

GRAPHEVO: EVALUATING SOFTWARE EVOLUTION USING MACHINE
LEARNING BASED CALL GRAPH ANALYTICS AND NETWORK PORTRAIT
DIVERGENCE

A Dissertation
IN
Computer Science
and
Computer Networking and Communication System

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
VIJAY WALUNJ

B.E., University of Mumbai, Maharashtra, India, 2008
M.S., University of Missouri–Kansas City, MO, USA, 2013

Kansas City, Missouri
2022

© 2022
VIJAY WALUNJ
ALL RIGHTS RESERVED

GRAPHEVO: EVALUATING SOFTWARE EVOLUTION USING MACHINE
LEARNING BASED CALL GRAPH ANALYTICS AND NETWORK PORTRAIT
DIVERGENCE

Vijay Walunj, Candidate for the Doctor of Philosophy Degree
University of Missouri–Kansas City, 2022

ABSTRACT

Understanding software evolution is essential for software development tasks, including debugging, maintenance, and testing. Unfortunately, as software changes, it becomes more prominent and more complicated, which makes it harder to understand. Software Defect Prediction (SDP) in the codebase is one of the most common ways artificial intelligence (AI) is used to improve the quality of agile products. But graph-based software metrics are seldom used in the software.

In this dissertation, we propose a graph-based software framework called GraphEvo based on deep learning modeling for graphs. We applied the recent network comparison advancement to software networks via information theory-based metric Network portrait divergence (NPD). NPD captures the structural changes to call graph-based software networks. The NPD-based method determines what significant software changes are, how much execution paths are affected, and how tests are improved concerning the code. All

of these factors affect how reliable the software is. To ensure that NPD-based software works well, version controls and Pull Requests (PRs) are used.

GraphEvo's most significant contributions are: (i) Find and show how software has changed over time using call graphs. (ii) Using a machine learning and deep learning techniques to understand the software and guess how many defects are in each code entity (such as a class). (iii) Use the NPD-based tooling to create a public bug dataset and machine learning to see how well it can predict software defects. (iv) Help with the PR review process by knowing how the changes to code and tests that go with them work.

We compared the performance of GraphEvo (i) across 66 software releases from five popular Java open-source systems to show that it works, (ii) for 9 Java projects and deep learning to make an SDP model, (iii) for 19 Java projects of different sizes and types from GitHub and to add bug information from other places, and (iv) for 627 PRs from 14 Java projects to see how vital tests are in PRs. These comprehensive experiments show that GraphEvo works well for debugging, maintaining, and testing software. We also received favorable responses from user studies, in which we asked software developers and testers what they thought of GraphEvo.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled “GraphEvo: Evaluating Software Evolution Using Machine Learning Based Call Graph Analytics And Network Portrait Divergence,” presented by Vijay Walunj, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair
Department of Computer Science Electrical Engineering

Sejun Song, Ph.D.
Department of Computer Science Electrical Engineering

Dianxiang Xu, Ph.D.
Department of Computer Science Electrical Engineering

Md Yusuf Sarwar Uddin, Ph.D.
Department of Computer Science Electrical Engineering

Cory Beard, Ph.D.
Department of Computer Science Electrical Engineering

Gharib, Gharibi, Ph.D.
TripleBlind, Inc., Kansas City, MO

CONTENTS

ABSTRACT	iii
ILLUSTRATIONS	ix
TABLES	xii
ACKNOWLEDGEMENTS	xiv
Chapter	
1 INTRODUCTION	1
1.1 Objective 1: Characterizing and Understanding Software Evolution Using Call Graphs	6
1.2 Objective 2: Defect Prediction Using Deep Learning with NPD for Soft- ware Evolution	7
1.3 Objective 3: NPD-based Tooling, Extendible Defect Dataset and Its As- sessment	8
1.4 Objective 4: Reviewing Pull Requests with Path-based Network Portrait Divergence and Tests	8
2 CHARACTERIZING AND UNDERSTANDING SOFTWARE EVOLUTION USING CALL GRAPHS	10
2.1 Introduction	10
2.2 System Overview	14
2.3 Related Work	19

2.4	Implementation	21
2.5	Evaluation and Discussions	22
2.6	Conclusions and Future Work	30
2.7	Tooling	31
3	DEFECT PREDICTION USING DEEP LEARNING WITH NPD FOR SOFT- WARE EVOLUTION	35
3.1	Introduction	35
3.2	Related Work	40
3.3	Proposed Work	49
3.4	Results and Discussions	57
3.5	Tooling	81
3.6	Threats to Validity	82
3.7	Conclusions and Future Work	86
4	NPD-BASED TOOLING, EXTENDIBLE DEFECT DATASET AND ITS AS- SESSMENT	88
4.1	Introduction	88
4.2	Related Work	90
4.3	Proposed Work	93
4.4	Dataset Creation	99
4.5	Evaluation	101
4.6	Threats to Validation	111
4.7	Conclusions and Future Work	112

5	REVIEWING PULL REQUESTS WITH PATH-BASED NPD AND TESTS . .	114
5.1	Introduction	114
5.2	Related Work	116
5.3	System Overview	120
5.4	Implementation	120
5.5	Results	124
5.6	User Study	128
5.7	Conclusions and Future Work	135
6	CONCLUSIONS AND FUTURE WORK	136
	Appendix	139
	BIBLIOGRAPHY	151
	VITA	168

ILLUSTRATIONS

Figure		Page
1	An Example of A Java Code Snippet and Its Corresponding Call Graph . . .	2
2	Overview of Our Approach	5
3	Call Graph of GraphEvo v.1 Java Library	12
4	System Overview of GraphEvo	15
5	Comparison View of GraphEvo Java Library v.1 and v1.1 Call Graphs . . .	18
6	The Landing Page of GraphEvo	23
7	Visualizing Results with GraphEvo	24
8	Sample of The JUnit Case Study Analysis Results	28
9	Software-level: Creating New Project	32
10	Software-level: Viewing Results	34
11	Overview of Our Approach	48
12	The Study Subjects: Applications and Their Versions	50
13	Snippet of The Call Graph Generated by GraphEvo	53
14	Example of Call Graph with Execution Paths	55
15	Example Illustrating A Snippet of The Dataset Featuring Metrics from The Ant System, Version 1.3.	58
16	Neural Network Structure Overview	62
17	Color-Coded Visualization Of Ant Call Graph Evolution	64

18	Pairwise Comparison of Top 10 Metrics	69
19	Top 10 Metrics	76
20	Univariate and Features Importance Analysis	77
21	Defect Prediction Evaluation Results	80
22	Class-level: Creating New Project Through UI And Uploading Jar Files .	82
23	Class-level: Viewing Complete Call Graph	83
24	Class-level: Viewing Class Metrics	84
25	Overview of Our GraphEvo Dataset Approach	94
26	A Snippet of The Call Graph with Functions Represented As Numbers . .	97
27	Software Ant Dataset Snippet	101
28	F-Measure(F) and AUC Analysis of Software Versions in GraphEvo Dataset Analysis	107
29	Feature Importance in GraphEvo Dataset	108
30	Comparison of The Size of The Datasets	109
31	Code Review Process	115
32	System Overview of GraphEvoPR	117
33	Screenshot of Software commons-lang Call Graph in GraphEvoPR Tool .	122
34	NPD Values as Observed in GraphEvo Dataset	122
35	Class DSCompiler in PR of math (math-10)	125
36	Class StrBuilder in PR of lang (lang-61)	126
37	Class StrBuilder in PR of lang (lang-48)	127
38	Details of User Study	130

39	Survey Participants Industry Experience	131
40	Likert Scale Representation of Survey Responses	135
41	Network Portrait Divergence Calculation Example 1 Of 4	140
42	Network Portrait Divergence Calculation Example 2 Of 4	141
43	Network Portrait Divergence Calculation Example 3 Of 4	142
44	Network Portrait Divergence Calculation Example 4 Of 4	143

TABLES

Tables		Page
1	The Study Subjects	25
2	Metric Values For The Subject Software Systems Based On Their Call Graphs	33
3	Classification Of Network Distances	47
4	Details Of Java Projects For Metric Analysis (Promise Dataset)	51
5	Static Code Metrics	52
6	Metric Values for The Selected Software Systems Based on Network Por- traits	61
7	Time Cost for S/W in Seconds(Promise Dataset)	66
8	Network Portrait Divergence Values for Ant	67
9	Software Ant: Class-level Comparison with Network Portrait Divergence	70
10	Software Lucene: Class-level Comparison with Network Portrait Diver- gence	72
11	Software Xalan: Class-level Comparison with Network Portrait Divergence	74
12	Classification Confusion Matrix	79
13	Related Work	92
14	Static Code Metrics	96
15	Details of Java Projects in GraphEvo Dataset	99

16	Predictive Capabilities in Datasets (F-Measure)	111
17	Static Code Metrics	118
18	Related Work	119
19	Code Complexity and Software Metric Classification	123
20	GraphEvo Dataset Machine Learning Results 1 Of 7	144
21	GraphEvo Dataset Machine Learning Results 2 Of 7	145
22	GraphEvo Dataset Machine Learning Results 3 Of 7	146
23	GraphEvo Dataset Machine Learning Results 4 Of 7	147
24	GraphEvo Dataset Machine Learning Results 5 Of 7	148
25	GraphEvo Dataset Machine Learning Results 6 Of 7	149
26	GraphEvo Dataset Machine Learning Results 7 Of 7	150

Acknowledgments

This work would not have been possible without the support and involvement of so many people. Thank you all! Mom and Dad, thank you for teaching me to be who I am today. Brother, I am pleased to have you in my life and enjoy your support in everything I do. I also thank my sisters, brothers, aunts, and uncles for their encouragement. Beloved late father-in-law, I am glad for the time we spent together, and you are greatly missed.

Thank you, Dr. Yugyung Lee, for teaching me how to be a better scholar and person. Thank you for your enduring motivation. I appreciate all of the time and effort you have put into me. I also would like to thank my previous employer H&R Block for supporting me in my Ph.D. journey.

Dr. Cory Beard, Dr. Sejun Song, Dr. Dianxiang Xu, and Dr. Yusuf Sarwar Uddin, thank you for your time and effort in being on my Ph.D. Committee. I appreciate all of your help and support. Dr. Gharib Gharibi, Sirisha Rella, Dr. Rakan Alanazi, Vasim Shaikh, Duy Ho, and others who worked with me and supported me; thank you for being there. I am grateful for your technical support throughout the project. I wish you a happy and successful life. My deepest gratitude goes to the School of Computing and Engineering faculty and staff.

I want to thank my friend Dr. Rohit Abhishek for constantly encouraging me to pursue Ph.D. and work towards its completion. Also, I want to thank my friends Anurag Thantharate, Rahul Paropkari, and Poonam Kankariya for supporting me spiritually throughout writing this thesis and my life in general. Lastly, I wouldn't have survived writing this thesis without the support of my wife, Priyanka Gaikwad.

CHAPTER 1

INTRODUCTION

Software comprehension is an imperative and indispensable prerequisite for software development activities such as maintenance, testing, and quality management [1, 2]. As a software system grows, its functionality and components' interactions increase in size and complexity. Without a proper understanding of the software system and its inner interactions, adding or changing a feature increases the risks of introducing errors and potentially undesirable behavioral changes. This problem becomes even more complicated when investigating the system's evolution, i.e., comparing the changes among several software releases. Software typically consists of specification, design and implementation, validation, and evolution. Software evolution is developing, maintaining, and updating software for various reasons such as requirement changes, defects, technical debt, and compatibility with other software. In modern software development, Software evolution affects all fundamental processes.

