MEASURING DEVELOPER EXPERIENCE WITH ABSTRACT SYNTAX TREES

STEVEN DEUTEKOM
Bachelor of Science, University of Lethbridge, 2020

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

MEASURING DEVELOPER EXPERIENCE WITH ABSTRACT SYNTAX TREES

STEVEN DEUTEKOM

Date of Defence: July 26, 2023

Dr. Jackie Rice	Professor	Ph.D.
Dr. John Anvik	Associate Professor	Ph.D.
Thesis Co-Supervisors		
Dr. Wendy Osborn	Associate Professor	Ph.D.
Thesis Examination Committee Member		
Dr. John Sheriff	Assistant Professor	Ph.D.
Thesis Examination Committee Member		
Dr. John Zhang	Associate Professor	Ph.D.
Chair, Thesis Examination Committee		

Dedication

To my family. Without their unconditional support I would be lost.

Abstract

Accurately representing a developer's programming knowledge and experience is difficult. Traditional metrics rely on counting the number of times a developer has used or made changes to pieces of code. When a developer has modified a file in the past they are less likely to introduce defects with a change. However, these metrics do not contain any general information on the structure or purpose of a piece of code and are only useful when developers work on a piece of code more than once. We investigated the use of several new metrics based on abstract syntax trees (ASTs) as a possible way to more completely measure a developer's experience. By using the ASTs of code previously modified by a developer we may be able to identify their experience with a piece of code they are modifying even if they have never modified that specific code before. Through statistical analysis and machine learning predictions we show that AST-based metrics capture a more general programming experience than count-based metrics. In their current form, AST-based metrics do not offer any significant improvements over existing metrics for defect prediction. However, our work offers a starting point for future use of ASTs for representation of knowledge and experience in defect prediction and other relevant areas.

Acknowledgments

I would like to thank my supervisors Dr. Jackie Rice and Dr. John Anvik. Both of them provided me with several opportunities to learn and grow before and during my graduate studies. I am extremely grateful their encouragement, support, and understanding.

Thank you to my committee members Dr. John Sheriff and Dr. Wendy Osborn for their work with me on my thesis.

Many thanks to Dr. Howard Chang for his encouragement and teaching through his direction of the Uleth contest programming club. I learned many things and gained many new friends in the programming club. Without it my time in university would have been much less fun.

I would also like to thank all of the other professors, instructors, and staff in the math and computer science department. I learned a lot from all of them in class and enjoyed getting to know them outside of class.

Finally, thanks to all the friends I made along the way. My classmates who worked late into the evenings on projects and studying for exams, my lab mates who gave ideas and opinions on my work and shared many laughs, and everyone who just made my time in Lethbridge memorable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Accurately measuring a developer's knowledge or experience is a fundamental problem in software engineering research. A new developer might want to find someone knowledgeable about a specific software module. A manager might want to assign a developer with the most knowledge of a system to fix a defect. Managers of an open-source project may want to assess the knowledge of new contributors before assigning them tasks. During quality control processes, developer experience might help determine which files in a project will contain the most defects.

Measuring experience for these tasks has been done on different levels. Finer-grained metrics include usage and implementation expertise. These count the number of times a developer has used or changed a module, class, or method in a software project [1, 24, 32]. Coarser-grained metrics include project and library expertise. These measure the number of times a developer has made changes to a project or made changes to code that uses a specific library or framework [5, 26]. Attempts have also been made at a more holistic view of a developer's expertise with projects, programming languages, and APIs to build a representation of a developer's skills [9].

These metrics help us better understand a developer's knowledge in general areas or specific parts of a code base. However, we may also want to measure a developer's knowledge of particular code structures. For example, given a method body, how well will a developer understand what the code does? The code may use libraries, classes, or methods they have used or modified in the past. It may be in a project they have worked on before, but that does not guarantee they will immediately understand the purpose of the code or how it accomplishes its task. The code may have a different organization than they are used to or use different programming constructs, styles, or paradigms. Simple experience counts cannot represent this fundamental knowledge; other source code metrics are necessary.

In other areas of software engineering research, there are many different source code metrics, such as counts of the number of lines of code, programming language keywords, code complexity, and how often code changes. Researchers have used these metrics to identify different groups of programmers based on their code and better understand how those groups program [28, 30]. Others have used similar metrics to identify the author of a program when no authorship information is available [3] and to predict defects in software [6, 12, 29]. Again, these metrics can succeed in specific applications but cannot accurately represent the code's organization or meaning. For example, keyword, operator, and method call counts give us an idea of some of the basic functionality of a piece of code but are basic enough to be included in most code, no matter its purpose.

In this work, we utilize the source code's abstract syntax tree (**AST**) to overcome some of these limitations and capture a more complete representation of a developer's code knowledge or expertise. This tree contains the entire structure and text of a piece of source code. The structured nature of the AST makes it possible to compare different ASTs to identify the substructures they share. The similarity between two substructures may be as small as a single keyword or as large as the entire tree. For example, say two methods have a for loop that calls another method. Even if these loops are in different places in each AST, we can identify them in both methods as a shared structure. Instead of counting individual keywords, we now count arbitrary combinations of keywords, operators, and other programming structures.

We hypothesize that **there are combinations of these structures that are more representative of the purpose of a piece of code and different programming constructs, styles, and paradigms**. Other works have provided evidence for this hypothesis. Alon *et* *al.* successfully used ASTs to predict the names of methods, which are representative of a method's purpose [2]. Kovalenko *et al.* used ASTs to compare and track implicit representations of developers' style over time [17]. Finally, Sahar *et al.* detected defects in software changes by comparing the AST of a changed method to other ASTs of methods known to contain defects or not [31].

We propose to use ASTs to track a developer's experience and use that experience to gauge their knowledge of a piece of code. To evaluate the success of this representation of expertise, we will work in the context of software defect prediction. We will use AST-based experience metrics to predict if a change to a method will introduce a defect. We expect that a developer with more experience with a piece of code will be more likely to change that code without introducing errors. If our metrics accurately represent some aspects of developer knowledge and experience, they should have a statistically significant association with defects. Because AST-based metrics capture more information about the code a developer has experienced, they should also be helpful when count-based experience is not. An example of this might be when a developer changes a file they have never worked on before. Working on code with similar ASTs may indicate knowledge of that code even if they have no direct experience with it.

Bird *et al.* showed that developers with the majority of contributions to a project[1] are less likely to introduce defects when changing that project [5]. In addition to investigating AST-based metrics, we will expand the contribution counts from Bird *et al.* to include the number of times a developer has made changes to files and methods. These count-based metrics will help us evaluate our AST-based metrics and serve as controls to determine whether our AST-based metrics do indeed capture more relevant experience information than simple experience counts.

[1]In this case, a binary in the Windows operating system.

1.1 Research Questions

We offer six research questions to be more precise about our goals. The first three focus on understanding how our new AST measures relate to existing measures and defects. The latter three evaluate the use of these new metrics for predicting defects. The majority of existing defect prediction techniques work at the file level. While we primarily investigate using AST-based experience metrics, we are also interested in defect prediction accuracy for method modifications with all available metrics. Our research questions are the following:

1. **How do AST-based experience metrics and count-based experience metrics relate to one another?** AST-based experience metrics and count-based experience metrics are generated using different information from a developer's method modification history. AST-based metrics capture a more complete picture of the source code of each method changed, but the experience numbers rely on measuring similarity between two methods. The worst-case for structural similarity would occur if two methods are considered as similar only when they are identical and otherwise are not considered similar. This situation would result in the same experience as counting the times a developer had previously modified a method. We need to examine the relationships between both types of metrics to determine whether our AST-based metrics are genuinely capturing new information.

2. **Do AST-based metrics have a statistically significant relationship with method modification defects?** We are evaluating the ability of AST-based metrics to capture developer knowledge and experience by evaluating them in the context of defect detection. We can better understand the relationship between AST-based experience metrics and method defects by using statistical tests and building statistical models. If there is a statistically significant relationship between these metrics and defects, it provides us with evidence that AST-based metrics can capture some aspects of developer knowledge and experience. We also investigate how these relationships change

when we account for count-based experience metrics and other metrics commonly used to detect defects.

3. **When count-based experience is not applicable, do AST-based experience metrics have a statistically significant relationship to defects?** We want to know if AST-based experience captures a more general measure of a developer's knowledge and experience. We can test this by examining how these metrics relate to defects when a developer modifies a method for the first time. In this scenario, a high similarity between previously modified code and the method a developer is modifying will come from similar structures in method ASTs and not because the developer has modified the same method more than once. If AST-based experience metrics maintain a statistically significant relationship to method modification defects in this context, there is evidence that they capture a sufficiently generic representation of developer knowledge.

4. **How successful is defect prediction using AST-based metrics and controls?** The previous questions focus on understanding how AST-based experience relates to method modification defects. We want to know if adding our AST-based metrics can improve defect predictions performed only using existing metrics. Also, the general approach for defect prediction is to predict the number of possible defects in a file before a large set of updates to a project are released. Our approach is different and targets each method a developer has modified when they finish a smaller set of changes to a project. We want to know how our AST-based metrics perform and how well existing metrics perform in this context.

5. **Which metrics from our full set of collected metrics are most important for predicting method modification defects?** Since detecting defects in single-method changes is less common, we want to investigate which metrics are the most important in this context. Identifying the most important metrics can aid future researchers,

and those implementing defect detection systems, in knowing which metrics might be most useful. We will use two metric-selection algorithms to rank all the metrics we collected and evaluate the highest-ranked metrics for predicting defects.

6. **How do defect prediction results change when method modifications and additions are considered?** For the majority of this work, we focus only on method modifications. In a software development setting, all developers' work must be checked for defects, not just modifications. We re-evaluate some of our defect detection attempts, including changes where a developer added new methods with changes when they modified existing methods.

1.2 Contributions

This work makes the following contributions:

- Investigate whether ASTs can measure developer knowledge and experience and how AST-based metrics compare to count-based metrics.
- Determine whether AST-based metrics have a statistically significant relationship to defects.
- Discover whether AST-based metrics can succeed when some count-based metrics are not applicable.
- Examine the performance of AST-based, count-based metrics, and existing metrics for method modification defect prediction.
- Identify the most important metrics for method modification defect prediction and evaluate their performance.
- Evaluate method-level defect prediction for software development, including method modifications and additions.

1.3 Thesis Structure

Chapter 2 provides technical background information and introduces related work. Chapter 3 explains our data collection process and the construction of our AST-based metrics. Chapter 4 presents the results of statistical analysis exploring the relationships between AST-based metrics and defects. Chapter 5 presents the results of machine-learning experiments predicting defects. Chapter 6 discusses our results. Chapter 7 presents our conclusions, future work, and threats to validity.

Chapter 2

Background and Related Work

This chapter introduces some essential terms and concepts. We present the software engineering and defect prediction processes and their related metrics. Following this, we discuss abstract syntax trees and the techniques we use to break them down and compare them. Finally, we will present the statistical and machine learning procedures used to evaluate our AST-based metrics and perform defect prediction.

2.1 The Software Development Process

The general software development process [33] is as follows:

1. A developer or user submits a bug report or feature request to the project's issue-tracking system. A bug is any problem with the software; we will refer to them as **defects** or **faults**.

2. A developer is assigned to make changes to the code to fix the defects or add the requested feature.

3. The developer creates a copy of the code in the current project using the **version control** system. This copy is called a **branch**.

4. The developer updates the code and adds tests necessary to ensure the fix or feature is working as expected.

5. When finished, the developer submits the changes for a **code review** by other developers. The request to combine the developer's branch of changes is a **pull request**.

6. Once the reviewers deem the code acceptable, they **merge** the changes with the main branch.

7. When fixes and features are ready, the updates are released to end users. Some projects will use a release cycle and release updated versions of software every few months. Other projects will release new features and important fixes as soon as possible.

The **version control** system is essential for tracking developer experience. This system tracks all modifications and additions to the code during the development process. A complete record of every change made to the source code over the lifetime of a project is maintained. Developers submit a group of changes as a **commit**, and the version control system records each addition and deletion made to the changed code. This history of changes makes it possible to inspect every modification made to a project and find the location and author of each change.

There are many different version control systems available to developers, but the most common are *Git*[2] and *GitHub*[3]. Git is a version control system, and GitHub is a cloud-based platform for managing projects that use Git.

2.2 Software Development Metrics

Many metrics are collected to understand better a piece of code or the circumstances in which it was changed. These metrics make it possible to represent a modification to a project as a set of numeric values. These values can be used to compare, analyze, or visualize all the changes made to a project. Below is a description of the most important metrics in our work. We present our new AST-based metrics in detail in Section 3.5 and give a complete list of all the specific metrics we use in Appendix A.

[2]https://git-scm.com/
[3]https://github.com/

2.2.1 Source Code Metrics

The following are metrics that are measurements taken directly from the source code. Listing 2.1 gives a simple Java class as an example for some metric explanations.

Listing 2.1: Example Java class with two small methods.

```
class Divide {
    // Divide two real numbers
    public static float divide (float a, float b) {
        return a / b;
    }
   // Divide two real numbers, and check for division by 0
    public static float checked_divide (float a, float b) {
        if (b == 0) {
            return NAN;
        } else {
            return divide(a, b);
        }
    }
}
```

- **Lines of Code** measures the total number of lines of source code in a piece of code. Blank lines and those that only contain comments are excluded. In Listing 2.1, the `Divide` class has 12 lines of code. The `divide` method has 3, and the `checked_divide` method has 7.

- **Number of Tokens** measures the number of tokens in a piece of code. A **token** is any character, symbol, or combination representing a single element in the code. For example, parenthesis, braces, and semi-colons are individual tokens. Compound symbols and all keywords and identifiers are individual tokens, such as `==`, `return`, `if`, `else`, and `class`. In Listing 2.1, there are 57 tokens in the `Divide` class, 18 tokens in `divide` and 35 tokens in `checked_divide`.

- McCabe's **Cyclomatic Complexity** [21] is a measure designed to indicate how difficult a piece of code will be to test. It was developed as an alternative to measuring code complexity by lines of code.

  Cyclomatic Complexity is calculated as the number of unique paths the execution that a piece of code can follow. A path through the code splits at each `if`, `else`, and `while` statement. Each of these splits increases the complexity by 1. The `divide` method has no control structures and can only be executed in one way, so its cyclomatic complexity is 1. The `checked_divide` has an `if` statement that makes it possible for the code to take one of two paths depending on whether or not the denominator is 0, so its cyclomatic complexity is 2. A long piece of code might have a low complexity if it comprises simple statements like function calls. However, a small amount of code with many `if` statements nested together could be highly complex.

2.2.2 Count-Base Experience Metrics

In general, *implementation expertise* metrics track the number of times a developer has made changes to different parts of a project's code. In this work, **count-based experience metrics** specifically measure how often a developer has changed individual methods in a project. For example, the number of changes made to a file is not the number of commits that include changes to that file, but the number of methods changed in that file across all commits. We use the following count-based experience metrics:

- **Method Changes** is the number of times a developer has changed a specific method. Throughout this work, unless otherwise specified, when we refer to a developer making a change we are talking about an individual method modification or addition.

- **File Changes** is the number of times a developer has modified a method in a file. If a developer modified more than one method in a file during a single commit, the count of their experience with this file would be greater than 1.

- **Project Changes** is the number of methods a developer has modified in each commit to a project.

2.2.3 Churn Metrics

Code churn measures how much code has changed over a period of time. It can be measured by the number of times a piece of code has changed or as a factor of the size of those changes by the number of additions and deletions. Research has shown code churn to be a reliable metric for predicting defects [12, 20, 23, 27]. We specifically measure churn as the changes made to code over the last four weeks. The rationale is that code that requires many changes over a short period has issues that will make it difficult to modify without introducing errors. We use the following churn metrics:

- **Developer's Churn** is the number of times the developer has modified a method in the last month. This number is also a part of their **method changes** experience, but it represents the most recent activity.

- **Other's Churn** is the number of times a method was changed in the last month by other developers who have worked on the project. Even if a developer has low individual churn, the piece of code may have high churn based on the activity of these other developers.

- **Additions/Deletions Churn** are the number of lines of code added or deleted in the file containing the modified method over the last month. Note with *Git*, editing done to a line of code is counted as both a deleted and an added line.

2.3 Defect Detection

Defect detection predicts whether defects are present in the source code. Traditionally, defect detection is done at the module level, usually involving individual files [4, 12, 22, 29, 35]. Near the end of a release period, when most changes to the code are finished, files are ranked by how many defects they are likely to contain. This ranking is done by extracting metrics from the source code and the version control history for each file and processing them with machine learning models. Research has shown that up to three-quarters of all defects can be found in 20% of a project's files [4]. By ranking the files, additional testing and quality assurance can prioritize the top-ranked files and identify most issues before release. Because there is a large amount of data for all the changes that have occurred over a months-long release period, these rankings can be very accurate. In addition, module-level defect detection only requires correctly identifying the ordering of the files with the most defects. It is unnecessary to correctly predict the number of defects as long as a file with many defects ranks near those with similar counts. However, there are limitations to this approach. Finding the defects in the identified files still takes a great deal of work. A file can have many contributors, so finding the most appropriate person to make changes can be difficult. Even when a developer has made most of the changes to a file, the details of those changes might be forgotten [16].

An alternative form of defect detection makes predictions when a change is made, usually at commit time [16, 20, 25, 31]. The files or methods in a change have source code and version control metrics extracted and processed by a machine learning model. The result is a prediction about whether or not the changes will introduce a defect. Commit-time defect predictions can catch a defect when the developer makes the changes, so these changes are still fresh in their mind. Identifying possible defects at commit time may increase the chance that errors will be found and improve the time required to fix them [16]. However, making these predictions is harder as less data is available for the change, especially if it is small. Incorrect predictions at commit time can also be more costly by causing developers to waste time searching code for defects that are not present.

2.4 Abstract Syntax Trees

An abstract syntax tree is a structured representation of source code. Figure 2.1 shows an AST for the expression `(1 + (3 - 5))`. Each inner node in the tree represents a rule for the language used to parse the expression. The **terminal nodes** at the bottom level of the tree contain the tokens from the text of the expression. The parentheses are tokens in the expression but are excluded in the AST as they are implicit in the tree's structure.

