. The feature envy smell arises if a method accesses data of another class more than its own data. Li and Shatnawi [23] focus on post-release object-oriented systems rather than the development process of systems. Their study shows that shotgun surgery, god method and god class smells are positively associated with the class error probability across and within the three error severity levels.

Danphitsanuphan and Suwantada [12] identify bad smells in software with a tool that helps detect the location of the source of code smell. They detected 7 types of bad smells: large class, long method, parallel inheritance hierarchy, long parameter list, lazy class, switch statements and data class. The parallel inheritance hierarchy smell occurs when an inheritance tree depends on another inheritance tree by composition. Both have a special relation that means one subclass of inheritance tree will be dependent on the subclass of another inheritance tree. The lazy class bad smell is a class that does not do anything. The switch statements bad smell is exemplified by a sequence of if or switch statements. Danphitsanuphan and Suwantada [12] analyzed 323 classes and were able to detect the lazy class, switch statements, parallel inheritance hierarchy and other three smells in the classes using their proposed tool. Their tool was implemented as Eclipse plugin using a predefined threshold of software metrics. The tool provides the detected code smells, software metric values, and the location of the bad

smell in the source code. They used several data mining techniques including Naïve Bayes and Association rule techniques [12] to discover the relationship between the code smell and software structural defects; in other words, which kind of software structure bug was caused by each type of code smell.

2.2 Empirical Evidence on Tracing Changes on the Issue Tracking System

To trace bugs between the issue tracking system and the change logs, there should be a link. The developer should mention the bugs' existence in the change logs. Both the issue tracking system and the change logs can contain large amount of information. These two systems can help developers evaluate and measure the quality of their systems.

Fischer et al. [14] trace the relation between the issue tracking system and the change logs. They suggest that these two systems provide insufficient support for a detailed analysis of software evolution aspects. They propose a solution to populate a release history database that combines version data with the bug tracking data. They use the CVS version control system and the Bugzilla bug tracking system in their data model. They import the CVS and Bugzilla data into their relational database, which is then used for obtaining meaningful views to show the evolution of a software project. Their findings demonstrate that such views enable more accurate reasoning of evolutionary aspects and facilitate the anticipation of software evolution. The idea of linking change logs on the bug tracking system was used in this study.

To demonstrate if bad smells contribute to later problems or not, Wu et al. [30] developed an automatic link recovery algorithm called ReLink. They discovered that there are missing links between bugs and the change logs. That happens due to the absence of bug references in the change logs. The ReLink algorithm automatically learns satisfaction criteria of features from explicit links and applies the learned criteria that checks if the link is unknown or valid. The algorithm helps identify the characteristics of links based on the bug IDs in change logs, find the links between bugs and change logs and recover the missing links [38]. They also

evaluate the impact of recovered links on software maintainability measurement and defect prediction.

Sliwerski et al. [35] analyze fix-inducing changes that cause problems on the change logs. They show how to automatically locate fix-inducing changes by linking the CVS version control system to the Bugzilla bug database. Their idea was to start with a bug report in Bugzilla that indicates the fixed problem, extract the associated change from CVS that gives the location of the fix, then determine if the earlier change at that location that was applied before the bug was reported. Their findings show that they were able to link the two systems to identify changes that cause a problem. To link the changes with the bugs, they used two independent levels of confidence: a syntactic level and semantic level.

2.3 Analysis of the Consequences of Bad Smell in Open Source Python System

There are multiple studies that focus on the impact of smells on faults using object-oriented languages such as Java, and C-like languages. However, there are not many studies that investigate the impact of Python smells. Chen et al. [10], [9] propose a tool called Pysmell to identify the code smells in Python projects. They introduce 10 Python code smells, 5 well-known and 5 new. They establish a metric-based detection method with three filtering strategies: experienced-based strategy, in which the threshold values for each metric is obtained by authors' past experiences; statistics-based strategy, the thresholds are derived through statistical measurement; and tuning machine strategy, is based on an example repository to identify Python smells and specify metric thresholds. They also conduct an empirical study and investigate the effects of each detection strategy on detecting Python smells and software maintainability. They report that long method and long parameter list are most commonly seen smells and the indicators of change-prone and fault-prone code in Python projects.

Vavrová et al. [37] investigate the differences of Python code smells from Java-based findings. They develop a tool called Design Defect Detector that detects five code smells and four

antipatterns defined in literature. Their study discovers that 8 out of 9 smells and antipatterns were detectable in Python. They found that the density of detected smells and antipatterns were lower in Python code than in Java Code. Their report shows that more than twice as many methods in Python can be considered as too long in Java. They also state that long parameter list smells are seven times less likely to be found in Python code than in Java code. To the best of our knowledge, there is currently no other empirical analysis of Python smells in open source systems.

Chapter 3

Research Methodology

There are several ways to detect bad smells in software systems. Sharma et al. [34] synthesize five methods for detecting smells: metric-based detection, code reviews, history-based detection, machine learning-based smell detection and optimization-based smell detection. Among these categories, the metric-based detections using tools are highly preferable and easy to implement. Metrics are used to measure the quality of a software product, but it is hard to assess the quality with only metrics. That is why metrics are used with their threshold values. In this research, the detection of some of the bad smells in a software project is performed using the code metrics or metric-based smell detection.

3.1 Object Oriented Design Metrics

Object-oriented design metrics are used to monitor progress, help quantify product complexity, and evaluate object-oriented software. They can be used to assess quality attributes such as complexity, understandability, usability, reusability, maintainability, cohesion, etc. They help developers understand design aspects of a software. They can help to identify problems in the code. Smells can also be defined in terms of thresholds on metrics [20]. The object-oriented design metrics were initially proposed by Chidamber and Kemerer [11], then refined by [23], [12], [32] and [37]. We employ the following object-oriented design metrics in our work:

1. LOC: This stands for the lines of code. It can be used as class level object-oriented metric.

2. NOA: This is the number of attributes. This metric is a class level object-oriented metric.

3. NOM: This is the number of methods in a class. It is used as class level metric.

4. PAR: This is the number of parameters. It is a method level metric.

5. CM: This notion stands for the Changing Method metric. This metric measures the number of distinct methods that access to an attribute, or a method of a given class.

6. ChC: This is the number of changing classes. This metric is the number of classes that access to an attribute, or a method of a given class.

7. AID: This is the Access of Import Data. It is the number of referenced variables that do not belong to the given class.

8. ALD: This stands for Access of Local data. This is the number of referenced variables that do belong to the given class.

9. NIC: This metric is the Number of Import Classes.

10. CC/VG: This notation refers to McCabe Cyclomatic complexity. This metric can be used to calculate the complexity of a method not a class. A method with a low cyclomatic complexity is generally better [32]. This is a method level object-oriented metric.

11. DIT: This is the depth of inheritance tree. This is a class level object-oriented metric. This metric provides each class to identify the length of the path until the root. This helps to calculate how far a class can go in the inheritance hierarchy. With the help of this metric, the ancestor classes can also be measured.

12. NCS: This stands for the count of the number of children that have inherited from a given class. This is a class level metric.

13. LMC: This is the length of message chaining by calling one method that calls another method.

14. LWMC: This metric is a weighted sum of all the methods defined in the class.

$$WMC = \sum_{i=1}^{n} Ci \tag{3.1}$$

where n is the total number of methods and Ci is the complexity of each method [11]. This is a class level metric.

15. LCOM: This metric is class level method. It stands for lack of cohesion method. It is the degree of similarity of methods. It can be calculated by finding the count of the number of methods pairs whose similarity is equal to 0 minus the count of method pairs whose similarity is different than 0 [32].

16. AIUR: This notion is the average inherence usage ratio.

17. MLOC: This is the method lines of code. This is a method level object-oriented metric.

3.2 Smells Definition and Detection Strategies

Fowler et al. [16] introduced 22 code bad smells. We detect the following bad smells using metrics: long method, large class, long parameters list, shotgun surgery, feature envy, parallel inheritance hierarchy, lazy class, message chains, data class and refused bequest. The metrics used for each bad smell shown in Table 3.1.

The metrics for the comments, incomplete library class, inappropriate intimacy, middle-man, switch statements, temporary field, primitive obsession, data clumps and divergent change bad smells are not provided because we are unaware of any.

Bad Smells	Metrics
Long Method	LOC: number of lines
Large Class	LOC: number of lines NOA: number of attributes NOM: number of methods
Long Parameter List	PAR: number of parameters
Shotgun Surgery	CM: number of changing methods ChC: number of changing classes
Feature Envy	AID: access of import data ALD: access of local data NIC: number of import classes
Parallel Inheritance Hierarchy	DIT: depth of inheritance tree NSC: number of children of a class
Lazy Class	NOM: number of methods NOC: number of attributes of a class DIT: depth of inheritance tree
Message Chains	LMC: length of message chain
Data Class	LWMC: weighted method per class LCOM: lack of cohesion of methods
Refused Bequest	AIUR: average inheritance usage ratio DIT: depth of inheritance tree

Table 3.1: Metrics for detecting Python smells

More than one metric may be needed to encapsulate a code smell characteristic. To construct a detection strategy for a given Python smell, two composition operators, “and” and “or”, are used. Their job is to combine metrics together. If two symptoms coexist for a given Python smell, the metrics would be connected by “and” operator; otherwise the “or” operator is used. Using the parallel inheritance hierarchy smell as an example, the number of children of the given class and depth of inheritance tree metrics are needed and connected by or operator point to the smell’s presence. Thresholds specified as minimum or maximum values allowed in the metric data set are also used for identifying Python smell. They define the corresponding limit of the metrics to avoid bad smells. Table 3.2 depicts a summary of the metrics and threshold values for code smells obtained from previous work.

3.2.1 Bad Smells Used for Investigation

Long Method

A long method is a method that has large executable statements. If a method is too long, it might be hard to understand or change. There are no certain numbers that shows if a method is long or not. To detect the long method bad smell in the code, “Rule of 30” was used from Refactoring in Large Software Projects [24]:

$$LOC > 30$$

where LOC is lines of code of a method. According to the rule, a method should not have more than an average of 30 code lines.

Large Class

A large class is a class that has an excessive number of methods, variables, or lines of code. Chen et al. [9] suggest detecting this smell with the following:

$$LOC \geq 200 \vee NOA + NOM > 40$$

Code Smell	Has Metrics Model?	If yes, metrics and threshold values
Duplicated Code	No	
Long Method	Yes	$LOC > 30$
Large Class	Yes	$LOC \geq 200 \vee (NOA + NOM) > 40$
Long Parameter List	Yes	$PAR \geq 5$
Divergent Change	No	
Shotgun Surgery	Yes	$((CMisintop20\%) \wedge (CM > 10 \wedge ChC > 5))$
Feature Envy	Yes	$AID > 4 \wedge (AIDisintop10\%) \wedge ALD > 3 \wedge NIC < 3$
Data Clumps	No	
Primitive Obsession	No	
Switch Statements	No	
Parallel Inheritance Hiearchy	Yes	$DIT > 3 \vee NSC > 4$
Lazy Class	Yes	$(NOM < 5 \wedge NOA < 5) \vee DIT < 2$
Speculative Generality	No	
Temporary Field	No	
Message Chains	Yes	$LMC \geq 4$
Middle Man	No	
Inappropriate Intimacy	No	
Alternative Classes with Different Interfaces	No	
Incomplete Library Class	No	
Data Class	Yes	$LWMC > 50 \vee LCOM > 0.8$
Refused Bequest	Yes	$(((AIURisinbottom25\%) \wedge (NOTDIT < 2))) \wedge AIUR < 33)$
Comments	No	

Table 3.2: The availability of metrics model for code smell and the appropriate threshold allocation for software metrics

where LOC is the lines of code, NOA is the number of attributes, and NOM is the number of methods in a class.

Long Parameter List

A method with more than 5 parameters is considered to have a long parameter list bad smell. This smell is detected using the number of parameters of a method. The model to detect this smell is adapted from Chen et al. [9]:

$$PAR \geq 5$$

Shotgun Surgery

A class has the shotgun surgery bad smell if a change made to one code segment cascades to changes to other code segments. Li and Shatnawi [23] propose:

$$((CMisintop20\%) \wedge (CM > 10 \wedge ChC > 5))$$

where CM is the number of changing methods and ChC is the number of changing classes in a system. According to the metrics model above, the result will be either zero or one, indicating, respectively the absence or presence of the bad smell.

Feature Envy

Feature envy bad smell occurs when a method uses other classes' data more than its own. Li and Shatnawi [23] suggest:

$$AID > 4 \wedge (AIDisintop10\%) \wedge ALD > 3 \wedge NIC < 3$$

where AID is the access of import data, ALD is the access of local data and NIC is the number of import classes. The result of this model will be an integer value that refers to the number of feature envy methods in the class.

Parallel Inheritance Hierarchy

Parallel Inheritance Hierarchy is a bad smell when an inheritance tree depends on another inheritance tree by composition. They maintain a special relation where one subclass of dependent inheritance must depend on a particular subclass of another inheritance. Danphitsanuphan and Suwantada [12] propose:

$$DIT > 3 \vee NSC > 4$$

where DIT is a depth of inheritance tree and NSC is the number of children of the given class.

Lazy Class

Any class that is created has a development and maintenance cost. Lazy class bad smell occurs if a class does not justify its cost. Danphitsanuphan and Suwantada [12] suggest detecting this smell with the following:

$$(NOM < 5 \wedge NOA < 5) \vee DIT < 2$$

where DIT is the depth of inheritance tree, NOM is the number of methods and NOA is the number of attributes of the given class.

Message Chains

The message chain bad smell arises when accessing an object through another object which assesses yet another object, and so forth. It goes through a long chain of attributes or methods by the dot operator [9]. Chen et al. [9] propose the following metric model:

$$LMC \geq 4$$

where LMC is the length of message chain.

Data Class

This smell occurs when a class contains fields and only getters/setter methods. Danphitsanuphan and Suwantada [12] suggest:

$$LWMC > 50 \vee LCOM > 0.8$$

where LWMC is the weighted methods per class and LCOM is the lack of cohesion of methods.

Refused Bequest

Subclasses that override functionality inherited from the parent class exhibit the refused bequest bad smell. Li and Shatnawi [23] propose:

$$(((AIURisinbottom25\%) \wedge (NOTDIT < 2))) \wedge AIUR < 33)$$

where AIUR is the average inheritance usage ratio and DIT is the depth of inheritance tree. This model produces a count of the methods the class inherits and does not use those inherited methods [23].

3.2.2 Bad Smells that are not investigated due to metrics model unavailability

There are other smells described Fowler and Beck [16] that have been investigated. However, we are not aware of any metrics model and also could not implement any that detects the smells below in the Python code yet.

Duplicated Code

Fowler [16] defines duplicated code as multiple occurrences of the same code segment. There is controversy among the studies whether duplicated code is the potential indicator of software problems or not.

Divergent Change

Divergent change bad smell occurs when a same single class is being changed for several different reasons. Fowler [16] explains that smell: "Well, I will have to change these three functions every time I get a new database; I have to change these four functions every time there is a new financial instrument," as an indication of divergent change. This violates the Single Responsibility principle that states every module should have one reason to be changed.

Data Clumps

This smell occurs when a set of unrelated data items, such as parameter lists of fields, that are grouped and used together throughout various parts of the program [16].

Primitive Obsession

Primitives are canonical data types built into programming languages. This smell occurs when the primitives are used to represent higher-level abstractions.

Temporary Field

Fowler [16] states that this bad smell occurs when an algorithm uses a temporary variable beyond its intended scope. We could not find any metric model to identify temporary field smell.

Middle Man

This bad smell occurs when a class delegates most of its work to other code components [16].

Alternative Classes with Different Interfaces

Two classes performing the similar functions, but with different method names illustrates this bad smell. We are unaware of any metrics to detect this smell.

Incomplete Library Class

This smell occurs when a library of methods does not adequately support the library's intended abstraction. We could not find any metrics to identify this smell in the code until now.

Comments

Comments are parts of the code that helps the developers understand what the code does. However, they can also be a sign of bad smell if the accompanying code cannot be understood without them. There are not enough resources to detect comments bad smells in the code that we are aware of.

Inappropriate Intimacy

This smell occurs when classes are too intimate into each other's private parts or inner data. Python gives the developer the ability to create public, private, and protected methods and variables within a class. Unlike other object-oriented programming languages, access modifiers in Python are determined by variable names. A name proceeded by a double underscore signifies private access; a single underscore indicates protected access. All the other variables and member functions of a class are public. Although we can discover the access modifiers in the code, we are not aware of any metrics model that detects inappropriate intimacy in the Python code yet.

Switch Statements

Switch statement bad smell occurs when you see the same switch statement in different places of your code. Unlike Java or C-like languages, Python does not have a switch or case statement. Dictionary mapping is used instead of switch-case statements. Switch statements bad smells are detected using VG, McCabe Cyclomatic Complexity, in other object-oriented programming

languages like Java, C-like languages, but we are unsure if this metric would be applied to Python code.

Speculative Generality

This smell occurs when there are unused fields, parameters, methods, or classes. There is no metrics have been found yet to identify this smell.

3.3 Research Approach

Our overarching research goal is to determine the extent to which bad smells lead to future defects. To accomplish this, we examined the change logs and defect logs of sample open-source Python projects to determine how many defects can be traced to code segments into bad smells and which bad smells contribute to the highest number of defects.

The research entails automating the identification of Python bad smells and automating a means of associating reported defects to subsequent code changes. The work results in a set of tools that analyze git repositories and defect logs for smell-to-defect correlation as well as an empirical analysis of at least two open source Python projects.

3.4 Research Methodology

We developed a tool that analyzes the Python projects. The functionality of our tool can be summarized as follows:

1. Our tool first downloads the open-source code repository that is going to be reviewed.

    1.1. An open-source repository is downloaded to the local directory using “git clone” command.

2. The tool extracts the information from each revision (e.g., Commit, date, and subject). It only extracts the modified Python files. It is not checking the deleted files because we

cannot prove that bad smells lead to bugs since it is deleted. It is not checking the added files because they are recently added to the repository, and they will not be having the previous version to compare to.

2.1. To be able to download/clone the open-source project, we used the GitPython library as a means by which to interact with Git repositories. With the help of GitPython, we are able to get the log reports using the expression below:

"git log —log-size —-name-status –pretty=format:

where –log-size refers to the size of the log, –name-status shows the list of files affected with added, modified and deleted information and –pretty-format shows commits in a format where we want only "%h-commit hash", "%cd-committer date" and "%s-subject" of the given commit to detect the files with bad smells. We used the expression above to get the list of modified Python files with their commitID, dates and subjects.