Software evolution is responsible for 50%-90% of the total development cost [3, 4]. In practice, software developers and testers often investigate the software system releases to grasp and depict the functionality changes. Software developers specifically need to understand software changes introduced in a specific release while working on fixing defects.

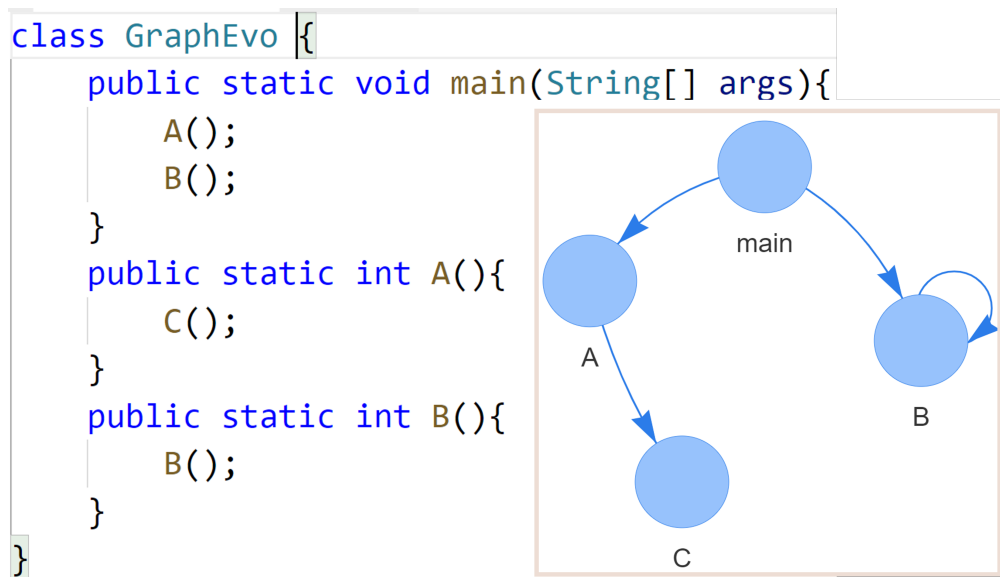


Figure 1: An Example of A Java Code Snippet and Its Corresponding Call Graph

A software defect is a deviation from the software specifications or end-user expectations and can lead to unpredictable results or failures. The Cost of Poor Quality Software in the US: A 2018 Report analysis [5] has found that in 2018, low-quality software cost more than \$2.8 trillion in the United States alone, of which 16.87% was spent on known/fixed faults cost while 37.46% were software failure losses. Furthermore, this demonstrates that finding and correcting defects are costly software development activities caused by internal or external effects. Internal costs such as waste, scraping, and/or rework occur before software goes into production. The external costs of production-ready software include reproducing, discovering, fixing, and verifying defects and their locations.

Miltiadis Allamanis et al. [6] pointed out that detecting defects is a core challenge

in software engineering because discovering defect patterns or defining appropriate measures is not trivial due to the complexity of software features and the diversity of software structure. Because software defects are expensive, the prediction of defects has been the research subject for the last several decades, leading to many tools and predictive models [7, 8].

These approaches help classify software artifacts (e.g., classes, modules, subsystems, and files) as being defect-prone or not. Even models predict defect density (the number of defects / SLOC) or the likelihood of a software artifact having a defect. Predicting malfunctions early in software development helps optimize the efforts of engineers and testers. Developers can prioritize code inspection and testing, starting with code areas having more defect-prone possibilities. Consequently, testers can efficiently utilize their limited resources and prioritize their test workflows.

Software defect prediction approaches are significantly cheaper than software measurement and reviews. Empirical studies have indicated that the probability of detecting software defects using prediction models could be higher than the possibility of discovery in current software reviews [9]. Many of these techniques, such as statistics-based models, parametric models, and machine learning-based models, are used [10, 11]. Defect prediction can improve software testing and has the potential to improve the overall software quality assurance process [12]. The recent advances in information theory have revolutionized the modeling and analysis of complex systems in many disciplines and practical problems, such as understanding biological systems [13], modeling the network's topology [14], and modeling software systems using complex graphs [15, 16]. Graph-based

techniques enable capturing the structure and essential properties of the system. It can unleash various methods and tools to investigate and study the design patterns, detect abnormalities, and forecast new trends. A call graph [17] is a visual representation of the software system structure in such a way that each node represents a function, and each directed edge represents a function relationship. Fig. 1 illustrates a sample code snippet written in Java and its corresponding static call graph. Node *main* is the system's starting point and the head of the execution path. Node *B* and Node *C* are the functions called from the main.

Open-source software has gained more and more popularity worldwide in the last decade. GitHub has emerged as a winner in hosting millions of such software and creating abundant data. Researchers have been probing through GitHub and coming to some insightful conclusions. PROMISE [18, 19] and Metrics Data Program (MDP) [20] from NASA are the popular publicly available datasets used for Software Defect Prediction (SDP). The researchers investigated different approaches to SDP and determined whether a new approach improves current ones [21, 22, 23, 24]. Only a few datasets have been the foundation for further research from too many datasets. Additionally, some of the work done on the dataset is not publicly available, and researchers must collect and process such data. Some of the datasets built on top of the popular datasets would lack repeatability [21].

Researchers have also focused their attention on pull request processes, researching many features such as the review process, how pull requests are assigned to different

reviewers, and under what conditions they are accepted [25, 26]. Pull requests are inherently social. Change requests are sent to the project reviewers and integrators, who hold a conversation and ultimately decide whether the changes are merged into the main branch. Researchers have also found that the developer’s reputation in sending the pull request and implementing new features appears to be a more critical acceptance consideration than any other criterion, including quality [27].

The below section defines the objectives and brief abstract of each chapter. The overall framework for each chapter is described in Fig. 2.

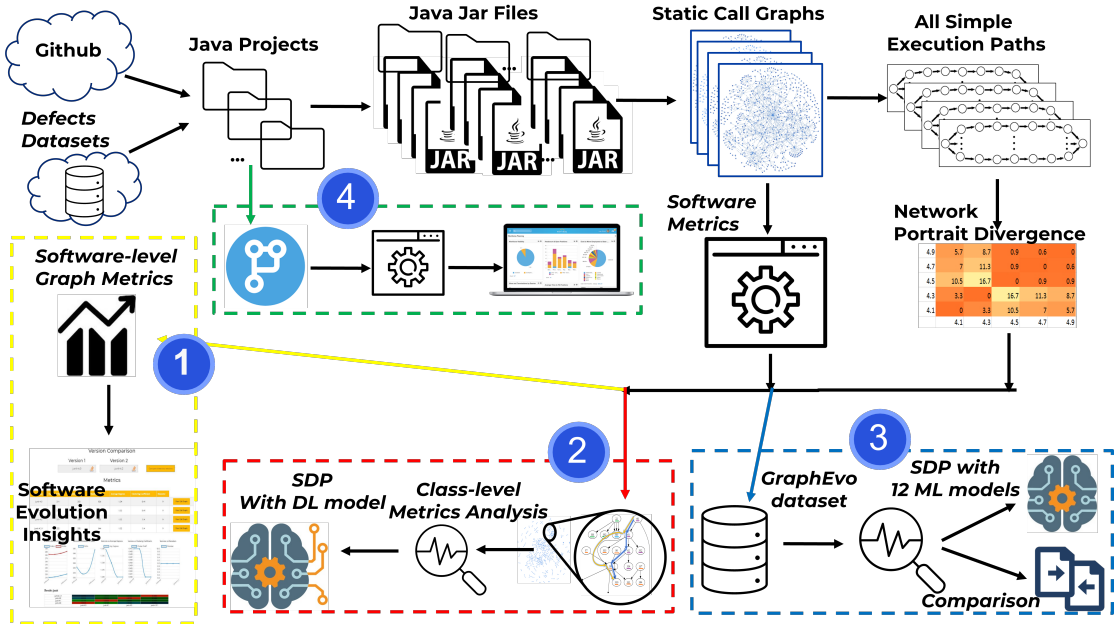


Figure 2: Overview of Our Approach

1.1 Objective 1: Characterizing and Understanding Software Evolution Using Call Graphs

As a software system evolves, its functionality and inner interactions increase in size and complexity. Without a proper understanding of the software system and its components' interactions, the tasks of adding or changing a feature increases the risks of making mistakes or changing the overall behavior of the software system [28]. This problem becomes even more challenging when investigating the system's evolution, i.e., comparing the changes among several software system releases over time. It's well-known that maintenance tasks, especially those resulting from evolving software, are the most expensive tasks in software development. Indeed, software evolution and its maintenance can cost from 40% to 80% of the overall development cost [3, 29].

Contributions

- Build GraphEvo utility to construct and visualize the call graph for one or more software releases.
- Calculate and display the set of graph-based metrics.
- Construct color-coded graphs and compare software evolution for several software systems.

1.2 Objective 2: Defect Prediction Using Deep Learning with NPD for Software Evolution

One of the primary practices in software evolution is predicting defects across software components in the codebase to improve agile product quality. While several software metrics exist, graph-based metrics have rarely been utilized in software quality. In this chapter, we explore recent network comparison advancements to characterize software evolution and focus on aiding software metrics analysis and defect prediction. We support our approach with an automated tool named *GraphEvoDef*.

Contributions

- Exploit graph-based metrics to study software evolution and identify code changes to aid software quality.
- Distinguish the metrics suited for defect prediction and compare the metrics to the NPD.
- Implement GraphEvoDef to help developers identify code modules that can have defects and predict defect counts.
- Provide an open-source Python tool to automate our study's analysis.

1.3 Objective 3: NPD-based Tooling, Extendible Defect Dataset and Its Assessment

Predicting software defects early in the release cycles can enhance resource efficiency and improve overall software quality. Researchers have been probing through popular datasets such as PROMISE, MDP, and datasets built on top of popular datasets to perform Software Defect Prediction (SDP). Only a few datasets have been the foundation for further research from too many datasets. In contrast, other datasets are not publicly available, lack reproducibility, lack the tooling needed to add new projects, or are incomplete. Information theory-based metrics such as NPD have proven helpful in SDP.

Contributions

- Provide an open-source Python tool to add new projects to the dataset.
- Build a dataset having class-level metrics with NPD and defect information.
- Evaluate the GraphEvo dataset using 12 machine learning techniques.
- Compare GraphEvo's performance to that of two other datasets.

1.4 Objective 4: Reviewing Pull Requests with Path-based Network Portrait Divergence and Tests

Pull requests (PRs) are widely used in open-source and industrial environments to assess and accept/reject feature contributions. Unlike the typical code review process, PRs provide a more lightweight approach for committing, reviewing, and managing code

changes. A PR contains the code and tests for a specific work item. Previous studies have reported that PR review is crucial for software development and that reviewers do not spend more time on test files than code files. At the same time, code reviewers are concerned that the tests accompanying the code modifications are adequate and cover all possible paths. Software metrics have been demonstrated to be helpful in different areas under software engineering.

Contributions

- Collect class-level metrics for code in PRs before and after modifications.
- Review a set of metrics and identify associated changes to tests.
- Provide an open-source Python tool to automate PR analysis.

CHAPTER 2

CHARACTERIZING AND UNDERSTANDING SOFTWARE EVOLUTION USING CALL GRAPHS

2.1 Introduction

Understanding the software structure and its organization is an imperative and indispensable prerequisite for software development activities such as reuse, testing, maintenance, and evolution [30]. As a software system evolves, its functionality and inner interactions increase in size and complexity. Without a proper understanding of the software system and its components' interactions, the tasks of adding or changing a feature increases the risks of making mistakes or changing the overall behavior of the software system [28]. This problem becomes even more challenging when investigating the system's evolution, i.e., comparing the changes among several software system releases over time. It's well-known that maintenance tasks, especially those resulting from evolving software, are the most expensive tasks in software development. Indeed, software evolution and its maintenance can cost from 40% to 80% of the overall development cost [29, 3].