2.4.1 Path Contexts

Alon *et al.* break an AST into its individual path contexts [2]. A **path context** is a path through a tree from one terminal node to another. A single terminal node will have a path to every other terminal node, following edges through its parents and back down to each terminal node of that parent. Paths are generated from left to right, and once a path between two terminal nodes has been added, it will not be added again in reverse. For example, the right-most terminal node will not appear as the first entry in any paths because all the paths it is a part of are already included. Each path starts and ends with a terminal node; no

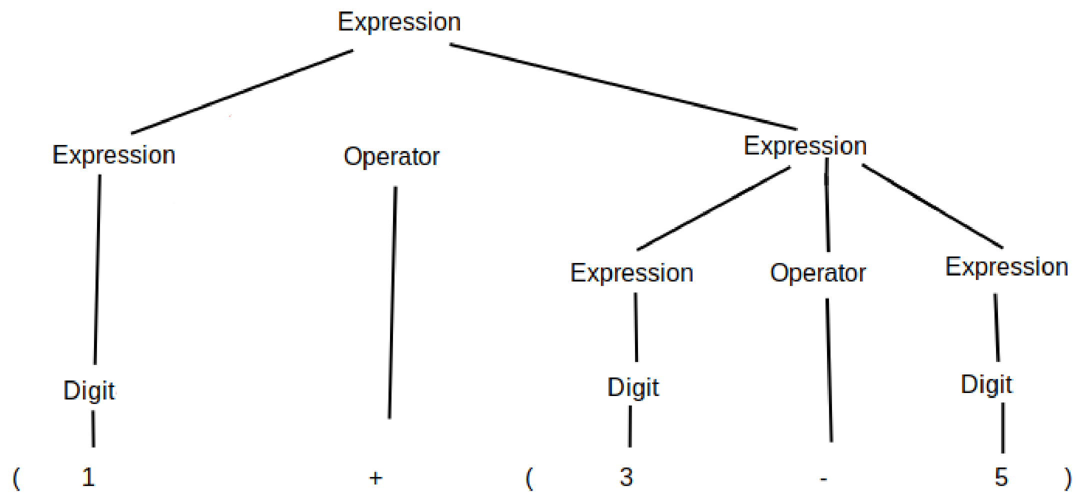


Figure 2.1: AST for a simple arithmetic expression

terminals appear in the middle.

Figure 2.2 shows some path contexts from the abstract syntax tree in Figure 2.1. We only include paths from 1 to the other terminal nodes for readability. All the path contexts in Figure 2.2 are as follows:

- **Red**: (1, Digit, Expression, Expression, Operator, +)
- **Green**: (1, Digit, Expression, Expression, Expression, Expression, Digit, 3)
- Yellow: (1, Digit, Expression, Expression, Expression, Operator, -)
- **Blue**: (1, Digit, Expression, Expression, Expression, Expression, Digit, 5)

The remaining paths are:

- (+, Operator, Expression, Expression, Expression, Digit, 3)
- (+, Operator, Expression, Expression, Operator, -)
- (+, Operator, Expression, Expression, Expression, Digit, 5)
- (3, Digit, Expression, Expression, Operator, -)

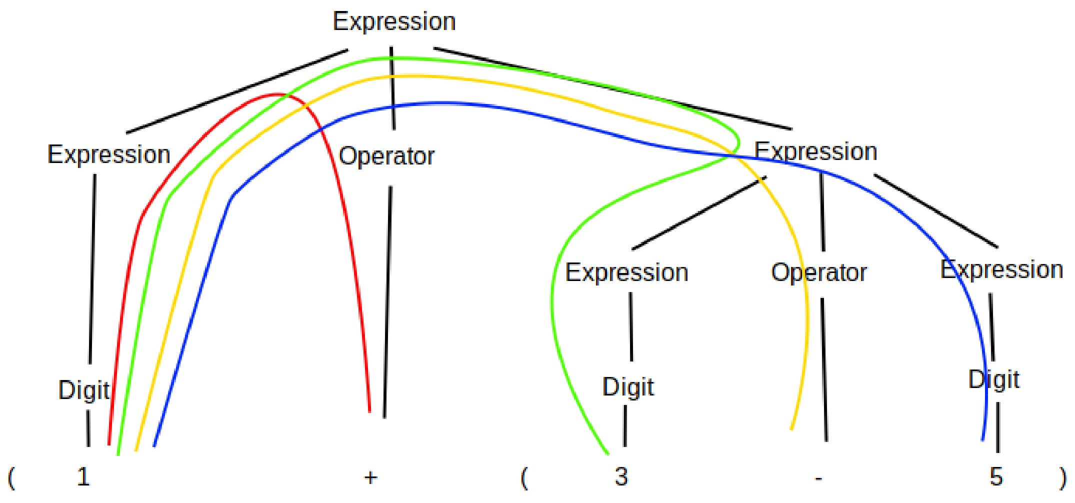


Figure 2.2: AST for a simple arithmetic expression with some path contexts.

- (3, Digit, Expression, Expression, Expression, Digit, 5)
- (-, Operator, Expression, Expression, Digit, 5)

While it may not be apparent from the example tree, a tree is generally larger than the code it represents. The set of path contexts from a tree is even larger. The tree in Figure 2.2 has ten paths. If we were to add two expressions of the same size $((1+(3-5))+(1+(3-5)))$ together, the resulting AST would have $20+25=45$ paths. The ASTs generated from a Java method with hundreds of tokens could have thousands of path contexts, each containing hundreds of labels.

Alon *et al.* generate vectors from path contexts and embed these in a high dimensional space to represent a method [2]. We take a similar approach and use a mathematical set of all the path contexts. We compare two ASTs by comparing their path context sets. How we do this is explained in detail in Section 3.5 when we describe how we calculate our AST-based experience metrics.

2.4.2 Tree Kernels

An alternative to using path contexts to compare trees is directly comparing them with **tree-kernels**. A tree kernel is a method of computing the numeric similarity of two trees by comparing the common sub-trees. Given two trees T_1 and T_2, we take the set of nodes in each tree, defined as N_T, and compute a difference between the sub-trees of each node in N_{T_1} with the sub-trees of each node in N_{T_2} and sum the results. This process is defined by Collins *et al.* [8] as

$$Tk(T_1,T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1,n_2) \tag{2.1}$$

where $\Delta(n_1,n_2)$ can be computed as:

- if sub-trees rooted at n_1 and n_2 are different then $\Delta(n_1,n_2)=0$

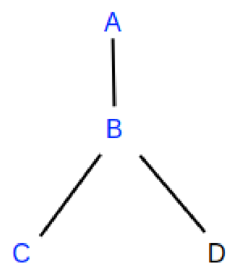


Figure 2.3: A simple tree with two terminal nodes.

- if the sub-trees rooted at n_1 and n_2 are the same, and n_1 and n_2 have only children that are terminal nodes then $\Delta(n_1, n_2) = \lambda$
- if the sub-trees rooted at n_1 and n_2 are the same, and n_1 and n_2 have one or more non-terminal children then $\Delta(n_1, n_2) = \lambda \prod_{j=1}^{nc(n_1)} (\Delta(c_{n_1}^j, c_{n_2}^j))$, where $nc(n_1)$, is the number of children of n_1 and c_{n^j} is the j^{th} child of n_1.

λ is a decay factor designed to reduce the contribution of larger trees. We used the KELP tree-kernel implementation to calculate tree kernels [4] in our experiments. This software has $\lambda = 0.4$ as a default.

Using the tree in Figure 2.3, we can calculate its similarity with itself as follows:

1. Let $\lambda = 0.4$
2. Given T_1 and T_2 let $N_{T_1} = A_1B_1$ and $N_{T_2} = A_2B_2$
3. Now calculate δ for each pair $n_1, n_2 \in N_{T_1}, N_{T_2}$, starting with the terminal nodes.
   - $\Delta(B_1, B_2) = 0.4$ as they are the same trees and only have terminal node children.
   - $\Delta(A_1, B_2) = 0$ as they are not the same same trees.
   - $\Delta(A_2, B_1) = 0$
   - $\Delta(A_1, A_2) = \lambda\Delta(B_1, B_2) = 0.4 \cdot 0.4 = 0.16$ as they only have one child node each, which is not a terminal node.

[4]https://github.com/SAG-KeLP/kelp-additional-kernels

4. Finally, $Tk(T_1, T_2) = 0.4 + 0 + 0 + 0.16 = 0.56$.

The KELP tree-kernel implementation also compares the terminal nodes. This means that with source code two pieces of code with identical AST structure, but different variable names, would have slightly different similarity scores. The similarity of ASTs includes the code's keywords, identifiers, and symbols. We can add this to the above result as follows:

- $\Delta(C_1, C_2) = 0.4$
- $\Delta(D_1, D_2) = 0.4$
- Finally, $Tk(T_1, T_2) = 0.4 + 0.4 + 0.56 = 1.36$.

2.5 Statistical Learning

Statistical learning is the process of trying to find a function that describes the relationship between a response variable Y and a set of one or more predictor variables $X = \{X_1, X_2, ..., X_n\}$.

$$Y = f(X) + \varepsilon \tag{2.2}$$

The function f maps values in X to values in Y, and ε is an error term independent of X with a mean of zero. Statistical learning aims to estimate f as closely as possible using existing data [15].

There are two different reasons to estimate f. The first is to learn more about the data, and the second is to make predictions about unseen data. We will discuss the former in Section 2.5.1 and the latter in Section 2.6. For both, our focus is on **binary classification**, where one of two categories is assigned to each data point in a dataset. For commit-time defect detection, these categories are *defect* or *no-defect* based on whether a change introduced a defect. When making predictions, we have a data point not yet assigned to one of these two categories, and we want to assign it to one based on its metric values.

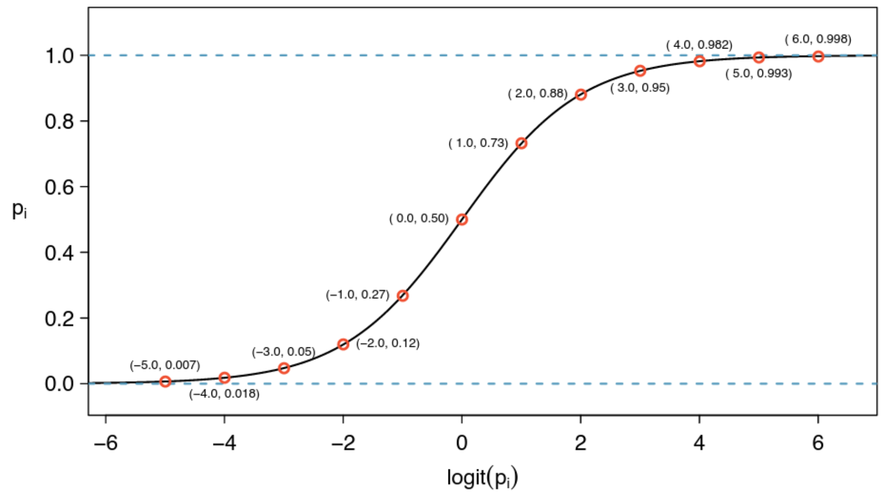


Figure 2.4: The logit function [10].

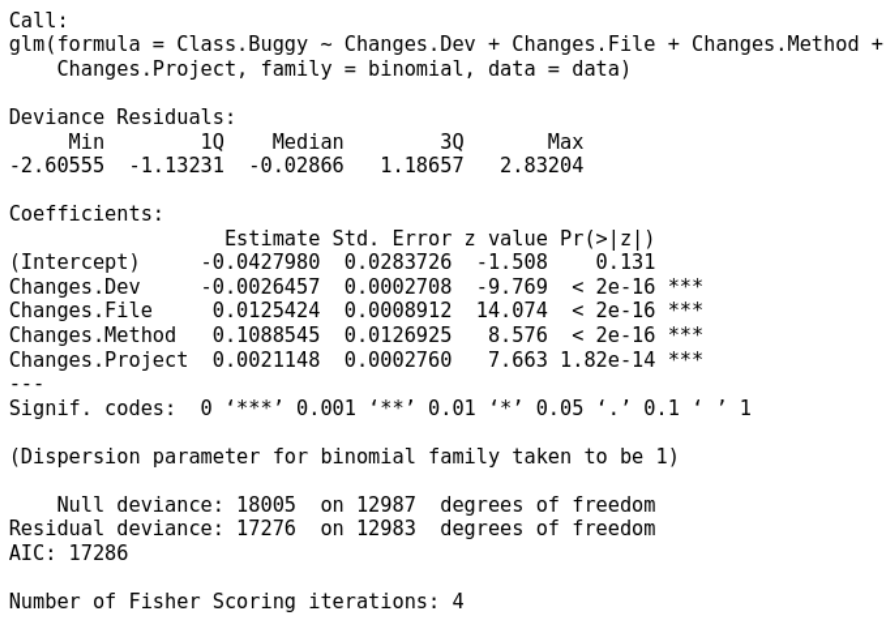
2.5.1 Logistic Regression

Equation 2.5 gives the logistic regression function [15].

$$\log(\frac{P(X)}{1-P(X)}) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n \tag{2.3}$$

where $P(X = [x_1, x_2, \ldots, x_n])$ is the probability that an instance of data X is in the group we are interested in predicting, also known as the success group. in our context, this is the group of changes that will cause a defect. The $\log(\frac{P}{1-P})$ is the **logit** function. Figure 2.4 shows the form of this function and its relationship to P. When the result is above 0, the probability of X being labelled a success is greater than 50%, and the data is predicted to be in the success group. The $\log(\frac{P}{1-P})$ gives us the log odds of an outcome. Each β_i corresponds to a metric in an instance of data and determines the increase in the log odds when that metric increases by 1, provided all other metrics stay the same [15].

Because solving a logistic regression equation gives us the coefficients, it can be used to understand better the relationships between the predictors and the categories. We can build logistic regression models with statistical software and inspect the coefficients. A sample logistic regression model summary is shown in Figure 2.5, which was built using tools from



```
Call:
glm(formula = Class.Buggy ~ Changes.Dev + Changes.File + Changes.Method +
    Changes.Project, family = binomial, data = data)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.60555  -1.13231  -0.02866   1.18657   2.83204

Coefficients:
                  Estimate Std. Error z value Pr(>|z|)
(Intercept)     -0.0427980  0.0283726  -1.508    0.131
Changes.Dev     -0.0026457  0.0002708  -9.769  < 2e-16 ***
Changes.File     0.0125424  0.0008912  14.074  < 2e-16 ***
Changes.Method   0.1088545  0.0126925   8.576  < 2e-16 ***
Changes.Project  0.0021148  0.0002760   7.663 1.82e-14 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 18005  on 12987  degrees of freedom
Residual deviance: 17276  on 12983  degrees of freedom
AIC: 17286

Number of Fisher Scoring iterations: 4
```

Figure 2.5: A summary of a logistic regression model built with R.

the R programming language[5]. This model is built from one of our datasets and shows four count-based experience metrics being used to predict whether a method modification will introduce defects.

In Figure Figure 2.5, the `Coefficients` section shows each predictor's β value in the `Estimate` column. The coefficient for `Changes.Method` is 0.1088. When making a method modification, increasing the number of previous modifications to the same method by one would increase the $\log(odds)$ of introducing a defect by 0.1088. We can convert this to an odds value by exponentiation $e^{0.1088} = 1.115$. This conversion changes our interpretation to apply the odds instead of the $\log(odds)$. Note that there is a multiplicative change in the probability. For a metric, an increase of one might be a large increase, but for another, it might be a very small increase. We could adjust our interpretation further to find how it affects the odds if a developer's previous modifications increase by ten $e^{10\cdot 0.1088} = 2.96 \approx 3$. So for this model, the odds of the developer with ten more modifications to the method will be three times more likely to introduce a defect. Note that the negative coefficients will

[5]https://www.r-project.org/

decrease the odds in the same way.

In order to evaluate whether a metric makes a statistically significant contribution to a model's predictions, we use the p-values in the `PR(>|Z|)` column of Figure 2.5. The p-value for a coefficient tells us the probability of the resulting coefficient having at least this value, in the positive or negative direction, if the coefficients true value were 0, i.e. the metric contributes nothing to the predictions. Typically, if a p-value is lower than 0.05, there is strong statistical evidence that the value of the coefficient is not 0 [10].

Another issue arises from the collection of metrics we include in the model. The p-value is only relevant for the particular model and not all models where the metric could be included. For example, the model in Figure 2.5 only has four experience metrics. In this model, they are all considered significant, but if we were to add other metrics that also might be related to defects, the result could change. So if we want to investigate the marginal impact of a single metric or a group of metrics, we first need to build a **control model** that includes all of the metrics we want to account for. These other metrics might be proven contributors to the outcome or metrics we believe might affect the outcome. For example, many health study models include age or how much the participants smoke or drink. We use metrics listed in Section 2.2 to test our new AST-based metrics.

2.6 Machine Learning

Machine learning is a form of statistical learning based on computational techniques to build models focusing on making predictions [15]. When performing the types of statistical analysis presented in Section 2.5.1, we build and inspect the model but do not use it for predictions. For analyzing the predictive power of our metrics we focus on the use of several machine learning algorithms that are commonly used for defect prediction.

In order to use a statistical or machine learning model for predictions, it must be tested to determine if it can make accurate predictions. In order to do this, a subset of that data is kept aside as **testing data** and the rest is used as **training data** to build the model. There are two complications when splitting the data. First, in order to capture all of the possible patterns and associations in the data we would want to use all of the data for training the model. By holding some data back we are possibly losing important information for making predictions. Second, the smaller we make our testing set of data the less likely we are to test a varied set of instances. One way to deal with these issues is to use cross-validation. **Cross-validation** splits the data n times and, for each split, uses a different $\frac{1}{n}$ of the data for testing and the rest for training. The results for each model are averaged to create the final result. This process ensures that the model is trained with the larger set of data for each split and still tested with every available data point at least once. With a well chosen n, cross-validation also helps prevent issues where the model fits the training data too well and provides very accurate results for testing but does not perform well when deployed (**over-fitting**). This is not necessarily true for all values of n and $n = 5$ or $n = 10$ are common values used as they have been shown to reduce test error rates from high bias or high variance [15].

2.6.1 Machine Learning Algorithms

We used five machine-learning algorithms in our work. We chose these algorithms because they are common in other defect detection research, and they come from different categories of algorithms [29]. As a result, we treat them primarily as black boxes and are only interested in their success in predicting defects and how those results compare to other work.

The algorithms we chose are:

- **Logistic Regression (LR)**: As described in Section 2.5.1.
- **Naive Bayes (NB)**: The Naive Bayes algorithm employs Bayes' theorem to find the probability of an outcome y given a set of metrics $x_1, ..., x_n$. Naive refers to an assumption that all conditional probabilities of the metrics to a category are independent. Each new data point computes this probability for each possible category, and the

data point is labelled as belonging to the category with the highest probability [13].

- **Support Vector Machines (SVM)**: Support Vector Machines separate points in n-dimensional space with a hyperplane. In 2-dimensions, it would amount to finding a vector that separates the data points by category as much as possible. A new instance would then be categorized based on which side of the vector it falls on [11].
- **K-Star (KS)**: The K-Star algorithm is an instance-based learning algorithm where a new data point is compared to other data points and uses the *k* most similar instances to categorize the new data point. K-Star computes the probability that the new data point belongs to each category using transformations between it and the data points in the category. It uses an entropy based distance function where the new data point is labelled as belonging to the category with the highest probability [7].
- **Random Forest (RF)**: The Random Forest algorithm builds multiple decision trees based on the training data. The resulting model predicts the category of a new data point by predicting it with each tree and taking the most common result. A decision tree is formed by choosing a metric that best splits the data into separate categories. If a split does not entirely separate, the data in that split is considered, and the tree splits again until the resulting split has only one category in each branch. The random forest creates a more diverse set of trees than similar techniques by restricting the available metrics for each split [11].