Once we get the list of modified files, we needed to get the state or copy of the each changed file using it's commit id. Commit id helps us to find the file content at that date and subject. We use the expression below to store the copy of each changed file:

"git show commitId:fileURL"

In this statement, git show indicates the file contents at that commit. We save each changed file in a local folder. We then apply our metrics to identify Python bad smells on the files. Figure 3.1 below is the flow of identifying bad smells in our tool:

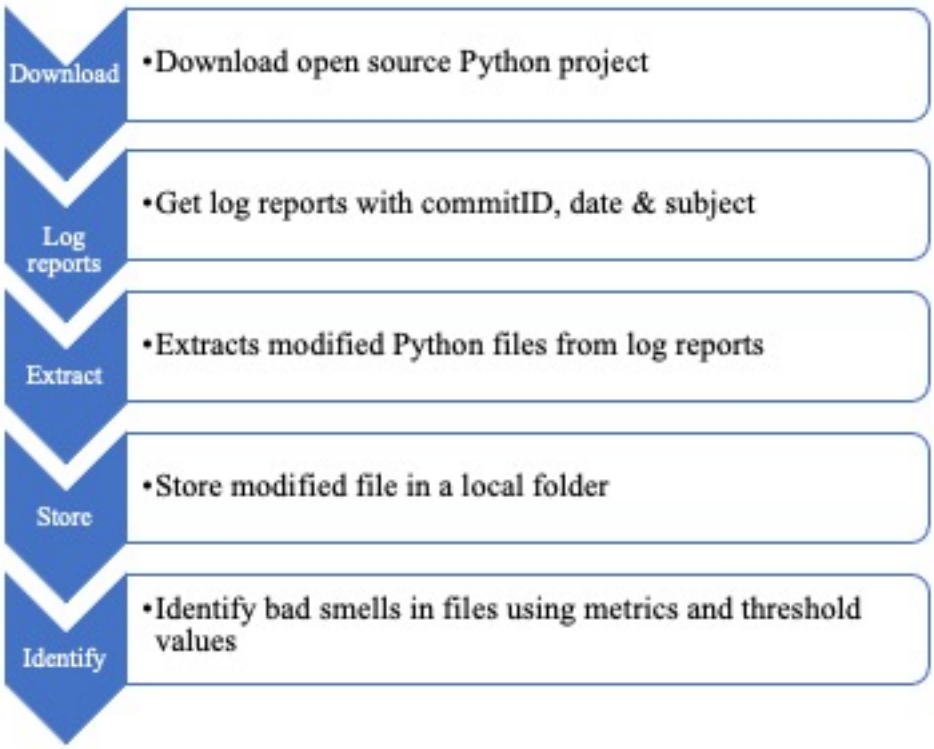


Figure 3.1: The steps of identifying bad smells in the research plan

3. Our tool checked the issue tracking system to find the links between commits (change logs) and the issues that are used to track ideas, tasks, fixes on the code and also bugs.

   3.1. The code downloads the bug reports from the issue tracking system.

   3.2. It extracts reports that are labeled as “Bug”.

   3.3. It links files with bugs and files with bad smells in it. In order to link bad smells to a bug, two independent level of confidence were adapted from Sliwerski et al. [35]:

      3.3.1. Syntactic analysis: the tool splits every log message into a stream of tokens. A token is one of the items below:

         A. Bug number: For a commit to include a bug number, it needs to have one of the regular expressions below:

            “bug[# \t]*[0-9]+” or “bug\?id=[0-9]+” or “\[[0-9]+\]”

B. Plain number: the tool will check if a string has only digits or not:

"[0-9]+"

C. Keyword: the tool will check for some keywords:

"fix(e[ds])?" or "bug(s)?" or "defect(s)?" or "patch"

D. Word: the tool will check if it is a string of alphanumeric characters or not.

According to Sliwerski et al. [35], for each link there will be a syntactic confidence value. It will initially be 0. The value of syntactic confidence will be increased by 1 when one of the following conditions is met and the value will always be an integer between 0 and 2:

- When the number is a bug number,
- When the change log contains a keyword, or the log message contains only plain or bug number.

3.3.2. Semantic analysis: this analysis validates a link between the change logs and bugs in the issue tracking system. For this analysis, we used Sliwerski et al. [35] and Cheung et al. [28] analysis. A semantic confidence value is calculated for each link according to the following:

A. The issue is tagged as "bug" and the state of this issue should be closed [35],

B. The short description of the bug report in issue tracking system is contained in the change log message [35],

C. The author of the change log message has been assigned to bug [35],

D. If the author of issue has been assigned to change log message and issue number is contained in the change log message [28].

Semantic confidence value is initially zero and is incremented by one whenever one of the conditions above is met.

The syntactic and semantic analysis results in links that satisfy the formula below in Equation 3.2:

$$semanticanalysis > 1 \vee (semanticanalysis = 1 \wedge syntacticanalysis > 0) \tag{3.2}$$

Once our project detects the bad smells on the files in change logs and then applies the semantic and syntactic analysis, we are then able to determine the extent to which bad smells contribute to later problems. Figure 3.2 shows the flow of our research plan:

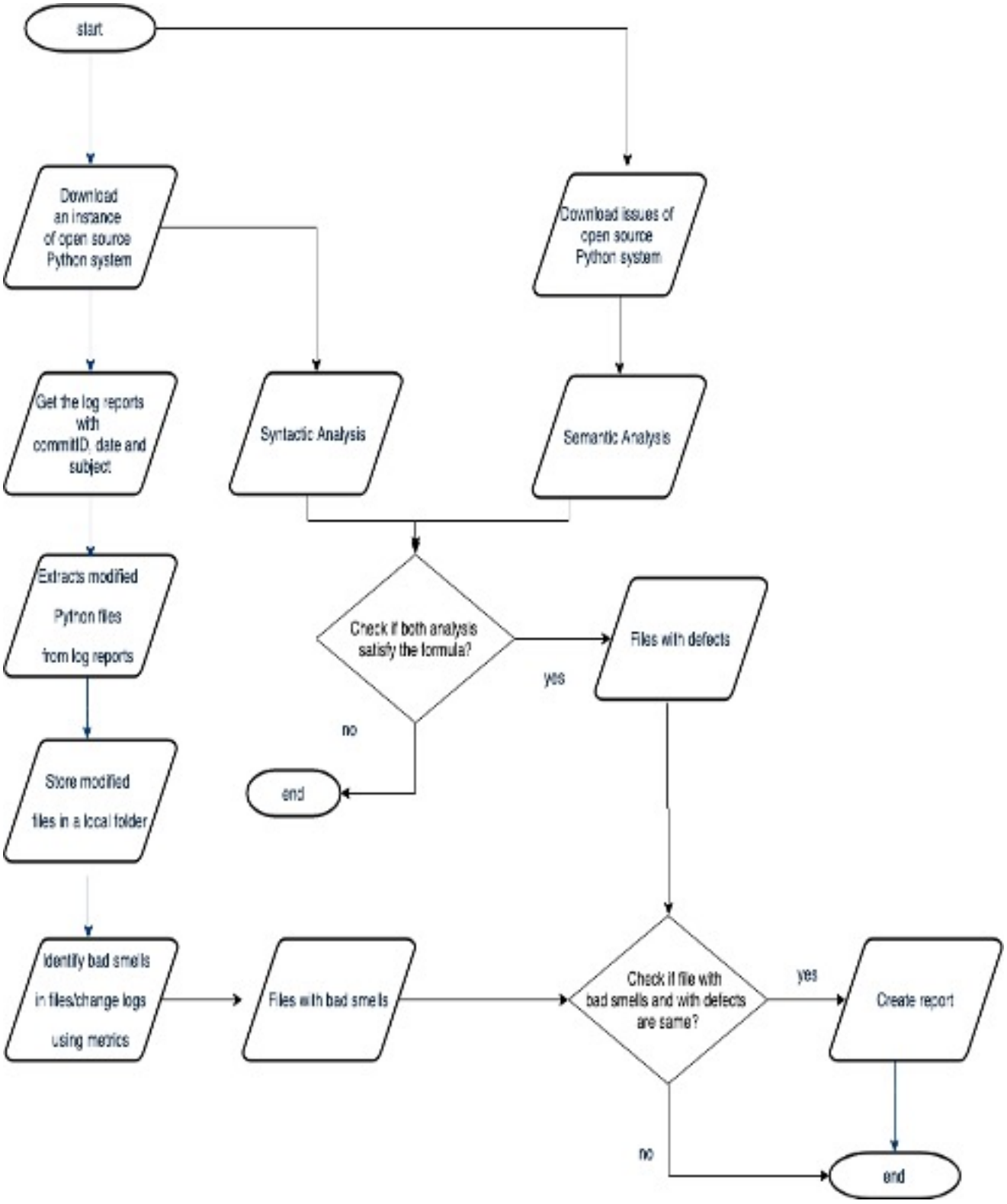


Figure 3.2: The flow of the research plan

Chapter 4

Validation

We implemented a tool to detect bad smells using the metrics model and thresholds, analyzed the issues and commits following the research plan explained in Chapter 3.4, and found the relation between bad smells and defects using the results that we got from our project. The project had three main components as shown in Figure 4.1. The leftmost part of Figure 4.1 begins with downloading the open-source project code, git log reports and issues of the open-source project and modified files across the life of the project. The information was provided to the detection component where the bad-smell metric model was applied. Issues and commits were then analyzed to determine if they contained bad smells. The number of bad smells removed as a result of issue repair was identified and reported.

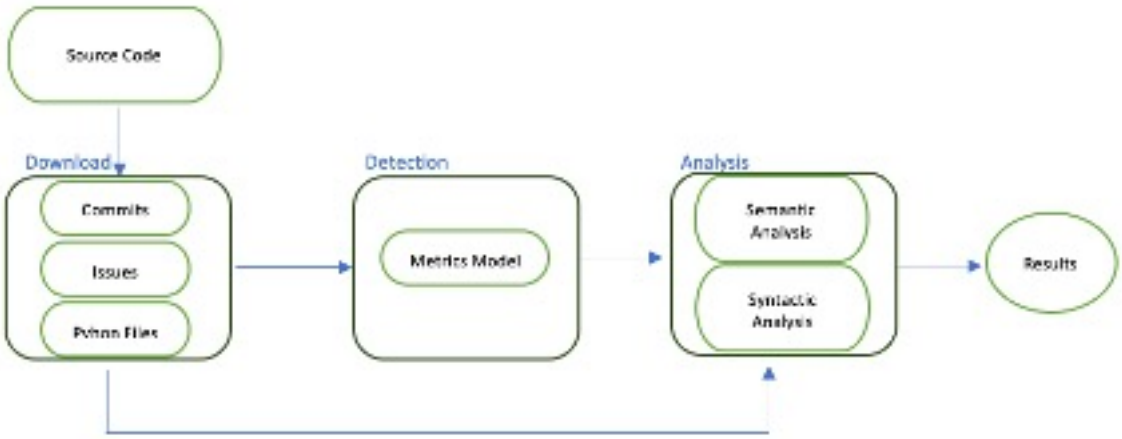


Figure 4.1: Main components of the Dissertation Project

We investigated the relation between bad smells and defects on five different Python open-source projects: NumPy [1], Keras [4], scikit-learn [4], zulip [3] and TensorFlow [2]. Table 4.1 below presents when the initial commit of each project was started, the number of lines, files, commits and issues that have occurred over the lifetime of each project.

Python Project	Initial Commits	Total Files	Total Lines	Total Commits	Total Issues
NumPy	Dec 18, 2001	351	83605	24342	17525
Keras-team	Mar 27, 2015	471	132614	5349	14239
scikit-learn	Jan 5, 2010	617	100769	26223	18678
zulip	Aug 28, 2012	916	74887	39398	16504
Tensorflow	Jan 12, 2016	1478	222595	5941	9342

Table 4.1: Python Open-Source Project Details

The issues and commits for all projects were downloaded for the analyses on Oct 28, 2020. Figure 4.2 is a snapshot of some commits obtained from NumPy that were used for our project. The 'commitID' on the figure is the revision number in the repository. The 'Name' and 'Email' have the information of the user who made the commit. The 'Date' shows when it is committed. The 'Subject' describes what the commit is about.

commitID	Name	Email	Date	Subject
822e00b9b	Ralf Gommers	ralf.gommers@gmai	Fri Oct 9 23:06:10 2020 +0100	Merge pull request #17520 from bjnath/show-real-license
60e131932	Charles Harris	charlesr.harris@gm	Fri Oct 9 16:03:20 2020 -0600	Merge pull request #17525 from rossbar/circled_envvar_fix
7f1d631581	Ross Barnowski	rossbar@berkeley.e	Fri Oct 9 13:42:57 2020 -0700	Fix yaml parsing error.
0485b2396	Ross Barnowski	rossbar@berkeley.e	Fri Oct 9 13:32:56 2020 -0700	Fix bash syntax
eed58620a	Ross Barnowski	rossbar@berkeley.e	Fri Oct 9 13:25:21 2020 -0700	CI: modify circleci merge ref grabbing.
156ed054e	Lisa Schwetlick	mail@lisaschwetlick.	Fri Oct 9 12:35:47 2020 -0700	ENH: add dtype option to csv and corrcoef (#17456)
84a4fcb5a	Ben Nathanson	github@bigriver.ayz	Fri Oct 9 12:27:34 2020 -0700	DOC: Fix empty 'C style guide' page (#17503)
328135c3a	Charles Harris	charlesr.harris@gm	Fri Oct 9 13:07:54 2020 -0600	Merge pull request #17344 from danbeibei/master
73c555fd5	Ben Nathanson	github@bigriver.ayz	Fri Oct 9 14:25:44 2020 -0400	DOC: Display real license on license page
55ed49819	Charles Harris	charlesr.harris@gm	Fri Oct 9 10:37:46 2020 -0600	BUG: Fix indentation.
2262a7d7a	Charles Harris	charlesr.harris@gm	Fri Oct 9 07:56:40 2020 -0600	Merge pull request #17508 from lucascolley/typing-final-feature
dee73994c	Charles Harris	charlesr.harris@gm	Fri Oct 9 07:54:50 2020 -0600	Merge pull request #17479 from person142/no-unicode

Figure 4.2: Example of commits obtained from NumPy project

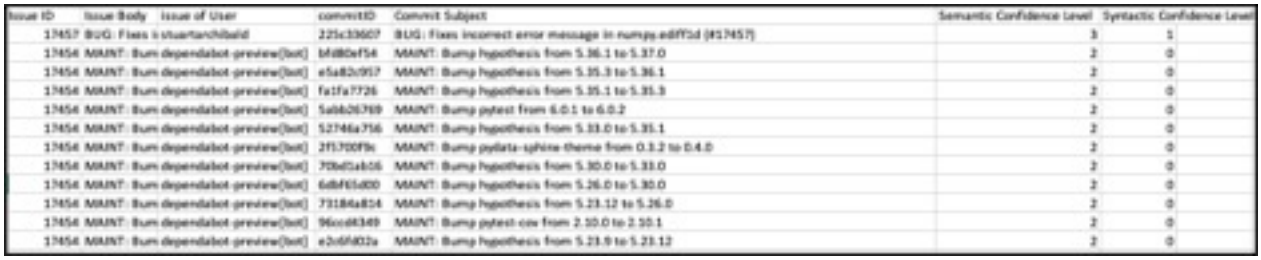
The issues on GitHub are a way to keep track of tasks, changes, and bugs for the projects. Issues for our project were filtered according to their label and status. Only issues saved as 'closed' and labeled as 'bug' or had 'bug' as a keyword in the issue's title or body were selected to use for the analysis. Figure 4.3 shows a screenshot of the issues that were used for our project. The 'id' on the figure is the issue identification number. 'Title' describes what the issue is all about. 'Body' provides a detailed description about the issue. 'User' is the person who is responsible for working on the issue at a given time. 'Label' shows what the type of issue is.

id	Title	Body	User	Label
17525	CI: Make merge ref grabbing condi	An attempt to fix #17524	rossbar	b'00 - Bug c
17502	BUG: remove `sys` from the type s	It is the builtin `sys`	person142	b'00 - Bug st
17501	BUG: Fix failures in master relatec	There are two failures	seberg	b'00 - Bug'
17499	Revert "BUG: allow registration of	Reverts numpy/numpy#1	mattip	None
17475	Error: Import Numpy and Subsequ	I do not know how to fix	hxwood	None
17472	TST: Fix doctest for full_like	<!--	jsignell	b'00 - Bug 0
17466	numeric exceptions in simple func	<!-- Please describe the	BishopWolf	None
17462	unexpected behavior of in for lists	The behavior of `in` with	tylerharvey	None

Figure 4.3: Example of Filtered Issue List

After the issues and commits were obtained, the syntactic and semantic analysis was run on the filtered issues. When both the syntactic and semantic analysis were done, only the commits that satisfied the formula in Equation 3.2 were used. That formula gave us the bug

fixed commits. Figure 4.4 below shows a snapshot of some of bug fixed commits found after the analysis. The columns in Figure 4.4 that have 'Issue ID', 'Issue Body' and 'Issue of User' on them have information about the bug fixed issue and the columns that are named as 'commitID', and 'Commit Subject' have information about the commits. The analysis was made to both issue and commit. The semantic and syntactic confidence level are noted in the 'Semantic Confidence Level' and 'Syntactic Confidence Level' columns, respectively.



Issue ID	Issue Body	Issue of User	commitID	Commit Subject	Semantic Confidence Level	Syntactic Confidence Level
17457	BUG: Fixes i	stuartarchibald	225c33607	BUG: Fixes incorrect error message in numpy.eldiff1d (#17457)	3	1
17454	MAINT: Bum	dependabot-preview[bot]	bfd80ef54	MAINT: Bump hypothesis from 5.36.1 to 5.37.0	2	0
17454	MAINT: Bum	dependabot-preview[bot]	e5a82c957	MAINT: Bump hypothesis from 5.35.3 to 5.36.1	2	0
17454	MAINT: Bum	dependabot-preview[bot]	fa1fa7726	MAINT: Bump hypothesis from 5.35.1 to 5.35.3	2	0
17454	MAINT: Bum	dependabot-preview[bot]	5ab6b2769	MAINT: Bump pytest from 6.0.1 to 6.0.2	2	0
17454	MAINT: Bum	dependabot-preview[bot]	52746a756	MAINT: Bump hypothesis from 5.33.0 to 5.35.1	2	0
17454	MAINT: Bum	dependabot-preview[bot]	2f1700f9c	MAINT: Bump pydata-sphinx-theme from 0.3.2 to 0.4.0	2	0
17454	MAINT: Bum	dependabot-preview[bot]	70bd2ab16	MAINT: Bump hypothesis from 5.30.0 to 5.33.0	2	0
17454	MAINT: Bum	dependabot-preview[bot]	6dbf65d00	MAINT: Bump hypothesis from 5.26.0 to 5.30.0	2	0
17454	MAINT: Bum	dependabot-preview[bot]	731846a814	MAINT: Bump hypothesis from 5.23.12 to 5.26.0	2	0
17454	MAINT: Bum	dependabot-preview[bot]	96ccd8349	MAINT: Bump pytest-cov from 2.10.0 to 2.10.1	2	0
17454	MAINT: Bum	dependabot-preview[bot]	e2c6fd02a	MAINT: Bump hypothesis from 5.23.9 to 5.23.12	2	0

Figure 4.4: Shows a screenshot of the spreadsheet after both the analysis are done

Only commits that had a semantic confidence level greater than 1 or those whose semantic confidence level was 1 and syntactic confidence level was greater than 0 were used. To illustrate this, all commits can be used in Figure 4.4 since all the commits' semantic confidence levels are greater than 1.