In practice, software developers and testers often need to investigate multiple software system releases to understand and depict the functionality changes over the releases. Software testers need to understand software changes introduced in a new release and focus their testing efforts on the new functions and execution paths to develop new test

scenarios, for example.

Recent information theory advances have revolutionized the modeling and analysis of complex systems in many disciplines, such as modeling the network's topology and software systems using call graphs [16]. A call graph [17] is a visual representation of the software system structure in such a way that each node represents a function, and each directed edge represents a function relationship.

Fig. 3 illustrates the static call graph of GraphEvo java library. The node *main* represents an entry point to the system, which is the head of the execution path and is not called by any other function in the system. It is represented as a node without incoming edges in the call graph. The leaf nodes represent exit points, and it does not call any other function in the system. It is represented as a node without outgoing edges in the call graph. An execution path is an ordered sequence of function calls from an entry point to an exit point; it does not visit the same function or function call twice. Note that the cycles within the call graphs can be eliminated or restricted to a specific number of iterations to reduce the length of the execution paths.

Static call graphs have been proven to assist software developers in understanding the overall system interactions and structure. For example, the tool in [31] can detect and visualize the static interactions between the classes of software systems written in Java. Code2graph [32] is a tool that automates the construction and visualization of static call graphs for programs written in Python. The work in [33] utilized machine learning algorithms to cluster the execution paths in large call graphs to facilitate system comprehension. The authors of [34] studied the call graphs of 223 releases of the Linux kernel to

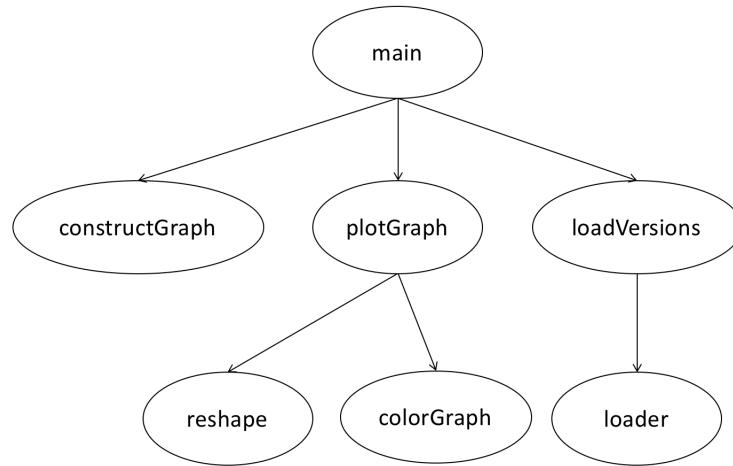


Figure 3: Call Graph of GraphEvo v.1 Java Library

identify similar graph structures. Bhattacharya et al. [16] analyzed the releases of well-known projects to evaluate the use of call graphs in understanding software evolution. Similar to the existing methods and tools, our tool aims at constructing and visualizing call graphs. However, unlike existing tools, our tool uses *static call graphs to visualize software evolution in a monolithic color-coded graph and graph metrics to quantify software change.*

In this chapter, we introduce a lightweight tool named GraphEvo (Graph Evolution) that can automatically construct and visualize color-coded call graphs to assist software developers and testers in better understanding software evolution and answer critical questions, such as “Which execution traces have changed, and where? Which new functions were added and where?” Notably, GraphEvo can automatically:

- Construct and visualize static call graphs for individual software releases to support system comprehension.

- Calculate a set of graph-based metrics to provide insights into the code structure and its characteristics.
- Present color-coded call graphs to illustrate software evolution in a monolithic graph to facilitate identifying and locating changes.
- Calculate the portrait divergence of each software release to quantify structural changes.

GraphEvo’s overarching goal is threefold. First, it is to help software developers understand the codebase and its structure by constructing and visualizing static call graphs. Second, it is to visualize and characterize software evolution. Third, assist software testers with decision-making by focusing on the codebase changes over time. GraphEvo applies to projects written in Java. However, it can be easily extended to other object-oriented programming languages, as explained in Section 2.4. GraphEvo can be launched and used in a command-line or web-based interface.

To evaluate the applicability and scalability of our tool and its underlying approach, we used five popular open-source Java systems: JUnit [35], jMock [36], Jgap [37], Guava [38], and SLF4J [39]; a total of 66 releases selected based on their popularity and availability. We report the evaluation and the results on the tool’s website (<https://goo.gl/8edZ64>).

The rest of this chapter is organized as follows. Section 2.2 presents an overview of GraphEvo and its features. Then we briefly discuss its implementation in Section 2.4. We evaluate and discuss our system in Section 2.5. Related work is summarized in Section

2.3. Finally, we conclude the chapter in Section 2.6.

2.2 System Overview

The approach behind our tool is to use the information theory of call graphs to facilitate and improve the characterization and visualization of software evolution. We introduce an open-source tool that can automatically visualize and characterize the software evolution for a given number of software releases with features to highlight the differences in execution paths to assist stakeholders in making better decisions. For example, assisting software testers in developing new test scenarios and answering a wide range of questions, such as “*What execution paths have been added that were not tested before?*”

The underlying approach of GraphEvo consists of two main steps (see Fig. 4). First, it constructs a call graph for each release of the software system under investigation. While the call graph can be used to understand the functionality and behavior of a single software system at a time, the goal here is to utilize the call graphs in characterizing the software evolution over time. Second, it calculates graph metrics across several system releases to quantify evolution. In addition, the tool provides a comparison and visual analysis features to help software testers better understand the code and execution-level changes. We explain the workflow and features of GraphEvo in the following steps using sample results from analyzing the evolution of GraphEvo itself as a test case throughout the rest of the chapter.

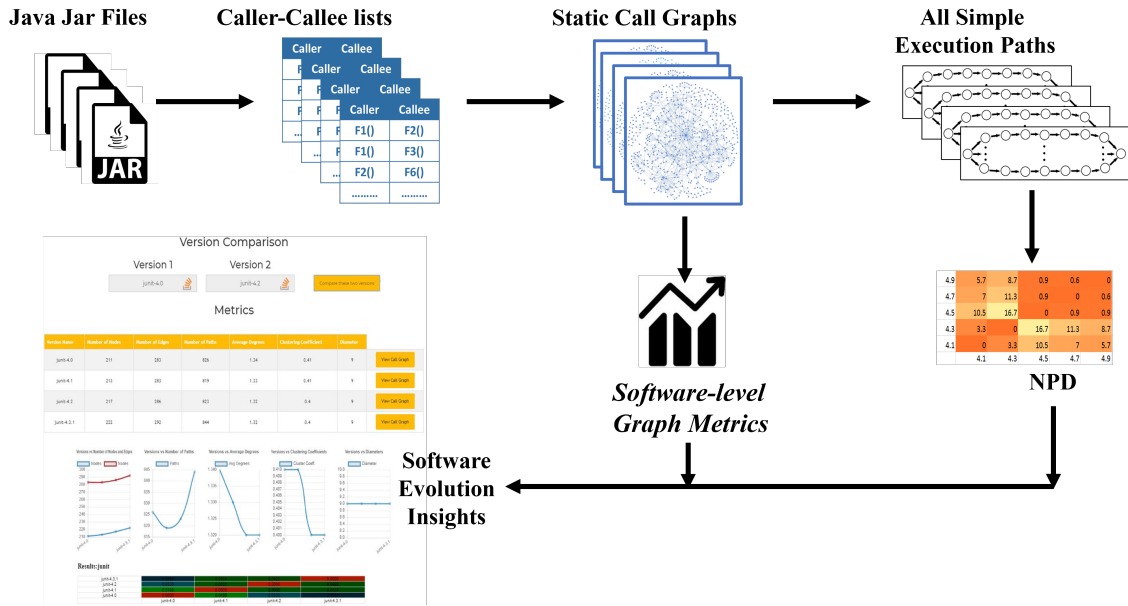


Figure 4: System Overview of GraphEvo

2.2.1 Constructing and Visualizing Call Graphs

The first step in building the call graph is extracting the caller-callee relationships from the codebase. We used an open-source tool [40] that can extract the function calls for a given Java codebase or Jar files. The tool generates a list of all function calls in the software system in the form of *caller – callee* pairs. We use this list to construct a generic call graph by adding each of the caller-callee relationships as a directed edge, (u, v) , to the graph where the caller, u , and the callee, v , represent nodes in the graph and a call from u to v . This generates a proper call graph for the entire system at the granularity level of function calls. This process is repeated for each of the software system releases under investigation. The call graphs are a prerequisite to measuring and understanding the software evolution over time and are used as an input to the second step in our study.

Visualizing the call graph of a single software system has proven to facilitate understanding the system’s behavior and functionality at the source-code level. In order to visualize the call graph of a single system, we translate its edges and nodes from the OOP language that they were written with to a graph description language, *DOT* [41], which in return can be rendered using different tools, such as *GraphViz* [42], to actual graphs in *jpg*, *svg*, or *pdf* formats.

2.2.2 Characterizing Software Structure

GraphEvo analyzes the call graphs generated in the previous step for each software release to quantify and visualize the software structure. First, we analyze meaningful graph metrics, including the number of nodes and edges, the graph average degree, the clustering coefficient, the graph diameter, and its modularity to capture the changes in the structure. Second, we compare the software releases based on their portraits using the Portrait Divergence method [43]. The overarching goal is to demonstrate that the graph metrics and portrait divergence can be used on software call graphs to quantify and visually illustrate the software evolution. Before we present and discuss the results, we first define the used graph-based metrics as follows:

- *Average degree*: The average degree of a graph is the largest vertex degree. A vertex degree is the number of edges incident to that vertex, with the loops counted twice. The average degree of a node, i.e., function, indicates its popularity; thus, it can quantify the graph coarseness. Graphs with a higher average degree tend to be tightly connected. The average degree is defined as $\bar{k} = \frac{2|E|}{|V|}$ where $|E|$ denotes the

number of edges and $|V|$ denotes the number of nodes.

- *Clustering coefficient*: It is a property of the nodes that measures to what degree the neighbors of a node v_i are also connected. A high value of the clustering coefficient means that the nodes in the neighborhood are tightly connected, and a value of 0 means that there are hardly any connections in the neighborhood. High values are discouraged in software engineering practices [3]. The neighborhood, N_i , of a vertex, v_i , is defined as its immediately connected neighbors and is given as $N_i = \{v_j : e_{ij} \in E \vee e_{ji} \in E\}$. The local clustering coefficient C_i for a vertex v_i , where k_i is the number of vertices in the neighborhood, is given as:

$$C_i = \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}$$

- *Graph diameter*: It is the longest and shortest path between any two vertices in the graph. Graphs with higher diameters are undesirable since they reflect deeper runtime stacks obstructing program comprehension, debugging, and maintenance.
- *Graph Modularity*: It is used to detect the community structures of a graph. A graph with a high modularity ratio between the nodes of a subgraph but sparse connections to the nodes in different subgraphs is desired. Inspired by the work in [16], a call graph modularity is defined as the ratio between its cohesion and its coupling values. It is well-known that the design of software systems is expected to have high cohesion and low coupling, which makes the system easier to test, debug,

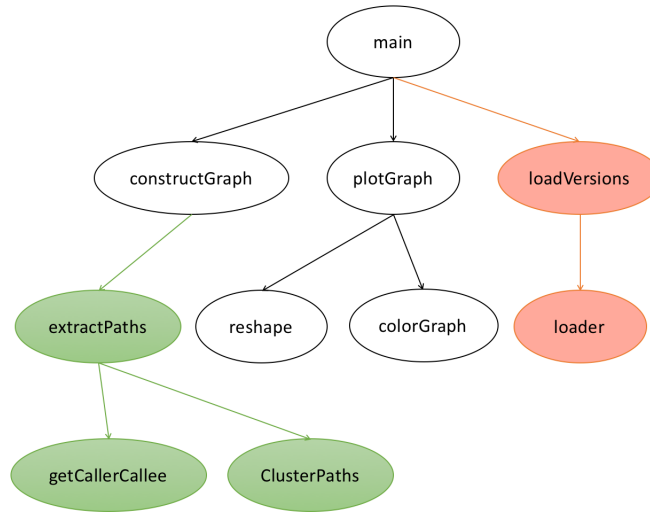


Figure 5: Comparison View of GraphEvo Java Library v.1 and v.1.1 Call Graphs

and maintain. The modularity of a subgraph A is defined as:

$$Modularity(A) = \frac{Cohesion(A)}{Coupling(A)}.$$

In addition, the tool compares the software releases based on their portraits using the Portrait Divergence metric [43]. The graph metrics and portrait divergence are used to quantify and visually illustrate software evolution and execution-path code changes, which can ultimately facilitate several software engineering tasks, significantly aiding integration tests.