2.6.2 Evaluation Metrics

There are several different measures to determine the prediction success of an algorithm. Three of the most common are **accuracy**, **precision** and **recall**. Each measures a particular aspect of the prediction results and can be more important in a given context.

In order to evaluate a model, a **confusion matrix** is constructed from the results of individual predictions. With two categories for prediction, a confusion matrix is a 2x2 table

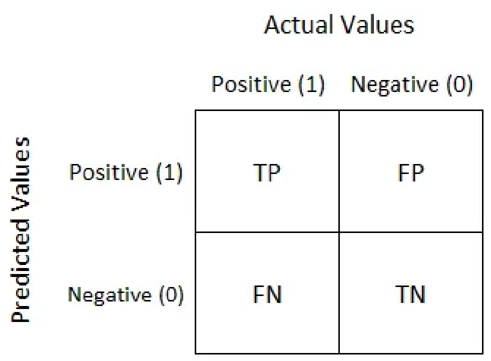


Figure 2.6: A 2x2 confusion matrix.

of the different outcomes of the prediction. Figure 2.6 gives an example confusion matrix. The table has four different classifications depending on where a prediction falls.

- **TP** is a true positive when the predicted category and the actual category are both in the positive category. In our work, this would be correctly predicting that a defect was introduced by a method modification.

- **TN** is a true negative when the predicted category and the actual category are both in the negative category. In our work, this would be correctly predicting that a defect was not introduced by a method modification.

- **FP** is a false positive when the predicted category is positive, but the actual category is negative. In our work, this would mean predicting that a defect was introduced when a defect was not introduced.

- **FN** is a false negative when the predicted category is negative, but the actual category is positive. In our work, this would mean predicting that a defect was not introduced when a defect was introduced.

Accuracy measures the total proportion correct predictions out of all predictions made. In our context a high accuracy would mean that most predictions were accurate, but it will not tell us how balanced predictions are between defect and no-defect changes. Equation

2.4 shows the calculation for accuracy.

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \tag{2.4}$$

Precision measures the proportion of true positives to all the predictions of the positive class. In our context, a high precision would mean that if we predicted a modification introduced a defect, there was a high probability that it was a defect. Equation 2.5 shows the calculation for precision.

$$Precision = \frac{TP}{TP+FP} \tag{2.5}$$

Recall measures the number of true positives over the number of true positives plus false negatives. In our work, a high recall would mean successfully finding most of the modifications that introduced defects. Equation 2.6 shows the calculation for recall.

$$Recall = \frac{TP}{TP+FN} \tag{2.6}$$

2.6.3 Metric Selection

One of the difficulties of machine learning is knowing what metrics or features to use. Too many metrics can lower prediction accuracy due to what is colloquially known as the curse of dimensionality [19]. The more metrics there are, the more dimensions in the feature space. With each dimension added, more data is needed to find a way to separate the categories. We use two algorithms to reduce the number of metrics and attempt to select those that are most important for defect detection in our experiments. As with the machine learning algorithms, we will mostly treat these feature selection algorithms as black boxes.

The algorithms we chose are:

- **InfoGain** uses entropy to determine a metrics value. It works by splitting a dataset with a metric and calculating how well that split separates data points into each category [13]. The main downside to this process is that it considers the metrics conditionally independent. If there are significant interactions between the metrics, the results will be less accurate. However, *InfoGain* is a popular feature selection algorithm.

- **RelieF** attempts to consider each metric's context. It does this by calculating distances between individual data points and using the distances of similar and dissimilar points when considering a metric's ability to split the data based on categories. By considering the locality of data points, the context of features is considered when determining the quality of a metric [19].

These algorithms rank all the features based on their ability to differentiate classes. We use this ranking to pick a smaller set of features that will perform similarly, or better, than the full set of metrics.

2.7 Related Work

In this section, we present additional details from previous works that were most influential to our work. These fall into two general categories. First, works that have explored the relationships between experience and defects. Second, works that used different representations of ASTs for tasks such as method name prediction and defect prediction.

2.7.1 Experience and Defects

To measure software expertise, Mockus and Herbsleb first proposed the idea of experience atoms [24]. These fundamental units measure a developer's experience with parts of a large code base. For example, when a developer changes a file, those changes can be recorded as developer experience. By measuring each developer's changes to a code base module, they created a tool called *experience browser* which helped developers find others with the most experience with a piece of code.

Bird *et al.* used change counts to the Windows operating system to investigate the effect that code ownership had on defects [5]. They define *ownership* as the ratio of commits made to a piece of code by a single developer to the total number of commits made to that code. Using this, they defined minor and major contributors as those that authored less or more than 5% of the commits to a piece of code. In this case, the unit of code was a binary file in the Windows operating system. These binary files are roughly equivalent to a project representing a single program in our data. Because they worked with binaries, they did not have access to full version control data but instead used release information of reported bugs in each binary.

They hypothesized that code that had a large number of minor contributors was more likely to have defects. They created several defect prediction models from contributor information and other software metrics, like lines of code, complexity and churn. Baseline models included only software metrics, and additional models added contribution data. The results showed a statistically significant increase in accuracy when including data on minor contributors.

Pandey *et al.* review release-time defect prediction literature and list some of the most common algorithms and metrics [29]. Algorithms include variations on Bayesian learners like Naive Bayes, support vector machines and decision trees like Random Forest. Most studies include metrics similar to lines of code and complexity. However, many also include metrics specific to object-oriented programming that measure the number of methods in a class or class dependencies. An overview of results showed several models achieving an accuracy of above 90% in predicting defects, but with mean and median values between 80-85%.

We want to build on these experience measures and better understand how experience correlates to defects. Bird *et al.* used a coarse grain measure on full programs successfully and showed that contributors with less experience introduce more defects into a codebase [5]. These works motivate us to investigate the effect of a finer-grained count-based experience on defect prediction. We build on the idea of experience by attempting to capture a more nuanced representation of experience and test it with similar techniques.

2.7.2 Abstract Syntax Trees

Alon *et al.* use a process they call *Code2Vec*, which converts code and its path contexts to vectors [2]. A collection of path context vectors represents a method body. They experimented with using this representation to predict method names. Each method is labelled based on its name, and the assumption is that most methods are named based on their purpose. Models trained with the path context vectors and method names can predict the most likely method name for an unseen method with greater than 90% accuracy. The process also weights the path contexts based on which are most important to the prediction.

Kovalenko *et al.* used the **Code2Vec** process to build implicit style representations of authors working on a single code base [17]. Their goal was to compare the style of these authors and to understand how those styles changed over time. They also wanted to know how a developer's experience working with others might change their style.

Both works use path contexts, though in a vectorized form and with neural networks. We do not build on these works but use their success in representing code for different purposes as motivation for using path contexts. We are not using vectorized versions of the path contexts, and we are not using neural networks for our predictions. However, the success of path contexts in these works gives us reason to include them in our work. Particularly, the success of method name prediction suggests some link between the AST and a method's purpose.

Sahar *et al.* take a different approach to represent the AST by using the AST directly [31]. They perform code duplication and defect detection using ASTs and tree-kernel methods. They use the similarity measure first to predict if the two methods are duplicates. They use the success of this similarity method to do defect detection in a novel way. Instead of looking at a file or module to determine how many defects it might contain, they compare single methods.

The process first includes labelling methods as having introduced defects. An unlabelled method is compared to a collection of labelled methods with tree-kernel methods. The top *k* most similar methods are then used to label the new method. The new method is labelled as a defect if the majority of similar methods are labelled as defects. They tested this method on several projects from the Apache Software Foundation. Some of their models achieved greater than 90% accuracy. For large enough projects, results are all above 60% and, in many cases, improve on other defect prediction methods used on the same project.

The success of tree-kernel methods in this context motivates us to use them for comparing ASTs and building AST-based experience metrics. Other forms of just-in-time defect detection focus on sets of changes or whole commits [16, 25]. Detecting defects in individual methods makes it even easier to focus a developer's attention on code that contains defects. We are motivated by the success of Sahar *et al.* for method-level of commit-time defect detection. However, where Sahar *et al.* use only tree kernel comparisons to identify if a method change will introduce a defect, we include many other common metrics along with our AST-based metrics.

2.8 Summary

In this chapter we covered the necessary background for our work. We discussed the software development process and introduced common source code and experience metrics. We explained the use of path contexts and tree kernels to work with ASTs. We introduced the statistical and machine learning methods that we use to evaluate our results. Finally, we summarized some other contributions that are related to our work.

Chapter 3

Methodology

In this chapter, we present the sources for our data and the tools we used for data collection. Following this, we outline our data collection process; how we turn project repositories into individual method changes and how we collect metrics from the data. Next we briefly detail the process and tools used to perform statistical analysis and defect prediction. Finally, we explain how we formulate our AST-based metrics.

3.1 Data Sources

We use the *Technical Debt Dataset*[6] [18] to select software projects and collect defect information. This dataset is a collection of different software quality measures for 33 projects from the Apache Software Foundation. The Apache Software Foundation is a group that funds, maintains, and supports many open source projects[7]. All projects measured for the *Technical Debt Dataset* must have been written in the Java programming language, be at least three years old, and have over 100 classes and 500 commits. Each commit for a project is labelled as *Fault-Inducing* for commits that introduced a defect and *Fault-Fixing* for commits that fixed existing defects.

Our data contains 27 projects from the *Technical Debt Dataset.* A total of five projects failed to process properly, likely due to extremely large commits or methods. Because the projects are all part of the same organization, some contributors have worked on multiple projects.

[6]https://github.com/clowee/The-Technical-Debt-Dataset. We used release 2.0.
[7]https://github.com/orgs/apache/repositories

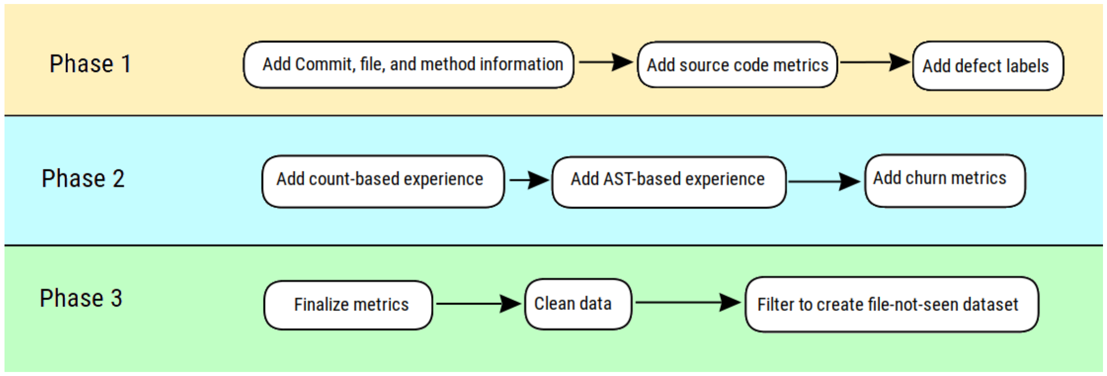


Figure 3.1: Data collection process

3.2 Data Collection Tools

We use the *pydriller*[8] Python library [34] to iterate over all the commits in a project. *Pydriller* can identify the source code before and after a change for each method changed in a commit and collects metrics such as lines of code and cyclomatic complexity.

We use the *JavaParser*[9] Java library to convert Java code into ASTs. We parse Java classes with a custom parser that transforms an AST into an s-expression for easy storage and processing.

3.3 Data Collection Process

The data collection process has three phases. The first phase processes each repository to produce a list of method changes with source code metrics, defect labels, and ASTs. At the end of phase one, we have a collection of records for each method change made to each project. The second phase sorts the records from each project by developer and adds developer experience. We then combine all of the records and add churn metrics. At the end of phase two, we have one large collection of method change records for each project with our metrics added. In phase three, we clean this set of records to create our final datasets.

Note that each project has a single GitHub repository. In many cases project and repository may be used interchangeably. We use repository in most of the following discussion

[8]https://pydriller.readthedocs.io/en/latest/
[9]https://javaparser.org/about.html

to reinforce that we are working with the version control data for a project.

3.3.1 Phase One

In phase one, we process each repository individually to identify method changes and add source code metrics and method ASTs. We iterate over all commits in the main branch of each repository from the earliest commit to the most recent. For each commit, we process each file where methods were changed. For each file, we process each changed method in the following manner:

1. *Add commit information*: author, time of the commit, number of files modified, and total line additions and deletions. Is it a *fault-inducing* commit? If yes, save the *fault-fixing* commit identifier to use when adding defect labels to method changes.

2. *Add file information*: file change type (*added*, *moved*, *modified*), file lines of code, file cyclomatic complexity, file line additions and deletions.

3. *Add method information*: method change type (*added*, *modified*), method source metrics (before and after the change), and method ASTs (before and after).

4. *Add method defect labels*: use *fault-inducing* and *fault-fixing* commit information to add one of two labels to the method change.

    - **Clean**: the commit was not labelled as *fault-inducing* or if the commit was labelled as *fault-inducing* the method was not part of modifications to fix the fault in a future *fault-fixing* commit. The changes made to the method are not seen to as part of the cause of a defect.

    - **Fixed**: the method was in a *fault-inducing* commit and was modified in a future *fault-fixing* commit. The changes to the method are seen to be the cause of a defect.

We exclude commits, files, and methods based on the following criteria:

- *A commit contained over 50 method changes.* Best software development practices are to commit to a small collection of related changes. We expect that commits that change many methods are likely automated changes made to adjust a variable or method name or perform a similar task. They can also be commits where a project was reorganized, and many files were moved to different locations.

- *A method has more than 100 lines of code.* A large method can have a very large AST. Storing and comparing large ASTs can be too computationally complex to be done in a reasonable amount of time. Large methods like this are rare. In a dataset of 14 million Java methods Alon *et al.* found that the average method length was 7 lines of code. We chose 100 to limit the impact of large methods on computation while reducing the number of available methods as much as possible.

Despite these exclusions, we found that processing all the repositories and adding experience information takes several days[10], and intermediate steps can generate over 10 GB of data. The computational loads could be somewhat mitigated with access to multiple CPUs such as a distributed or cloud computing environment. As discussed below, the experience data is added to each author individually. While as much of the experience collection for a single author is run in parallel, it would likely be possible to adjust the process to also collect each author's experience data in parallel by distributing the collection for each author to a separate CPU.

In order to add experience data, all commits must be organized by the author. There were a total of 528 individual authors in the dataset. However, several were removed or combined with others. The final data contained only 490 authors. Author numbers are summarized in Table 3.1.

Some authors are present in multiple projects, so we combine these authors' commits. Some authors had slightly different user names between projects. We manually identified

[10]All data collection steps were executed on an 11th Gen Intel(R) i7-11700k processor (8-Core/16-Thread, 16M Cache, 3.6GHz to 5.0GHz

and combined all possible duplicate authors, so each had all their changes under a single name. For example, names like `Alex-Herbert` and `aherbet` were combined, as well as names like `Carl-Hall` and `Carl-Franklin-Hall`. In a case where two authors had very similar names, but we could not be sure they were the same author, we excluded both from the data. We excluded a total of 6 users for this reason. We expect that even within a large organization, it is unlikely that there were multiple users with identical names. The most likely explanation for repeat usernames is migrating a project from one version control system to another, for example, from a local company server to GitHub. However, in these cases, it is possible that if two users were combined who are not the same developer, some experience information could be affected. We found and combined 21 users that had multiple names and combining them reduced the total author count by 26.

Other problems caused the removal of several other authors. Five authors were removed because the size of their associated data was larger than 40MB. Adding experience to an author with more than 20MB took large amounts of time to process. The number 40MB was chosen to reduce computational costs while removing as few authors as possible. Finally, a single author was removed because errors were encountered during processing that could not be fixed.

Table 3.1: Dataset author numbers

	Total	Name Issues	Too Large	Errors	Duplicates	Final
Authors	528	6	5	1	26	490

3.3.2 Phase Two

The second phase adds experience and churn information to method change records. We process each author individually and iterate over their earliest to most recent commits to collect experience metrics as follows:

1. *Add count-based experience*: count changes made to the method, file, and project over the previous six months.

2. *Add AST-based experience*: determine *path coverage*, *path similarity*, and *tree kernel* experience based on the changes from the previous six months. We present the details of these metrics in Section 3.5.

3. *Move current methods to experience list*: Once a commit is fully processed, all individual method changes are added to the list of previously changed methods. The list is filtered to ensure only the previous six months commits are included.

After we add all individual experience metrics for each developer, we combine the complete set of method change records into a single database. We process this complete database to add the remaining churn metrics. These metrics include the number of times all developers changed a method and the number of line additions and deletions over the last six months.

3.3.3 Phase Three

Before performing our analysis, we must make some final metric adjustments and additions.

1. *Set category labels*: 1 for *Defect* and 0 for *No-Defect*. Based on the defect labels added in Section 3.3.1, we label *Fixed* methods as defects.

2. *Add log values*: for example, $\log(LinesOfCode)$. Some models performed better when including various metrics on a log scale [4].

3. *Remove method additions*: most of our analysis focuses on method modifications to understand how developers' experience affects their ability to understand and modify code without introducing defects. A developer's experience differs when they must understand and modify existing code instead of creating the solution from scratch. We are removing the change instance for when a method was added, not all data related to the addition of code. So, while the instances of method additions themselves are not included, their effect on developer experience is.

Table 3.2: Datasets and Sizes

Dataset	Initial	Cleaned			Final Balanced
	Total	Total	Defect	No-Defect	Total
Full Modification	76963	74454	66153	8301	16602
File-Not-Seen	27093	26503	24539	1964	3928

This cleaned dataset contains all method modification data that applies to our analysis. We will refer to it as the *full modification data*. We use this full dataset to produce an additional dataset that includes only first-time changes to a file or method. First time changes are those where a person was not responsible for the original addition of the code or any modifications to it over the projects lifetime. In order to understand how well our AST-based measures capture information on the code's general structure, we require changes where their experience does not include the exact code they are modifying. We refer to this additional dataset as the *file-not-seen modification data* because it only contains method modifications for changes where a developer has not previously worked on the file.

Analysis and prediction generally perform best when each class has an even number of instances. In the datasets, the number of defects is roughly one for every ten method modifications. We balance our datasets by selecting all of the *defect* changes and then selecting a random subset of the *no-defect* changes of the same size. We give the dataset sizes of the two in Table 3.2.

3.4 Statistical Analysis and Machine Learning

In order to better understand how AST-based metrics relate to defects, we used statistical modelling. We also used machine learning algorithms to evaluate the possible contribution of these metrics to existing defect prediction. More detailed descriptions of these processes are given in Chapter 4 and Chapter 5. Here we give a description of the tools and the general approaches used.

The statistical analysis was carried out using the R[11] statistical software. We used the

[11]https://www.r-project.org/

R package *ggplot2*[12] to visualize the relationships between AST-based metrics and defects. We also visualized the relationships between AST-based and count-based metrics to see how they related to one another. In addition to visualizations, we used R functions `cor` for correlations between metrics and `t.test` for t-tests.