After the bug fixed commits were identified, and modified files were downloaded with their version numbers, the modified files that had the same commit ID (version number) with the bug fixed commits were identified in order to find if a bad smell led to a defect or not. Once the files with bug fix commit IDs were found, previous version number of those corresponding files were also found and investigated. Figure 4.5 is an example to show how we found the relation between the bad smell and a defect. The left side of Figure 4.5, 'Commits' shows a list of commits that have all commits including bug fix commits in it and the right side shows list of folders that have modified files with their version number (commitID) and modified date. Figure 4.5 illustrates that the commit list has one bug fix commit, which is 'Commit15'. The version or commit number of 'Commit15' according to the figure is X. The project code iterates

through the modified files within the Folders, the right side of Figure 4.5 and looks for the files that were modified at version number X. According to the figure, 'File_CommitX_atTime6' in 'Folder1' is modified at version number X at 'Time6'. The versions should already be identified according to their modification date before any analysis. Once the file with version number X is found, then the code checks when that file was modified. According to the figure, 'File_CommitX_atTime6' in 'Folder1' is modified at version number X at 'Time6' and the previous version of the same file is modified at 'Time5'. Once both versions are found, a metrics model for bad smells is performed on both versions. If the file modified at 'Time5' has a bad smell, but the file at 'Time6' doesn't have a bad smell, that may indicate the modification on the file is due to bad smell.

Figure 4.5: Shows an example of the process about how the project works to find the relation between bad smells and defects

4.1 Accuracy of Smell Detection

The precision and recall in detecting python smells were measured to compare the accuracy of smell detection of our project. In order to validate the results, we manually detected the bad smells in NumPy project and got the result using this project code. In order to calculate the precision and recall, the methodology is listed below [29]:

- True positives are instances (classes or methods) present in the code smell reference list that are reported by the project being assessed

- False positives are instances that are not present in the reference list, but they were reported by the project.
- False negatives are instances that are present in the reference list that were not reported by the tool.
- True negatives are instances that are not present in the reference list and were also not reported by the tool.

The formulas for the computation of recall, precision and F-measure are provided below:

$$recall = \frac{number of true positives}{number of instances in the reference list} \tag{4.1}$$

$$precision = \frac{number of true positives}{number of instances reported by the project} \tag{4.2}$$

$$F - measure = \frac{(2 * precision * recall}{(precision + recall} \tag{4.3}$$

For precision, recall and F-measure, the score should be close to 1 to be able to assess the accuracy of the computations.

4.2 Bayes Theorem Calculation on Defect-Smell Analysis

To be able to determine the relation between bad smells and defects, Bayes' theorem was used. This theorem determines the likelihood of an event occurring based on a previous outcome occurring or the probability of an event based on new data obtained from that event [19]. It relies on the inclusion of prior probability distributions to produce posterior probabilities. Prior probability is the probability of an outcome based on the current data before new data is obtained and posterior probability is the updated probability of an outcome occurring after new data is obtained. In Bayes' theorem, posterior probability is calculated by updating the prior probability. Bayes' theorem [19] given in Equation 4.4 provides the calculation of the probability of

event A occurring given even B:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) * P(B|A)}{P(B)} \tag{4.4}$$

Where $P(A)$ is the probability of an event A occurring, $P(B)$ is the probability of an event B occurring, $P(B \mid A)$ is the probability of an event B occurring given event A. In this project, Bayes' formula was used to determine the probability of if a bad smell leads to defect or not.

4.3 Measuring Association Between Code Smells and Defects

Correlation analysis is used to measure the relationship between the number of code smells and the probability of defects occurring on python open-source projects for this research. The CORREL formula in Excel is used to calculate the correlation coefficients. The correlation factor (r) is a number between -1 and 1. There are many rules of thumb for interpreting the size of a correlation coefficient. However, for this study, Table 4.2 by Hinkle et al. [18] was used.

Correlation Coefficient(r)	Description
.90 to 1.00 (-.90 to -1.00)	Very high positive (negative) correlation
.70 to .90 (-.70 to -.90)	High positive (negative) correlation
.50 to .70 (-.50 to -.70)	Moderate positive (negative) correlation
.30 to .50 (-.30 to -.50)	Low positive (negative) correlation
.00 to .30 (.00 to -.30)	Negligible correlation

Table 4.2: Correlation Coefficient Description

4.4 Code Smell-Defect Relation Findings on Python Open-Source Projects

4.4.1 Large Class

This smell occurs when a class has too many responsibilities. Classes usually start small, but then the developers include many fields, methods, and lines of code to the class over time. As

the class gets bloated, it becomes complex, typically exhibiting low and inner-class cohesion [28].

Accuracy of Large Class Smell Detection

To assess the accuracy of large class detection strategy of the project, the precision, recall, F-measure of detection results were computed. The tool found 24 large classes and 20 large classes were found with manual detection in NumPy project. That gives us the recall as 0.83 and precision as 1 and F-measure as 0.907.

Large Class Analysis

Open-source Python projects were analyzed to determine if a large-class bad smell was related to defects in which the bad smell was removed. Naïve Bayes Equation which is given in Equation 4.5 is to resolve the relation between large class bad smell and defects.

$$P(Defect = Yes|BadSmell = LargeClass) = \frac{P(Defect=Yes)\times P(BadSmell=LargeClass|Defect=Yes)}{P(BadSmell=LargeClass)} \quad (4.5)$$

Large Class in NumPy

NumPy is a small-scale open-source project. It is the smallest project in Table 4.1. The results show that there are 634 classes in total in NumPy. Only 24 out of 634 classes were found to be large class in NumPy project. 3 of 24 large classes were modified and 12 of 610 regular classes removed from the project because they led to defects. Table 4.3 displays the frequency table of NumPy project.

Numpy Frequency Table			
isLargeClass	Defect=Yes	Defect=No	Total
Yes	3	21	24
No	12	609	610
			634

Table 4.3: Large Class Frequency table of 'NumPy'

Using Table 4.3, the probability of a defect occurring when a class has a large class smell and the probability of each event occurring is calculated and presented in Table 4.3.

Numpy Likelihood Table	
P(Defect=Yes)	15/64
P(Bad Smell=Large Class | Defect=Yes)	3/15
P(Bad Smell=Large Class)	24/634
P(Defect=Yes | Bad Smell=Large Class)	(3/15 * 15/634) / (29/634) = 0.103

Table 4.4: Large Class Likelihood table of 'NumPy'

Table 4.4 shows that the probability of a defect occurring due to a large class smell in NumPy project is 0.103

Large Class in Keras-Team

Keras-Team is the second smallest project in Table 4.1. The results of Keras-Team project show that there are 107 classes in total. There are 20 large classes in the Keras-Team project. There were not any class removed or modified from the project because they led to defects. Table 4.5 below shows the frequency table of Keras-Team project.

Keras-Team Frequency Table			
isLargeClass	Defect=Yes	Defect=No	Total
Yes	0	20	20
No	0	87	87
			107

Table 4.5: Large Class Frequency table of 'Keras-Team'

Using the table above, the probability of defect occurring when a class has a large class bad smell and the probability of each event occurring is computed and the results are provided in Table 4.6. It shows that the probability of a defect occurring due to a large class smell in Keras-Team project is 0.

Keras-Team Likelihood Table	
P(Defect=Yes)	0/107
P(Bad Smell=Large Class | Defect=Yes)	0
P(Bad Smell=Large Class)	20/107
P(Defect=Yes | Bad Smell=Large Class)	((0/107 * 0) / (20/107) = 0

Table 4.6: Large Class Likelihood table of 'Keras-Team'

Large Class in scikit-learn

scikit-learn is the third largest project according to Table 4.1. The results of scikit-learn project show that there are 413 classes in total. 36 large classes were found in this project. There were no regular classes removed from the project because they led to defects, but only 1 class was modified because it led to defect. Table 4.7 shows the frequency table of scikit-learn project.

scikit-learn Frequency Table			
isLargeClass	Defect=Yes	Defect=No	Total
Yes	1	35	36
No	0	377	377
			413

Table 4.7: Large Class Frequency table of 'scikit-learn'

The probability of a defect happening when a class has a large class bad smell and the probability of each event occurring are presented in Table 4.8. It is calculated as 0.027.

scikit-learn Likelihood Table	
P(Defect=Yes)	1/413
P(Bad Smell=Large Class | Defect=Yes)	1/1
P(Bad Smell=Large Class)	36/413
P(Defect=Yes | Bad Smell=Large Class)	(1/1 * 1/413) / (36/413) = 0.027

Table 4.8: Large Class Likelihood table of 'scikit-learn'

Large Class in Zulip

zulip is the second largest project with 525 classes in total. We found that there were only 10 large classes in this project. 1 class was modified, and 6 classes were removed from the project because they led to defects. Table 4.9 illustrates the frequency table of zulip project.

Zulip Frequency Table			
isLargeClass	Defect=Yes	Defect=No	Total
Yes	1	9	10
No	6	509	515
			525

Table 4.9: Large Class Frequency table of 'Zulip'

Using the table above, we calculated the probability of defect occurring due to a large class bad smell in zulip project and the results presented in Table 4.9.

Zulip Likelihood Table	
P(Defect=Yes)	7/525
P(Bad Smell=Large Class | Defect=Yes)	1/7
P(Bad Smell=Large Class)	10/525
P(Defect=Yes | Bad Smell=Large Class)	(1/7 * 7/525) / (10/525) = 0.1

Table 4.10: Large Class Likelihood table of 'Zulip'

Large Class in Tensorflow

Tensorflow is the largest project that was used for this analysis. It had 180 classes in total and only 10 classes had the large class bad smell. There were 1 class change found due to defects. Table 4.11 shows the frequency table of Tensorflow project.

Tensorflow Frequency Table			
isLargeClass	Defect=Yes	Defect=No	Total
Yes	1	9	10
No	0	170	170
			180

Table 4.11: Large Class Frequency table of 'Tensorflow'

Table 4.12 shows the probability of defect occurring when a class has a large class bad smell and the probability of each event occurring and it is calculated as 0.1.

Tensorflow Likelihood Table	
P(Defect=Yes)	1/180
P(Bad Smell=Large Class | Defect=Yes)	1/1
P(Bad Smell=Large Class)	10/180
P(Defect=Yes | Bad Smell=Large Class)	(1/180 * 1/1) / (10/180) = 0.1

Table 4.12: Large Class Likelihood table of 'Tensorflow'

Large Class Findings

Table 4.13 shows us the results of five different size Python projects for large class smell.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Large Classes	Probability of a Defect due to a Large Class Smell
NumPy	351	83605	24342	2074	29	0.103
Keras-team	471	132614	5349	21	20	0
scikit-learn	617	100679	26223	99	36	0.027
zulip	916	74887	39398	5066	10	0.1
Tensorflow	1478	222595	5941	23	10	0.1

Table 4.13: Results - Probability of a defect due to a large class bad smell of all projects

The relationships between number of bug-fixed commits and number of classes with large class smell and between the probability of a defect occurring and number of classes with large class smell were determined using scatter plots. A visual inspection of the graphs shows that distribution in Figure 4.6 and Figure 4.7 are random. Large class findings indicate that there exists a low negative correlation (-0.428) between the number of bug fixed commits and large class smell.

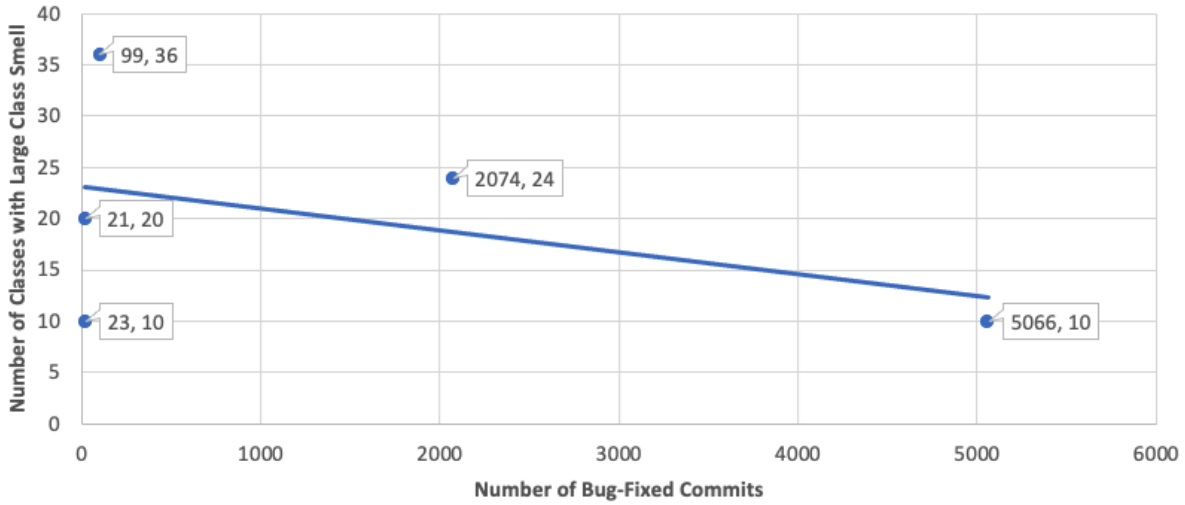


Figure 4.6: The relation between the probability of a defect due to large class and number of bug fixed commits

The correlation coefficient between the number of large classes and probability of a defect happening because of a large class bad smell were also computed (Figure 4.7). The analysis reports -0.568 as correlation coefficient. However, it is not a perfect negative correlation because when the number of large classes increases, the probability of a defect occurring due to large class smell does not necessarily increase. Given the values for both ranges, it is undecidable whether classes with large class smell lead to defects.

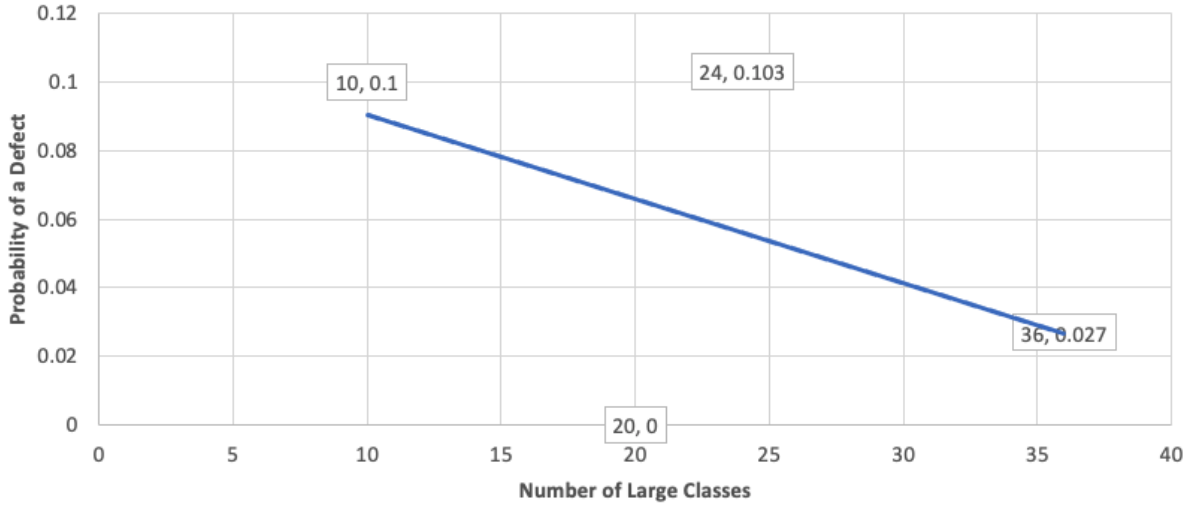


Figure 4.7: The relation between the probability of a defect due to Large Class Smell and Number of Large Classes

4.4.2 Long Parameter List

Long parameter list in a method definition is a bad smell. This might happen when a method needs to do too many things, or it is created to minimize dependencies between objects. Methods with long parameter lists are hard to read and understand. As the methods grow, they will be hard to use.

Accuracy of Long Parameter List Smell Detection

The long parameter list smell detection was also validated using precision and recall. 241 methods with long parameter list smell were manually detected and the tool found 248 methods in total in the NumPy open-source project. The recall is calculated as 0.846 and precision as 0.822 and F-measure as 0.833.