2.2.3 Characterizing and Visualizing Software Evolution

Portrait divergence was recently developed to compare networks using their portraits. Unlike the previous ad-hoc comparison measures (see Section 2.3), portrait divergence is based on information theory, which provides a principled interpretation of the

divergence measure. Thus, it can compare the networks based on their topology structures and not assume that the networks are defined on the same nodes. Moreover, unlike the current expensive node matching optimization methods, portrait divergence is graph invariant and is relatively computationally efficient. Note that this approach can treat both directed and undirected networks similarly.

To calculate the portrait divergence among the portraits of several releases of a software system, GraphEvo (1) extracts all simple execution paths of each of the software releases; (2) constructs a set of network portraits for each of the releases based on their execution paths and (3) calculate the portrait divergences and visualizes differences among the releases of the software system using a color-coded matrix and call graphs. The color codes include *red* to identify nodes removed from a version and *green* to identify new nodes. These processes are carried out automatically by GraphEvo. The user can view each one of these steps and its outcomes for further system comprehension. Moreover, these outcomes (e.g., execution traces) are a stepping stone for further software analytics applications. Fig. 5 illustrates a call graph representing two analyzed versions of GraphEvo.

2.3 Related Work

The recent surge in graph data across various scientific domains has led to new tools and methods for understanding and evaluating big graphs. While call graphs have been used to facilitate the tasks of software comprehension, they are still limited to capturing the functionality of a single software system at a time. However, as a software system

evolves, understanding the similarities and differences across multiple releases become a crucial and challenging tasks. To this end, our research focused on automating the process of quantifying and visualizing the changes across multiple releases of a software system based on an information-theoretic approach to comparing the software call graphs.

Call graphs have been thoroughly studied to represent the software system's inner interactions in function calls. Most existing studies either focus on generating and analyzing the call graphs of one system at a time or the power law of the node degree distribution. For example, the researchers in [31] built a tool to detect and visualize the static interactions between the classes of software systems written in Java; the work in [34] studied the call graphs of 223 releases of the Linux kernel to identify similar graph structures; while [16] analyzed the releases of well-known projects to evaluate the use of information theory in understanding software evolution.

In addition, other studies have focused on software collaborative graphs. For example, [44] examined the collaborative software graphs and found them scale-free, small-world networks similar to those found in other sociological and biological systems. Unlike these tools, GraphEvo focuses on *static* code analysis to generate and use *static* call graphs.

Nevertheless, most of the current approaches in this field examine the structural properties of a single software system at a time, including model dependency [45], class collaboration graphs [44], and inheritance graphs. Other related works have discussed motifs [46]. These approaches focus on a single software system. They cannot capture the changes between two networks of the same topology, i.e., the exact arrangement of

the edges and their directions between the nodes, but different sets of nodes. Other approaches use graph edit distance matrices to measure the number of edges and nodes needed to transfer one network into another. Similar to the basic approaches, distance measures are limited to only capturing the network topology. Advanced research in this area proposes the comparison of network subgraphs and motifs.

Overall, GraphEvo surpasses the tools in its category in two ways: First, it extends and combines several graph metrics to capture the structural and functional software evolution across several releases of popular open-source systems written in Java or Python. Second, it provides color-coded call graphs to identify and locate changes in software releases.

2.4 Implementation

The implementation tasks were slightly straightforward. First, we extended and integrated an existing tool, *Constructor* [40], that extracts a list of the caller-callee functions for a given Java project. Second, we used the caller-callee relationships to construct a graph using a generic graph data structure. Third, once the graph object is generated, we can either visualize it to the user by translating it into a DOT language or use it as an input to the metrics calculator, which can calculate and visualize the aforementioned graph metrics. Fourth, based on the portrait divergence results, we can find and localize the changes and visualize them in a color-coded call graph.

The tool is fully implemented in Python and operated locally using a web-based interface. We used HTML and CSS for the visuals and designs, along with Javascript

for client-side requests to data retrieval from the server. For the backend server development, we utilized Node.js, a well-known open-source, cross-platform Javascript runtime environment that significantly simplifies the process of server-side programming for web applications. Therefore, to make the application even more dynamic and flexible, we employed Express.js, another Javascript library that functions on top of Node.js and is meant for convenient, scalable, and efficient routing of the whole system, without which it may add layers of complexity upon determining correct and accurate routing components from the standard Node.js environment.

Fig. 6 illustrates GraphEvo's landing page. On the top navigation bar, users can perform one of four actions: view the welcome screen, learn more about our system, consider the previously executed projects, and analyze their software releases. For the last action, they are prompted to select a list of software versions and enter their preferred project name that is displayed and stored in the server accordingly. The highlighted files are then analyzed simultaneously by the backend server. After results are finalized, they are returned to the frontend via various formats such as text and JSON files, which will be used to render visual components and illustrations, as shown in Fig. 7.

2.5 Evaluation and Discussions

Our evaluation was based on a total of 66 releases of five open-source Java software systems, including software libraries and applications, namely JUnit, jMock, Jgap, Guava, and SLF4J. JUnit is a unit testing framework for Java. jMock is a library that supports test-driven Java code development with mock objects. Jgap is a Java Genetic

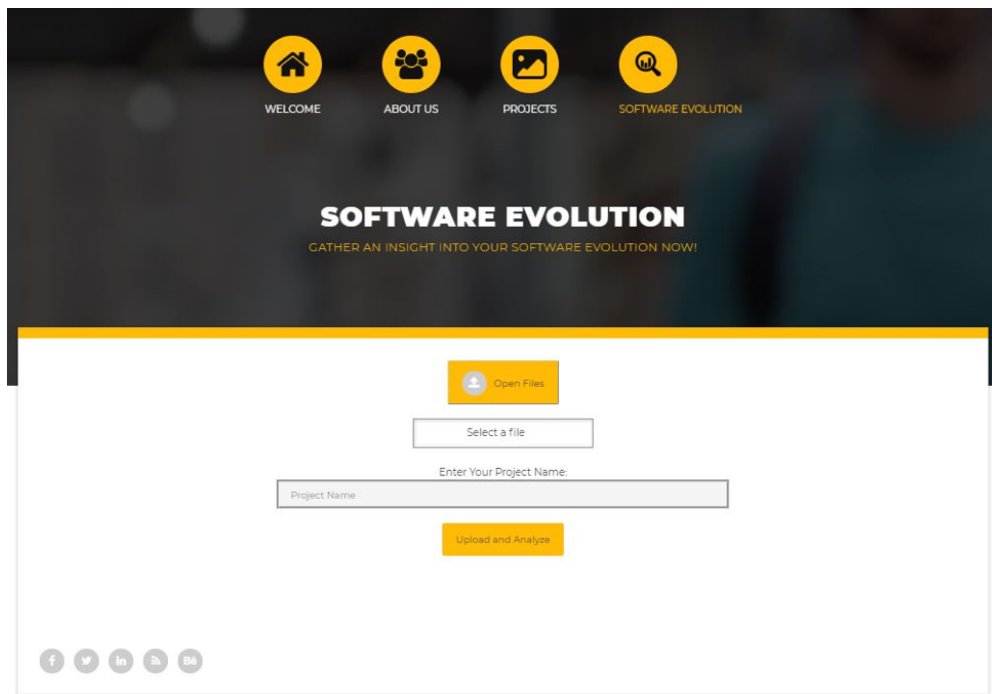


Figure 6: The Landing Page of GraphEvo

Algorithms package. Guava is a set of core libraries that includes new collection types, immutable collections, a graph library, functional types, and other utilities. SLF4J is a simple logging facade for Java. These software systems were selected based on the following criteria: (1) they must be open-source systems written in Java, (2) they must have a long history of releases, (2) they must be of significant size (thousands of source-line of code -KSLOC), and (3) they must be popular, i.e., have a large user base. After identifying a pool of such systems, we collected all available releases of each software system. The search for these software systems was carried out using Google BiGQuery [47]. Table 1 lists the applications involved in our study along with some critical properties, including name, time span, and size in KSLOC.



Figure 7: Visualizing Results with GraphEvo

One of the essential driving questions of our evaluation is: *Can graph-based techniques facilitate the comprehension and analysis of software evolution using the software system call graph?* In this section, we answer this question by discussing the results of the examined software systems using GraphEvo. Specifically, we show how the changes in graph metrics over several releases can indicate non-obvious events in the software evolution, which could affect various software engineering concerns. The metrics results, including the number of nodes, the number of edges, the number of execution paths, the average degree of the graph, the clustering coefficient, the graph diameter, and the modularity ratio, are listed in Table 2. Note that the table includes the results for only five releases of each case study due to space constraints. Examples of the result graphs, metrics diagrams, and portraits for Junit are listed in Fig. 8.

The number of nodes, edges, and execution paths: We first observed, as one would expect, that the number of nodes, edges, and execution paths increased as the software system evolved. Some systems exhibit linear growth in these metrics, namely jMock and Jgap. However, Guava experienced the highest growth among the five systems in the releases 20.0 and 21.0, while its size decreased from 132.96 KSLOC to 106.85 KLOC. We believe this change happened due to code re-factoring since the number of

Table 1: The Study Subjects

Software System	Time Span	# Releases Tested
Junit	2007-2014	20
jMock	2007-2018	13
Jgap	2010-2012	5
Guava	2011-2017	13
SLF4J	2010-2017	15

nodes and edges sharply grew in these two releases compared to the previous releases while the system's overall size decreased. This implies that significant code changes had happened to increase system quality. It was also apparent that most used releases of the SLF4J system were stable and did not grow over the first four releases compared to slight growth in the last release.

In contrast, the JUnit system illustrated a compelling case. We observed that in the early stages of the software evolution, it exhibited a sharp growth in the number of nodes, edges, and execution paths between the 4.1 and 4.3 releases. We believe the sharp growth in earlier versions was due to adding new features. We manually inspected the release's repository to find that the number of defects reported drastically increased. This explains the sudden drop in these metrics after the sharp increase. Furthermore, graph metrics could also be used to examine the nodes and paths that were mostly affected or were vulnerable to change.