For statistical modelling, we used the R function `glm` for logistic regression with the option `family="binomial"`. We built many models with this function. First, we built models with only a single AST-based metric to test how these metrics related to defects on their own. We then built a control model using common defect prediction metrics such as *Lines of Code* and *Cyclomatic Complexity*. The R summary of the control models are given in Appendix B. Finally, we built models that extended the control model with one of the AST-based metrics to see if the AST-based metrics had statistically significant relationships to defects when we considered well-established metrics for defect prediction.

The machine learning analysis was carried out using the WEKA machine learning framework. For each experiment, the relevant dataset was loaded into the WEKA Explorer. Once a dataset is loaded, an algorithm can be selected and the type of training and testing split can be chosen. As previously stated, we used 10 fold cross-validation for all of our machine learning models. We used all algorithms with WEKA's default settings. As these were our first experiments of this type, there was no rationale for changing the defaults. Using the defaults means that it is easy for others to reproduce and expand on our experiments in the future. With a selected algorithm, models can be built and tested with the click of a button. The results of the model and the aggregation of cross-validation results are summarized when the process is complete. We collected for accuracy, precision, and recall from these results. We built control models for each algorithm using the same metrics as in the statistical analysis. We also built models with only AST-based metrics and various other models using all the metrics we collected and some smaller metrics sets chosen with WEKA's metrics selection tools. We then compared the various models to

[12]https://ggplot2.tidyverse.org/

determine whether our AST-based metrics improved on existing metrics and to determine which metrics have the most predictive power for just-in-time defect prediction.

As noted earlier, WEKA was also used for metric (feature) selection. A separate tab in the WEKA application offers several feature selection algorithms. These algorithms function the same way as the modelling algorithms. Once a dataset is loaded, an algorithm is selected and run. Results of the feature selection are given in the app as a ranking or subset of the features in the data. Both the *InfoGain* and *RelieF* algorithms we used provide a ranking of the features based on their expected importance for predicting defects from the data. We chose the top 10 features from each algorithm in each of our datasets.

3.5 AST-Based Experience Metrics

This section discusses how we formulate each of our five AST-based metrics. We begin with some definitions:

- **Current method** is the method modification or addition we are examining.
- **Previous Methods** are all the methods the developer added or modified in the six months prior to the *current method*.
- **Before-AST** is the AST of a method before a change is made.
- **After-AST** is the AST of a method after a change is made.
- **Coverage** is defined as the proportion of items in set *A* that are also in set *B*:

$$coverage = \frac{|A \cap B|}{|A|} \tag{3.1}$$

- **Jaccard Distance** [14] is a set similarity measure defined as

$$jaccard = \frac{|A \cap B|}{|A \cup B|} \tag{3.2}$$

3.5.1 Path-Based Experience

We calculate the following three experience metrics using sets of path contexts from the *current method* and the *previous methods*. Each is a slight variation on previous metrics and has different possible benefits and issues.

Total Path Coverage

Total path coverage measures how many path contexts in the *current method* are covered by the complete set of path contexts from the *previous methods*:

1. All path context sets from each of the *previous methods* are unioned. If applicable, we include path context sets from both *before* and *after* ASTs from each method.

2. Coverage is calculated as in Equation 3.1, where A is the set of path-contexts from the *current method* and B is the result of Step 1.

The potential benefit of this metric is that it will reflect whether a developer has any experience with a single path context in the *current method*. If they change a method that shares small pieces with several other methods, they will have high coverage, even if no previous method is identical. However, a possible issue with this metric is that when a developer changes a method more than once, and no other developers make changes in between, they will have 100% coverage of the path contexts in the method. If too much time has passed between those changes, the 100% coverage may not reflect their knowledge properly.

Top K Path Coverage

Top *k* path coverage measures average coverage over the path contexts in *current method* using individual methods from *previous methods*:

1. Coverage between each of the path-context sets from each of the *previous methods* and the path-context set from the *current method* is calculated as in Equation 3.1. *A* is

the *current method*, and B is a method from *previous methods*. For *previous methods* with both a *before* and *after* AST, the largest of the two coverage values is used.

2. The top k highest coverage values are kept. We investigate $k = 1, 5, 10$.
3. The average of the top k coverage values is computed. Note that if there are less than k *previous methods*, the missing methods are assumed to have zero coverage.

A possible benefit of this metric is that it smooths our instances where a developer works on the same method several times in a row. In order to have 100% coverage, a developer must work on a method k times. A possible issue with this metric comes from comparing individual methods. The resulting coverage no longer measures all the paths from *previous methods* but how many similar methods a developer has worked on previously. The final value will be low if a method shares small parts with many *previous methods* but is not highly similar to a single method.

Top K Path Similarity

Top k path similarity is calculated the same as for *Top K Path Coverage*, except *jaccard* is substituted for *coverage*. Instead of calculating one method's coverage over another, we are considering their overall similarity. This difference offers a different approach to measuring method experience. The main caveat is that we can no longer think of coverage as a direct proportion of experience. We can only say that there is a high or low similarity between the method a developer is currently working on and the methods they have worked on previously.

3.5.2 Tree Kernel Similarity

Tree kernels work similarly but with ASTs instead of path contexts. The following two metrics use tree kernels and two different adjustments for similarity calculations. The adjustments are necessary because the similarities between ASTs are highly dependent on the size of the trees [8]. For example, two identical trees with 4 nodes might have a similarity of 1.36. In contrast, two identical trees with 18 nodes might have a similarity of 5.85[13].

Top K Normalized Tree Kernel Similarity

Top k normalized tree kernel similarity normalizes the tree kernel values. We will call this Tk' using Tk as defined in Equation 2.1. Collins *et al.* define the normalization formula [8] as:

$$Tk'(A,B) = \frac{Tk(A,B)}{\sqrt{Tk(A,A) \cdot Tk(B,B)}} \tag{3.3}$$

In order to remove some extreme outliers we excluded values that fell above 3 for the $k = 1$ metric where no averages were taken. Figure 4.3 shows that for the $k = 5$ metric the range is approximately 0 to 1.5 with only a small number of data points falling after 1.2. We chose the range based on outlier analysis using inter-quartile range outlier analysis [10] and adjustments to exclude the most extreme outliers while excluding as little data as possible from each dataset.

These adjustments removed approximately 1000 entries from the full modification dataset and 500 entries from the the file-not-seen modification dataset. In both cases these entries were removed before the dataset was balanced for defect and no-defect modifications. This results in less than 2% reduction in the data for each dataset.

The process to calculate the similarity is the same as Section 3.5.1 using Equation 3.3 instead of *coverage* (Equation 3.1). The result is the average of the top k normalized tree kernel similarities between the *current method* and each of the *previous methods*.

With this metric we are calculating similarities using the AST directly. Path contexts include terminal nodes from the AST as the first and last elements. Using path contexts code that uses large numbers of custom identifiers may have a low similarity to another

[13]These values were calculated with two random trees and will vary depending on the shape and depth of the tree. The difference between the similarities will remain.

method that performs the exact same task but has different variable names. The tree kernels consider the terminal nodes separately from the inner nodes, so two identical methods with different variable names will still be highly similar. As with *Top K Path Similarity*, the main caveat is that we can no longer think of this similarity as a direct proportion of experience. We can only say that there is a high or low similarity between the method a developer is currently working on and the methods they have worked on previously.

Top K Self-Adjusted Tree Kernel Similarity

Calculating top k self-adjusted tree kernel similarity is done in the same way as Section 3.5.1, except we substitute Equation 3.4 for coverage and call this value Tk''.

$$Tk''(A,B) = \frac{Tk(A,B)}{Tk(A,A)} \tag{3.4}$$

The benefit of this metric is that it considers the context of the similarity like *coverage*. For example, consider the following $Tk(A,B)$ and $Tk(B,A)$. Using Equation 3.3 both similarities are the same. If A is small and appears as a sub-tree in B, we might want $Tk(A,B) = 1$. We would get this result if we took $coverage(paths_A, paths_B)$. We might also want the reverse $Tk(B,A) < 1$ because the structure of A only covers a small amount of B.

The issue is that tree kernel calculations are more complex than path context *coverage* and in some situations, has unexpected results. For example, if A is contained in B three times, we might get $Tk''(A,B) \approx 3$ but $Tk''(B,A) \approx 0.33$. We might expect that one of these results should be 1. This situation will be rare in ASTs in real-world projects, but it is important to note the possibility. It also means that not all values will fall between 0 and 1.

We again performed outlier analysis to exclude the most extreme values from the data before balancing defect and no-defect modifications. In this case, we excluded values above 40 for the $k = 1$ metrics. Figure 4.3 shows that the majority of the data is below 20 for the $k = 5$ metric. These adjustments resulted in approximately 3% of modifications being excluded from each dataset. However, there is likely overlap in outliers between the two metrics, so at most 5% of the data was removed from each dataset.

3.6 Summary

This chapter presents the methods we used to collect data and create gather software development and experience metrics. We collected 27 Apache Software Foundation open-source Java projects with defect information in the *Technical Debt Dataset* [18]. We split these projects into individual method changes with information from projects, commits, files, and methods. Each change has source code metrics, churn metrics, and developer experience metrics added. We clean the data to create a final dataset representing an equal number of method changes labelled as *defect* and *no-defect*. We take a subset of this data and create a second dataset with only method changes where a developer has never worked on the file.

We also explain how our AST-based metrics are constructed. We presented three metrics built with path-context sets and two metrics that use ASTs and tree kernels. All of these metrics are slightly different in order to fully investigate what techniques are most representative of developer knowledge and experience.

Chapter 4

Experience Metric Analysis

In this chapter, we present the results of the statistical analysis used to answer our first three research questions:

- **R1**: How do AST-based experience metrics and count-based experience metrics relate to one another?
- **R2**: Do AST-based metrics have a statistically significant relationship with method modification defects?
- **R3**: When count-based experience is not applicable, do AST-based experience metrics have a statistically significant relationship to defects?

For AST-based metrics, we separated the previous six months into two time periods: the most recent month (**M1**) and the five months that came before it (**M5**). A six month period was chosen for experience to ensure that code changes counted as experience were recent enough to affect a developers knowledge of the code. Because one of our churn measurements uses count-based experience over the most recent month, when exploring metric relationships for both AST-based and count-based metrics, we will focus on M5 in order to exclude experience from churn. For all AST-based metrics top-k averages, we focus on $k = 5$ (**K5**) in the presented results as those metrics performed best. In addition, all metrics used in this analysis measure developer experience with the code *before* they made a modification.

Table 4.1: Correlation between count-based experience and AST-based experience metrics for full modification data. Moderate and strong correlations are presented in bold.

Measure	Method Changes	File Changes	Project Changes
Total Path Cover	0.290	0.244	0.261
Top K Path Cover	0.382	0.341	0.294
Top K Path Similarity	**0.416**	0.324	0.268
Tree Kernel Similarity	0.242	0.195	0.331
Tree Kernel Experience	-0.083	0.006	0.176

4.1 Relationships Between Experience Metrics

Our first research question asks how AST-based and count-based metrics relate. To answer this question, we will explore the correlation between all experience metrics and the general relationship that each experience metric has to defects.

4.1.1 Metric Correlations

The AST-based metrics contain a record of the developer's experience with specific methods they modified. However, we hypothesize that the AST-based metrics contain more general programming knowledge and experience than count-based metrics. If AST-based metrics are too sensitive to a specific method, they may only have a high value when comparing identical methods. If this is the case, we expect the correlation between the AST-based and count-based metrics will be strong. However, if the AST-based metrics capture more subtle similarities than count-based metrics, we expect the two metrics will be weakly correlated. We consider a correlation value over +/- 0.7 as strong, between +/- 0.4 and +/- 0.7 as moderate, and below +/- 0.4 as weak.

Table 4.1 shows correlations for AST-based metrics and count-based metrics for method

Table 4.2: Correlation between each count-based experience metric for full modification data.

Measure	Method Changes	File Changes
Method Changes	-	-
File Changes	0.335	-
Project Changes	0.181	0.248

modifications over a 6 month period and with $k = 5$. There are no strong correlations between the AST-based and count-based metrics. A moderate correlation exists between *method changes* and *Top-K Path Similarity*, but the rest are all weakly correlated to count-based metrics. The *tree-kernel experience* metric has the weakest correlation to the count-based metrics. These results suggest that the two classes of metrics do record different kinds of experience information.

Table 4.2 shows the correlations between count-based metrics. Again, there are no moderate or strong correlations between the different types of counts. *Method changes* and *file changes* have the highest correlation, although they are still weak. This result makes sense as a change to a method is also a change to a file, but with many methods in a file there should generally be more file changes unless modifications to the file are focused on a single method.

Table 4.3 shows the correlations between each AST-based metric. Each AST-based metric is a variation of the other metrics, so we expect stronger correlations. Here we see that the path-based metrics are highly correlated with one another. These three metrics use the same strategy for decomposing an AST and use slightly different set similarity measures, so we expect them to be highly similar. The *tree-kernel similarity* is moderately similar to the path-based metrics. This result is somewhat unexpected because the tree kernel uses the AST differently from the path-based metrics. However, both use the AST, and the comparisons will have similarities. The *tree-kernel experience* is again very weakly correlated to the other metrics, even the other tree-kernel metric. Therefore, the calculations

Table 4.3: Correlation between each AST-based experience metric for full modification data. Moderate and strong correlations are presented in bold.

Measure	Total Cover	Top-K Cover	Top-K Sim	Tree Kernel Sim
Total Path Cover	-	-	-	-
Top K Path Cover	**0.825**	-	-	-
Top K Path Similarity	**0.816**	**0.952**	-	-
Tree Kernel Similarity	**0.538**	**0.534**	**0.532**	-
Tree Kernel Experience	0.022	0.066	-0.014	0.092

used to normalize/adjust the tree kernel results are significant.

The correlation values are instructive, but it is also helpful to visualize these relationships. Figure 4.1 has several plots showing the relationships between count-based and AST-based experience metrics. The x-axis is the AST-based experience metrics. The y-axis is the count-based experience metrics. Each dot represents a method modification, where **red** dots are modifications that *did* introduce a defect while **blue** dots are modifications that *did not* introduce a defect.

We are looking for obvious patterns between two metrics in Figure 4.1. We would expect to see the points concentrated around lines or curves if there were strong correlations between the metrics. Instead, for most of the plots, we see large masses of of mostly spread out points indicating weak correlations.

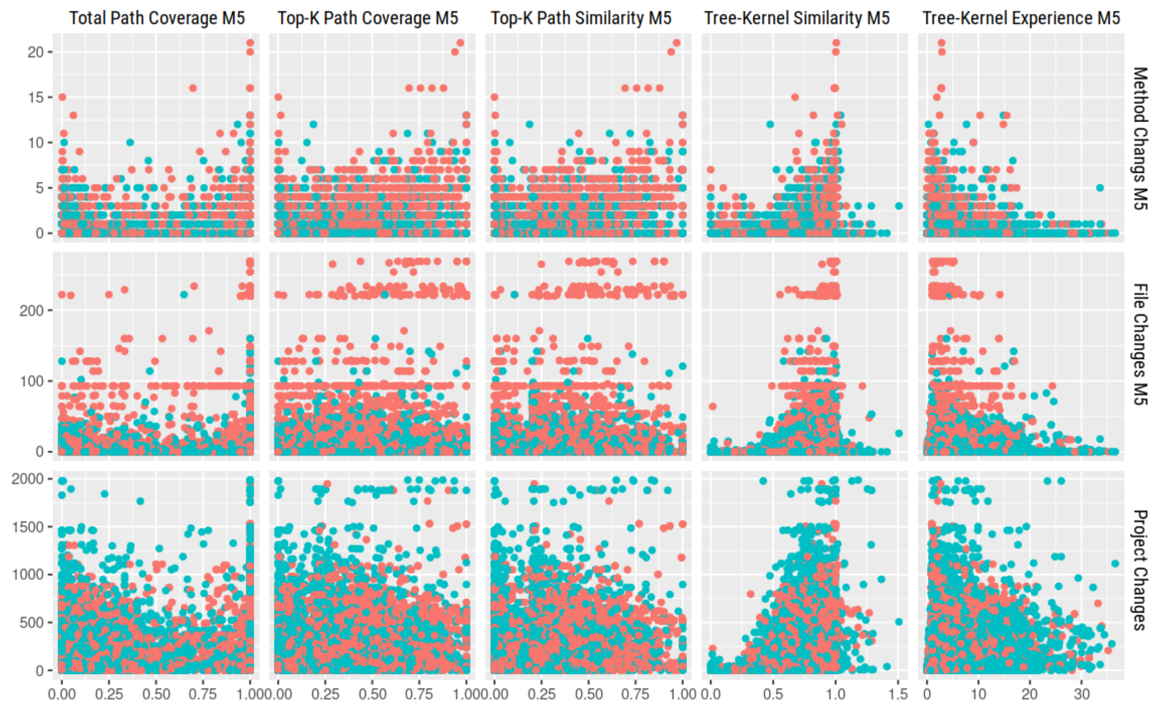


Figure 4.1: AST-based experience metrics v.s. Count-based experience metrics. Red indicates a modification induced a defect.

4.1.2 Relationships Between Metrics and Defects

Another way to understand the differences between the types of experience metrics is how they generally relate to defects. Figure 4.1 shows us some of these differences. However, it is difficult to determine whether an increase or decrease in any experience metric results in more or fewer defects. The exception is the *file changes* plots in row 2. There is a large concentration of red points above 200. This concentration of points indicates that when a developer has changed a file more than 200 times[14] it is highly probable that the change will result in a defect. As mentioned when discussing churn, this relationship is likely because a file that has required 200 or more requires these changes because it contains many defects or has other issues that make changing it without causing defects difficult.

In order to get a general understanding of how each experience metric relates to defects, we used Welch's two-sample t-test [10]. The t-test compares the average (**mean**) difference for a metric between two groups. Table 4.4 shows the results for t-tests measuring the mean value for each experience metric between the group of modifications that *did* introduce a defect and the group of modifications that *did not* introduce a defect. A positive difference indicates an increase in defects as the metric values increase, and a negative difference indicates a decrease in defects as the metric values increase. A p-value of below 0.05 tells us that the result is statistically significant. In this case, it indicates a low probability that we would see a given result if the means in the two groups were identical.

Table 4.4 shows the results of the t-test for both count-based and AST-based metrics. There is a statistically significant difference in defect, and no-defect means for all experience metrics except *project-changes*. We are most interested in the magnitude and sign of the means for each metric. Perhaps the most interesting observation is that all metrics except *tree-kernel experience* have a positive relationship with defects. This relationship says that as experience increases, the chance of introducing a defect also increases. This result is not what we might expect from experience metrics, especially since these metrics

[14]Note that these experience metrics exclude the most recent month.

Table 4.4: T-test results for experience metrics excluding the most recent month (**M5**) for full modified data.