Long Parameter List Analysis

The probability of a python open-source project leads to defects when a method has long parameter list (LPL) smell was calculated using the formula in Equation 4.6.

$$P(Defect = Yes|BadSmell = LPL) = \frac{P(Defect=Yes)\times P(BadSmell=LPL|Defect=Yes)}{P(BadSmell=LPL)} \quad (4.6)$$

The results for each of open-source Python project for the long parameter list smell is provided in Tables 4.14 through 4.23:

Numpy Frequency Table			
isLongParameterList	Defect=Yes	Defect=No	Total
Yes	11	237	248
No	94	1598	1692
			1940

Table 4.14: Long Parameter List Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	105/1940
P(Bad Smell=LPL | Defect=Yes)	11/105
P(Bad Smell=LPL)	248/1940
P(Defect=Yes | Bad Smell=LPL)	(11/105 * 105/1940) / (248/1940) = 0.044

Table 4.15: Long Parameter List Likelihood table of 'NumPy'

Keras-team Frequency Table			
isLongParameterList	Defect=Yes	Defect=No	Total
Yes	0	86	86
No	10	438	448
			534

Table 4.16: Long Parameter List Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	10/534
P(Bad Smell=LPL | Defect=Yes)	0/10
P(Bad Smell=LPL)	86/534
P(Defect=Yes | Bad Smell=LPL)	(10/534 * 0/10) / (86/534) = 0

Table 4.17: Long Parameter List Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isLongParameterList	Defect=Yes	Defect=No	Total
Yes	4	236	240
No	11	832	843
			1083

Table 4.18: Long Parameter List Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	15/1083
P(Bad Smell=LPL | Defect=Yes)	4/15
P(Bad Smell=LPL)	240/1083
P(Defect=Yes | Bad Smell=LPL)	((4/15* 15/1083) / (240/1083) = 0.016

Table 4.19: Long Parameter List Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isLongParameterList	Defect=Yes	Defect=No	Total
Yes	2	163	165
No	41	1928	1969
			2134

Table 4.20: Long Parameter List Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	43/2134
P(Bad Smell=LPL | Defect=Yes)	2/43
P(Bad Smell=LPL)	165/2134
P(Defect=Yes | Bad Smell=LPL)	(2/43* 43/2134) / (165/2134) = 0.012

Table 4.21: Long Parameter List Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isLongParameterList	Defect=Yes	Defect=No	Total
Yes	0	100	100
No	22	696	716
			816

Table 4.22: Long Parameter List Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	22/816
P(Bad Smell=LPL | Defect=Yes)	0/22
P(Bad Smell=LPL)	100/816
P(Defect=Yes | Bad Smell=LPL)	(22/816* 0/22) / (100/816) = 0

Table 4.23: Long Parameter List Likelihood table of 'Tensorflow'

Long Parameter List Findings

Table 4.24 summarizes the projects findings for long parameter list bad smell. The result of this smell suggests that when the number of methods with long parameter list smell increases, the probability of defect occurring due to long parameter list smell also increases.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Methods with LPL	Probability of a Defect due to a LPL
NumPy	351	83605	24342	2074	248	0.044
Keras-team	471	132614	5349	21	86	0
scikit-learn	617	100679	26223	99	240	0.016
zulip	916	74887	39398	5066	165	0.012
Tensorflow	1478	222595	5941	23	100	0

Table 4.24: Results - Probability of a defect due to a long parameter list bad smell of all projects

Figure 4.8 is a graph below displays the correlation between number of bug fixed commits and the methods with long parameter list smell in it. The correlation coefficient is computed as 0.234 in this case. The correlations for these two rankings turned out low because as the number of bug fixed commits increase, the number of methods with smells does not necessarily increase. This graph suggests that there is weak correlation between two rankings.

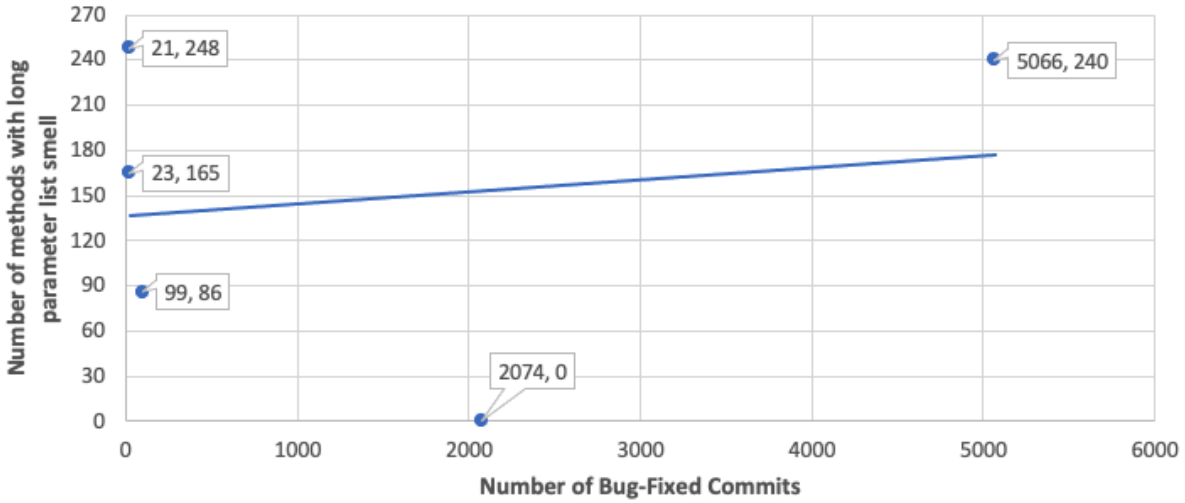


Figure 4.8: The relation between the Number of Bug-fixed Commits and methods with long parameter lists

The correlation analysis was performed to determine the relation between long parameter list smell and defects. The correlation coefficient between those two sets detected is 0.8518. These analyses show that there is a high positive relation between the number of methods with long parameter list smell and the probability of a defect occurring shown in Figure 4.9. The presence of long parameter list is a strong indication of the presence of defects. The findings illustrate projects having more than 100 methods with long parameter list may lead to defects. As the number of methods with smell increase, the probability of a defect occurring due to long parameter list smell also increases.

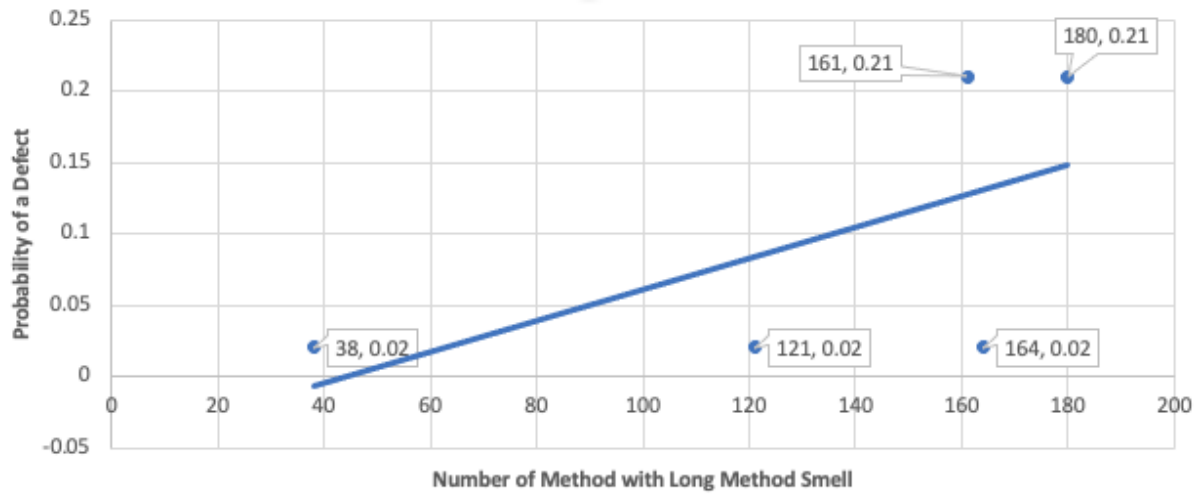


Figure 4.9: The relation between the probability of a defect and Number of Long Parameter List

4.4.3 Parallel Inheritance Hierarchy

When you introduce a subclass to a class, you then need to introduce another subclass of another class. It occurred mostly because of misunderstanding the responsibility or the wrong relationships between classes. This smell leads to lots of duplicate and unmaintainable code.

Accuracy of Parallel Inheritance Hierarchy Smell Detection

The tool found 53 classes that have parallel inheritance hierarchy smell and 51 classes with the smell were found with manual detection. Using the total number of classes for manual and tool, the recall was calculated as 0.94, the precision 0.98 and F-measure 0.96 for this smell.

Parallel Inheritance Hierarchy Analysis

The objective here was to determine the probability of a Python open-source project leading to defects when a class has the parallel inheritance hierarchy (PIH) smell using Naïve Bayes

Equation shown in Equation 4.7:

$$P(Defect = Yes|BadSmell = PIH) = \frac{P(Defect=Yes)\times P(BadSmell=PIH|Defect=Yes)}{P(BadSmell=PIH)} \quad (4.7)$$

Table 4.25 through 4.34 illustrates the findings for the long parameter smell.

Numpy Frequency Table			
isParallelInheritanceHierarchy	Defect=Yes	Defect=No	Total
Yes	6	47	53
No	41	540	581
			634

Table 4.25: Parallel Inheritance Hierarchy Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	47/634
P(Bad Smell=PIH | Defect=Yes)	6/47
P(Bad Smell=PIH)	53/634
P(Defect=Yes | Bad Smell=PIH)	(47/634* 6/47) / (53/634) = 0.113

Table 4.26: Parallel Inheritance Hierarchy Likelihood table of 'NumPy'

Keras-team Frequency Table			
isParallelInheritanceHierarchy	Defect=Yes	Defect=No	Total
Yes	0	47	47
No	0	60	60
			107

Table 4.27: Parallel Inheritance Hierarchy Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	0/107
P(Bad Smell=PIH | Defect=Yes)	0
P(Bad Smell=PIH)	47/107
P(Defect=Yes | Bad Smell=PIH)	(0/107* 0) / (47/107) = 0

Table 4.28: Parallel Inheritance Hierarchy Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isParallelInheritanceHierarchy	Defect=Yes	Defect=No	Total
Yes	0	94	94
No	12	307	319
			413

Table 4.29: Parallel Inheritance Hierarchy Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	12/413
P(Bad Smell=PIH | Defect=Yes)	0/12
P(Bad Smell=PIH)	94/413
P(Defect=Yes | Bad Smell=PIH)	(12/413* 0/12) / (94/413) = 0

Table 4.30: Parallel Inheritance Hierarchy Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isParallelInheritanceHierarchy	Defect=Yes	Defect=No	Total
Yes	5	53	58
No	39	428	467
			525

Table 4.31: Parallel Inheritance Hierarchy Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	44/525
P(Bad Smell=PIH | Defect=Yes)	5/44
P(Bad Smell=PIH)	58/525
P(Defect=Yes | Bad Smell=PIH)	(44/525* 5/44) / (58/525) = 0.086

Table 4.32: Parallel Inheritance Hierarchy Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isParallelInheritanceHierarchy	Defect=Yes	Defect=No	Total
Yes	0	6	6
No	5	169	174
			180

Table 4.33: Parallel Inheritance Hierarchy Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	5/180
P(Bad Smell=PIH | Defect=Yes)	0/5
P(Bad Smell=PIH)	6/180
P(Defect=Yes | Bad Smell=PIH)	(5/180* 0/5) / (6/180) = 0

Table 4.34: Parallel Inheritance Hierarchy Likelihood table of 'Tensorflow'

Parallel Inheritance Hierarchy Findings

Table 4.35 presents the results obtained from Python projects for the parallel inheritance hierarchy smell.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Classes with Parallel Inheritance Hierarchy Smell	Probability of a Defect due to a Parallel Inheritance Hierarchy Smell
NumPy	351	83605	24342	2074	53	0.113
Keras-team	471	132614	5349	21	47	0
scikit-learn	617	100679	26223	99	94	0
zulip	916	74887	39398	5066	58	0.086
Tensorflow	1478	222595	5941	23	6	0

Table 4.35: Results - Probability of a defect due to a parallel inheritance hierarchy bad smell of all projects

Correlation analysis was applied to all Python projects for parallel inheritance hierarchy smell. Figure 4.10 and Figure 4.11 are plots that show the distribution between two ranges. The correlations calculated for the relation in Figure 4.10 is negligible (correlation of 0.102). It indicates that there is no correlation between the number of bug-fixed commits and classes with parallel inheritance hierarchy smell.

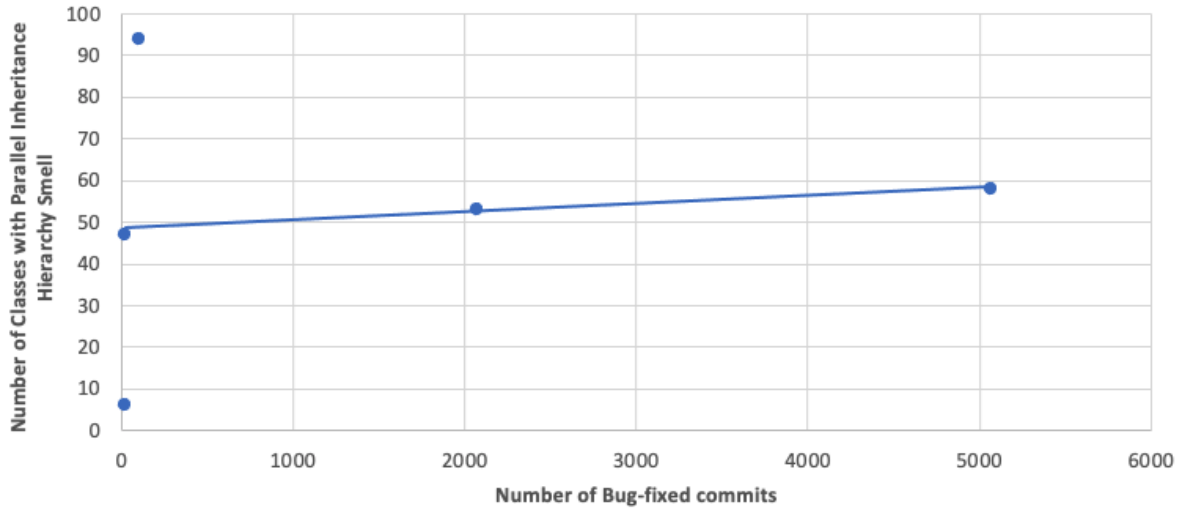


Figure 4.10: The relation between number of Bug-fixed commits and classes with parallel inheritance hierarchy smell

There exists little to no correlation between classes with parallel inheritance hierarchy smell and the probability of a defect (correlation of 0.138). Figure 4.11 reveals no significant correlation of parallel inheritance hierarchy with the defects. This also shows that the number of classes with parallel inheritance hierarchy does not affect the probability of defect occurring.

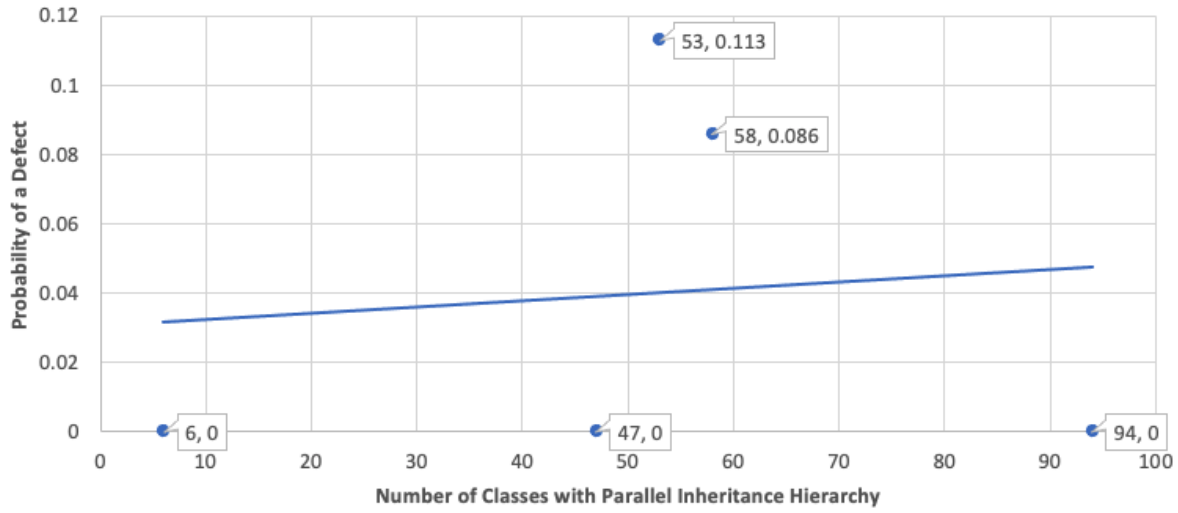


Figure 4.11: The relation between classes with parallel inheritance hierarchy and the probability of defects occurring

4.4.4 Lazy Class

Classes that perform little-to-no functionality are considered *lazy classes*. They arise, typically, from a previous refactoring or an unimplemented feature. They are classified as a bad smell because they add unnecessary complexity.

Accuracy of Lazy Class

A manual examination of NumPy revealed 389 classes that fell into the lazy category. The tool found 437 lazy classes, resulting in a recall of 0.89, a precision of 0.97, and an F-measure of 0.93.

Lazy Class Analysis

The lazy class smell was then analyzed to determine the probability of a defect when a class has lazy class smell in it. Equation 4.8 provides the formula to calculate the problem for lazy

class bad smell.

$$P(Defect = Yes|BadSmell = LazyClass) = \frac{P(Defect=Yes)\times P(BadSmell=LazyClass|Defect=Yes)}{P(BadSmell=LazyClass)} \quad (4.8)$$

The results for each of the subject libraries is illustrated in Tables 4.36 through 4.45.

NumPy Frequency Table			
isLazyClass	Defect=Yes	Defect=No	Total
Yes	44	514	558
No	1	75	76
			634

Table 4.36: Lazy Class Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	45/634
P(Bad Smell=Lazy Class | Defect=Yes)	44/45
P(Bad Smell=Lazy Class)	558/634
P(Defect=Yes | Bad Smell=Lazy Class)	(45/634 * 44/45) / (558/634) =0.078

Table 4.37: Lazy Class Likelihood table of 'NumPy'

Keras-team Frequency Table			
isLazyClass	Defect=Yes	Defect=No	Total
Yes	0	48	48
No	0	58	58
			106

Table 4.38: Lazy Class Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	0/106
P(Bad Smell=Lazy Class | Defect=Yes)	0
P(Bad Smell=Lazy Class)	48/106
P(Defect=Yes | Bad Smell=Lazy Class)	(0/106* 0) / (48/106) = 0

Table 4.39: Lazy Class Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isLazyClass	Defect=Yes	Defect=No	Total
Yes	5	261	266
No	6	141	147
			413

Table 4.40: Lazy Class Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	11/413
P(Bad Smell=Lazy Class | Defect=Yes)	5/11
P(Bad Smell=Lazy Class)	266/413
P(Defect=Yes | Bad Smell=Lazy Class)	(11/413*5/11)/(266/413)=0.18

Table 4.41: Lazy Class Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isLazyClass	Defect=Yes	Defect=No	Total
Yes	109	382	491
No	4	30	34
			525

Table 4.42: Lazy Class Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	113/525
P(Bad Smell=Lazy Class | Defect=Yes)	109/113
P(Bad Smell=Lazy Class)	491/525
P(Defect=Yes | Bad Smell=Lazy Class)	(113/525 * 109/113) / (491/525) = 0.22

Table 4.43: Lazy Class Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isLazyClass	Defect=Yes	Defect=No	Total
Yes	6	133	139
No	0	41	41
			180

Table 4.44: Lazy Class Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	6/180
P(Bad Smell=Lazy Class | Defect=Yes)	6/6
P(Bad Smell=Lazy Class)	139/180
P(Defect=Yes | Bad Smell=Lazy Class)	(6/180* 6/6) / (139/180) = 0.43

Table 4.45: Lazy Class Likelihood table of 'Tensorflow'

Lazy Class Findings

Table 4.46 displays the results of lazy class bad smell on five open-source Python projects.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Classes with Lazy Class Smell	Probability of a Defect due to a Lazy Class Smell
NumPy	351	83605	24342	2074	558	0.078
Keras-team	471	132614	5349	21	48	0
scikit-learn	617	100679	26223	99	266	0.18
zulip	916	74887	39398	5066	491	0.22
Tensorflow	1478	222595	5941	23	139	0.43

Table 4.46: Results - Probability of a defect due to a lazy class bad smell of all projects

The correlation between the lazy class smell and the defect and with the lazy class and number of bug fixed commits were investigated. The lazy class smell was found to be strongly correlated with the number of bug fixed commits with correlation of 0.768 (Figure 4.12). As

the presence of number of bug-fixed commits increases, the presence of classes with lazy class smell also increases.