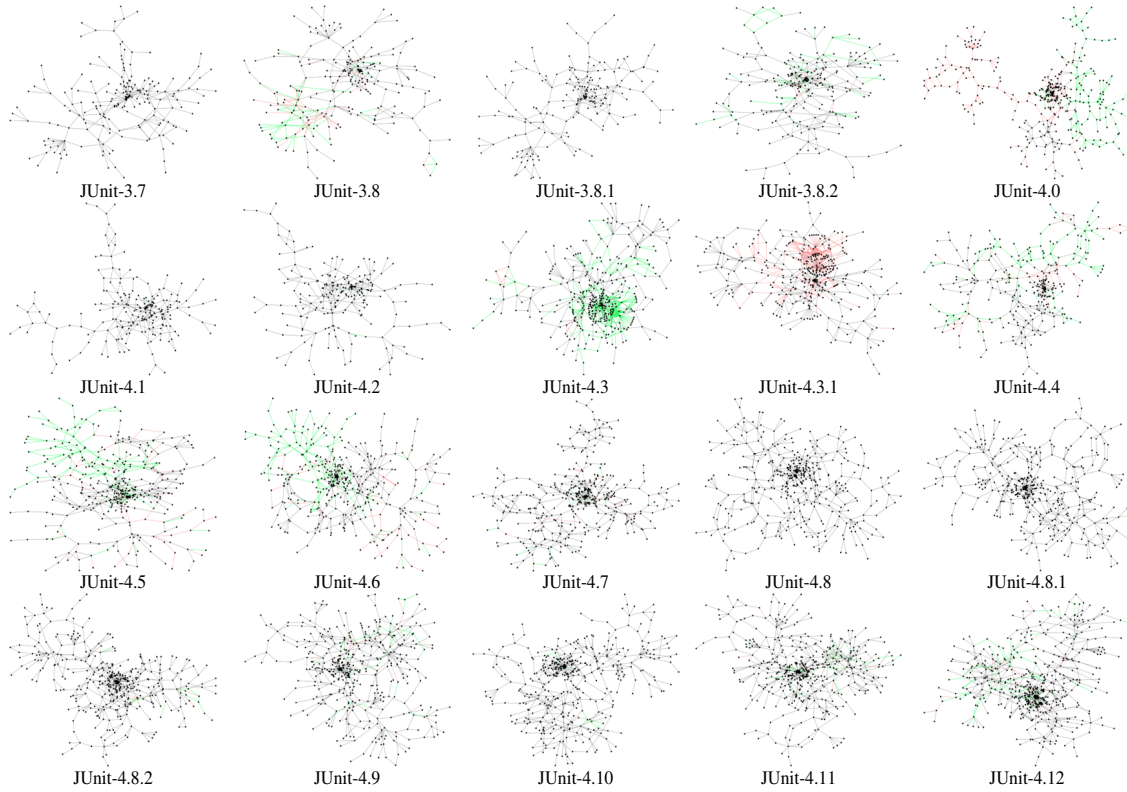
Average degree: It has been proven in previous work that graphs with a high average degree are more tightly connected [48]. In a call graph representing a software system's inner function interactions, the average degree illustrates the function's coarseness or popularity. One interesting observation of the system's average degree is that it had a narrow range from 0.7 - 1.3. Most of the systems exhibited a slow and steady increase. An outlier was the jMock system: its average degree changed from 0.702 to 6.92 and then to 6.73 in the releases 2.2, 2.5, and 2.6, respectively, then turned back to 0.687 again. We believe this change was triggered by the increase in edges from 95 to 133 from the 2.2 to 2.5 releases.

In contrast, while the number of edges in the JUnit system increased from 283 to 573, the average degree increased from 1.33 to 1.55. We believe that this different behavior in the two systems (jMock and JUnit) is mainly affected by the type and distance of the edges. Increasing the number of edges between the nodes of a subgraph will proportionally increase its average degree and vice versa.

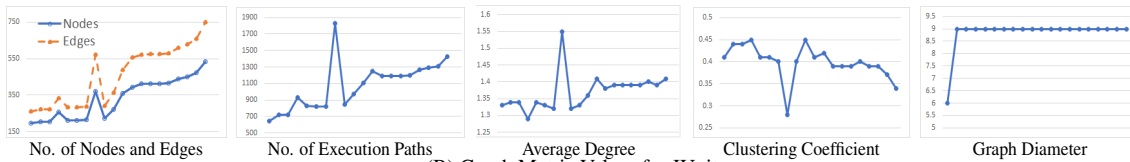
Clustering coefficient: It is a property of the nodes that measures to what degree the neighbors of a node are also connected. A high value of the clustering coefficient means that the nodes in the neighborhood are tightly connected, and a value of zero indicates that there are hardly any connections in the neighborhood. Specifically, a coefficient of zero means that the graph is bipartite. A bipartite graph, also known as a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. A high clustering coefficient value indicates poor software engineering practice since the nodes are tightly connected, and the graph cannot form hierarchical abstraction levels, complicating software evolution.

Table 2 show that the clustering coefficient values for jMock and SLF4J are zero throughout the software releases we examined, indicating good software engineering development practices. Similarly, the rest of the systems had very low clustering coefficient values. Both Jgap and Guava had very stable values throughout their releases. In contrast, the values in JUnit fluctuated between 0.002 and 0.02, which could be derived from the changes in the system's overall structure.

Graph diameter: We observe from Table 2 that the diameter values tend to stay constant in all systems, except for a slight change in Guava. The values ranged from 1, in



(A) Color-Coded Visualization of JUnit Call Graph Evolution



(B) Graph Metric Values for JUnit

junit-4.12	0.9107	0.975	0.975	1.2113	0.7411	0.6831	0.6505	1.7926	0.6625	0.4555	0.1761	0.2328	0.0802	0.0802	0.0802	0.0796	0.0696	0.0698	0.0386	0
junit-4.11	0.7186	0.7849	0.7849	1.0144	0.6116	0.5439	0.5097	1.4709	0.5121	0.3582	0.1304	0.1466	0.0393	0.0393	0.0393	0.0275	0.0272	0.0168	0	0.0386
junit-4.10	0.5877	0.672	0.672	0.9047	0.5475	0.4879	0.454	1.3674	0.4789	0.3188	0.1125	0.1094	0.0306	0.0306	0.0306	0.0253	0.0197	0	0.0168	0.0696
junit-4.9	0.6305	0.6553	0.6553	0.8555	0.5238	0.4645	0.4517	1.3793	0.4817	0.2846	0.1146	0.1214	0.0279	0.0279	0.0279	0.0192	0	0.0197	0.0272	0.0696
junit-4.8.2	0.582	0.6154	0.6154	0.8555	0.5137	0.444	0.4193	1.3992	0.4627	0.3049	0.0927	0.0974	0.0162	0.0162	0.0162	0	0.0192	0.0253	0.0275	0.0796
junit-4.8.1	0.5515	0.5921	0.5921	0.8057	0.4747	0.4081	0.383	1.3548	0.44	0.2597	0.0541	0.0684	0	0	0	0.0162	0.0279	0.0306	0.0393	0.0802
junit-4.8	0.5515	0.5921	0.5921	0.8057	0.4747	0.4081	0.383	1.3548	0.44	0.2597	0.0541	0.0684	0	0	0	0.0162	0.0279	0.0306	0.0393	0.0802
junit-4.7	0.5515	0.5921	0.5921	0.8057	0.4747	0.4081	0.383	1.3548	0.44	0.2597	0.0541	0.0684	0	0	0	0.0162	0.0279	0.0306	0.0393	0.0802
junit-4.6	0.331	0.3951	0.3951	0.5551	0.283	0.2378	0.2071	1.0323	0.2554	0.1535	0.0356	0	0.0684	0.0684	0.0684	0.0974	0.1214	0.1094	0.1466	0.2328
junit-4.5	0.5042	0.4912	0.4912	0.6194	0.2897	0.2596	0.2354	3.4771	0.2704	0.1563	0	0.0356	0.0541	0.0541	0.0541	0.0927	0.1146	0.1125	0.1304	0.1761
junit-4.4	0.4567	0.3044	0.3044	0.3023	0.1106	0.0883	0.0988	3.0316	0.1094	0	0.1563	0.1535	0.2597	0.2597	0.2597	0.3049	0.2846	0.3188	0.3582	0.4555
junit-4.3.1	0.4705	0.3849	0.3849	0.3848	0.056	0.0534	0.0422	2.8142	0	0.1094	0.2704	0.2554	0.44	0.44	0.44	0.4627	0.4817	0.4789	0.5121	0.6625
junit-4.3	0.7206	2.6622	2.6622	0.7541	2.8582	2.3476	2.3654	0	2.8142	0.0316	3.4771	1.0323	1.3548	1.3548	1.3548	1.3992	1.3793	1.3674	1.4709	1.7926
junit-4.2	0.4043	0.3112	0.3112	0.3314	0.033	0.0085	0	2.3654	0.0422	0.0988	0.2354	0.2071	0.383	0.383	0.383	0.4193	0.4517	0.454	0.5097	0.6505
junit-4.1	0.4566	0.3076	0.3076	0.2997	0.0199	0	0.0085	2.3476	0.0534	0.0883	0.2596	0.2378	0.4081	0.4081	0.4081	0.444	0.4645	0.4879	0.5439	0.6831
junit-4.0	0.485	0.311	0.311	0.2921	0	0.0199	0.033	2.8582	0.056	0.1106	0.2897	0.283	0.4747	0.4747	0.4747	0.5137	0.5238	0.5475	0.6116	0.7411
junit-3.8.2	0.4241	0.1323	0.1323	0	0.2921	0.2997	0.3314	0.7541	0.3848	0.3023	0.6194	0.5551	0.8057	0.8057	0.8057	0.8555	0.8555	0.9047	1.0144	1.2113
junit-3.8.1	0.1991	0	0	0.1322	0.311	0.3076	0.3112	2.6622	0.3849	0.3044	0.4912	0.3951	0.5921	0.5921	0.5921	0.6154	0.6553	0.672	0.7849	0.975
junit-3.8	0.1991	0	0	0.1323	0.311	0.3076	0.3112	2.6622	0.3849	0.3044	0.4912	0.3951	0.5921	0.5921	0.5921	0.6154	0.6553	0.672	0.7849	0.975
junit-3.7	0	0.1991	0.1991	0.4241	0.485	0.4566	0.4043	0.7206	0.4705	0.4567	0.5042	0.331	0.5515	0.5515	0.5515	0.582	0.6305	0.5877	0.7186	0.9107
	junit-3.7	junit-3.8	junit-3.8.1	junit-3.8.2	junit-4.0	junit-4.1	junit-4.2	junit-4.3	junit-4.3.1	junit-4.4	junit-4.5	junit-4.6	junit-4.7	junit-4.8	junit-4.8.1	junit-4.8.2	junit-4.9	junit-4.10	junit-4.11	junit-4.12

(C) Portrait Divergence Values for JUnit

Figure 8: Sample of The JUnit Case Study Analysis Results

both JMock and SLF4J, to 9 in JUnit. Generally, lower diameter values are desired since high values mean deep runtime stacks, which can complicate software maintenance and debugging.

Modularity: Inspired by the work in [16], a call graph modularity is defined as the ratio of its cohesion to its coupling values. A higher value indicates better software engineering practices. Overall, the examined software releases exhibited steady modularity growth over time.

Portrait divergence: The previous graph metrics can exploit the essential characteristics of the graph and its evolution. However, when changes are made in a software system while the topology does not change, i.e., the number of nodes did not change, the previous metrics fail to detect such changes. Therefore, along with the previous metrics, we used the portrait divergence metric that can compare two graphs despite the number and order of their nodes. It can quantify the changes in software evolution and represent them in network portraits, which subsequently provide insights on the execution path changes to aid integration tests. This metric is based on generating the graph portraits first using the execution paths of the call graph, graph invariant.