Metric	Mean Defect	Mean No-Defect	Difference	P-value
Method Changes	0.72	0.42	0.30	$<$ **2.2e-16**
File Changes	10.69	2.94	7.75	$<$ **2.2e-16**
Project Changes	188.70	188.48	0.22	0.9531
Total Path Coverage	0.47	0.37	0.10	$<$ **2.2e-16**
Top-K Path Similarity	0.20	0.15	0.05	$<$ **2.2e-16**
Top-K Path Coverage	0.16	0.11	0.04	$<$ **2.2e-16**
Tree Kernel Similarity	0.74	0.70	0.04	$<$ **2.2e-16**
Tree Kernel Experience	3.41	4.76	-1.35	$<$ **2.2e-16**

exclude the most recent month of changes (churn), and is the opposite of what has been observed in previous research [5]. Because the metrics have different scales, comparing each difference's magnitude is difficult. However, we can compare the magnitude of the difference relative to the defect mean values. As we saw in Figure 4.1, the *file changes* appear to have the largest difference relative to its scale, the difference is approximately 72% of the defect mean value. Again, a developer with many file changes appears to indicate a high probability that they will introduce a defect. The rest of the differences between means is relatively small. Therefore both count-based and AST-based metrics have similar relationships with defects. This result is expected since both metrics measure experience from previously modified code, even if they measure that experience differently.

4.2 AST-based Experience and Method Modification Defects

Our second research question asks whether AST-based metrics are statistically significant indicators of defects in method modifications. The t-tests in Section 4.1.2 and the plots in Figure 4.1 illustrated the general relationship between AST-based experience metrics and defects. The t-tests only consider each variable in isolation and the plots were difficult to use to discern any meaningful relationship between the metrics and defects. In this section we present some density plots for the AST-based experience metrics to see better how defects are distributed for each metric. We then present the results of various statistical models that improve on our t-test results by accounting for additional metrics that have proven strong relationships to defects such as *lines of code* and *cyclomatic complexity*.

4.2.1 AST-Based Metrics And Defects

Figure 4.2 and Figure 4.3 show the density of method modifications (y-axis) for each level of experience (x-axis). Experience over the most recent month (M1) is shown in the left column and experience for the five months before the most recent month (M5) is shown in the right column. Again, the modifications that did introduce a defect are shown in **red**. Using these plots, we can better see the increase in defects with an increase in experience.

The path-based metrics shown in Figure 4.2 have some interesting features. The most notable is *total path coverage* in the first row. This metric has two large peaks at the lowest and highest levels of experience. This metric uses a set of all the path contexts a developer has seen, meaning if they work on a method more than once in a short period, this metric will show high level of experience. For the M5 metric, defects are more common when working on code where a developer has high experience. This result looks more like churn than we might expect. However, the M1 which measures the same period as churn has fewer defects at higher levels. So there may be more complicated reasons for these relationships. The other two rows have very similar densities. There are no longer spikes of high levels of experience which is to be expected, given that these are averages of the top five most similar methods. In order to get high levels of experience, one would have to work on a highly similar method five or more times. The M5 (column 1) plots for both show the same decrease in defects for low experience changes than shown in row 1. Both also show higher levels of defects for higher levels of experience.

The tree-kernel-based metrics shown in Figure 4.3 have a different shape than the path-based metrics. The similarity metric in row 1 has a more normal distribution with a small second peak at low levels. Compared to the path-based metric plots, the main difference is that most changes are centred at higher experience levels. However, the tree-kernel similarity metric in row 1 also has more defects at higher experience levels. The tree-kernel experience metric in row 2 is the only metric with fewer defects at higher experience levels. Like the averaged path-based metrics, most changes are made at lower experience levels. The curve as experience increases is also smoother than the averaged path-based metrics, which have some small peaks at higher experience levels.

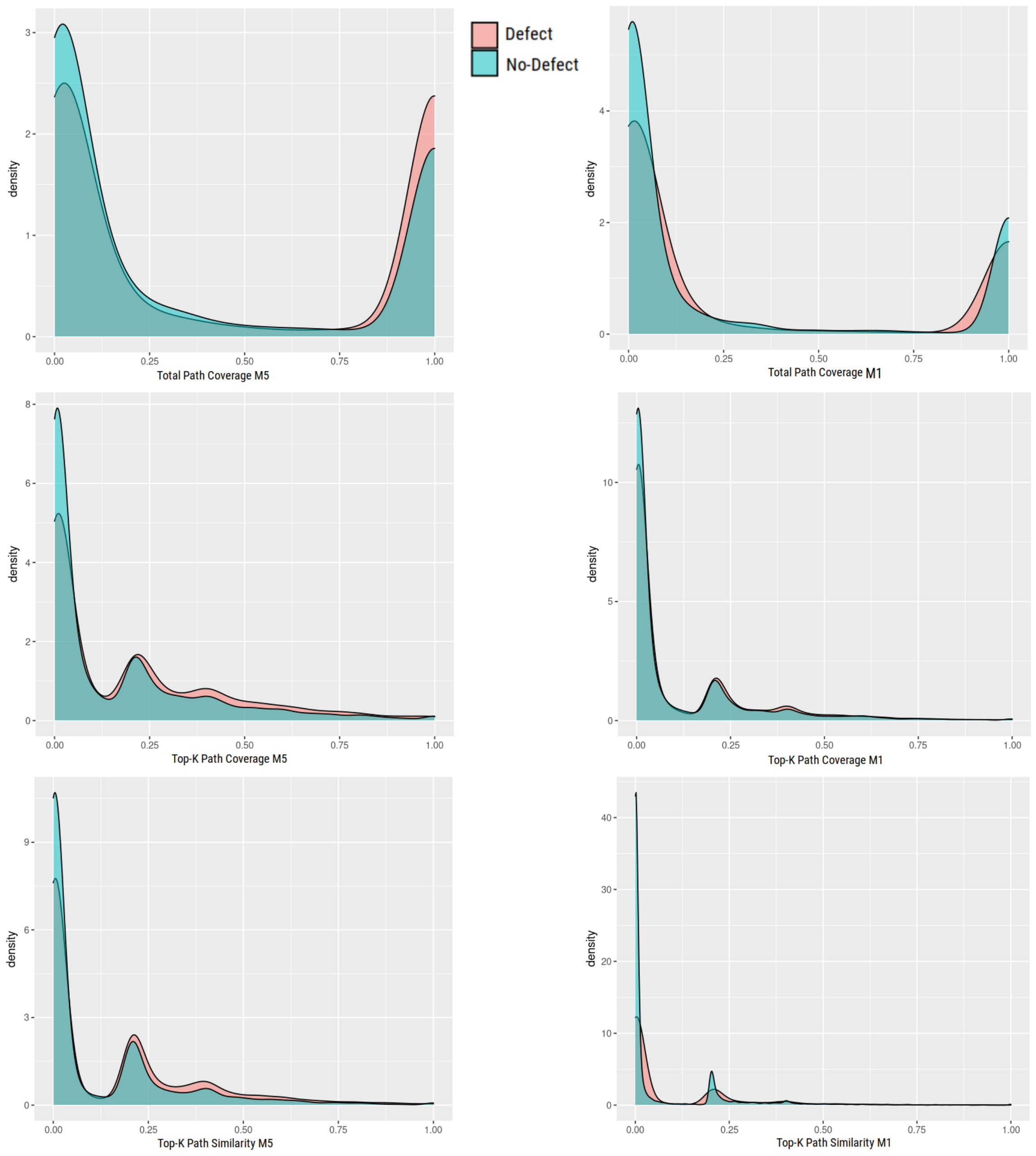


Figure 4.2: Path-based experience densities.

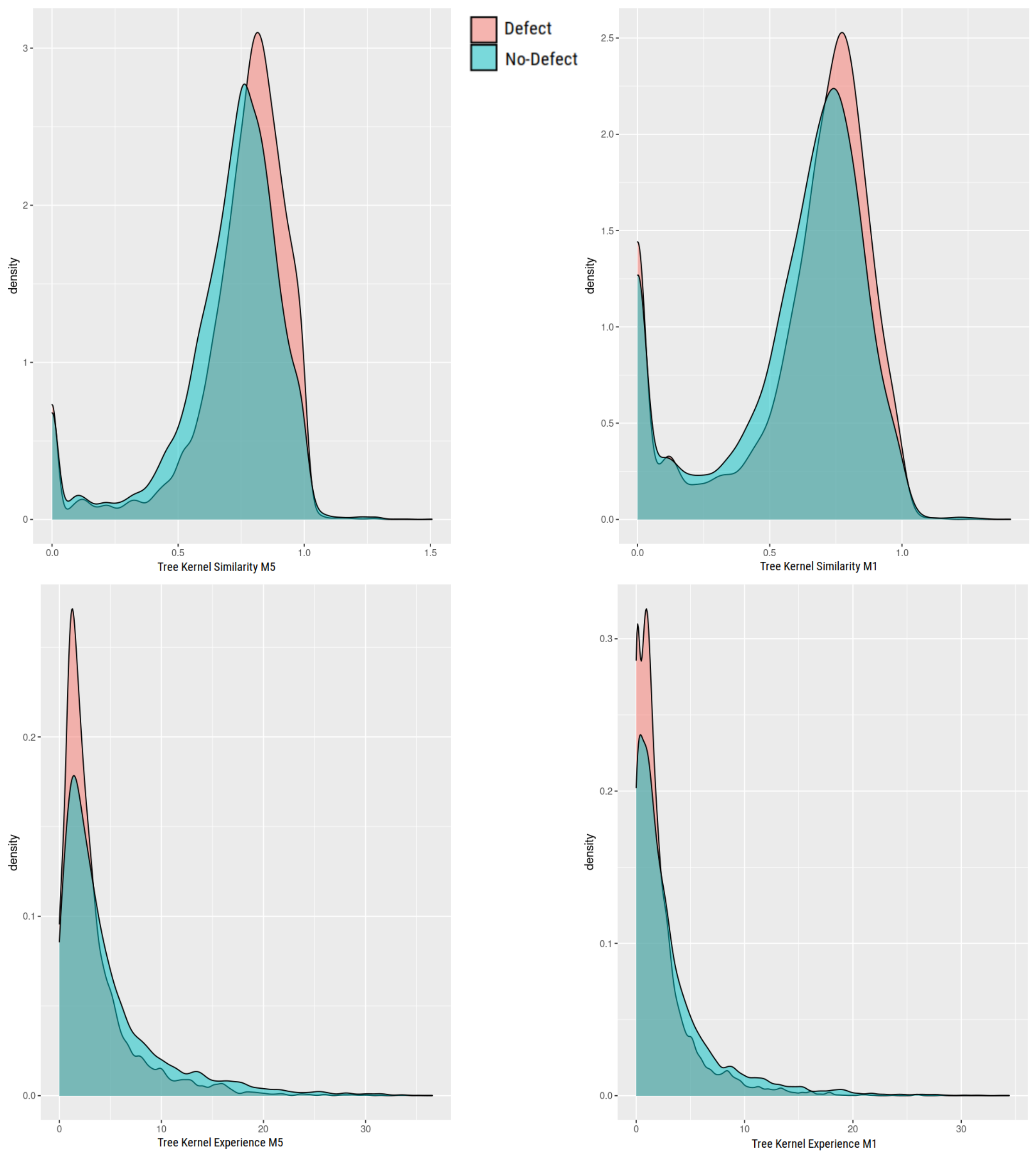


Figure 4.3: Tree-kernel-based experience densities.

4.2.2 Modelling AST-Based Metrics and Defects

Here we use logistic regression models to better understand the relationships between AST-based experience metrics and defects. One of the values of the logistic regression model is that it can give us an idea of how a change in a metric's value will affect the odds of causing a defect. In our context, the model's coefficients give us the $\log(odds)$ that a change of 1 in the metric's value will cause a defect, provided all other metrics' values remains the same.

Table 4.5 contains the coefficients and p-values for each AST-based metric in a model where that metric is the only variable included in the control model. The control model includes common defect prediction metrics such as *Lines of Code*, *Cyclomatic Complexity*, *Churn*, and count-based experience metrics. As model is large, a summary is included in Appendix B. As mentioned above, the coefficients represent the change (multiplicative) in the $\log(odds)$ of causing a defect if the metric changes by 1. The p-value tells us the probability that the coefficient of a metric would have an estimated value at least this far from 0, if that coefficient was 0. We present any p-value below 0.05 in bold for easy identification. The first two columns show the results for a metric when it was the only metric included in the model. The next two columns show the results for a metric when it was added to the control model.

The models illustrate a few important things. First, each metric is statistically significant when it is the only metric used. All metrics are less significant when added to the control model, but only the path-based M1 metrics are no longer considered statistically significance at the 0.05 threshold. This result provides evidence that AST-based experience metrics do indeed capture information that the control metrics do not. The direction

Table 4.5: Coefficients and p-values for individual AST-based metrics experience measures in logistic regression models for predicting modification defects for full modification data.

Metric	Individual		With Controls	
	Coefficient	P-value	Coefficient	P-value
Total Path Cover M5	0.486	$<$ **2e-16**	0.352	**2.22e-13**
Total Path Cover M1	0.209	**1.8e-8**	0.113	0.058
Top-K Path Cover M5	0.852	$<$ **2e-16**	0.4236	**9.57e-5**
Top-K Path Cover M1	0.338	**7.83e-5**	-0.7362	0.646
Top-K Path Sim M5	1.111	$<$ **2e-16**	0.476	**3.17e-4**
Top-K Path Sim M1	0.575	**9.57e-9**	-0.143	0.502
Tree-Kernel Sim M5	0.938	$<$ **2e-16**	0.489	**2.71e-9**
Tree-Kernel Sim M1	0.325	**3.37e-9**	0.224	**0.00142**
Tree-Kernel Exp M5	-0.071	$<$ **2e-16**	-0.027	**1.07e-9**
Tree-Kernel Exp M1	-0.085	$<$ **2e-16**	-0.027	**7.90e-7**

of the relationships between each metric and defects also matches those in the t-test, and these directions do not change from individual to control models when the metric remains significant.

Previous methods did not indicate the magnitude of the relationship between the experience metrics and defects. For example, the *Tree-Kernel Sim M5* metric has a coefficient of 0.938 in the individual model and 0.489 when included in the control model. This result indicates that a developer with a *Tree-Kernel Sim M5* value of 1 is $e^{0.938} \approx 2.55$ times (individual model) and $e^{0.489} \approx 1.6$ times (control model) more likely to introduce a defect when modifying a method compared to a developer with *Tree-Kernel Sim M5* of 0. For this metric a change in 1 is large. The scale of this experience is between 0 and 1.5 as discussed in Section 3.5.2 and visualized in Figure 4.3. It is more likely that there will be smaller changes in experience between developers or for a developer over time. We can adjust the scale and see how a 0.1 change affects the odds of defects. A developer is $e^{0.1 \cdot 0.938} \approx 1.098$ times (individual model) and $e^{0.1 \cdot 0.0438} \approx 1.050$ times (control model) less likely to introduce a defect when making a change compared to a developer with 0.1 less tree kernel experience. While the results show a statistically significant relationship, the small changes in odds suggest that it may not be useful to select developers to make a change using their AST-based experience unless the difference between them is large.

4.3 AST-based Experience and First Time Method Modifications

Our third research question asks how relationships between our AST-based experience metrics change when a method is modified for the first time. One of the main issues with using method and file change counts as experience is that they only tell us the developer's experience with specific code and do not apply to any other code or general programming knowledge. Total and project-based counts can tell us whether a developer has made many modifications over a period of time. However, they cannot give us any idea of the developer's general programming knowledge. The benefit of AST-based metrics is that they can still be used in this situation, and they have the potential to capture more general programming knowledge. Because this knowledge is implicit, the programmer is not likely to consciously recall pieces of the ASTs from code they have modified in the past. However, we can test to see if their implicit knowledge makes a difference in making successful defect-free method modifications.

In Section 4.2.2, the AST-based metrics were statistically significant, but the data included instances where a developer may have modified a method many times. These results therefore do not tell us if the information captured by AST-based metrics is general knowledge or similar to information captured by count-based metrics. In this section, we examine how the AST-based metrics perform when there is no chance that a specific method was modified previously. In these cases, the AST-based experience is gained only from the structure of similar but not identical code.

As discussed in Chapter 3, we created a second dataset with only modifications made by developers who had at least ten previous commits and had not previously modified the file containing the current method. All analysis in this section uses this *file-not-seen* dataset. Note that because it is more common for developers to work on the same portion of a project, this dataset is much smaller than the full modification dataset used in Section 4.2.

4.3.1 Metric Correlations

First, we re-examine the correlations between count and AST-based metrics. Table 4.6 shows these correlations in a single table. The method and file entries are excluded as their values are all zero. Again, all of the AST metrics weakly correlate with project counts, although the correlations are slightly weaker than with the full modification data. The *Tree-Kernel Exp* correlation with path-based metrics increased slightly but is still very weak. Otherwise, the two datasets have no notable changes in experience metric correlations.

Table 4.7 shows the t-test results for the file-not-seen data. There are two notable changes. First, the project changes metric is now statistically significant. So when a de-

Table 4.6: Correlation between each AST-based experience metric for file-not-seen modification data. Moderate and strong correlations are in bold.

Measure	Project Chgs	Total Cover	Avg Cover	Avg Sim	Kernel Sim
Total Cover	0.214	-	-	-	-
Top K Cover	0.211	**0.863**	-	-	-
Top K Path Sim	0.192	**0.773**	**0.871**	-	-
Tree-Kernel Sim	0.225	0.265	0.214	0.298	-
Tree-Kernel Exp	0.164	0.112	0.150	0.059	-0.093

veloper has not seen a file before, the number of changes they have made to the project in the past is important for determining whether they may introduce a defect. This metric also has a negative association with defects. So the more changes to a project the developer has made, the lower the chance they will introduce a defect when modifying a method. The other change is that the path-based metrics and the tree kernel similarity metric have much lower differences between their means. The two path-based metrics that use coverage are no longer statistically significant. The means for defect and no-defect changes have increased for the tree-kernel experience metric, from 3.41 to 3.79 and 4.76 to 5.23, respectively. Average *Tree-Kernel Exp M5* has increased even though there are no modifications to identical methods, which is interesting and unexpected.

The density plots for path-based metrics in Figure 4.4 have changed noticeably. The most significant change is in the *total path coverage* plot in row 1. The spike at 1.0 experience is almost gone now. This result is expected because there are no identical methods in

Table 4.7: T-test results for experience metrics excluding the most recent month (**M5**) for file-not-seen modified data.