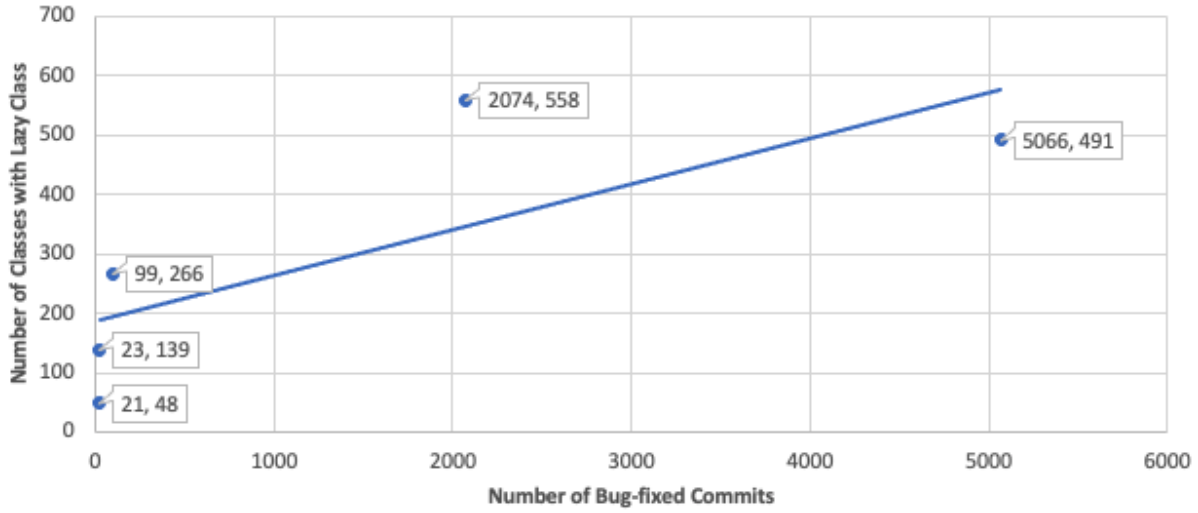


Figure 4.12: The relation between the number of bug-fixed commits and lazy classes

However, Figure 4.13 shows that there exists no such relation between lazy class smell and a defect (correlation of -0.094). For instance, Tensorflow project has the smallest number of lazy classes, but the probability of a defect due to lazy class in that project has the highest probability. On the other hand, NumPy has the highest number of lazy class smell, and it has the second lowest number of probabilities among all projects. Therefore, we cannot correlate the lazy class smell and a defect using these projects results.

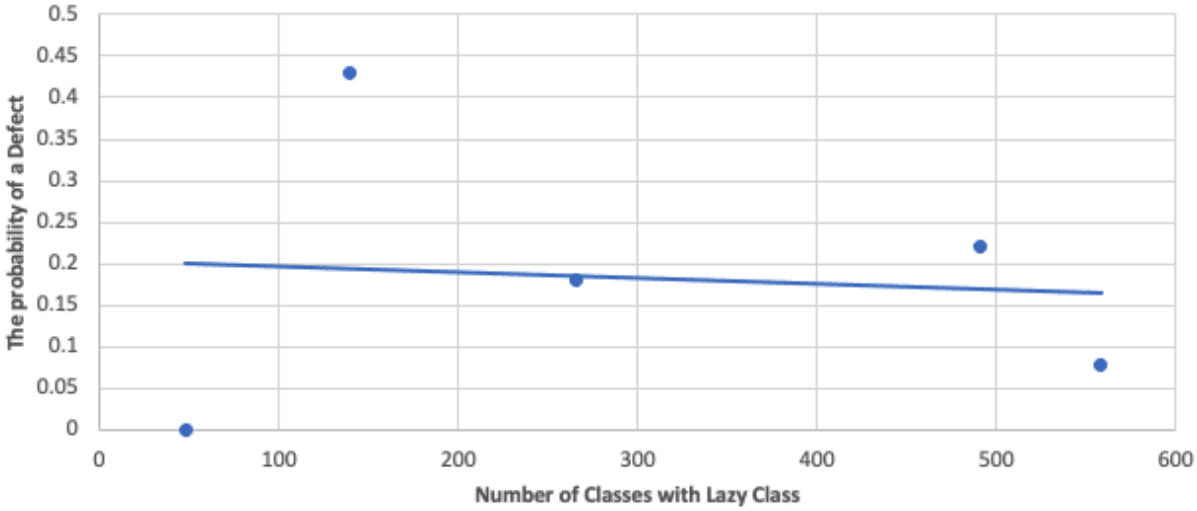


Figure 4.13: The relation between lazy class smell and the probability of defects occurring

4.4.5 Data Class

A class that is used solely as a repository of unencapsulated data elements is referred to as a *data class*. This is considered a bad smell because data elements are directly accessible without benefit of intermediary functions that ensure integrity.

Accuracy of Data Class Detection

We relied on Radon [21] and lcom [6] for LWMC and LCOM metrics to detect the bad smell. Radon is Python tool that computes cyclomatic complexity, lines of code, comment lines, blank lines, Halstead metrics and maintainability index. It is used to detect the LWMC metric. lcom is a tool to measure Python class cohesion and, thus, determine the LCOM metric.

Data Class Analysis

The same formula is applied that used for the other smells to determine the probability of a defect when a class has Data class smell in it. Naïve bayes Equation in Equation 4.9 provides

the formula:

$$P(Defect = Yes|BadSmell = DataClass) = \frac{P(Defect=Yes)\times P(BadSmell=DataClass|Defect=Yes)}{P(BadSmell=DataClass)} \quad (4.9)$$

The results of each python projects for data class smell were analyzed and provided in Table 4.47 through 4.56.

NumPy Frequency Table			
isDataClass	Defect=Yes	Defect=No	Total
Yes	7	81	88
No	43	503	546
			634

Table 4.47: Data Class Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	50/634
P(Bad Smell=Data Class | Defect=Yes)	7/50
P(Bad Smell=Data Class)	88/634
P(Defect=Yes | Bad Smell=Data Class)	(50/634 * 7/50) / (88/634) = 0.079

Table 4.48: Data Class Likelihood table of 'NumPy'

Keras-team Frequency Table			
isDataClass	Defect=Yes	Defect=No	Total
Yes	0	32	32
No	0	75	75
			107

Table 4.49: Data Class Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	0/107
P(Bad Smell=Data Class | Defect=Yes)	0
P(Bad Smell=Data Class)	32/107
P(Defect=Yes | Bad Smell=Data Class)	(0/107* 0) / (32/107) = 0

Table 4.50: Data Class Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isDataClass	Defect=Yes	Defect=No	Total
Yes	2	39	41
No	12	360	372
			413

Table 4.51: Data Class Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	14/413
P(Bad Smell=Data Class | Defect=Yes)	2/14
P(Bad Smell=Data Class)	41/413
P(Defect=Yes | Bad Smell=Data Class)	(14/413*2/14)/(41/413)=0.048

Table 4.52: Data Class Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isDataClass	Defect=Yes	Defect=No	Total
Yes	5	22	27
No	42	456	498
			525

Table 4.53: Data Class Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	47/525
P(Bad Smell=Data Class | Defect=Yes)	5/47
P(Bad Smell=Data Class)	27/525
P(Defect=Yes | Bad Smell=Data Class)	(47/525 * 5/47) / (27/525) = 0.18

Table 4.54: Data Class Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isDataClass	Defect=Yes	Defect=No	Total
Yes	2	29	31
No	5	144	149
			180

Table 4.55: Data Class Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	7/180
P(Bad Smell=Data Class | Defect=Yes)	2/7
P(Bad Smell=Data Class)	31/180
P(Defect=Yes | Bad Smell=Data Class)	(7/180 * 2/7) / (31/180) = 0.064

Table 4.56: Data Class Likelihood table of 'Tensorflow'

Data Class Findings

The result of data class bad smell on the projects is presented in Table 4.57.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Classes with Data Class Smell	Probability of a Defect due to a Data Class Smell
NumPy	351	83605	24342	2074	88	0.079
Keras-team	471	132614	5349	21	32	0
scikit-learn	617	100679	26223	99	41	0.048
zulip	916	74887	39398	5066	27	0.185
Tensorflow	1478	222595	5941	23	31	0.064

Table 4.57: Results - Probability of a defect due to a data class bad smell of all projects

The correlation coefficient is used to determine the relation between two ranges. Figure 4.14 shows that the number of bug-fixed commits exhibits no correlation with the number of classes with data class smell (Correlation of 0.025).

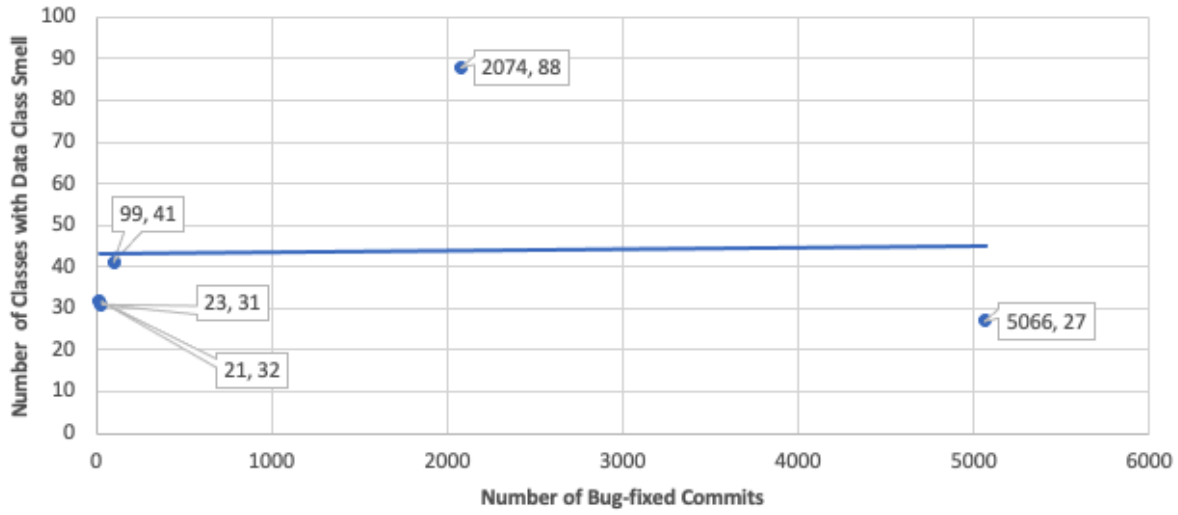


Figure 4.14: The relation between classes with data class smell and number of bug-fixed commits

Negative correlation was found between the number of classes with data class smell and the probability of a defect occurring because of data class smell with correlation of -0.082 (Figure 4.14). This does not indicate any significant correlation. It suggests that data class smells are not closely related with defects.

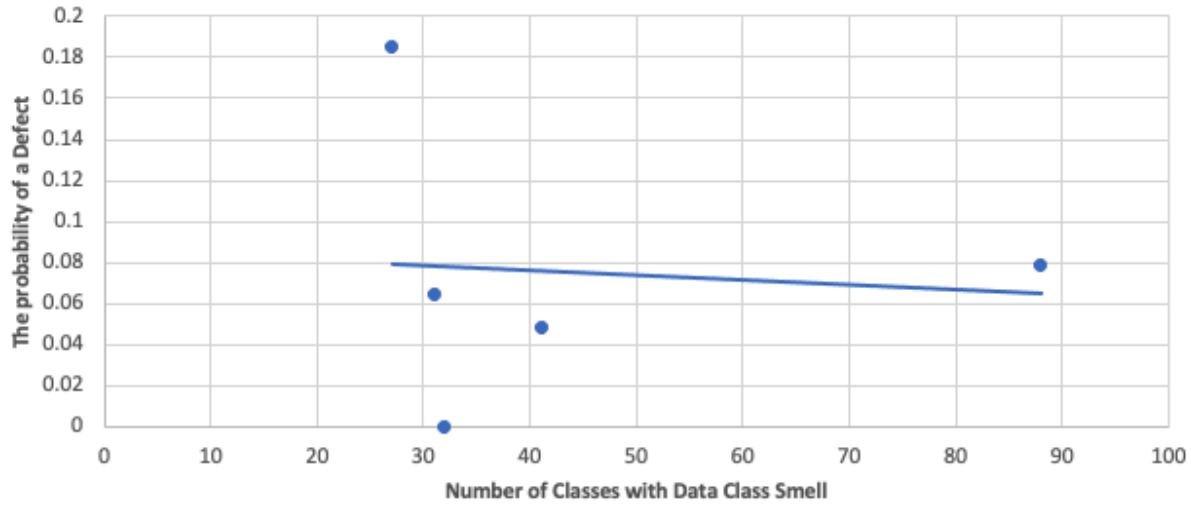


Figure 4.15: The relation between number of classes with data class smell and the probability of a defect occurring

4.4.6 Long Method

A method which includes many lines of code is a sign to long method code smell. Longer methods are hard to read [16]. When there is a change needed in a method, a developer needs to understand every line of code. As the method gets longer, it gets more complicated. That would make the method hard to maintain.

Accuracy of Long Method Detection

The reliability of long method smell detection is investigated using the manual inspection of NumPy project. The tool has a %83 recall, a %98 precision, and a %81 F-measure.

Long Method Analysis

In the analysis of the correlation between long method code smell and defects, the formula below is used.

$$P(Defect = Yes|BadSmell = LongMethod) = \frac{P(Defect=Yes)\times P(BadSmell=LongMethod|Defect=Yes)}{P(BadSmell=LongMethod)} \quad (4.10)$$

The result from the analysis of the correlation between long method code smell and defects is provided in Table 4.58 through 4.67.

NumPy Frequency Table			
isLongMethod	Defect=Yes	Defect=No	Total
Yes	38	142	180
No	180	1580	1760
			1940

Table 4.58: Long Method Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	218/1940
P(Bad Smell=Long Method | Defect=Yes)	38/218
P(Bad Smell=Long Method)	180/1940
P(Defect=Yes | Bad Smell=Long Method)	(38/218 * 218/1940) / (180/1940) = 0.21

Table 4.59: Long Method Likelihood table of 'NumPy'

Keras-team Frequency Table			
isLongMethod	Defect=Yes	Defect=No	Total
Yes	1	37	38
No	6	490	496
			534

Table 4.60: Long Method Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	7/534
P(Bad Smell=Long Method | Defect=Yes)	1/7
P(Bad Smell=Long Method)	38/534
P(Defect=Yes | Bad Smell=Long Method)	(1/7 * 7/534) / (38/534) = 0.02

Table 4.61: Long Method Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isLongMethod	Defect=Yes	Defect=No	Total
Yes	4	160	164
No	14	905	919
			1083

Table 4.62: Long Method Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	18/1083
P(Bad Smell=Long Method | Defect=Yes)	4/18
P(Bad Smell=Long Method)	164/1038
P(Defect=Yes | Bad Smell=Long Method)	(4/18 * 18/1083) / (164/1083) = 0.02

Table 4.63: Long Method Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isLongMethod	Defect=Yes	Defect=No	Total
Yes	34	127	161
No	406	1567	1973
			2134

Table 4.64: Long Method Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	440/2134
P(Bad Smell=Long Method | Defect=Yes)	34/440
P(Bad Smell=Long Method)	161/2134
P(Defect=Yes | Bad Smell=Long Method)	(440/2134 * 34/440) / (161/2134) = 0.21

Table 4.65: Long Method Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isLongMethod	Defect=Yes	Defect=No	Total
Yes	3	118	121
No	12	683	695
			816

Table 4.66: Long Method Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	15/816
P(Bad Smell=Long Method | Defect=Yes)	3/15
P(Bad Smell=Long Method)	121/816
P(Defect=Yes | Bad Smell=Long Method)	(3/15 * 15/816) / (121/816) = 0.02

Table 4.67: Long Method Likelihood table of 'Tensorflow'

Long Method Findings

Table 4.68 below presents the summary of findings obtained from Python open-source projects. From the observations of source codes, the average method lines of code vary from 13 to 18. Methods were considered long when the number of lines of code of a method is greater than 30 [24].

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Methods with Long Method Smell	Probability of a Defect due to a Long Method Smell
NumPy	351	83605	24342	2074	180	0.21
Keras-team	471	132614	5349	21	38	0.02
scikit-learn	617	100679	26223	99	164	0.02
zulip	916	74887	39398	5066	161	0.21
Tensorflow	1478	222595	5941	23	121	0.02

Table 4.68: Results - Probability of a defect due to a Long Method bad smell of all projects

Naive Bayes in Equation 4.10 was performed to correlate the long method code smell with the defects. Figure 4.16 shows the relation between number of methods with long method smell and the probability of a defect. The value of correlation coefficient in this case (r=0.60) indicates that there is moderate positive correlation between long method smells and defects occurring.

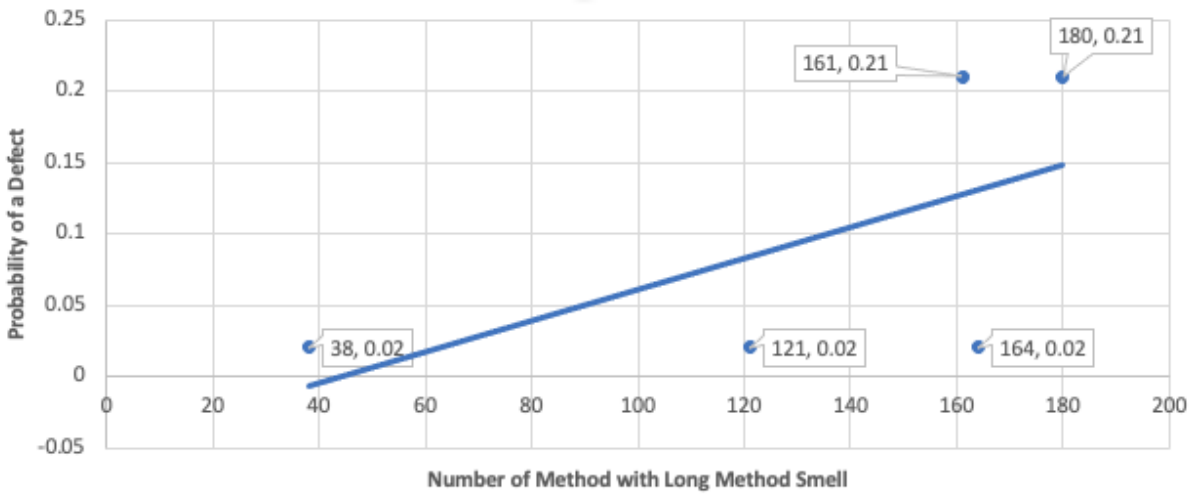


Figure 4.16: The relation between long method code smell and probability of a defect occurring

The correlation of 0.479 was found between the number of bug-fixed commits and the long method code smell. That indicates a low positive correlation. As the number of bug fixed commits increases, the number of methods with long method smell do not increase.