We observed that portrait divergence could efficiently quantify the changes among software releases. In JUnit call graph evolution, the metrics of each release and its color-coded portrait divergence matrix are listed in the Appendix. A higher number means a higher difference between the two releases. The value zero means that the two releases are identical. We observe that the two most different releases, i.e., the most changes, happened between releases 4.3 and 4.5, which are the ones that exhibited a sharp drop in the

number of execution paths after the sharp growth in these values in releases 4.1 and 4.3. Not only can portrait divergence quantify code change, but it identifies the locations of the changes on the function level, which we argue can assist software testers in understanding the system evolution.

Given the previous results, we can strongly suggest that network metrics can exploit and assist software evolution. It was also evident from the JUnit test case and its sharp growth in the numbers of nodes, edges, and execution paths followed by a sharp drop in these numbers that critical evolution cases could be captured and predicted using graph analysis. Not only do the metrics illustrate several structural changes across the releases of a single software system, but it was also evident that the structure was surprisingly similar among the examined software systems. With a quick look at the numbers in Table 2, one can observe the overall similarity across the examined software systems and their evolution. It was also evident from our visual analysis results that color-coded call graphs can assist developers in identifying system-path changes and planning integration tests accordingly, significantly reducing testing time and cost.

2.6 Conclusions and Future Work

In this chapter, we investigated the use of information theory for analyzing and visualizing software evolution. In particular, (1) we introduced a method to construct a static call graph for a given Java project, (2) we examined a set of metrics to capture the dissimilarities among several releases of a software system, and (3) provided the tool

support, GraphEvo, for our approach. We evaluated the approach and tool using 66 software releases of five popular Java open-source systems. The study illustrated that graph metrics can exploit the similarities and differences in the structure and evolution of a software system. GraphEvo can be used to re-produce our study using any open-source Java project with two or more releases.

Our near-future work will focus on increasing the number of software releases and systems. Moreover, we aim to study the similarities and differences among domain-specific applications to extract the minimum set of functions required to create a similar application, which has a wide range of applications in autonomous software systems and code generation.

2.7 Tooling

Here is the screenshot of the visualization we used as part of the software-level evaluation in Fig. 9 and Fig. 10.

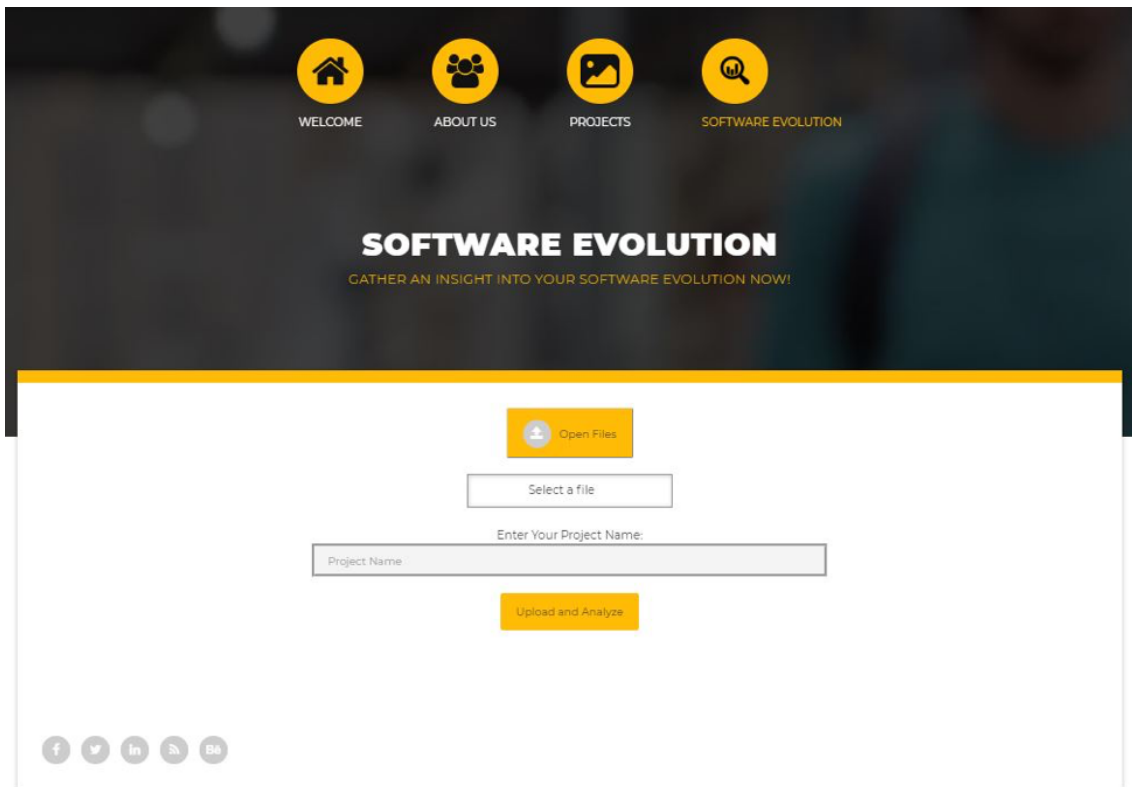


Figure 9: Software-level: Creating New Project

Table 2: Metric Values For The Subject Software Systems Based On Their Call Graphs

S/W	Release	Nodes	Edges	Paths	Avg-Degree	Clstring-Coef	Diameter	Modularity
JUnit	4.1	213	283	819	1.33	0.41	9	0.865
	4.3	370	573	1828	1.55	0.28	9	0.772
	4.5	361	490	1104	1.36	0.41	9	0.875
	4.7	414	573	1193	1.38	0.39	9	0.876
	4.9	439	611	1266	1.39	0.39	9	0.872
jMock	2.1	84	94	76	0.706	0	1	0.876
	2.2	85	95	76	0.702	0	1	0.881
	2.5	114	133	112	0.692	0	1	0.916
	2.6	127	117	118	0.673	0	1	0.926
	2.8	128	120	121	0.687	0	1	0.93
Jgap	3.5	1748	2336	4783	1.335	0.016	6	0.868
	3.6	1763	3218	5042	1.337	0.016	6	0.872
	3.6.1	1765	3219	5123	1.336	0.016	6	0.869
	3.6.2	1772	3242	5185	1.343	0.016	6	0.868
	3.6.3	1772	3242	5185	1.343	0.016	6	0.867
Guava	17.0	1981	2593	3604	1.098	0.043	4	0.943
	18.0	1987	2576	3466	1.092	0.042	4	0.943
	19.0	1975	2582	3466	1.1	0.042	4	0.936
	20.0	2218	2945	4277	1.094	0.041	5	0.945
	21.0	2288	3035	4416	1.097	0.04	5	0.943
SLF4J	1.6.2	11	32	31	1.091	0	1	0.119
	1.6.3	11	32	31	1.091	0	1	0.119
	1.6.6	11	32	31	1.091	0	1	0.119
	1.7.0	11	32	31	1.091	0	1	0.119
	1.7.25	13	37	36	1.154	0	1	0.152

Version Comparison

Version 1

Version 2

junit-4.0



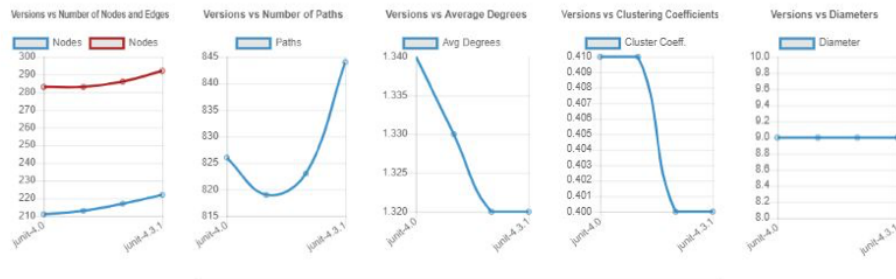
junit-4.2



Compare these two versions

Metrics

Version Name	Number of Nodes	Number of Edges	Number of Paths	Average Degrees	Clustering Coefficient	Diameter	
junit-4.0	211	283	826	1.34	0.41	9	View Call Graph
junit-4.1	213	283	819	1.33	0.41	9	View Call Graph
junit-4.2	217	286	823	1.32	0.4	9	View Call Graph
junit-4.3.1	222	292	844	1.32	0.4	9	View Call Graph



Results:junit

junit-4.3.1	222	292	844	1.32	0.4	9
junit-4.2	217	286	823	1.32	0.4	9
junit-4.1	213	283	819	1.33	0.41	9
junit-4.0	211	283	826	1.34	0.41	9
	junit-4.0	junit-4.1	junit-4.2	junit-4.3.1		

Figure 10: Software-level: Viewing Results

CHAPTER 3

DEFECT PREDICTION USING DEEP LEARNING WITH NPD FOR SOFTWARE EVOLUTION

3.1 Introduction

Software comprehension is an imperative and indispensable prerequisite for software development activities such as maintenance, testing, and quality management [1, 2]. As a software system grows, its functionality and components' interactions increase in size and complexity. Without a proper understanding of the software system and its inner interactions, adding or changing a feature increases the risks of introducing errors and potentially undesirable behavioral changes. This problem becomes even more complicated when investigating the system's evolution, i.e., comparing the changes among several software releases. Software development often includes specification, design and implementation, validation, and evolution. Software evolution refers to the process of creating, maintaining, and upgrading software for a variety of reasons, including requirement changes, faults, technical debt, and compatibility with other software. Software evolution has an impact on all essential processes in modern software development. Software evolution is responsible for 50%-90% of the total development cost [3, 4].

In practice, software developers and testers often investigate the software system releases to grasp and depict the functionality changes. Software developers specifically need to understand software changes introduced in a specific release while working on

fixing defects.

A software defect is a deviation from the software specifications or end-user expectations and can lead to unpredictable results or failures. [5] analysis has found that in 2018, low-quality software cost more than \$2.8 trillion in the United States alone, of which 16.87% was spent on known/fixed faults cost while 37.46% were software failure losses. Furthermore, this demonstrates that finding and correcting defects are costly software development activities caused by internal or external effects. Internal costs such as waste, scraping, and/or rework occur before software goes into production. The external costs of production-ready software include reproducing, discovering, fixing, and verifying defects and their locations.

[6] pointed out that detecting defects is a core challenge in software engineering because discovering defect patterns or defining appropriate measures to detect them is not trivial due to the complexity of software features and the diversity of software structure. Because software defects are expensive, the prediction of defects has been the research subject for the last several decades, leading to many tools and predictive models [7, 8].

These approaches help classify software artifacts (e.g., classes, modules, subsystems, and files) as being defect-prone or not. Even models predict defect density (the number of defects / SLOC) or the likelihood of a software artifact having a defect. Predicting malfunctions early in software development helps optimize the efforts of engineers and testers. Developers can prioritize code inspection and testing, starting with code areas having more defect-prone possibilities. Consequently, testers can efficiently utilize their limited resources and prioritize their test workflows.

Software defect prediction approaches are significantly cheaper than software measurement and reviews. Empirical studies have indicated that the probability of detecting software defects using prediction models could be higher than the possibility of discovery in current software reviews [9]. Many of these techniques, such as statistics-based models, parametric models, and machine learning-based models, are used [10, 11]. Defect prediction can improve software testing and has the potential to improve the overall software quality assurance process [12]. The recent advances in information theory have revolutionized the modeling and analysis of complex systems in many disciplines and practical problems, such as understanding biological systems [13], modeling the network's topology [14], and modeling software systems using complex graphs [15, 16]. Graph-based techniques enable capturing the structure and essential properties of the system. It can unleash various methods and tools to investigate and study the design patterns, detect abnormalities, and forecast new trends.