Metric	Mean Defect	Mean No-Defect	Difference	P-value
Project Changes	140.69	154.67	-13.98	**0.0317**
Total Path Coverage	0.10	0.11	0.01	0.105
Top-K Path Similarity	0.04	0.05	0.01	**0.016**
Top-K Path Coverage	0.02	0.02	0.00	0.086
Tree Kernel Similarity	0.70	0.68	0.02	**3.12e-7**
Tree Kernel Experience	3.79	5.23	-1.44	$<$ **2.2e-16**

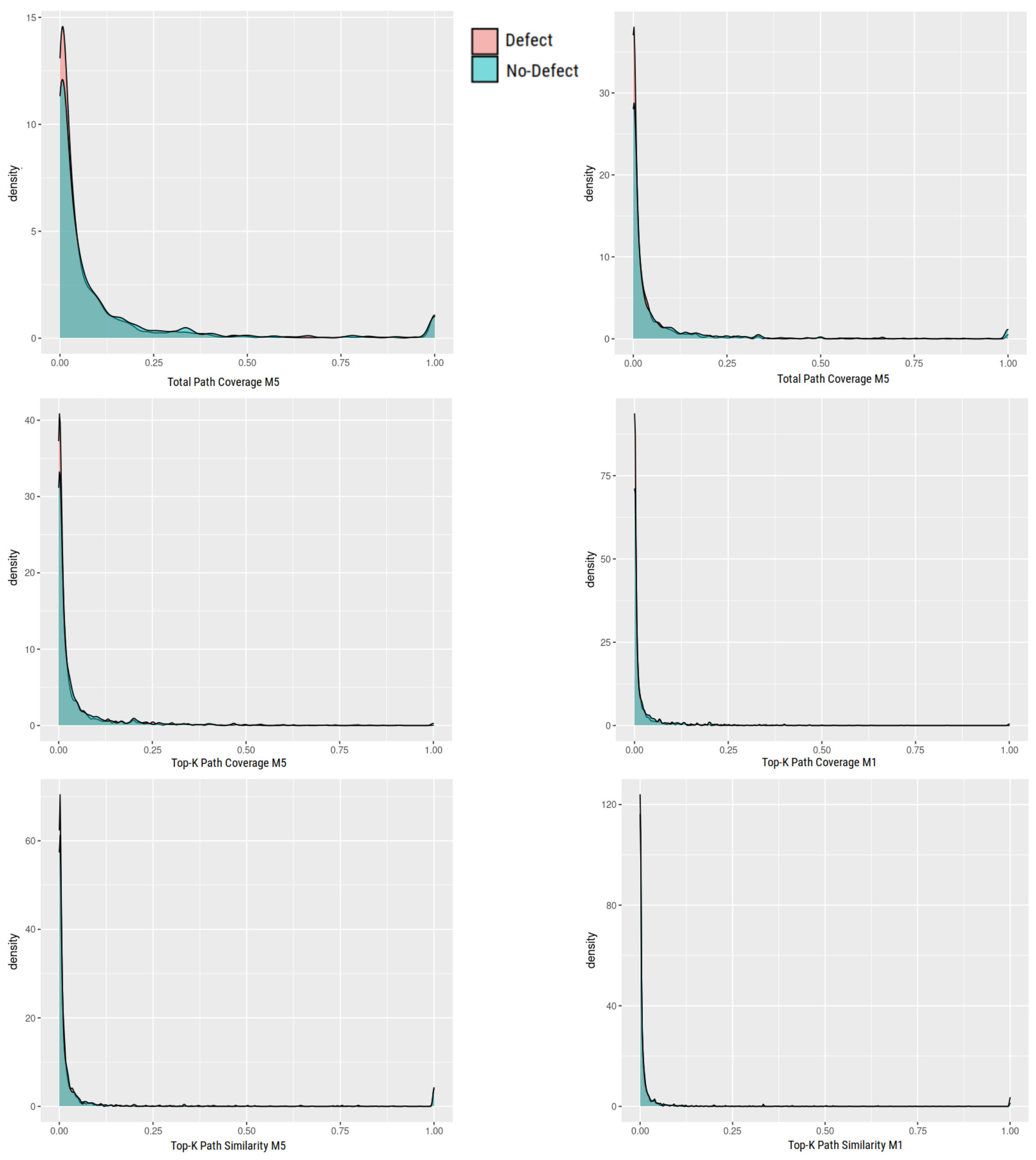


Figure 4.4: Path-based experience measure densities **(FNS)**.

a developer's experience. There is now a slightly higher level of defects at 0 as well. The other two path-based metrics have very low experience levels, making it difficult to see differences between defect and no-defect densities. As shown with the t-tests, the differences between means for the path-based metrics are very small.

The density plots for the tree-kernel-based metrics in Figure 4.5 look almost identical to their full modification data. The only noticeable change is that the peak at low levels for the similarity metric in row 1 is gone in the left column (M5).

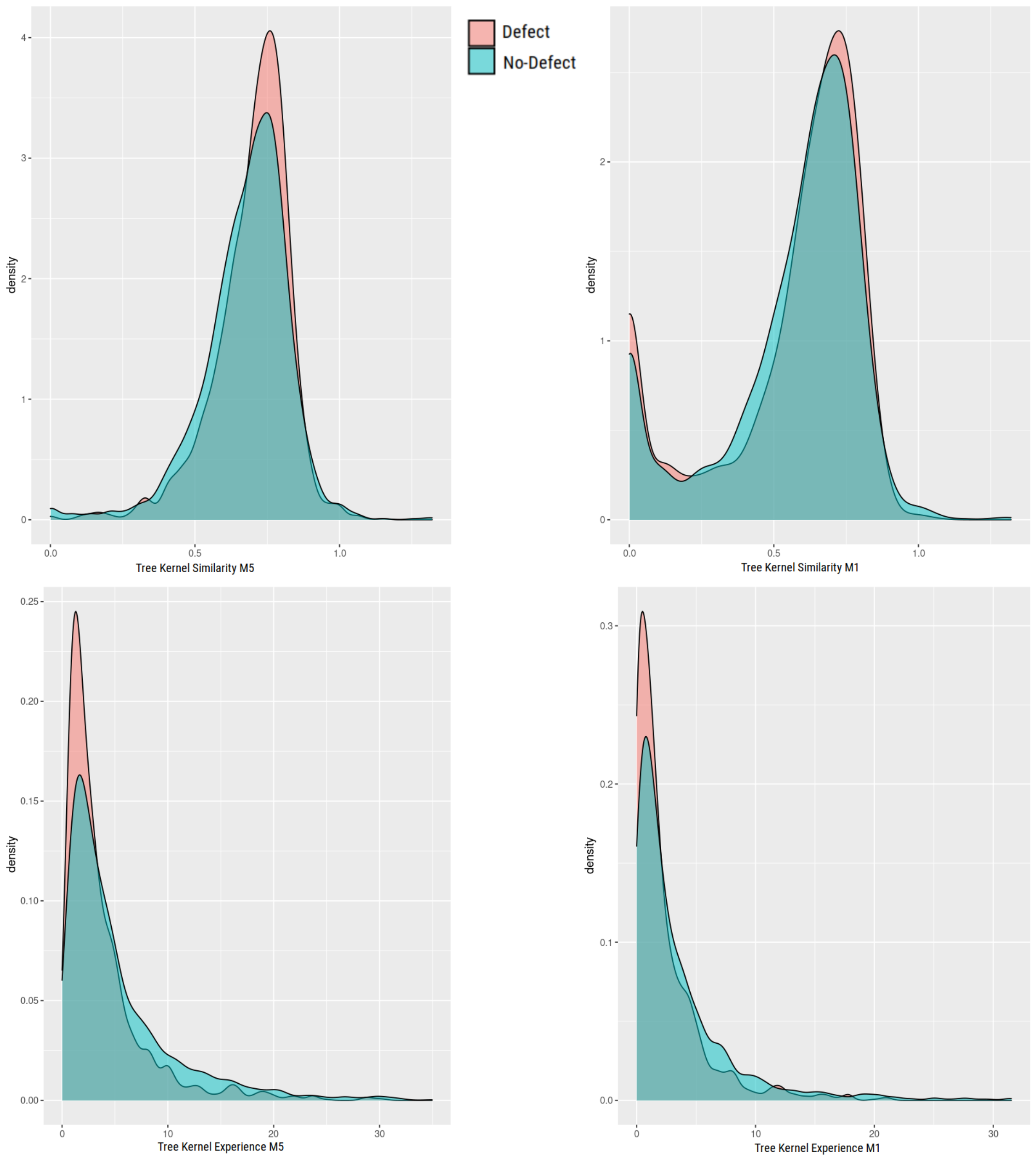


Figure 4.5: Tree-kernel-based experience measure densities **(FNS)**.

4.3.2 Statistical Modelling

Table 4.8 has the coefficients and p-values from logistic regression models built in the same manner as for Table 4.5. The changes for metrics in the individual models are similar

Table 4.8: Coefficients and p-values for individual AST-based metrics experience measures in logistic regression models for predicting defects for file-not-seen modification data.

Metric	Individual		With Controls	
	Coefficient	P-value	Coefficient	P-value
Total Path Cover M5	-0.237	0.105	0.117	0.293
Total Path Cover M1	-0.849	**1.05e-4**	-0.152	0.507
Top-K Path Cover M5	-0.721	**0.0169**	0.093	0.771
Top-K Path Cover M1	-2.015	**3.97e-5**	-0.573	0.227
Top-K Path Sim M5	-0.709	0.0881	-0.184	0.673
Top-K Path Sim M1	-2.776	**8.81e-4**	-1.281	0.081
Tree-Kernel Sim M5	1.170	**3.89e-7**	0.983	**3.38e-4**
Tree-Kernel Sim M1	-0.123	0.331	0.029	0.842
Tree-Kernel Exp M5	-0.067	$<$ **2e-16**	-0.035	**1.06e-4**
Tree-Kernel Exp M1	-0.078	$<$ **2e-16**	-0.030	**0.00357**

to what we saw from the t-test in Table 4.7. The path-based metrics that use coverage are no longer significant on their own. Unlike in Table 4.5, all path-based M1 metrics are now significant and have negative associations with defects. This result suggests that when there is no chance of changing the same method over and over in a short period of time, defects decrease as path-based experience increases. However, no path-based metric is statistically significant when added to the control model. So we have to be careful when interpreting this change.

The tree-kernel-based metrics remain significant, although the similarity metric is only significant for M5. The coefficients for *tree-kernel similarity M5* have increased. However, they are still positively associated with defects, so more experience increases the likelihood of causing defects or perhaps more defects increase the likelihood of experience. Given the swap in the direction of the associations for path-based metrics and the negative association of *tree-kernel experience*, this may suggest a flaw with *tree-kernel similarity* as an experience metric. The *tree-kernel experience* coefficients have retained their association direction, and more experience lowers the odds of introducing a defect. Their values are small, but the scale is larger than the other metrics. Given a roughly 10% increase in experience using the control model, a developer with a *Tree-Kernel Exp M5* value 3.5 higher than another developer would be $e^{3.5 \cdot 0.035} \approx 1.13$ times less likely to introduce a defect. This odds value is still unlikely to be practically significant when choosing developers for a task. However, the statistically significant relationship provides evidence that AST-based metrics can capture a more general programmer experience and knowledge than count-based metrics.

4.4 Summary

In this chapter, we used statistical analysis to explore our new AST-based metrics and their relationship to existing experience metrics and defects. The new AST-based metrics are similar to the existing count-based metrics but not so similar that they capture the same information. The *tree-kernel experience* metric was the least correlated with other metrics and had a negative relationship with defects, as we would expect from programming experience. This relationship held for both datasets. We saw the path-based metrics and even the finer-grained count-based metrics having positive relationships with defects, even when excluding churn, which was not expected. We will discuss the implications of this further in Chapter 6. Overall, there is evidence that our AST-based metrics capture statistically significant information on developer experience and knowledge. However, the direct relationship between these metrics to defects is not very strong.

Chapter 5

Defect Prediction

In this chapter, we present the results of machine learning and feature selection to answer our final three research questions:

- **R4**: How successful is defect prediction using AST-based metrics and controls?
- **R5**: Which metrics from the full set of collected metrics are most important for predicting method modification defects?
- **R6**: How do defect prediction results change when method modifications and additions are considered?

5.1 Predicting Method Modification Defects

Our fourth research question asks how well we can predict method modification defects. In this section, we build and evaluate several machine learning models to predict defects in method modifications. We still want to understand how our AST-based metrics perform, so we build models with only AST-based metrics and then add them to a model built with the same metrics as the control model in Chapter 4. Note that the control model contains the count-based experience metrics. Doing so investigates whether these AST-based experience metrics can be used to successfully predict method defects. Table 5.1 and Table 5.2 show these models' accuracy, precision, and recall.

As discussed in Chapter 2, accuracy is a good measure for identifying how many instances were correctly predicted. However, it does not give us a full picture of our success

Table 5.1: Accuracy, recall, and precision for models using AST-based experience and control metrics using the full modification dataset. The largest value in each column is in bold.

Measure	AST-based	Experience		Control	Metrics	
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
NB	0.574	0.572	**0.670**	0.604	0.654	0.443
LR	0.593	0.595	0.638	0.634	0.662	0.548
SVM	0.579	0.589	0.521	0.625	0.666	0.500
KS	0.577	0.572	0.615	0.743	0.728	**0.776**
RF	**0.620**	**0.614**	0.645	**0.773**	**0.775**	0.769

in predicting defects. Precision tells us how many of the defects we predicted were defects. A high precision means that we may have missed some defects, but if we predict that a change caused a defect, there is a high probability that it did. Recall tells us how many of the defects we successfully predicted. A high recall might mean false positives, but we missed very few defects.

Recall that the algorithms we used are Naive Bayes (**NB**), Logistic Regression (**LR**), Support Vector Machine (**SVM**), K-Star (**KS**) and Random Forest (**RF**).

As we might expect, Table 5.1 shows that the control model performs better than the AST-based experience model. It contains many metrics, some of which, such as lines of code and cyclomatic complexity, have been shown in previous research to be highly associated with defects [5, 6, 12, 29]. The Random Forest algorithm achieves the best result at approximately 77% accuracy with similar precision and recall. Only the Random Forest and K-Star algorithms using the control model achieve over 70% accuracy. These results seem to agree with the statistical analysis in Chapter 4 and indicate some relationship between the AST-based experience metrics and method modification defects. However, their predictive power is relatively low on their own, with a max accuracy of 62%.

Table 5.2 shows the results of building a model with the AST-based experience metrics and the control group. Again the Random Forest algorithm performs the best at 73% accuracy. However, the results are lower than the control model by itself in Table 5.1. So

Table 5.2: Accuracy, recall, and precision when combining AST-based experience and control metrics using the full modification dataset. The largest value for each column is in bold.

Measure	AST-based	Experience	w/ Controls
	Accuracy	Precision	Recall
NB	0.593	0.613	0.507
LR	0.642	0.655	0.600
SVM	0.621	0.653	0.515
KS	0.656	0.648	0.680
RF	**0.733**	**0.742**	**0.716**

adding the AST-based metrics reduces the predictive power of the control model. This result suggests that our AST-based metrics are not valuable additions to commit-time defect detection.

5.2 Using Full and Reduced Sets of Metrics

Our fifth research question asks about the most important metrics for predicting method modification defects. In Section 5.1, the AST-based metrics resulted in reduced accuracy for the control model. This section focuses on using all the metrics we collected, including those we did not use in the control model. We can find out how successful we can be at predicting commit-time defects and which metrics are the most useful for this type of defect detection.

The full set of metrics introduces some additional collected metrics that we did not include in previous experiments. These include metrics such as counting how many times other developers changed a file in the last six months or how many methods a developer modified in the current commit. A full list of metrics is included in Appendix A. These metrics are either not experience metrics or are not common predictors of defects in other work. However, they were easy to collect and may offer additional predictive power for commit-time defect prediction.

We use *InfoGain* and *RelieF* feature selection algorithms to create two sets of 10 metrics. These methods were briefly discussed in Chapter 2. Each algorithm ranks the full

Table 5.3: Accuracy, recall, and precision with the full set of metrics using the full modification dataset. The largest value for each column is in bold.

Measure	AST-based	Experience	w/ Controls
	Accuracy	Precision	Recall
NB	0.621	0.631	0.581
LR	0.672	0.667	0.684
SVM	0.656	0.655	0.661
KS	0.691	0.673	0.741
RF	**0.764**	**0.767**	**0.760**

set of metrics based on their expected predictive power. We use this to investigate which metrics may be the most useful for commit-time defect prediction and determine whether models built with the reduced sets can perform similarly to the full model.

Table 5.3 shows the machine learning results using the full set of metrics. These models perform roughly similar to the control models in Table 5.1, changing from 77% accuracy to 76% accuracy. It appears that with our data, adding more metrics does not improve accuracy.

Table 5.4 shows the top 10 metrics from the rankings of each feature selection algorithm. The top three metrics in both sets are from the additional metrics added at this stage. *Project Changes Others* and *Project Commits Others* measure the number of method modifications and commits made by other developers. *Changes This Commit* is the number of methods the developer modified in the same commit as the current method.

The *InfoGain* list focuses on file and project changes by the developer and others. It also includes metrics tracking methods and file sizes. The *RelieF* list focuses on project changes for other developers but total changes for the developer making the changes. It also focuses on file complexity and the number of lines added and deleted from the file during the commit. There are no AST-based experience metrics in the top ten for either algorithm, and most of the metrics chosen have to do with changes to the code rather than the structure of the source code.

Table 5.5 shows the result of building models with the top 10 feature sets given in Ta-

Table 5.4: Top 10 features ranked by *InfoGain* and *RelieF* feature selection algorithms for full modification data.

Rank	*InfoGain*	*RelieF*
1	Project Changes Others M6	Project Changes Others M6
2	Project Commits Others M6	Changes This Commit
3	Project Changes Others M1	Project Commits Others M6
4	File Lines of Code Log	File Additions This Commit Log
5	File Lines of Code	Project Commits Others M1
6	Developer Project Changes M5	File Complexity Log
7	Method Lines Of Code	Project Changes Others M1
8	Method Lines Of Code Log	File Deletions This Commit Log
9	Project Commits Others M1	Developer Total Changes M6
10	Developer File Changes M5	Developer Total Changes M5

ble 5.4. The metric subset generated by *RelieF* has the highest accuracy out of all the results, though the differences are relatively small and may not be statistically significant. The Random Forest algorithm achieved approximately 80% accuracy and a similar precision and recall. This result is an improvement of approximately 3% from the control model. Again, only K-Star and Random Forest see accuracy above 70%; in this case, K-Star also has an increase in accuracy by approximately 3%. The results from the *InfoGain* metrics selection also show increases in accuracy over the control model, but not as much as *RelieF*. The K-Star and Random Forest algorithms increase by approximately 1.5% over the control model. These results confirm that for commit-time defect detection on our data, a small selection of metrics can be as effective or more effective and that achieving a reasonable 80%

Table 5.5: Accuracy, recall, and precision using top 10 metrics as selected by *InfoGain* and *RelieF* using the full modification dataset. The largest value for each column is in bold.

Measure	*InfoGain*			*RelieF*		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
NB	0.622	0.678	0.646	0.573	0.549	0.823
LR	0.635	0.631	0.650	0.600	0.596	0.623
SVM	0.644	0.634	0.681	0.631	0.621	0.671
KS	0.758	0.747	0.750	0.778	0.752	**0.830**
RF	**0.785**	**0.782**	**0.792**	**0.802**	**0.800**	0.806

prediction accuracy is possible. These results are promising, and the differences between the selected and traditionally successful features in defect detection are interesting.

5.3 Predicting Defects with Method Modifications and Additions

Our final research question asks how prediction performance will change if we include changes where a developer adds a new method and those where they modify an existing method. Previous results allow us to examine a developer's experience and knowledge of an existing piece of code that must be modified. However, for a software development project, developers add new code and modify existing code. To properly evaluate the application of method-level defect prediction to software development, we must include both types of changes. We still exclude changes to a project that we expect to represent moving existing code in a project. However, some additions may have come from code being extracted from larger methods to make multiple smaller methods. These types of changes represent a smaller proportion of modifications made to a project, but it is worth noting that they may be present in the data.