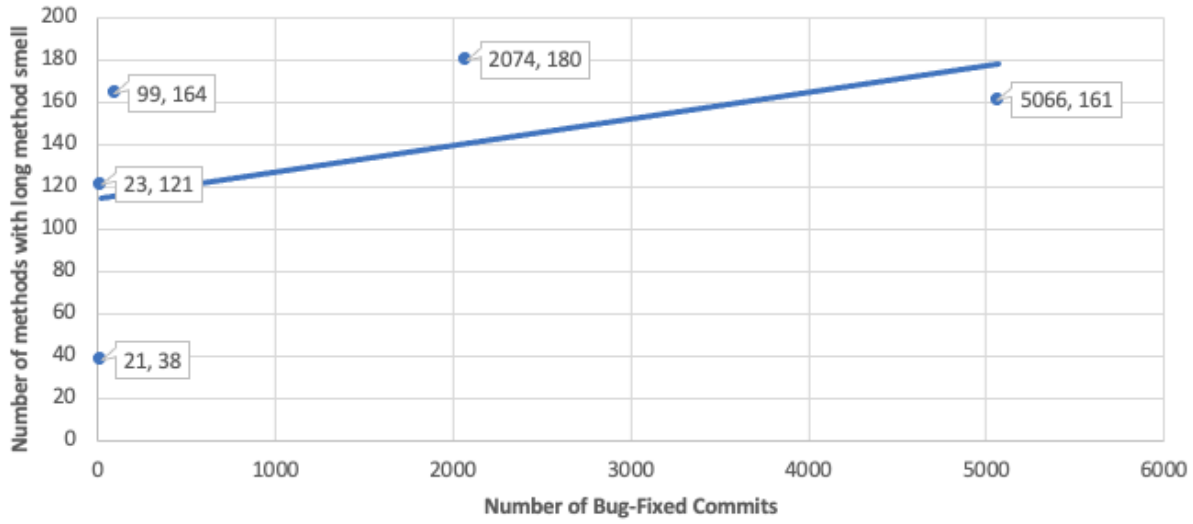


Figure 4.17: The relation between number of bug-fixed commits and probability of a defect occurring for long method code smell

Our findings for long method code smell shows that the existence of number of bug fixed commits have no impact on the long method code smells in source code and there is no direct correlation between long method and the occurring of defects in these python open-source projects.

4.4.7 Refused Bequest

A class can inherit variables and methods from another class. However, if a class extends another class, but do not use any of its attributes or behavior, this is an inappropriate use of inheritance. That is a sign to refused bequest code smell in object-oriented systems.

Accuracy of Refused Bequest Detection

Refused bequest detection metric model was applied to the classes that met conditions below:

- A class should have a superclass.
- The superclass cannot be built-in module. It should be user-defined methods.

Refused bequest detection metric model was evaluated using manual detection in NumPy project with a precision of 0.96, recall of 0.81 and F-measure of 0.88.

Refused Bequest Analysis

The correlation between refused bequest code smell and a defect is determined using the formula below.

$$P(Defect = Yes|BadSmell = RefusedBequest) = \frac{P(Defect=Yes)\times P(BadSmell=RefusedBequest|Defect=Yes)}{P(BadSmell=RefusedBequest)} \quad (4.11)$$

The results of correlation analysis for refused bequest is presented between Table 4.69 through 4.78.

NumPy Frequency Table			
isRefusedBequest	Defect=Yes	Defect=No	Total
Yes	7	65	72
No	7	28	35
			107

Table 4.69: Refused Bequest Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	14/107
P(Bad Smell=Refused Bequest | Defect=Yes)	7/14
P(Bad Smell=Refused Bequest)	72/107
P(Defect=Yes | Bad Smell=Refused Bequest)	(14/107 * 7/14) / (72/107)=0.09

Table 4.70: Refused Bequest Likelihood table of 'NumPy'

Keras-team Frequency Table			
isRefusedBequest	Defect=Yes	Defect=No	Total
Yes	1	64	65
No	0	25	25
			90

Table 4.71: Refused Bequest Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	1/90
P(Bad Smell=Refused Bequest | Defect=Yes)	1/1
P(Bad Smell=Refused Bequest)	65/90
P(Defect=Yes | Bad Smell=Refused Bequest)	(1 * 1/90) / (65/90) = 0.01

Table 4.72: Refused Bequest Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isRefusedBequest	Defect=Yes	Defect=No	Total
Yes	7	52	59
No	2	26	28
			87

Table 4.73: Refused Bequest Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	9/87
P(Bad Smell=Refused Bequest | Defect=Yes)	7/9
P(Bad Smell=Refused Bequest)	59/87
P(Defect=Yes | Bad Smell=Refused Bequest)	(9/87 * 7/9) / (59/87) = 0.11

Table 4.74: Refused Bequest Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isRefusedBequest	Defect=Yes	Defect=No	Total
Yes	10	30	40
No	5	78	83
			123

Table 4.75: Refused Bequest Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	15/123
P(Bad Smell=Refused Bequest | Defect=Yes)	10/15
P(Bad Smell=Refused Bequest)	40/123
P(Defect=Yes | Bad Smell=Refused Bequest)	(15/123 * 10/15) / (40/123) = 0.25

Table 4.76: Refused Bequest Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isRefusedBequest	Defect=Yes	Defect=No	Total
Yes	0	9	9
No	0	2	2
			11

Table 4.77: Refused Bequest Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	0/11
P(Bad Smell=Refused Bequest | Defect=Yes)	0/0
P(Bad Smell=Refused Bequest)	9/11
P(Defect=Yes | Bad Smell=Refused Bequest)	(0/11 * 0) / 9/11= 0

Table 4.78: Refused Bequest Likelihood table of 'Tensorflow'

Refused Bequest Findings

Table 4.79 shows the results of Python open-source projects for the refused bequest smell.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Classes with Refused Bequest Smell	Probability of a Defect due to a Refused Bequest Smell
NumPy	351	83605	24342	2074	72	0.09
Keras-team	471	132614	5349	21	65	0.01
scikit-learn	617	100679	26223	99	59	0.11
zulip	916	74887	39398	5066	40	0.25
Tensorflow	1478	222595	5941	23	9	0

Table 4.79: Results - Probability of a defect due to Refused Bequest bad smell of all projects

Using the projects results above, the correlation coefficient analysis is applied between the number of bug-fixed commits and number of classes with refused bequest smell. Figure 4.18 shows there is no correlation (r=0.105) relationship between those two. In other words, those two variables are independent from each other. As the number of bug-fixed commits increase, the number of classes with refused bequest smell will not be increased.

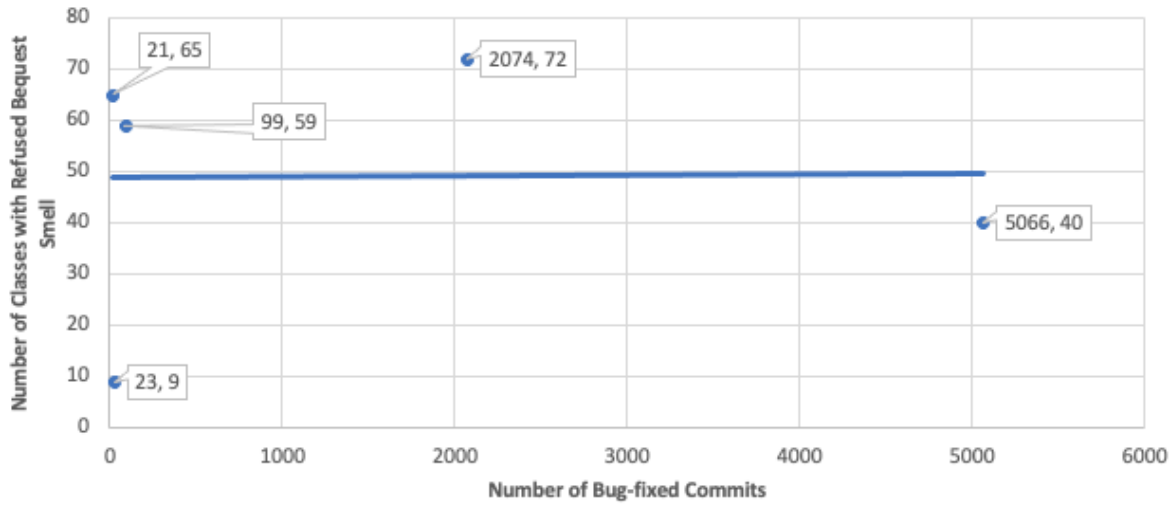


Figure 4.18: The relation between Number of Bug-fixed commits and classes with Refused Bequest Smell

There is a correlation factor of 0.011 between the number of classes with refused bequest smell and the probability of a defect occurring due to feature envy bad smell in Figure 4.19. That is a negligible correlation factor.

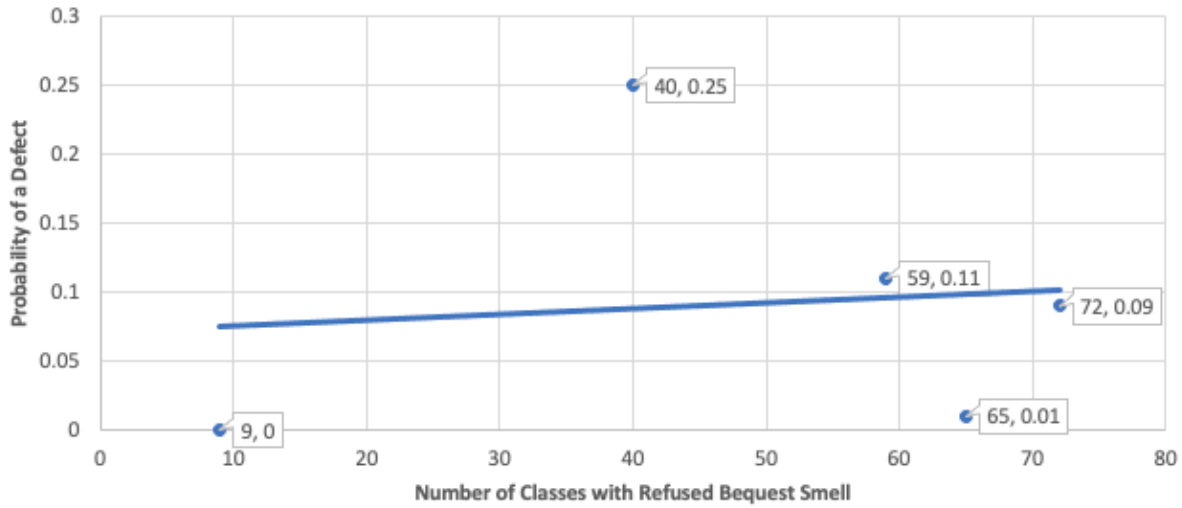


Figure 4.19: The relation between the Probability of a Defect and number of classes with Refused Bequest Smell

The results obtained from five different sized Python projects shows that the correlation coefficient is almost equal to 0 and that does not imply that there is a relationship between refused bequest smell and the defects.

4.4.8 Feature Envy

Feature envy is a bad smell when an object of a class uses functionality from another class more than its own class. The problem with this bad smell is that it increases coupling and reduces the cohesion of the object's own class.

Accuracy of Feature Envy Detection

The accuracy of feature envy detection technique is calculated on NumPy project. The project code detected that there are 7 classes that have feature envy smell while 8 classes were detected manually. The accuracy results show that feature envy smell detection has 87.5% precision, 100% recall and 93% recall.

Feature Envy Analysis

The equation below is used to calculate the relation between feature envy code smells and defects.

$$P(Defect = Yes|BadSmell = FeatureEnvy) = \frac{P(Defect=Yes)\times P(BadSmell=FeatureEnvy|Defect=Yes)}{P(BadSmell=FeatureEnvy)} \quad (4.12)$$

Equation 4.12 is applied to all the projects and the frequency and likelihood tables of each project are provided from Table 4.80 to 4.90:

NumPy Frequency Table			
isFeatureEnvySmell	Defect=Yes	Defect=No	Total
Yes	0	14	14
No	32	588	620
			634

Table 4.80: Feature Envy Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	32/634
P(Bad Smell=Feature Envy | Defect=Yes)	0/32
P(Bad Smell=Feature Envy)	14/634
P(Defect=Yes | Bad Smell=Feature Envy)	(32/634 *0/32) / (14/634) = 0

Table 4.81: Feature Envy Likelihood table of 'NumPy'

Keras-team Frequency Table			
isFeatureEnvySmell	Defect=Yes	Defect=No	Total
Yes	0	8	8
No	0	99	99
			107

Table 4.82: Feature Envy Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	0/107
P(Bad Smell=Feature Envy | Defect=Yes)	0
P(Bad Smell=Feature Envy)	8/107
P(Defect=Yes | Bad Smell=Feature Envy)	(0/100 *0) / 8/107 = 0

Table 4.83: Feature Envy Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isFeatureEnvySmell	Defect=Yes	Defect=No	Total
Yes	0	12	12
No	10	391	401
			413

Table 4.84: Feature Envy Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	10/413
P(Bad Smell=Feature Envy | Defect=Yes)	0/10
P(Bad Smell=Feature Envy)	12/413
P(Defect=Yes | Bad Smell=Feature Envy)	(10/413 * 0/10) / 12/413 = 0

Table 4.85: Feature Envy Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isFeatureEnvySmell	Defect=Yes	Defect=No	Total
Yes	0	6	6
No	63	456	519
			525

Table 4.86: Feature Envy Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	63/525
P(Bad Smell=Feature Envy | Defect=Yes)	0/63
P(Bad Smell=Feature Envy)	6/525
P(Defect=Yes | Bad Smell=Feature Envy)	(63/525 * 0/63) / 6/525 = 0

Table 4.87: Feature Envy Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isFeatureEnvySmell	Defect=Yes	Defect=No	Total
Yes	0	2	2
No	5	173	178
			180

Table 4.88: Feature Envy Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	5/180
P(Bad Smell=Feature Envy | Defect=Yes)	0/5
P(Bad Smell=Feature Envy)	2/180
P(Defect=Yes | Bad Smell=Feature Envy)	(5/180 * 0/5) / 2/180 = 0

Table 4.89: Feature Envy Likelihood table of 'Tensorflow'

Feature Envy Findings

The probability of occurrence of a defect of a class with feature envy smell is calculated and below is the summary of all results obtained from all projects.

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Classes with Feature Envy Smell	Probability of a Defect due to a Feature Envy Smell
NumPy	351	83605	24342	2074	14	0
Keras-team	471	132614	5349	21	8	0
scikit-learn	617	100679	26223	99	12	0
zulip	916	74887	39398	5066	6	0
Tensorflow	1478	222595	5941	23	2	0

Table 4.90: Results - Probability of a defect due to Feature Envy bad smell of all projects

We also studied two particle correlation analysis to determine the relation between classes with feature envy and defect occurring and the relation between bug fixed commits and again with defect occurring for feature envy smell. Table 4.90 and Figure 4.20 and Figure 4.21 show that any increase or decrease in the amount of feature envy smell has no effect on the

probability of the software code to be defective. The results show that the feature envy smell is not the cause of defects.

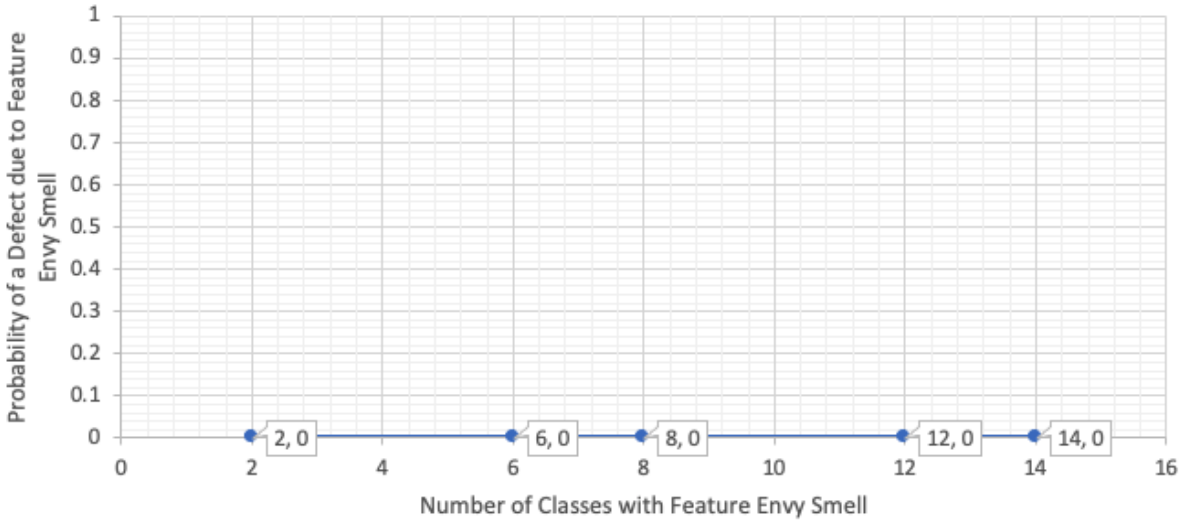


Figure 4.20: The relation between classes with feature envy smell and probability of a defect occurring

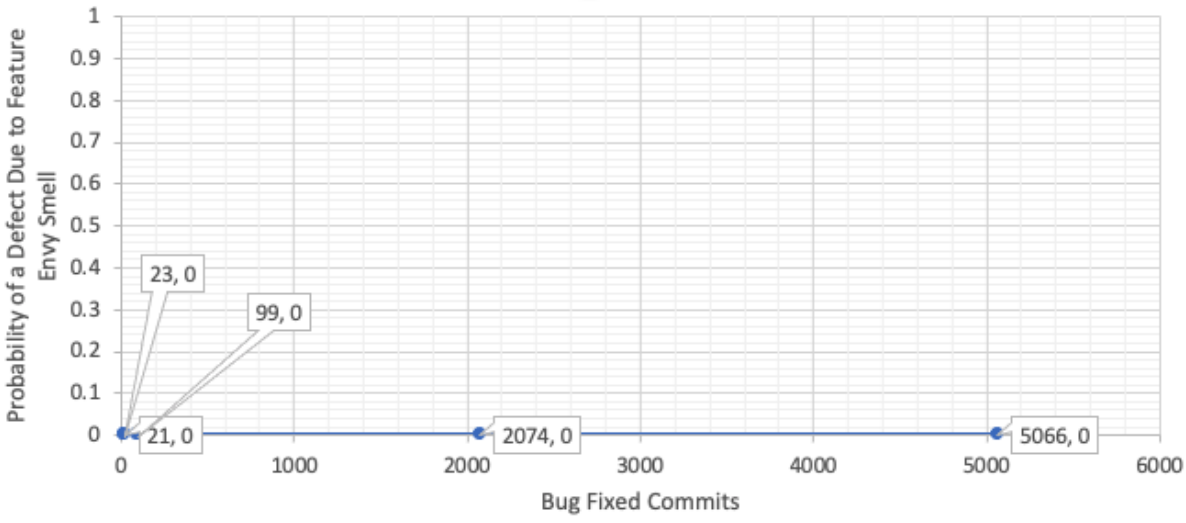


Figure 4.21: The relation between probability of a defect and number of bug fixed commits for feature envy smell

4.4.9 Shotgun Surgery

Shotgun Surgery smell happens when a method is called by many times from many other classes. That case requires lots of time and changes throughout the code to implement a new requirement on the selected method and the other methods and classes who call the selected method.