In this chapter, we aim to employ deep learning techniques on metrics and measurements extracted from software call graphs [17] to study and investigate software evolution and defect prediction. A call graph helps visualize the software system function calls in which each node in the graph corresponds to a function, and each directed edge depicts a function relationship [49]. The call graph contains its logical workflows and execution pathways composed of caller-callee relationships that may accurately depict the constantly evolving state of the system.

Specifically, we investigate whether graph-based techniques can facilitate the comprehension and analysis of software systems during software evolution using static call

graphs. Through this work, we answer the following research questions:

- **RQ 3.1:** Can Network Portrait Divergence, like other graph-based metrics, detect significant events in software evolution?
- **RQ 3.2:** How does the proposed software class-level metric Network Portrait Divergence compare to other metrics for software defect prediction?
- **RQ 3.3:** Can Network Portrait Divergence help to improve the prediction of software defects?

We hypothesize that network patterns and measures can help represent the software system as a graph. This can be analyzed to capture the software evolution’s essential properties and improve software quality through defect prediction. To prove our hypothesis, we study and analyze 29 open-source software systems, such as JUnit [35], Cassandra [50], Camel [51], ZooKeeper [52] over their entire lifespan, a total of 384 releases. We have chosen Java applications as our case studies so that our work is comparable to the most recent work in this domain [53, 19] and also because the majority of these applications are available as open-source projects with proper defect datasets.

We map the source code into a path-based execution model and analyze where and how software releases have occurred. We also investigate the impacts of changes identified by the level-based possible paths. This study consists of three main steps. First, we construct the static call graph for each release of the software system and extract software metrics. Second, we analyze the software metrics, including the Network Portrait Divergence, and select the set of metrics with the strongest relationship to the defects.

Third, we train two deep learning models for defect prediction: (1) a binary classifier to predict whether or not a specific software entity (class or module) includes a defect and (2) a regression model to estimate the number of defects in a given software entity.

Additionally, we provide a semi-automated tool to compare the call graphs and their metrics and visualize them to the developers and testers who can further build upon our results. We show that the applied graph-based methods and metrics can appropriately detect significant structural properties, quantify, and visualize the similarities and differences among several software releases. Our contributions can be summarized as follows:

- *We exploit graph-based metrics to study software evolution and identify code changes to aid software quality.* Our study illustrates that call graph analysis and graph-based metrics can help understand software evolution and identify code changes to the software releases. Specifically, we propose to answer questions on code changes such as: *Has the code structure changed significantly? and where.*
- *Our study distinguishes the metrics suited for defect prediction and compares the metrics to the Network Portrait Divergence.* We leverage this metric with existing software metrics as the base features to train deep learning models for defect prediction.
- *We implemented GraphEvoDef to help developers identify code modules that can have defects and predict defect counts.* We also provide an open-source Python tool to automate our study's analysis. We equip our chapter with a tool that can automatically (1) construct and visualize call graphs for a given Java code-base or

Jar files and (2) compare and visualize the analyses results for software evolution and defect prediction tasks.

3.2 Related Work

The recent rise of network data across different scientific domains has led to new tools, and methods [49] for understanding and evaluating networks. Although call graphs have promoted software comprehension tasks, they are still limited to capturing a single software system’s functionality at a time. However, as a software system evolves, understanding multiple releases’ similarities and differences become a crucial and daunting task. To this end, our research focused on processing multiple versions of a software system and measuring the changes based on an information-theoretic approach.

3.2.1 Software Metrics

- *Code and Complexity Metrics* [54, 55]: Complexity metrics indicate how complex a code block is. A module with a complex piece of code and many paths would have a higher risk. Researchers proposed and implemented many complexity metrics (i.e., the very well-known Cyclomatic Complexity (CC) metric of Thomas McCabe), mostly calculated based on the source code. CC metric measures independently testable paths for that module (method/class/package). Some examples of code metrics are lines of code and lines of comment. Examples of complexity metrics are *system complexity*, *McCabe* and Halstead’s *cyclomatic complexity*, and *essential complexity*.

- *Object-Oriented Metrics* [56, 57]: Object-oriented programming paradigm produces these metrics, and they work with the specific programming concepts, such as *inheritance*, *class*, *cohesion*, and *coupling*. Researchers have proposed several object-oriented metrics suites. The most popular one has been the Chidamber-Kemerer (CK) metrics suite. The CK suite has 6 metrics which are *Weighted method per class (WMC)*, *Coupling between object classes (CBO)*, *Lack of cohesion in methods (LCOM)*, *Depth of inheritance tree (DIT)*, *Number of children (NOC)* and *Response for a class (RFC)*.
- *Change or Process Metrics* [58, 59, 60, 61]: Changes made during the software development process are collected throughout the software life cycle across its multiple releases. Some process metrics are *code churn measures*, *change bursts*, and *code deltas*. The code churn metric shows how code evolves, while the change burst metric considers the sequences of the progressive changes [61]. The code delta metric computes the difference between two builds in terms of a specific metric, such as *code lines*.
- *Developer Metrics* [62, 61]: As each developer contributes to the software, these metrics are recorded. These metrics are the cumulative number of developers revising a module, the aggregate count of developers who changed the file over all the releases, and the developer's code.
- *Network Metrics* [63, 64, 65]: These metrics are the most recent ones for fault prediction. Dependency relation between different entities produces the network

metrics. The network nodes are classes or interfaces or any entities, while the directed edges represent dependency between the two such entities. The application of network analysis on this graph provides the network metrics values to find a correlation between the dependencies and defects. Some of these network metrics are *degree centrality*, *closeness centrality*, *betweenness centrality*, *eigenvector centrality*, *size measures*, *constraint measures*, and *ego network measures*.

These categories and their underlying software metrics signal a vast research area in software metrics selection. Better prediction models can be built when these subsets are combined appropriately.

3.2.2 Network Portrait Divergence

Portrait divergence [66] was recently developed to compare networks using their portraits. Unlike the previous ad-hoc comparison measures (see Section 3.3), *Network Portrait Divergence* is based on information theory, which provides a reliable interpretation of the divergence measure. Thus, it can compare the networks based on their topology structures and not assume that they are defined on the same nodes. Moreover, unlike the current expensive node matching optimization methods, Network Portrait Divergence is a graph invariant and is relatively computationally efficient. Note that this approach can treat both directed and undirected networks similarly.

To calculate the Network Portrait Divergence among the portraits of several releases of a software system, we need to

- Extract all possible execution paths of each of the software releases using their call

graphs

- Construct a set of network portraits for each of the releases based on their execution paths
- Calculate the portrait divergences among the software system releases under investigation using their portraits.

Existing works considered all nodes to have the same weightage [67]. This chapter will calculate Network Portrait Divergence for each class/module by iterating over classes and assigning weightage. We will describe this in detail in Section 3.3.

3.2.3 Defect Prediction and Deep Learning

Various methods have been developed to promptly predict the most probable locations of defects in large code-bases and can be categorized as either classification or regression models [68]. These focus on approaches that correlate with potentially defective code. Machine Learning techniques have serviced most of the defect prediction models. Those techniques derive features from software code and feed them to standard classifiers such as Support Vector Machine, Naïve Bayes, Random Forests, and Decision Tree. Researchers have been carefully designing features that can distinguish defective code from non-defective code, such as *code size*, *code complexity*, *code churn metrics*, *code change*, or *process metrics*, as described in the Software Metrics section [69, 70, 71].

[72] built a defect prediction model using decision tree regression (DTR) for CPDP and WPDP and found similar performances for both scenarios. With the help

of DTR, [73] could forecast the number of faults for intra- and inter-release situations. Additionally, several empirical studies [74, 75, 76, 72] showed that DTR showed the best performance among the other machine learning algorithms. Support Vector Regression (SVR) is an extension of support vector machine (SVM) that is based on the concept of structural risk reduction [77]. Many defect prediction empirical software engineering research has utilized the SVR approach [78, 79].

Deep learning is currently becoming more widespread in the field of software engineering. Zhao and Huang [80] proposed a new approach, DeepSim, to measure the code's functional similarities. They have used a deep neural network (DNN) model to learn semantic representation features and direct binary classification. Some researchers have also used word embeddings, followed by RNN and code semantics, to make code suggestions [81], while others have considered multi-class classification [82].

A novel deep neural network named Code-Description Embedding Neural Network (CODEnn) was proposed to help developers perform code search [83]. The DNN-based code search was built on high-dimensional vector space using code snippets and natural language descriptions. Some more approaches automatically discover discriminating features in the source code and help detect code clones [84]. The deep learning-based defect prediction model (i.e., DPNN model) introduced in [53], first obtained 11 metrics in the MIS dataset [85], and 21 metric variables from the KC2 dataset from the PROMISE repository [18], and then used a DNN regression model to learn features from the matrix and predict the number of defects. DPNN consists of two separate neural networks, one for the MIS dataset and one for the KC2 dataset, each with four layers and

11 and 21 inputs. SVR, FSVR, and DTR were all outperformed by DPNN. According to the authors, DPNN significantly reduces the mean squared error (MSE) and increases the squared correlation coefficient.

3.2.4 Call Graphs

Call graphs have been extensively studied to show the software system's inner interactions in function calls. Some existing studies either focus on generating and analyzing the call graphs of one system at a time [32, 33, 86] or focus on the power law of the node degree distribution. For example, the researchers in [31] built a tool to detect and visualize the static interactions between the classes of software systems written in Java; the work in [34] studied the call graphs of 223 releases of the Linux kernel to identify similar graph structures; while [16] analyzed the releases of well-known projects to evaluate the use of information theory in understanding software evolution. Other studies have focused on collaborative software graphs. The software components as software networks and systems were assessed based on design quality principles [87]. Code2Vec [88] has used Abstract Syntax Tree to extract all software paths as vectors and later aggregate them to form Code2Vec neural model. The neural model is employed to predict method names.

3.2.5 Studies on Network Comparison

Typical software networks such as call graphs have nodes representing functions; edges are the function calls between nodes and specific degree distribution. This network evolves with each software release as new functions are introduced, and old ones are

deleted or rewired, resulting in exhibiting small-world, scale-free network properties [44]. When comparing such networks, the methods may be divided into two categories: Known Node-Correspondence (KNC) and Unknown Node-Correspondence (UNC)[89].

- KNC - Both networks have a common node-set (or at least a portion), and their pairwise connection is known. As a result, only graphs of comparable size and application scope may be compared. This may be the case of some minor releases when no new functions are introduced to the code; it is only that certain functions have been rewired.
- UNC - Any two networks, regardless of their scale, density, or application sector, can be compared. These techniques often employ one more statistic, which then determines a distance. This might be the case for almost all versions where features are introduced or modified. It is more influenced when a few software versions undergo a complete revamp.

We see several attempts to quantify dissimilarities, such as social networks, time-evolving networks, biological networks, power grids, infrastructure networks, and software networks. [89] compared real-world datasets using the distances mentioned in Table 3 for directed/undirected and weighted/unweighted scenarios. With the same size and density networks, most distances in the table could achieve perfect classification. They also concluded that UNC distances, such as spectral distances, graph-based measures, and Network Portrait Divergence, are particularly well suited for structural comparisons because they provide information on the amount and significance of changes in graph

structures. Also, [90] systematically compared graph distances using package `netrd` [91] and found that the Network Portrait Divergence is well suited for real-world network comparisons.