We saw that the highest-performing models in Section 5.2 did not contain our new AST-based metrics. For this reason, we do not repeat the analysis with only AST-based metrics and the control model. We instead test the full set of metrics and use feature selection to find the top 10 best metrics for the new data.

Table 5.6: Accuracy, recall, and precision with the full set of metrics using a dataset that includes both method additions and modifications. The largest value for each column is in bold.

Measure	AST-based	Experience	w/ Controls
	Accuracy	Precision	Recall
NB	0.640	0.657	0.587
LR	0.675	0.670	0.691
SVM	0.664	0.650	0.709
KS	0.704	0.684	0.757
RF	**0.768**	**0.763**	**0.777**

Table 5.7: Top 10 features ranked by *InfoGain* and *RelieF* feature selection algorithms using modification and addition data.

Rank	*InfoGain*	*RelieF*
1	Project Changes Others M6	Changes This Commit
2	Method Lines Of Code After	Project Changes Others M6
3	Method Lines Of Code After Log	Project Commits Others M6
4	Method Tokens After	File Complexity After Log
5	Method Tokens After Log	File Additions This Commit Log
6	File Lines Of Code Log	Project Commits Others M1
7	File Lines Of Code	Project Changes Others M1
8	Project Commits Others M6	File Deletions This Commit Log
9	Method Complexity After	Developer Total Changes M6
10	Method Complexity After Log	File Lines Of Code After Log

Table 5.6 shows the machine learning results using the full set of metrics. These models perform roughly the same as the full models in Table 5.3. Incorporating the method addition data has not altered the ability of the models to predict defects in any significant way. This result is somewhat interesting given that there is over 150% more data, and adding new code is quite different than modifying existing code. Note that in this case, any measurements of the source code can only be taken after the change has been made, as additions do not have code before the change.

Table 5.7 shows the two metric subsets selected with the *InfoGain* and *RelieF* algorithms. The *RelieF* set is almost identical to the set in Table 5.4. The rank of the metrics has changed slightly, but otherwise, *Developer Total Changes M5* has been replaced by *File Lines Of Code After Log*. The *InfoGain* set has changed more than the *RelieF* set. All but two metrics are now method and file source code metrics that track complexity, token counts, and lines of code. Again, note that the metrics explicitly add *After* to indicate that they are from the code after the method was added or modified.

Table 5.8 shows the result of building models with the top 10 feature sets given in Table 5.7. Again, the metric subset generated by *RelieF* has the highest accuracy at approximately 80% accuracy and a similar precision and recall. There is a negligible im-

Table 5.8: Accuracy, recall, and precision using top 10 metrics selected by *InfoGain* and *RelieF* using the dataset with both method additions and modifications. The largest value for each column is in bold.

Measure	*InfoGain*			*RelieF*		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
NB	0.639	0.671	0.545	0.602	0.579	0.744
LR	0.650	0.642	0.680	0.613	0.608	0.634
SVM	0.656	0.635	0.732	0.634	0.620	0.690
KS	0.721	0.708	0.752	0.783	0.755	**0.838**
RF	**0.749**	**0.739**	**0.769**	**0.808**	**0.807**	0.811

provement over the model in Table 5.5. The results from the *InfoGain* metrics selection are a drop in accuracy by 3-4% compared to those in Table 5.5. These results indicate that even though most of our work focuses on method modifications, our commit-time defect detection methodology still applies to the general software development process.

5.4 Summary

In this chapter, we saw evidence that suggests predicting defects for modified or added methods is possible with reasonable accuracy. It is also possible, in a general software development context, to achieve fairly high accuracy and precision when predicting method-level defects. These results are promising and are close to the average results achieved by other defect detection studies using similar data [31], although still lower than the higher accuracy models for release-time defect detection that can achieve over 90% accuracy [29].

We learned that our new AST-based experience metrics are less important for defect prediction than other metrics. We saw that the AST-based metrics are capable of some small prediction accuracy by themselves, but when included in control models, they lowered the accuracy of those models. However, some of the additional features we collected that are less commonly used in previous works were among the most predictive metrics. Change-based and file-based software metrics were ranked the highest with metric selection algorithms and were used in the models with the highest accuracy.

Chapter 6

Discussion

We answered our six research questions in Chapter 4 and Chapter 5. Here we discuss the results and their implications for AST-based experience and commit time defect detection. First, we discuss what we learned about experience metrics. This discussion focuses primarily on our AST-based metrics and how well they capture more general programming knowledge. However, we also explore other issues, such as higher experience indicating more defects. Finally, we discuss what we learned about commit-time method-level defect detection. We saw good results that are comparable to other works.

6.1 Experience Metrics

The results of our analysis were mixed. Our new AST-based metrics did not correlate highly with other count-based experience metrics. All long-term (M5) metrics had statistically significant relationships to defects when including all method modifications. Tree-kernel-based metrics maintained this statistical significance when developers had no direct experience with a method they were modifying. However, there were some unexpected relationships. All but one experience metric, including count-based metrics, were positively associated with defects meaning that a higher likelihood of defects is associated with higher levels of experience. This is unexpected because previous work with defect detection has found that more count-based experience reduces the chances of defects [5]. The direction of these relationships changed for some metrics when using file-not-seen data, but path-based metrics were no longer significant when added to the control model. Below we discuss the implications of these results for AST-based metrics and the general association of experience and defects.

6.1.1 Path-Based Metrics

The most important result for path-based metrics is that they were not significant when added to controls in the file-not-seen models. This result means the experience they capture is highly specific to the code they have seen and not the general underlying structure of that code. When discussing metric construction in Section 3.5, we said that one of the weaknesses of path contexts was that they start and end with terminal nodes. The difficulty with this representation is that two paths can represent an identical sub-structure in an AST but be different because they used different identifier names. This result means that we are far less likely to see paths that match between two methods unless those methods are some variation of the same method. Because of this, path-based metrics are more likely to mirror count-based metrics for method changes.

When compared to count-based metrics, path-based metrics will capture a more nuanced picture of the code that a developer has worked on in the past. Our analysis confirmed that the path-based metrics are not highly correlated with count-based metrics, though *top-k path similarity* was moderately correlated to method changes. However, the fact that path-based metrics are not applicable when method and file change counts are also not applicable suggests they offer little improvement over those count-based metrics. Figure 4.2 and Figure 4.4 show that for each dataset, experience levels are generally low for the path-based metrics, especially for the file-not-seen data. So as we used them, path contexts are not the most promising way to represent an AST when measuring programming experience.

The value of using path contexts is that it is easy to separate individual components of a method and compare them, such as with set similarity measures. An adjustment to remove terminal nodes from path contexts might have been able to represent the AST more generally. It may also be possible to include the terminal node pairs in some way that does not interfere with the general AST comparison. The tree-kernel implementation considers terminal nodes separately, so there is precedence for doing this. Another option may be to use some kind of type information for terminal nodes. This would make the terminal nodes more generic while still allowing differences to be compared.

6.1.2 Tree-Kernel-Based Metrics

The most important result for tree-kernel-based metrics is that they remain significant in all our statistical analyses. They are still statistically significant in the file-not-seen dataset, suggesting they capture a more general programming experience and knowledge. They were not highly correlated to count-based metrics, and the *tree-kernel experience* metric had a very weak correlation to all other experience metrics. The tree-kernel similarity also seems to do better than path contexts in capturing meaningful information from the AST. Further work can be done to investigate using experience and knowledge metrics using tree kernels.

While results were promising, some things could be improved with our tree-kernel-based metrics. One issue is that even though the results were statistically significant, the practical significance is questionable. With the control metrics, a moderate increase in experience only gives a small change in the likelihood that a developer will introduce a defect. Also, the two metrics we created do not agree on whether more experience will make a developer more or less likely to introduce a defect. So, making decisions about developers based on this type of AST-based experience alone is impractical.

Another area for improvement is the difficulty in using tree-kernel numbers as experience when the sizes of methods greatly affect the size of similarity numbers. We chose two ways to adjust the similarity numbers so that similarity numbers could be compared across methods of different sizes. The first was taken from tree-kernel literature [8] to normalize the values. A normalization function should result in all values being between 0-1; however, our results were not entirely in this range. The reason for this was unclear and will require further investigation to determine what AST comparisons caused the higher values.

The second adjustment was an attempt to measure the similarity as experience based on the size of only the current method being changed. This metric was statistically significant in all cases and matched our expectation of how programming experience should relate to defects. However, given that it differs from all the other experience metrics, we must be careful how we interpret this success. Is this metric a good measure of general programming experience and knowledge? Or does it only seem that way because it matches our expectations? More work will need to be done to confirm whether this representation is correct or if it can be improved upon. Tree-kernel metrics seem more promising than path-based metrics, but work still needs to be done.

6.1.3 Experience and Defects

The most unexpected and interesting relationship we saw in our analysis was that higher experience levels correlated to higher defects. We expect developers with more experience with a method, file, module, or project to be less likely to introduce defects when making a change. This result is especially surprising given that other works have found significant relationships showing high experience leads to fewer defects [5, 25]. The differences may come from our dataset or our experience collection methodology. Future work must be done to find the cause of the differences. The other place we see this relationship between experience and defects is churn. The expectation is that the more a developer works on a specific piece of code, the more likely it is to have defects. The explanation is that the need to work on the code repeatedly is because it is low-quality code with issues. So even though the developer has experience with the code, successfully changing it is not easy.

Our results suggest that even over a longer period of time, and excluding the most recent changes generally considered by churn, finer-grained count-based experience metrics may still indicate some of the code issues we would expect with high churn. This result may also apply to path-based metrics as well given that we have indicated they are still highly connected to the specific methods developers have changed in the past. We did see that when there was no chance of churn, the available experience methods had the expected relationship to defects; more experience means fewer defects. However, when added to the control model, most of these experience metrics were no longer significant. So the correlations are there, but they are not associated with defects when we control for metrics such as code size and complexity.

Another consideration is that we cannot guarantee that developers with low experience levels are inexperienced. They may have experience with other companies and personal projects. In addition, new project contributors are less likely to be assigned to more complex problems or additions to a project. In these cases, high experience might cause more defects. If a developer is new to the project but is an experienced programmer, they may work on assignments that are very easy to complete without defects. These are only a few possible scenarios, but they give some of the reasons that measured experience may not match up with our expectations.

The relationship between experience and defects is complex, and it is difficult to measure effectively. There may be different types of experience to consider. A count-based experience is useful for a project when trying to find developers with knowledge of the purpose of a module or its general connection with other parts of a project [1, 24]. There is also a use for this coarse-grained count-based experience when a larger project comprises small projects, and developers might work mostly on one sub-project and occasionally contribute to other sub-projects when necessary [5]. The finer-grained experience is better for finding if a piece of code is causing problems, even over a larger period of time, rather than being used to identify developers with knowledge of that code. These types of experience are also limited to a single code base, as the experience with one method, file, module, or project cannot be transferred to another project.

A more general programming experience is useful for understanding a developer's programming knowledge or whether that knowledge is relevant to a piece of code. However, this type of knowledge and experience is much more challenging to measure. We have discussed some of the challenges in creating our AST-based metrics, but there is also the challenge of gathering enough information to represent general programming knowledge. In research and industry, there is often a single project or collection of sub-projects of interest. It is challenging to collect experience information outside of this context. Professional developers may have worked on proprietary projects in the past. Their previous work is unavailable, and even if it is, accessing all of that experience will often be difficult.

6.2 Method-Level Commit-Time Defect Detection

The most important prediction result is that the AST-based metrics we created were absent in the most predictive models. They even lowered the accuracy of the control model when they were included. While Section 6.1 presents that these metrics have promise as a representation of programming knowledge and experience, the prediction results are better without them. In a practical context, it is not worth the extra effort of collecting AST data and the time required to convert it into experience metrics. Most metrics that were present in the most accurate models are simple to collect from a project's Git history. However, as this is an early attempt to use ASTs to capture programming knowledge and experience, more work may find better variations of these metrics that are significant predictors of defects.

Our overall defect prediction results were reasonably high and comparable to other works that have performed just-in-time defect detection. Our best results achieve just over 80% accuracy, precision, and recall. Kamei *et al.* performed just-in-time defect prediction with logistic regression models and achieved an average of 69-71% accuracy [16]. Sahar *et al.* used tree-kernels to perform just-in-time defect detection using projects from the Apache Software Foundation [31]. They did their defect detection by project rather than combining all of the projects, and their results ranged from 60-90%. From Pandety *et al.*, we see results for release-time defect prediction where the best performing models are above 90% accuracy[29]. However, as previously mentioned, release-time defect detection is more difficult, so our 80% accuracy is appropriate for just-in-time defect detection.

We built our final model from a data set that included method addition and modification data. The success of our models shows that it is possible to add this just-in-time defect detection to the software development process. Data collection could be even more precise for a single project with more information available from changes. For example, if automatic refactoring tools make a change, they can be guaranteed to be excluded, and no changes that a developer makes will be excluded. This situation may lead to even higher accuracy in practice. Defect detection at commit time could help to catch more defects early in the process and make it easier to know where changes need to be made. Even with false positives, because the ratio of defect to no-defect changes is low, there will not be many changes that will need to be re-checked for no reason [16].

Chapter 7

Conclusion

Accurately representing a developer's programming knowledge and experience is difficult. Traditional metrics rely on counting the number of times a developer has used or made changes to pieces of code. When a developer has modified a file in the past they are less likely to introduce defects with a change. However, these metrics do not contain any general information on the structure or purpose of a piece of code and are only useful when developers work on a piece of code more than once. We investigated the use of several new metrics based on abstract syntax trees (ASTs) as a possible way to more accurately measure a developer's experience.

In order to determine the effectiveness of these new metrics we answered six research questions.

1. **How do AST-based experience metrics and count-based experience metrics relate to one another?** AST-based metrics that use path-contexts are more highly correlated to count-based metrics than those that use tree kernel methods. However, all AST-based metrics are weakly correlated with count-based metrics. This means that our AST-based metrics capture a different type of experience information than count-based metrics.

2. **Do AST-based metrics have a statistically significant relationship with method modification defects?** We found that AST-based metrics do have a statistically significant relationship with defects even when controlling for other common defect prediction metrics. However, the nature of these relationships showed that for all but one

of the new metrics higher experience levels were associated with a higher probability of defects. Given that some of the count-based metrics also showed this relationship we expect that this result is due either to our specific data or our approach of predicting defects for individual method changes rather than for files at release time.

3. **When count-based experience is not applicable, do AST-based experience metrics have a statistically significant relationship to defects?** When controlling for other common defect prediction metrics only one of the AST-based metrics maintained statistical significance. This provides evidence that an AST-based approach can capture a more general programming knowledge and experience.

4. **How successful is defect prediction using AST-based metrics and controls?** We found that on their own, our AST-based metrics were able to predict defects in method modifications with approximately 60% accuracy. This is lower than models using common metrics without including AST-based metrics which achieved around 70% accuracy. When adding the AST-based metrics to the other models the accuracy declined.

5. **Which metrics from our full set of collected metrics are most important for predicting method modification defects?** Using metric selection algorithms the highest ranked metrics were those that tracked how much a project was changing over time, the complexity of the code being changed, the nature of the changes, and a developers total change experience. Random Forest models built with these metrics achieved our highest accuracy at approximately 80%. This accuracy is in line with other works for just-in-time method-level defect prediction.

6. **How do defect prediction results change when method modifications and additions are considered?** Using both method modification and addition data our results were unchanged.

Overall, this work found that AST-based metrics have statistically significant relationships to defects and that they can be used to represent general programming experience and knowledge. These results are promising for the use of ASTs in measuring and comparing developer knowledge and experience, however, AST-based metrics require more work to be done in order to tailor them to defect-prediction or to use them in other contexts. Our just-in-time method-level defect prediction was successful, but the most successful models were not built using our AST-based metrics. The most predictive metrics are easy to collect and well-researched. More work could be done to produce more accurate results for method-level defect detection, or as the more predictive metrics are not method-based, using files or full changes instead might be a better approach.

7.1 Future Work

In this section, we provide some ideas for future work. Our mixed results leave some room for improvement when working with AST-based metrics and commit-time defect detection. Some simple adjustments and directions could give additional insight or improve our approaches. Some of the success of using ASTs for experience and knowledge could also be applied to other contexts.

There are some simple steps to expand or improve our work:

- Experiment with different experience windows. We investigated the use of a developer's experience from over six months. It is possible that developer experience is relevant for longer than this or is not relevant for this long. We also partitioned experience at the most recent month into a one-month and five-month groups. Other groups could be tested, such as two periods of three months each or three periods of two months each.

- Experiment with different k values for averaged metrics. We only showed results for $k = 5$, but we tried $k = 1, 10$ as well with similar results. Different values may be more appropriate for specific metrics, projects, or datasets.

- Experiment with coarser-grained commit-time defect detection. Given that the most successful machine learning models used coarser-grained metrics, predicting defects at the file or commit level may be better. This approach could still offer some value over waiting until release time, though it would sacrifice some specificity for narrowing down the defect's location.

- Experimenting with other machine learning techniques. We chose five common machine learning algorithms from different categories. Other algorithms may perform better, particularly in the same category as Random Forest which was our highest performing algorithm across all of our models. Similar experiments could also be performed with Neural Networks.

- Experiment with different feature selection methods. We chose two of the more common feature selection algorithms, but others might fit the data better. Some feature selection techniques use a specific machine learning algorithm to rank features. The Random Forest approach could be used to select optimal features for the algorithm and our data and potentially offer higher performance.

- Experiment with model parameters. We used the default settings in *WEKA* for each of the machine learning algorithms we used. We also used the default of $k = 10$ for cross validation. There may be tweaks that could be made to the higher-performing algorithms to achieve better performance or more accurate evaluation of the models.

In addition to looking for specific improvements to our methods and data, other steps could be taken to address some of the limitations of our work. Other applications for AST-based techniques and metrics exist outside of defect detection.

We combined many projects to create our datasets; however, these projects all come from the same organizations. Our results may not generalize to other projects or software development in general. New experiments could be done on other projects, perhaps combining projects from different organizations. The experience we gather is also very project specific. We worked to mitigate this by choosing projects from a larger organization with the expectation that we could capture more of a developer's experience over time if they worked on multiple projects. Even in this situation, only some developers worked on multiple projects. Since we are trying to capture a more general experience and knowledge, tracking all of a developer's work is best if possible. It may be possible with tools like GitHub to gather more experience data on each author from other projects outside a single organization or from a developer's personal public projects. There will always be limitations to gathering all of a developer's experience. However, many of the low-experience developers in our dataset might have large amounts of programming experience elsewhere that we were not able to account for.

Similarly, when focusing on projects from a single organization, many developers may have high general programming experience and low project experience. If gathering all the programming experience for each developer is impossible, the data could be partitioned by metrics like *project changes*. There may be different relationships between experience and defects when we look at developers with a large amount of project experience and those without. Regarding commit-time defect detection, there could be different strategies for predicting defects based on how much experience data is available for a developer.