Shotgun Surgery Analysis

We planned to detect the shotgun surgery smell using the metrics model proposed by Li and Shatnawi [23]. It has two metrics that needs to be determined:

- CM is the changing methods defined as the methods that call a method of the host class.
- CC is the changing classes which is known as the number of classes that call the method of the host class.

In the Li and Shatnawi [15] model, there is another case that we need to identify other than finding the count of CM and CC metrics, which is to measure if the distinct methods that call the measured method is in top 20%. The fixed threshold value of CM for that case does not provide a clear rational on how this value is devised. Therefore, we decided to use the detection strategy proposed by Lanza et al. [22] below in Figure 4.22. Although the strategy uses generally accepted threshold values for CM and CC metrics, it is not clear what values they used as the value for Short Memory Cap and for Many.

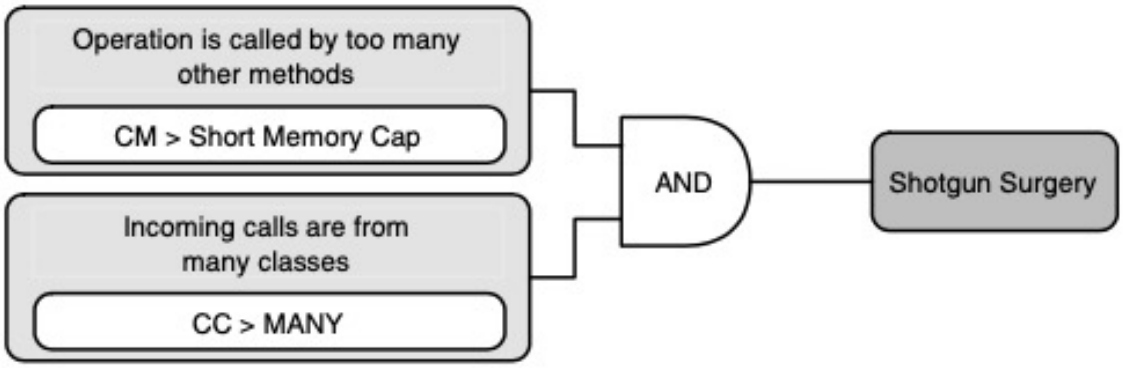


Figure 4.22: Shotgun Surgery Smell Detection Strategy

Therefore, we used the extracted threshold values proposed by Fontana et al. [15]. They proposed a data-driven method using a benchmark dataset of software systems to extract the metric thresholds for defining the code detection rules to five bad smells. Their extracted thresholds for shotgun surgery smell are same as below in Table 4.91.

Shotgun Surgery	Low	Medium	High
CM	5	7	12
CC	4	6	10

Table 4.91: The extracted threshold values for shotgun surgery detection strategy

The shotgun surgery detection strategy with the extracted threshold values was evaluated using manual detection in NumPy. The recall of 0.90, precision of 0.96 and F-measure of 0.93 were calculated to determine the accuracy of shotgun surgery detection strategy with the extracted threshold values. The analysis of shotgun surgery bad smell on five open-source projects with low, medium, and high values are provided in Appendices: A, B and C. The correlation coefficient analysis is also applied to the results of Appendices: A, B and C. The result of each analysis is provided in Table 4.92 below.

	Correlation Coefficient
Shotgun Surgery Analysis with Low Threshold Values	0.20
Shotgun Surgery Analysis with Medium Threshold Values	0.58
Shotgun Surgery Analysis with Large Threshold Values	0.34

Table 4.92: The correlation analysis of shotgun surgery smell with difference threshold values

It shows that there exists no strong correlation relationship between defects and shotgun surgery smell. That means the occurrence of shotgun surgery smell with the extracted threshold values has no effect on the probability of defect occurring.

Shotgun Surgery Smell Findings

We examined that the derived thresholds available in the literature to detect shotgun surgery may not be suitable to the context of shotgun surgery smell detection because the thresholds are not derived to determine the relation between the shotgun surgery smell and defects. They are derived to have more useful and adequate bad smell-based evaluations of source code quality [15]. When the metrics model proposed by Lanza et al. [22] with different threshold values was used, we checked the CM and CC values before and at the time of bug fixed commit. However, it was not checked if the methods and classes who call the selected method of the host class changed or not. Therefore, we decided to look at the dependents of the selected method. The classes that do not have any dependents work independently. However, when a class has a method that is called by many other classes, that might cause complexity and high coupling between classes. High coupling might complicate the projects because it makes methods and classes hard to change or understand when a new feature needs to be added. Therefore, the projects should be implemented with the weakest coupling between classes. Table 4.92 shows the dependent analysis of the summary of results from NumPy, zulip, scikit-learn, Keras-team and Tensorflow. The result of each project is available on Appendix D. Level on the table states

the number of connections between classes. For instance, in table 4.93, Level 2 for a selected method means it is called by 2 other class methods.

	Keras-team	Tensorflow	scikit-learn	NumPy	Zulip
Level 1	N/A	0%	20.84%	25.02%	12.5%
Level 2	0%	N/A	3.23%	8.82%	98.15%
Level 3	0%	N/A	0%	0%	N/A
Level 4	N/A	N/A	0%	0%	0%
Level 5	N/A	N/A	0%	0%	N/A
Level 6	N/A	N/A	N/A	0%	N/A
Level 7	N/A	N/A	N/A	0%	N/A
Level 9	N/A	N/A	N/A	0%	N/A
Level 11	N/A	N/A	N/A	0%	N/A
Level 20	N/A	N/A	N/A	0%	N/A
Level 71	N/A	N/A	0%	N/A	N/A

Table 4.93: The result of shotgun surgery method dependent analysis

Figure 4.23 shows that methods with three, four or more dependents have no change on their dependents. The results show that when a method with one dependent requires a change, its dependent is also changed. Methods with two dependents have similar behavior with one dependent. However, when the dependents increase, the results show no change on the dependents but the method itself. That shows the coupling between classes should be at Level 2 at most according to our result.

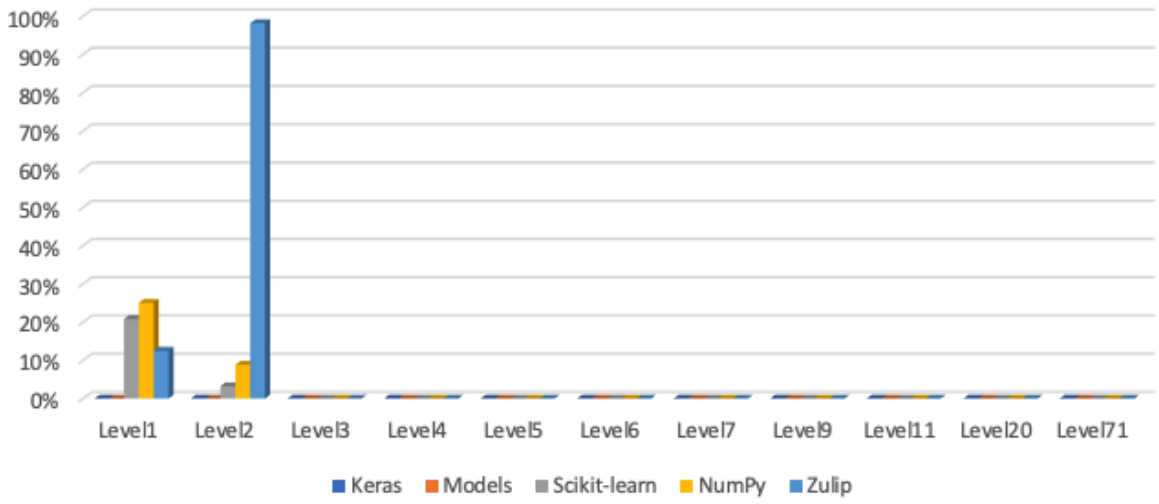


Figure 4.23: Shotgun Surgery Dependent Analysis

Chapter 5

Conclusion

5.1 Summary

Bad smells are particular patterns that might indicate a deeper problem. Bad smells are not necessarily incorrect, but are indicative of code that is complex, non-intuitive, or difficult to understand. Having them in code does not mean the software will not work; their presence degrades code quality, making the software susceptible to failure in the future. Although bad smells are not normally a bug, there are studies and common wisdom suggests that they could lead to bug when the software is later modified. It takes less time to prevent a bug occurring rather than fixing the bug from the code in the future.

This research provided an empirical study that investigates the impact of code smells on later problems. We aimed to address three issues: if bad smells can be shown to metastasize over time, if recorded defects traced back to bad smells, and if there are bad smells that are more virulent than others. We explored the effects of bad smells on five popular Python open-source software projects.

The overarching research goal of this study was to determine the extent to which bad smells lead to future defects. The investigation began with downloading Python open-source project code, git log reports (change logs), issues of the projects and modified files across the life of the projects. The change logs and issues were analyzed to determine the bug fixed commits. After that, the files modified in those bug fixed commits and files at previous version

of bug fixed commits were identified. The modified files were analyzed for the bad smells. We then checked if bad smells were removed from the modified files at previous version of bug fixed commits. This methodology was applied to five open-source Python projects and the probability of a defect occurring due to a bad smell was measured using the correlation analysis. The validation was performed in two phases. Phase one was investigating the relation on an open-source Python project: NumPy. Phase two was conducted on four other open-source Python projects namely Tensorflow, scikit-learn, Keras-team and Zulip. The validation process included validating smell detection strategies. Although bad smells on each project were detected using detection strategies, detection strategies were validated using manual inspection. This study focused on 10 bad smells, Large Class, Long Parameter List, Parallel Inheritance Hierarchy, Lazy Class, Data Class, Refused Bequest, Long Method, Feature Envy and Message Chains. The detection of bad smells was performed using object-oriented metrics models adapted from the literature. There was no evidence of message chains occurring in any of the open-source Python projects. However, we were able to detect the other nine bad smells in all Python projects.

After obtaining results from open-source Python projects, we investigated the following research questions.

- *RQ1. Do bad smells contribute to later problems?*

  Yes, some of the bad smells analyzed in this study do contribute to later problems. The results as provided in Appendix E.1 indicate that there is a positive effect of Long Parameter List smell on later problems. Long Method, Large Class and Shotgun Surgery, when detected using medium threshold values have moderate effect on defects occurring. However, Parallel Inheritance Hierarchy, Lazy Class, Data Class, Refused Bequest, Feature Envy and Shotgun Surgery, when detected with low and high threshold values have no significant effect on the presence of defects.

- *RQ2. What percentage of defects in open-source software can be traced to bad smells?*

To answer this research question, the total number of defects and how much of these defects belong to bad smells were calculated (Appendix E.3). The result of each project is analyzed individually. For NumPy, Lazy Class has the highest defect density. It is followed by Refused Bequest, Long Method, Data Class, Parallel Inheritance Hierarchy, Long Parameter List, Large Class, and Shotgun Surgery. There was no evidence found for Feature Envy effect on defects of this project. For Keras, Refused Bequest smell has the highest defect density. Long Method is the next smell that has an effect on defect density. No other smell appeared to correlate with defects. For scikit-learn project, Refused Bequest smell has the highest defect density. The smells that have effect on defects are, respectively, Lazy Class, Data Class, Long Parameter List, Long Method and Large Class. Parallel Inheritance Hierarchy, Feature Envy and Shotgun Surgery smells have no effect on density for scikit-learn project. For Zulip, Lazy Class was found to be the highest density on defects. It is then followed by Refused Bequest, Long Method, Data Class, Parallel Inheritance Hierarchy, Large Class and Long Parameter List. Feature Envy and Shotgun Surgery have zero defect density in Zulip. Lastly, Tensorflow project was analyzed. The result of this project shows that Lazy Class has the highest defect density. Data Class, Large Class and Long Method have also effect on defects, respectively. Refused Bequest, Feature Envy, Long Parameter List and Parallel Inheritance Hierarchy were found to have no impact on defects for this project.

- *RQ3. In what order should bad smells be fixed in open-source software?*

  The result of this study shows that bad smells have some effect on later problems. The correlation analysis was performed to determine how strong the relationship between bad smells and defects. The findings in Appendix E.1 illustrate that Long Parameter List smells have the highest correlation coefficient number. That indicates that the Long Parameter List smell has a high positive effect on the presence of defects. We propose that the correlation coefficient should determine the order of bad smells fix in open-source

software. Our results suggest that Long Parameter List, Long Method, Shotgun Surgery when detected with medium threshold values, Large Class, Shotgun Surgery if detected with high threshold values, Shotgun Surgery if detected with low threshold values, Refused Bequest, Parallel Inheritance Hierarchy, Data class, Lazy Class and Feature Envy should be fixed, respectively.

5.2 Observations

This study offers two observations:

- *Highlighting the existence of classes that are not doing enough.*

  Lazy class was the most common, appearing in 88% in NumPy, 45% in Keras-team, 64% in scikit-learn, 94% in Zulip and 77% in Tensorflow (in Appendix E.2). It appears Lazy Class smell occurs frequently in these projects, especially in Zulip. There was no evidence found in this study that Lazy Class lead to defects, but it is a smell that occurs when a class does not do enough. Developers should either need to have enough responsibility for a class or remove the classes that do little in the system.

- *Emphasizing the misuse of inheritance.*

  We observed that there was misuse of inheritance on the projects that were analyzed for this research. The Child Class ignores the functionalities from the Base Class(parent of Child Class) but inherits the functionalities directly from the Super Class(parent of Base Class). The Child Class should mention the Base Class in the definition of the Child Class to access the attributes and methods of the Base Class. It should not access any properties from its Parent Class without inheriting it unless the properties of Parent Class are implemented in the Base Class. This usage breaks encapsulation because the Parent Class will be exposed to any new class which inherits the current Child Class.

5.3 Contributions

This research aimed to shed light on the effect of bad smells on the presence of defects. The main contributions of this research are:

- *The effects of long parameter list smell on defects.*

  The value of correlation coefficient for the Long Parameter List has the highest correlation coefficient factor (r=0.85). The result of this study shows that the source code that contains Long Parameter List smell has the most pronounced effect on bug. As the number of methods with Long Parameter List smell increases, the probability of a defect occurring due to the Long Parameter List smell also increases. Based on our findings, the Long Parameter List bad smell is the only smell that is likely to be associated with defects more than other code smells examined in this study. We suggest that it be refactored if it exists in the code segment.

- *The detection strategy for Shotgun Surgery smell might not be working.*

  Shotgun Surgery detection strategy needs two metrics: CC and CM. The filtering mechanism for CC and CM metrics should be connected by an *and* operator. When the entities in CM and CC values of a class method are greater than threshold values, it is referred to as Shotgun Surgery smell. Thresholds for the filtering mechanism were defined mostly from past experiences, or empirical studies. Shotgun Surgery smell is detected using the combination of metrics and threshold values from the literature. The fixed threshold values in Shotgun Surgery Smell detection strategy for Python was not providing a clear rationale on how some of the values were devised. That brings us the question whether the detection strategy for shotgun surgery smell is accurate. We first identified the Changing Methods(CM) and Changing Classes(CC) of the selected class method before and at the bug fixed commit and found the class methods that have the Shotgun Surgery smell instances before and at the time of a bug fix by looking at the selected methods CMs

and CCs values. Although the detection strategy checks the dependents of the selected class method and decides whether there was Shotgun Surgery smell instance, the result of this study shows that the strategy does not work accurately, because it does not check if the dependents do change when the selected method changes. This does not reflect the concept of Shotgun Surgery smell because any change in the selected method should cause a cascade of changes in its CMs and CCs. Therefore, we think that the detection strategy proposed for Shotgun Surgery smell should not be used when looking for the relation between Shotgun Surgery smell and defect occurring.

5.4 Future Work

The topics for this research include:

- *Increasing the number of Python projects to perform the same analysis.*

  We worked on five different sized projects for this study. From Keras-team, scikit-learn and Tensorflow, we obtained a relatively limited number of bug-fixed commits which, in turn, constrained where we looked for smells. Larger size of projects having bigger sizes might have a higher number of bug-fixed commits. As the number of bug-fixed commits increase, the number of identified smells may increase. That may give more data to work on to find the correlation between code smells and defects occurring. Therefore, further validation on a larger size of projects might be beneficial in future work.

- *To what extent Python bad smells are correlated with each other?*

  The smells can be correlated with each other. There are tools available to identify some smells on the projects. However, there are not enough metrics model for all Python bad smells in the literature. Investigating correlations between code smells might be useful in detecting the presence of undetectable code smells. It is worth exploring the correlation

of Python bad smells with each other and analyze whether the correlation could be used to improve bad smell detection.

- *Defining the detection rules for all Python smells and investigating the effects of the smells on later problems.*

  Seven Python bad smells were explored in this study. However, there are 22 structures in total considered as bad smells. We are unaware of any object-oriented metrics model for the smells that were not investigated in this paper. Additional future work might be useful if this study would be conducted on defining the code detection rules for the other smells and identifying the correlation between defects and that smells.