Some experiments in a more practical context would also provide valuable insight into commit-time defect detection. Work could be done to integrate defect detection into a project and see how developers react. Does this kind of defect prediction make a code reviewer's job easier? If method changes are identified as having a high likelihood of introducing a defect, can the reviewer and the developer identify the necessary fixes? Do false positives waste developers' time or cause them to second guess decisions and introduce mistakes trying to fix non-existent defects?

The evidence that ASTs can be used to represent experience and more general programming knowledge could also be explored more deeply. Experiments more focused on specific knowledge could be conducted in a more controlled setting such as giving developers or students code samples and then asking them to modify code with similar and dissimilar styles, algorithms, or data structures.

Finally, AST-based methods could be applied to contexts where knowledge or code similarity is useful. It may be possible to use AST similarity to identify a piece of code's author(s). It may also be possible to use ASTs to measure similarities of code written by different groups of developers. Like defect prediction, these research areas have succeeded with simple metrics like keywords, code length and code complexity [3, 28, 30]. The addition of AST-based metrics could improve these methods or identify new similarities.

7.2 Threats to Validity

Collecting our data and performing our analysis, we had to make many choices. In this section we mention how these choices might have affected our results.

We gathered data from 27 projects, all maintained by the Apache Software Foundation. Even with a large amount of data, we cannot say these projects represent all software projects. The results of our analysis may not generalize to all software projects.

There were over 500 different developers in our data. There is always a large difference between developers contributing a few commits to a project and developers contributing a majority of the commits. This situation can be especially difficult when considering experience, which will not accurately represent a developer who may work on other code regularly, but only contribute to a project in our data once.

Due to the computationally complex nature of some of our AST methods, we had to make some choices about the code we included. We also recognize that not all changes to the code are equal. Sometimes developers use tools to refactor large pieces of the code automatically or move code around in ways we cannot detect easily. We ignored commits that changed more than 50 methods at once to mitigate large changes that tools may do. We also excluded methods with more than 100 lines of code to reduce the computational complexity. These excluded commits and changes could alter our results in a significant way.

We chose a six-month experience window for our experience measures. However, it is unclear how long experience should be relevant. Longer experience windows may help reduce the effect of churn on experience measures. Shorter windows may more accurately represent the amount of time a developer's experience with a piece of code is relevant to their understanding. Some weighting based on a change's age could allow for even more nuance.

The defect labelling came from an external dataset. Labelling defect-inducing and defect-fixing commits relies on an algorithm that is not guaranteed to be 100% accurate. This will affect our defect labels and could skew our results. Similarly, we used commits to label methods as inducing defects. We expect that a change made to a method to fix a defect means that changes to that method introduced the defect. However, this may not always be the case and may result in methods being labelled incorrectly or defects being missed.

Some of our statistical analysis, specifically t-tests, assumes that each data point is independent of other data points. However, given that each change to a piece of code will rely on previous changes, this cannot be guaranteed. The churn metric is a good example of this, where subsequent changes might have defects because the previous changes were ineffective. We can still use these statistical techniques, but the results must be reviewed with this in mind.

Bibliography

[1] Sharmin Akter. Recomending expert developers using usage and implementation experience. *University of Lethbridge*, 2014.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019.

[3] Parinaz Bayrami. Code authorship attribution using content-based and non-content based features. *University of Lethbridge*, 2018.

[4] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Looking for bugs in all the right places. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 61–72, New York, NY, USA, 2006. ACM.

[5] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 4–14, New York, NY, USA, 2011. ACM.

[6] Maria Caulo. A taxonomy of metrics for software fault prediction. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1144–1147, New York, NY, USA, 2019. ACM.

[7] John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In *12th International Conference on Machine Learning*, pages 108–114, 1995.

[8] Michael Collins and Nigel Duffy. Convolution kernels for natural language. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[9] Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, ICSE '21, page 995–1007. IEEE Press, 2021.

[10] David M. Diez, Christopher D. Barr, and Mine Cetinkaya-Rundel. *OpenIntro Statistics*. OpenIntro, 4th edition, 2019.

[11] Bradley Efron and Trevor Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press, 2016.

[12] Todd L. Graves, Alan F. Karr, U. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26, 2000.

[13] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[14] John Hancock. *Jaccard Distance (Jaccard Index, Jaccard Similarity Coefficient)*. 10 2004.

[15] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.

[16] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[17] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Building implicit vector representations of individual coding style. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 117–124, New York, NY, USA, 2020. ACM.

[18] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, sep 2019.

[19] Huan Liu and Hiroshi Motoda. *Computational Methods of Feature Selection (Chapman and Hall/Crc Data Mining and Knowledge Discovery Series)*. Chapman and Hall/CRC, 2007.

[20] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, page 11–19. IEEE Press, 2017.

[21] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[22] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 13–23, New York, NY, USA, 2008. ACM.

[23] Andrew Meneely and Oluyinka Williams. Interactive churn metrics: Socio-technical variants of code churn. *SIGSOFT Softw. Eng. Notes*, 37(6):1–6, nov 2012.

[24] Audris Mockus and James D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 503–512, New York, NY, USA, 2002. ACM.

[25] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[26] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287, 2019.

[27] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005.

[28] Fariha Naz. Do sociolinguistic variations exist in programming? *University of Lethbridge*, 2015.

[29] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications*, 172:114595, 2021.

[30] Mahmudul Hasan Rafee. Computer program categorization with machine learning. *University of Lethbridge*, 2017.

[31] Hareem Sahar, Yuxin Liu, Abram Hindle, and Denilson Barbosa. An empirical evaluation of the usefulness of tree kernels for commit-time defect detection in large software systems. *arXiv*, 2021.

[32] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, page 121–124, New York, NY, USA, 2008. ACM.

[33] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010.

[34] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories*. 2018.

[35] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 91–100, New York, NY, USA, 2009. ACM.

Appendix A

Full List of Metrics

Below is a full list of metrics used in this work. The boldface name is the readable name. It is followed by a variable name present in model summaries and tables. Finally, there will be a short description explaining the metric.

Many metrics have several variations. These are indicated by `Before`, `After`, `M6`, `M5`, `M1`, `K10`, `K5`, `K1`. Where possible we will condense these for a single metric indicating the variations with square brackets: [Before/After] or M[6/5/1]. Some metrics also have logarithmic variants. These will be indicated with a `[.Log]` following the metric.

For example, `Method.LinesOfCode.[Before/After][.Log]` represents the following four metrics:

- `Method.LinesOfCode.Before`
- `Method.LinesOfCode.After`
- `Method.LinesOfCode.Before.Log`
- `Method.LinesOfCode.After.Log`

A.1 Method Metrics

- **Method Lines Of Code**
  `Method.LinesOfCode.[Before/After][.Log]`
  - The number of lines of code in the modified/added method. Additional before/after metrics for the logarithm of the value.
- **Method Cyclomatic Complexity**
  `Method.Complexity.[Before/After][.Log]`
  - The cyclomatic complexity of the modified/added method. Additional before/after metrics for the logarithm of the value.
- **Method Number Of Parameters [Before/After]**
  `Method.ParamCount.[Before/After]`
  - The number of parameters the modified/added method takes.
- **Method Number Of Tokens [Before/After] [Log]**
  `Method.TokenCount.[Before/After][.Log]`
  - The number of tokens in the modified/added method. Additional before/after metrics for the logarithm of the value.

- **Method Change Type**
  `Method.ChangeType`
  - Whether the method was modified or added.

- **Method Churn [Developer/Others]**
  `Method.Churn.[Developer/Others]`
  - The number of times the developer/other developers changed a modified method over the previous month.

- **Method Unique Authors**
  `Method.UniqueAuthors`
  - The number of different developers that changed the modified method over the previous six months.

- **Method Unique Authors Churn**
  `Method.Churn.UniqueAuthors`
  - The number of different developers that changed the modified method over the previous month.

- **Method Change Type**
  `Method.ChangeType`
  - Whether the method was added or modified.

A.2 File Metrics

- **File Lines Of Code [Log]**
  `File.LinesOfCode[.Log]`
  - The number of lines of code in the file containing the modified/added method. Additional metric for the logarithm of the value.

- **File Cyclomatic Complexity [Log]**
  `File.Complexity[.Log]`
  - The cyclomatic complexity of the file containing the modified/added method. Additional metric for the logarithm of the value.

- **File [Additions/Deletions] This Commit [Log]**
  `File.Commit.[Additions/Deletions][.Log]`
  - The number of lines added and deleted during the commit the method was changed. Additional metric for the logarithm of the value.

- **File Commit Change Type**
  `File.Commit.ChangeType`
  - Whether the file was added or modified.

- **File Churn [Developer/Others] [Log]**
  `File.Churn.[Developer/Others][.Log]`
  - The number of times the developer/other developers modified/added a method in

the file of the changed method over the previous month. Additional metric for the logarithm of the value.

- **File Churn [Additions/Deletions] [Log]**
  `File.Churn.[Additions/Deletions][.Log]`
  - The number of lines added or deleted in the file of the changed method over the previous month. Additional metric for the logarithm of the value.

- **File Unique Authors**
  `File.UniqueAuthors`
  - The number of different developers that changed the modified method over the previous six months.

- **File Unique Authors Churn**
  `File.Churn.UniqueAuthors`
  - The number of different developers that changed the modified method over the previous month.

- **File Change Type**
  `File.Commit.ChangeType`
  - Whether the file was added or modified.

A.3 Project/Total Metrics

- **Project Churn [Developer/Others]**
  `Project.Churn.[Developer/Others]`
  - The number of times the developer/other developers modified/added a method in the project of the changed method over the previous month.

- **Project Churn [Developer/Others] Commits**
  `Project.Churn.[Developer/Others].Commits`
  - The number of commits the developer/other developers in the project involving the changed method over the previous month.

- **Total Changes Made This Commit**
  `Commit.TotalChanges`
  - The total number of methods that the developer changed in the commit for the modified/added method.

- **Total Churn Developer**
  `Total.Churn.Developer`
  - The number of times the developer modified/added a method in any tracked project over the previous month.

A.4 Developer Experience Metrics

- **Developer Method Changes Excluding Churn**
  `Developer.Method.Changes.ExcludeChurn`

- The number of times the developer changed the added/modified method in the previous six months, excluding changes made in the most recent month.

- **Developer File Changes Excluding Churn**
  `Developer.File.Changes.ExcludeChurn`
  - The number of times the developer changed a method in the file containing the added/modified method in the previous six months, excluding changes made in the most recent month.

- **Developer Project Changes Excluding Churn**
  `Developer.Project.Changes.ExludeChurn`
  - The number of times the developer changed a method in the project containing the added/modified method in the previous six months, excluding changes made in the most recent month.

- **Developer Total Changes Excluding Churn**
  `Developer.Total.Changes.ExcludeChurn`
  - The number of times the developer changed a method in the previous six months, excluding changes made in the most recent month.

- **Developer Method Changes**
  `Developer.Method.Changes`
  - The number of times the developer changed the added/modified method in the previous six months.

- **Developer File Changes**
  `Developer.File.Changes`
  - The number of times the developer changed a method in the file containing the added/modified method in the previous six months.

- **Developer Project Changes**
  `Developer.Project.Changes`
  - The number of times the developer changed a method in the project containing the added/modified method in the previous six months.

- **Developer Total Changes**
  `Developer.Total.Changes`
  - The number of times the developer changed a method in the previous six months.

- **Developer Method Commits**
  `Developer.Method.Commits`
  - The number of commits where the developer changed the added/modified method in the previous six months.

- **Developer File Commits**
  `Developer.File.Commits`
  - The number of commits where the developer changed a method in the file containing the added/modified method in the previous six months.

- **Developer Project Commits**
  `Developer.Project.Commits`
  - The number of commits where the developer changed a method in the project containing the added/modified method in the previous six months.

- **Developer Total Commits**
  `Developer.Total.Commits`
  - The number of commits where the developer changed a method in the previous six months.

- **Developer Path Coverage**
  `Paths.Coverage.Total.[M1/M5/M6].[Before/After]`
  - The coverage measure comparing the path-contexts in the modified/added method to the set of path-contexts in all methods added/modified by the developer in the previous month (M1), six months (M6), or six months excluding the previous month (M5).

- **Developer Path Coverage Each**
  `Paths.Coverage.Each.[M1/M5/M6].[Before/After]`
  - The coverage measure comparing the path-contexts in the modified/added method to the set of path-contexts for each individual method added/modified by the developer in the previous month (M1), six months (M6), or six months excluding the previous month (M5). Top K(1/5/10) coverage measures are averaged for the final value.

- **Developer Path Similarity Each**
  `Paths.Similarity.Each.[M1/M5/M6].[Before/After]`
  - The similarity measure comparing the path-contexts in the modified/added method to the set of path-contexts for each individual method added/modified by the developer in the previous month (M1), six months (M6), or six months excluding the previous month (M5). Top K(1/5/10) coverage measures are averaged for the final value.

- **Developer Tree Kernel Similarity**
  `TreeKernel.Similarity.[M1/M5/M6].[K1/K5/K10].[Before/After]`
  - The tree kernel similarity measure comparing the modified/added method to methods added/modified by the developer in the previous month (M1), six months (M6), or six months excluding the previous month (M5). Top K(1/5/10) similarities are averaged for the final value.

- **Developer Tree Kernel Experience**
  `TreeKernel.AltNormal.[M1/M5/M6].[K1/K5/K10].[Before/After]`
  - The tree kernel experience measure comparing the modified/added method to methods added/modified by the developer in the previous month (M1), six months (M6), or six months excluding the previous month (M5). Top K(1/5/10) experience measures are averaged for the final value.

A.5 Other Developer Change Metrics

- **Others Method Changes**
  `Others.Method.Commits`
  - The number of times other developers changed the added/modified method in the previous six months.

- **Others File Changes**
  `Others.File.Changes`
  - The number of times other developers changed a method in the file containing the added/modified method in the previous six months.

- **Developer Project Changes**
  `Dev.Project.Changes`
  - The number of times other developers changed a method in the project containing the added/modified method in the previous six months.

- **Other Developer File Commits**
  `Others.File.Commits`
  - The number of commits where other developers changed a method in the file containing the added/modified method in the previous six months.

- **Other Developer Project Commits**
  `Others.Project.Commits`
  - The number of commits where other developers changed a method in the project containing the added/modified method in the previous six months.

Appendix B

Logistic Regression Control Models

The control models were not included in Chapter 4, because they contain a large number of metrics. Here we present the R model summaries for the control model used with both datasets in both Chapter 4 and Chapter 5. Listing B.1 is the R model summary for the control model for the full modification data. Listing B.2 is the R model summary for the control model for the file-not-seen modification data.

Note that the developer metrics for count-based experience have their names shortened to fit the page. I.e. `Developer.Project.Changes.ExcludeChurn` is changed to `Dev.Project.Changes.NoChurn`.

Listing B.1: R summary for control model, full modification data.

```
Deviance Residuals:
     Min        1Q    Median        3Q       Max
-3.12017  -1.04650  -0.06582   1.14435   2.03664

Coefficients:
                              Estimate Std. Error z value Pr(>z)
(Intercept)                 -5.560e-01  3.735e-02 -14.887  < 2e-16 ***
File.Complexity              5.699e-04  5.760e-04   0.989  0.32246
File.LinesOfCode             2.619e-04  1.392e-04   1.881  0.05992 .
File.Commit.Additions        2.764e-03  3.289e-04   8.404  < 2e-16 ***
File.Commit.Deletions       -3.454e-03  3.464e-04  -9.971  < 2e-16 ***
File.Churn.Developer         3.801e-03  2.609e-03   1.456  0.14526
File.Churn.Others           -5.419e-03  7.320e-03  -0.740  0.45914
File.Churn.Additions        -1.080e-05  5.798e-06  -1.863  0.06244 .
File.Churn.Deletions        -9.053e-05  2.004e-05  -4.518 6.23e-06 ***
Method.ParamCount.Before    -6.403e-03  1.165e-02  -0.550  0.58244
Method.Complexity.Before    -4.329e-02  7.153e-03  -6.051 1.44e-09 ***
Method.LinesOfCode.Before    4.776e-02  2.851e-03  16.750  < 2e-16 ***
Method.TokenCount.Before    -3.062e-03  3.583e-04  -8.547  < 2e-16 ***
Method.Churn.Developer       1.056e-01  2.544e-02   4.149 3.34e-05 ***
Method.Churn.Others          2.784e-02  5.649e-02   0.493  0.62208
Project.Churn.Developer     -1.336e-03  1.684e-04  -7.931 2.17e-15 ***
Project.Churn.Others        -2.618e-04  1.127e-04  -2.323  0.02015 *
Dev.Method.Changes.NoChurn   4.626e-02  1.736e-02   2.664  0.00772 **
Dev.File.Changes.NoChurn     2.226e-02  1.753e-03  12.698  < 2e-16 ***
Dev.Project.Changes.NoChurn -3.070e-04  7.605e-05  -4.037 5.42e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

    Null deviance: 23015  on 16601  degrees of freedom
Residual deviance: 21201  on 16582  degrees of freedom
AIC: 21241

Number of Fisher Scoring iterations: 5
```

Listing B.2: R summary for control model, file-not-seen modification data.

```
Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.52117  -1.12678   0.06796   1.15007   2.05228

Coefficients:
                               Estimate Std. Error z value Pr(>z)
(Intercept)                  -1.482e-01  7.222e-02  -2.051 0.040222 *
File.Complexity               4.211e-03  1.264e-03   3.332 0.000863 ***
File.LinesOfCode             -6.991e-04  3.173e-04  -2.203 0.027572 *
File.Commit.Additions         3.589e-03  7.231e-04   4.964 6.92e-07 ***
File.Commit.Deletions        -5.388e-03  7.498e-04  -7.186 6.69e-13 ***
File.Churn.Others             1.001e-01  6.070e-02   1.650 0.098963 .
File.Churn.Additions         -1.584e-03  1.318e-03  -1.202 0.229346
File.Churn.Deletions          5.272e-04  6.690e-04   0.788 0.430700
Method.ParamCount.Before      1.504e-02  2.383e-02   0.631 0.527913
Method.Complexity.Before     -4.638e-02  1.473e-02  -3.148 0.001641 **
Method.LinesOfCode.Before     3.528e-02  6.271e-03   5.626 1.85e-08 ***
Method.TokenCount.Before     -1.535e-03  7.732e-04  -1.985 0.047107 *
Method.Churn.Others           1.174e-01  3.437e-01   0.342 0.732653
Project.Churn.Developer      -1.468e-03  3.117e-04  -4.709 2.49e-06 ***
Project.Churn.Others         -1.191e-03  2.633e-04  -4.524 6.07e-06 ***
Dev.Project.Changes.NoChurn  -7.823e-05  1.742e-04  -0.449 0.653383
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

    Null deviance: 5445.4  on 3927  degrees of freedom
Residual deviance: 5184.4  on 3912  degrees of freedom
AIC: 5216.4

Number of Fisher Scoring iterations: 4
```