- *Is bug classification accurate on issue tracking system?*

  Issue tracking system on GitHub helps the developer to track the project. It allows its user to create label to group issues into categories. There are default labels for every repository and labels that a user with access can create. 'bug' is one of default the categories that is available for the user and any issue with 'bug' label refers to an unexpected problem or unintended behavior. For this study, the issues labeled as 'bug' were assumed to be accurately categorized and then they were extracted from each project's issue tracking system and used to identify bug fixed commits. Although issue tracking system permits its users to categorize the issues with different labels, there is not a detection mechanism which checks if an issue is categorized with an accurate label. Therefore, tagging issues on the issue tracking system needs further investigation.

References

[1] NumPy. https://numpy.org/, 2020 (accessed on 28 November, 2020).

[2] TensorFlow. https://www.tensorflow.org, 2020 (accessed on 28 November, 2020).

[3] zulip. https://zulip.com, 2020 (accessed on Oct, 2020).

[4] Keras-team. https://keras.io, 2020 (accessed on Oct 28, 2020).

[5] PMD - Rule Set: Code Size Rules @ONLINE. https://pmd.sourceforge.io/pmd-4.2.5/rules/codesize.html, 2020 (accessed on Oct 28, 2020).

[6] Lack of Cohesion of Methods. https://github.com/potfur/lcom, 2021 (accessed on July, 2021).

[7] checkstyle – Metrics. http://checkstyle.sourceforge.net/config_metrics.html#JavaNCSS, 2021 (accessed on Oct 28, 2020).

[8] Clone Digger. https://sourceforge.net/projects/clonedigger/, 2021 (accessed on Oct 28, 2020).

[9] Z. Chen, L. Chen, W. Ma, and B. Xu. Detecting code smells in python programs. *Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016*, pages 18–23, 11 2016.

[10] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu. Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology*, 94:14–29, 2 2018.

[11] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20 (6). *Software Engineering, IEEE ...*, 20(6):476 – 493, 1994.

[12] P. Danphitsanuphan and T. Suwantada. Code smell detecting tool and code smell-structure bug relationship. *2012 Spring World Congress on Engineering and Technology, SCET 2012 - Proceedings*, (May 2012), 2012.

[13] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, pages 1–12, New York, New York, USA, 2016. ACM Press.

[14] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 23–32. IEEE Comput. Soc, 2003.

[15] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita. Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM*, 2015-Augus:44–53, 8 2015.

[16] M. Fowler and K. Beck. *Refactoring : improving the design of existing code*. Addison-Wesley, 1999.

[17] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–39, 9 2014.

[18] D. E. Hinkle, W. Wiersma, and S. G. Jurs. *Applied Statistcis for the behavioral sciences*. Houghton Mifflin, Boston, 5 edition, 2003.

[19] J. Joyce. Bayes' Theorem. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 201 edition, 2019.

[20] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[21] M. Lacchia. radon. `https://radon.readthedocs.io/en/latest/`, 2020.

[22] M. Lanza, R. Marinescu, and S. Ducasse. *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, Germany, 2007.

[23] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 7 2007.

[24] M. Lippert and S. Roock. Refactoring in large software projects. page 280, 2006.

[25] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE.

[26] R. C. M. Martin. *Clean Code*. Prentice Hall, Upper Saddle River, NJ, 2008.

[27] S. McConnell. *Code Complete*. Microsoft Press, Redmont, Washington, 2 edition, 2004.

[28] T. B. D. of Electrical Engineering, B. S. . b. Computer Science, South Dakota State University, L. Y. D. of Electrical Engineering, and B. S. . y. Computer Science, South Dakota

State University. Reducing the Large Class Code Smell by Applying Design Patterns. *IEEE International Conference on Electro Information Technology(EIT)*, pages 590–595, 2019.

[29] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7, 12 2017.

[30] R. Peters and A. Zaidman. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 411–416. IEEE, 3 2012.

[31] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 72–81. IEEE, 5 2010.

[32] L. H. Rosenberg and L. E. Hyatt. Software Quality Metrics for Object Oriented System Environments – A report of SATC's research on OO metrics. *Crosstalk Journal*, pages 1–7, 1997.

[33] M. Sangeetha, P. Sengottuvelan, M. Sangeetha, A. Professor, M. Sangeetha, P. Sengottuvelan, and M. Sangeetha. Fascinating Observation Monitor-based Clamant Code Smell Detection Using Python, 2017.

[34] T. Sharma and D. Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 4 2018.

[35] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1, 2005.

[36] G. Travassos, F. Shull, M. Fredericks, V. R. Basili, G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs. In *Proceedings of the 14th*

ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '99, volume 34, pages 47–56, New York, New York, USA, 1999. ACM Press.

[37] N. Vavrová and V. Zaytsev. Does Python Smell Like Java ? The Art , Science , and Engineering of Programming.

[38] R. Wu, H. Zhang, S. Kim, S. C. Cheung, S. Kim, R. Wu, and H. Zhang. ReLink: Recovering links between bugs and changes. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–25, 2011.

[39] M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Prioritising Refactoring Using Code Bad Smells. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 458–464. IEEE, 3 2011.

[40] M. Zhang, T. Hall, and N. Baddoo. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, 4 2011.

Appendices

Appendix A

Shotgun Surgery Smell Detection with Low Threshold Values

A.1 Shotgun Surgery Smell Analysis with Low Threshold Values

The frequency and likelihood tables of each project with low threshold values are provided from Table A.1 to A.11:

NumPy Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	5	4	9
No	787	1282	2069
			2078

Table A.1: Shotgun Surgery Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	792/2078
P(Bad Smell=Shotgun Surgery | Defect=Yes)	5/792
P(Bad Smell=Shotgun Surgery)	9/2078
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(792/2078 * 5/792) / (9/2078)=0.55

Table A.2: Shotgun Surgery Likelihood table of 'NumPy'

Keras-team Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	3	3
No	4	940	944
			947

Table A.3: Shotgun Surgery Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	4/947
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/4
P(Bad Smell=Shotgun Surgery)	3/947
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(4/947 * 0/4) / (3/947) = 0

Table A.4: Shotgun Surgery Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	2	2
No	75	916	991
			993

Table A.5: Shotgun Surgery Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	75/993
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/75
P(Bad Smell=Shotgun Surgery)	2/993
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(75/993 * 0/75)/(2/993)=0

Table A.6: Shotgun Surgery Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	0	0
No	267	1071	1338
			1338

Table A.7: Shotgun Surgery Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	267/1338
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/267
P(Bad Smell=Shotgun Surgery)	0/1338
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(267/1338 * 0) / 0 = 0

Table A.8: Shotgun Surgery Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	1	0	1
No	57	591	648
			649

Table A.9: Shotgun Surgery Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	58/649
P(Bad Smell=Shotgun Surgery | Defect=Yes)	1/58
P(Bad Smell=Shotgun Surgery)	1/649
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(58/649*1/58)/(1/649)=1

Table A.10: Shotgun Surgery Likelihood table of 'Tensorflow'

A.2 Shotgun Surgery Smell Findings with Low Threshold Values

The result of shotgun surgery bad smell analysis with low threshold values are provided in table A.11 below:

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Class Methods with Shotgun Surgery Smell	Probability of a Defect due to a Shotgun Surgery Smell
NumPy	351	83605	24342	2074	9	0.55
Keras-team	471	132614	5349	21	3	0
scikit-learn	617	100679	26223	99	2	0
zulip	916	74887	39398	5066	0	0
Tensorflow	1478	222595	5941	23	1	1

Table A.11: Projects Results - Probability of a defect due to Shotgun Surgery bad smell with low threshold values

Appendix B

Shotgun Surgery Smell Detection with Medium Threshold Values

B.1 Shotgun Surgery Smell Analysis with Medium Threshold Values

The frequency and likelihood tables of each project with medium threshold values are provided from Table B.1 to B.11:

NumPy Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	9	0	9
No	785	1284	2069
			2078

Table B.1: Shotgun Surgery Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	794/2078
P(Bad Smell=Shotgun Surgery | Defect=Yes)	9/794
P(Bad Smell=Shotgun Surgery)	9/2078
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(794/2078 * 9/794) / (9/2078) = 1

Table B.2: Shotgun Surgery Likelihood table of 'NumPy'

Keras-team Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	3	3
No	4	940	944
			947

Table B.3: Shotgun Surgery Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	4/947
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/4
P(Bad Smell=Shotgun Surgery)	3/947
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(4/947 * 0/4) / (3/947) = 0

Table B.4: Shotgun Surgery Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	0	0
No	76	917	993
			993

Table B.5: Shotgun Surgery Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	76/993
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/76
P(Bad Smell=Shotgun Surgery)	0/993
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(76/993 * 0) / 0 = 0

Table B.6: Shotgun Surgery Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	0	0
No	267	1071	1338
			1338

Table B.7: Shotgun Surgery Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	267/1338
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/267
P(Bad Smell=Shotgun Surgery)	0/1338
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(267/1338 * 0) / 0 = 0

Table B.8: Shotgun Surgery Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	1	0	1
No	57	591	648
			649

Table B.9: Shotgun Surgery Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	58/649
P(Bad Smell=Shotgun Surgery | Defect=Yes)	1/58
P(Bad Smell=Shotgun Surgery)	1/649
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(58/649*1/58)/(1/649)=1

Table B.10: Shotgun Surgery Likelihood table of 'Tensorflow'

B.2 Shotgun Surgery Smell Findings with Medium Threshold Values

The result of shotgun surgery bad smell analysis with medium threshold values are provided in table B.11 below:

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Class Methods with Shotgun Surgery Smell	Probability of a Defect due to a Shotgun Surgery Smell
NumPy	351	83605	24342	2074	9	1
Keras-team	471	132614	5349	21	3	0
scikit-learn	617	100679	26223	99	0	0
zulip	916	74887	39398	5066	0	0
Tensorflow	1478	222595	5941	23	1	1

Table B.11: Projects Results - Probability of a defect due to Shotgun Surgery bad smell with medium threshold values

Appendix C

Shotgun Surgery Smell Detection with High Threshold Values

C.1 Shotgun Surgery Smell Analysis with High Threshold Values

The frequency and likelihood tables of each project with high threshold values are provided from Table C.1 to C.11:

NumPy Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	3	1	4
No	791	1283	2074
			2078

Table C.1: Shotgun Surgery Frequency table of 'NumPy'

NumPy Likelihood Table	
P(Defect=Yes)	794/2078
P(Bad Smell=Shotgun Surgery | Defect=Yes)	3/794
P(Bad Smell=Shotgun Surgery)	4/2078
P(Defect=Yes | Bad Smell=Shotgun Surgery)	((794/2078)*(3/794))/ (4/2078)=0.75

Table C.2: Shotgun Surgery Likelihood table of 'NumPy'

Keras-team Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	3	3
No	4	940	944
			947

Table C.3: Shotgun Surgery Frequency table of 'Keras-team'

Keras-team Likelihood Table	
P(Defect=Yes)	4/947
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/4
P(Bad Smell=Shotgun Surgery)	3/947
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(4/947 * 0/4) / (3/947) = 0

Table C.4: Shotgun Surgery Likelihood table of 'Keras-team'

scikit-learn Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	0	0
No	76	917	993
			993

Table C.5: Shotgun Surgery Frequency table of 'scikit-learn'

scikit-learn Likelihood Table	
P(Defect=Yes)	76/993
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/76
P(Bad Smell=Shotgun Surgery)	0/993
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(76/993 * 0) / 0 = 0

Table C.6: Shotgun Surgery Likelihood table of 'scikit-learn'

Zulip Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	0	0	0
No	267	1071	1338
			1338

Table C.7: Shotgun Surgery Frequency table of 'Zulip'

Zulip Likelihood Table	
P(Defect=Yes)	267/1338
P(Bad Smell=Shotgun Surgery | Defect=Yes)	0/267
P(Bad Smell=Shotgun Surgery)	0/1338
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(267/1338 * 0) / 0 = 0

Table C.8: Shotgun Surgery Likelihood table of 'Zulip'

Tensorflow Frequency Table			
isShotgunSurgery	Defect=Yes	Defect=No	Total
Yes	1	0	1
No	57	591	648
			649

Table C.9: Shotgun Surgery Frequency table of 'Tensorflow'

Tensorflow Likelihood Table	
P(Defect=Yes)	58/649
P(Bad Smell=Shotgun Surgery | Defect=Yes)	1/58
P(Bad Smell=Shotgun Surgery)	1/649
P(Defect=Yes | Bad Smell=Shotgun Surgery)	(58/649*1/58)/(1/649)=1

Table C.10: Shotgun Surgery Likelihood table of 'Tensorflow'

C.2 Shotgun Surgery Smell Findings with Large Threshold Values

The result of shotgun surgery bad smell analysis with large threshold values are provided in table C.11 below:

Python Project	Number of Files	Number of Lines	Number of Commits	Number of Bug Fixed Commits	Number of Class Methods with Shotgun Surgery Smell	Probability of a Defect due to a Shotgun Surgery Smell
NumPy	351	83605	24342	2074	4	0.75
Keras-team	471	132614	5349	21	3	0
scikit-learn	617	100679	26223	99	0	0
zulip	916	74887	39398	5066	0	0
Tensorflow	1478	222595	5941	23	1	1

Table C.11: Projects Results - Probability of a defect due to Shotgun Surgery bad smell with large threshold values

Appendix D

Shotgun Surgery Dependent Analysis Results

D.1 Keras-team

Keras:	Level1	Level2	Level3	Total Records
Level1	0	N/A	N/A	0
Level2	0	0	N/A	4
Level3	0	0	0	6

Figure D.1: Keras-team Shotgun Surgery Dependent Analysis

D.2 Tensorflow

Models:	Level1	Total Records
Level1:	0	30

Figure D.2: Tensorflow Shotgun Surgery Dependent Analysis

D.3 scikit-learn

scikit-learn:	Level1	Level2	Level3	Level4	Level5	Level71	Total Records
Level1	79	N/A	N/A	N/A	N/A	N/A	379
Level2	1	0	N/A	N/A	N/A	N/A	31
Level3	0	26	0	N/A	N/A	N/A	26
level4	0	0	0	0	N/A	N/A	5
Level5	0	0	0	0	0	N/A	11
Level71	0	0	0	1	0	0	2

Figure D.3: scikit-learn Shotgun Surgery Dependent Analysis

D.4 Zulip

zulip	Level1	Level2	Level3	Level4	Total Records
Level1	27	0	0	0	216
Level2	53	0	0	0	54
Level3	0	0	0	0	0
Level4	0	0	0	0	6

Figure D.4: Zulip Shotgun Surgery Dependent Analysis

D.5 NumPy

numpy	Level1	Level2	Level3	Level4	Level5	Level6	Level7	Level8	Level9	Level10	Level11	Level20	Total Records
Level1	271	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1083
Level2	4	9	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	102
Level3	2	0	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	20
Level4	0	0	0	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
Level5	0	0	0	0	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
Level6	0	0	0	0	0	0	N/A	N/A	N/A	N/A	N/A	N/A	69
Level7	0	0	0	0	0	0	0	N/A	N/A	N/A	N/A	N/A	1
Level8	0	0	0	0	0	0	0	0	N/A	N/A	N/A	N/A	0
Level9	2	0	0	0	0	0	0	0	0	N/A	N/A	N/A	2
Level11	0	0	0	0	0	0	0	0	0	0	0	N/A	1
Level20	0	0	8	0	0	0	0	0	0	0	0	0	8

Figure D.5: NumPy Shotgun Surgery Dependent Analysis

Appendix E

Code Smell and Defect Density Analysis

E.1 Correlation Coefficient Analysis of Code Smells

Code Smell	**Correlation Coefficient**
Large Class	-0.54
Long Parameter List	0.85
Parallel Inheritance Hierarchy	0.10
Lazy Class	-0.09
Data Class	-0.08
Refused Bequest	0.11
Long Method	0.60
Feature Envy	0
Shotgun Surgery w/ Low Threshold Values	0.20
Shotgun Surgery w/ Medium Threshold Values	0.58
Shotgun Surgery w/ High Threshold Values	0.34

Table E.1: The result of correlation coefficient for each code smell

E.2 Code Smell Density Analysis Results

Code Smell	NumPy	Keras-team	scikit-learn	Zulip	Tensorflow
Large Class	4%	19%	9%	2%	6%
Long Parameter List	13%	16%	22%	8%	12%
Parallel Inheritance Hierarchy	8%	44%	23%	11%	3%
Lazy Class	88%	45%	64%	94%	77%
Data Class	14%	30%	10%	5%	17%
Long Method	9%	7%	15%	8%	15%
Refused Bequest	67%	72%	68%	33%	82%
Feature Envy	2%	7%	3%	1%	1%
Shotgun Surgery Detected w/ Low Threshold Value	0.4%	0.3%	0.2%	0%	0.2%
Shotgun Surgery Detected w/ Medium Threshold Value	0.4%	0.3%	0%	0%	0.2%
Shotgun Surgery Detected w/ High Threshold Value	0.2%	0.3%	0%	0%	0.2%

Table E.2: Code Smell Density of Python projects

E.3 Defect Density Traced to Code Smell Analysis Results

	Numpy		Keras-team		scikit-learn		Zulip		Tensorflow	
Bad Smell	Defects Density Traced to Code Smell	Defect Density	Defects Density Traced to Code Smell	Defect Density	Defects Density Traced to Code Smell	Defect Density	Defects Density Traced to Code Smell	Defect Density	Defects Density Traced to Code Smell	Defect Density
Large Class	0.5%	2.4%	0%	0%	0.2%	0.2%	0.2%	1.3%	0.6%	0.6%
Long Parameter List	0.6%	5.4%	0%	1.9%	0.4%	1.4%	0.1%	2%	0%	2.7%
Parallel Inheritance Hierarchy	0.9%	7.4%	0%	0%	0%	2.9%	1%	8.4%	0%	2.8%
Lazy Class	6.9%	7.1%	0%	0%	1.2%	2.7%	20.8%	21.5%	3.3%	3.3%
Data Class	1.1%	7.9%	0%	0%	0.5%	3.4%	1%	9%	1.1%	3.9%
Refused Bequest	6.5%	13.1%	1.1%	1.1%	8%	10.3%	8.1%	12.2%	0%	0%
Long Method	2%	11.2%	0.2%	1.3%	0.4%	1.7%	1.6%	20.6%	0.4%	1.8%
Feature Envy	0%	5%	0%	0%	0%	2.4%	0%	12%	0%	2.8%
Shotgun Surgery w/ Low Threshold Value	0.2%	38.1%	0%	0.4%	0%	7.6%	0%	20%	0.2%	8.9%
Shotgun Surgery w/ Medium Threshold Value	0.4%	38.2%	0%	0.4%	0%	7.7%	0%	20%	0.2%	8.9%
Shotgun Surgery w/ High Threshold Value	0.1%	38.2%	0%	0.4%	0%	7.7%	0%	23.5%	0.2%	8.8%

Table E.3: Code Smell and Defect Density Findings