Table 3: Classification Of Network Distances

Network Distance Name	Type	Description
Euclidean	KNC	The shortest distance between two points in N-dimensional space.
Manhattan [92]	KNC	The sum of the lengths of the line segment projections from the points onto the coordinate axes.
Canberra [93]	KNC	A numerical measure of the distance between two points in a vector space.
Jaccard [94]	KNC	A dissimilarity between sample sets.
DeltaCon [95]	KNC	A distance calculated using the fast belief-propagation method for determining node affinities.
Clustering Coefficient	UNC	A metric for how closely nodes in a graph tend to cluster together.
Diameter	UNC	The maximum distance between the pair of vertices.
DGCD-129 [96]	UNC	Between two networks, the Directed graphlet correlation distance (DGCD-129) is defined as the Euclidian distance between their upper triangle values in their Directed graphlet correlation matrices.
MI-GRAAL [97, 98]	UNC	A confidence score derived using multiple node statistics (degree, coefficient clustering, etc.) assigned to each pair, and then the nodes are aligned from the lowest to the highest score.
Network Portrait Divergence [66]	UNC	An information-theoretic measure for comparing networks by constructing graph-invariant distributions contained in portraits.

Most of the current approaches in this field examine the structural properties of a single software system at a time, including model dependency [99, 100], class collaboration graphs [99, 101], and inheritance graphs [102]. Other related works have discussed motifs [103, 104]. They cannot capture the changes between the same topology networks, i.e., the same arrangement of the edges and their directions between the nodes and different sets of nodes.

Other approaches [105] used graph edit distance matrices to measure the number of edges and nodes required to transfer one network into another. Similar to the primary approaches, distance measures are limited to only capturing the network topology. Advanced research in this area proposes the comparison of network subgraphs and motifs.

A motif is an interconnected set of nodes of a complex network of a given size and types [46]. The number of nodes in a motif represents its size, and the topology represents its type. The variations between the networks in these methods are examined based on differences in subgraph counts and patterns. The network structure across a diverse set of domains was explained using the structural diversity of real-world networks [106]. They suggested that the origin of a network, i.e., technological, social, or biological, could not necessarily be a factor in creating similar network structures.

Overall, our research surpasses the discussed related studies in two ways: First, we extend and combine several graph metrics to capture the structural and functional software evolution across several releases of popular Java-based open-source systems. Second, we provide a tool to facilitate this study’s reproduction using other software systems as test cases.

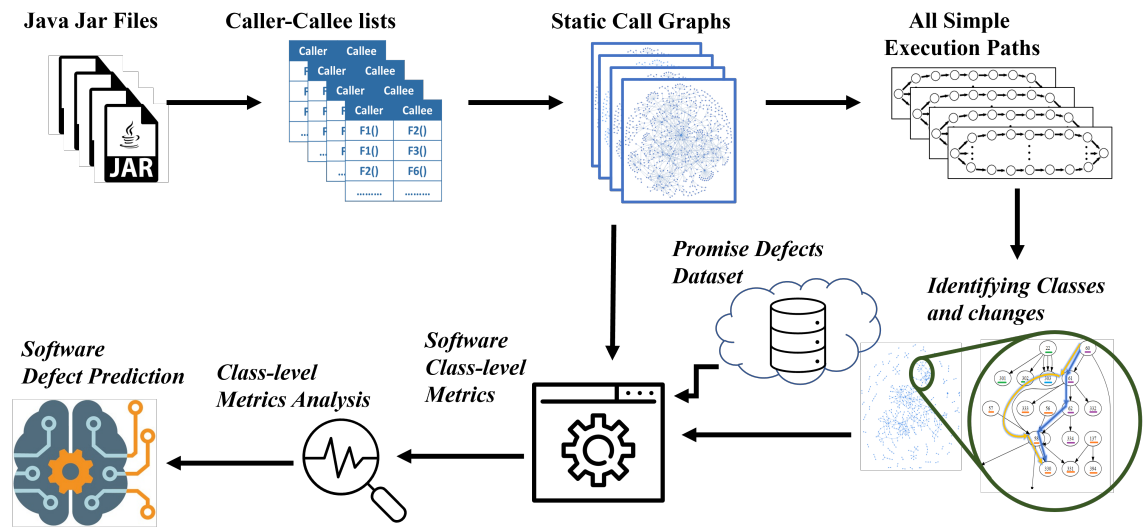


Figure 11: Overview of Our Approach

3.3 Proposed Work

Our study explores Network Portrait Divergence to identify significant events in software evolution, compare it to other software metrics, and apply it to software defect prediction.

We also equip our study with an open-source Python tool that implements our method; i.e., it can automatically visualize and define the software evolution for a variety of software releases with features to highlight variations in execution paths to help the software engineers identify and quantify software changes made in a given software release. For example, the tool can help answer the questions: *What are the new execution paths that are added to the software? And where?*

Our approach consists of two main steps, illustrated in Fig. 11. First, we construct a call graph for each release of the software system under investigation. While the call graph can explain the features and behavior of a single software system’s functionality, we aim to utilize call graphs in characterizing software evolution over time. Second, we study, analyze, and evaluate graph metrics across several releases to characterize the system’s evolution. In addition, we provide comparison and visualization features to help software engineers better understand the code and execution-level changes. Then, based on the evaluated metrics, we train and build defect prediction models. Before explaining the methods mentioned above in more detail, we first present the study subjects used in our evaluation in the following subsection.

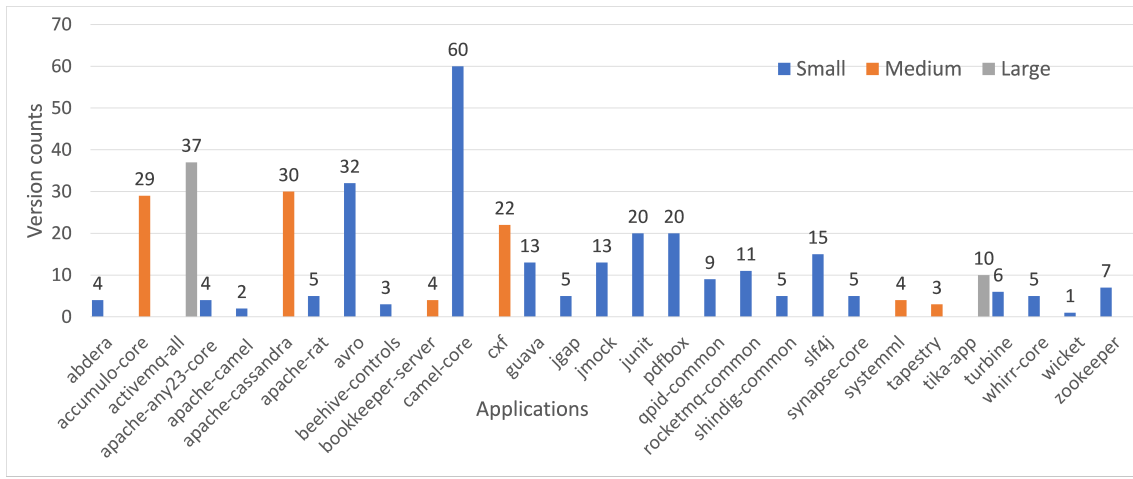


Figure 12: The Study Subjects: Applications and Their Versions

3.3.1 Study Subjects

The analysis is focused on 384 releases of 29 open-source Java software systems, including software libraries and applications, namely [35], [50], [51], [52], and more. A list of the study subject applications and the number of versions used in our study are listed in Fig. 12. These software systems have been chosen based on the following criteria: (1) they must be open-source systems written in Java, (2) they must have a minimum of six-month history with at least two versions of the software, (3) they must be of substantial size, thousands of source-line of code (KSLOC), and additionally (4) they are preferred to be highly rated and well-followed by developers. The search for these software systems was carried out using Google [47]. Fig. 12 also lists the application code-base size, which can be small, medium, and largely based on KSLOC. Small applications are those with less than 100 KSLOC, medium applications are those with 100 KSLOC to 500 KSLOC, and applications with more than 500 KSLOC are categorized as large applications.

We also used a dataset of 9 open-source Java-based projects from the PROMISE dataset [18]. Projects contain more than 4546+ classes and 1,251,619+ lines of code. The projects’ descriptive statistics are shown in Table 4. #Instances/Classes, #Count of Versions #LOC, #Buggy Instances, % of Buggy Instances, and #Defects are the number of instances or classes, the number of code lines, the number of buggy instances, the percentage of buggy instances, and the number of defects respectively. Each instance represents a class file that contains 21 static code metrics (e.g., CBO, WMC, RFC, LCOM), which present all the variables involved in our study. Table 5 lists the metric names and their description [107]. These metrics are used as dependent variables to analyze in the metric selection section.

Table 4: Details Of Java Projects For Metric Analysis (Promise Dataset)

No	Project	Versions	#Classes	LOC	# Buggy (% of Buggy Inst.)	# Defects
1	Ant	5	745	208,653	166 (22.3%)	338
2	Camel	2	965	113,055	188 (19.5%)	500
3	Ivy	2	352	87,796	40 (11.4%)	56
4	Lucene	3	340	102,859	203 (59.7%)	632
5	Pbeans	2	745	208,653	166 (22.3%)	338
6	Poi	3	442	129,327	281 (63.6%)	500
7	Velocity	3	229	57,012	78 (34.1%)	190
8	Xalan	4	885	411,737	411 (46.4%)	1,213
9	Xerces	2	588	141,180	437 (74.3%)	1,596

3.3.2 Constructing and Visualizing Call Graphs

A call graph is defined as a directed graph, $\vec{G} = (V, E)$, where V is the set of vertices, i.e., nodes of the graph, and E is the set of directed edges of the graph. The nodes represent functions in a software call graph, and the edges represent function

Table 5: Static Code Metrics

Metric Suite (Number of Metrics)	Acronym	Metric Full Name
CK suite (6)	WMC	Weighted method per class
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion in methods
	RFC	Response for a class
	CBO	Coupling between object classes
	NOC	Number of children
Martins metrics (2)	CA	Afferent couplings
	CE	Efferent couplings
QMOOM suite (5)	DAM	Data access metric
	NPM	Number of public methods
	MFA	Measure of functional abstraction
	CAM	Cohesion among methods
	MOA	Measure of aggregation
Extended CK suite (4)	IC	Inheritance coupling
	CBM	Coupling between methods
	AMC	Average method complexity
	LCOM3	Normalized version of LCOM
McCabe's CC (2)	AVG_CC	Mean values of methods within the class
	MAX_CC	Maximum values of methods in the class
Others (3)	LOC	Lines of code
	NPD	Measures the software complexity changes
	BUG	Non-buggy or buggy

calls. Note that the term “function” in our research encompasses the different types of program procedures, including class member functions (also known as methods), static, and stand-alone functions. Edges are ordered pairs of nodes, $e = (u, v)$, where each edge is considered to be directed from node u to node v . The node u , initiating a call, is named the caller function, and the node v is named the callee. For a vertex u the number of incoming calls also called arrows, is called the *indegree* and denoted $deg^-(u)$, and the number of the outgoing calls, arrows, is called the *outdegree* and is denoted $deg^+(u)$. A

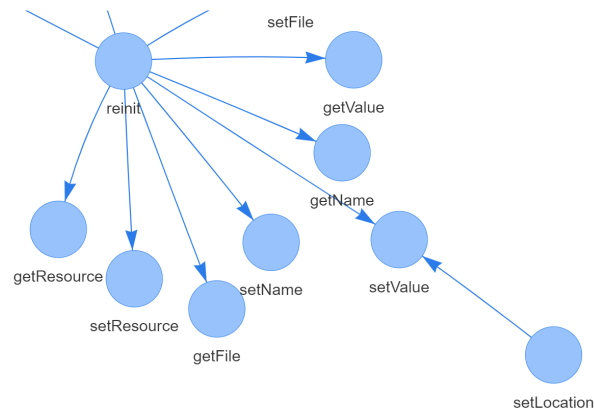


Figure 13: Snippet of The Call Graph Generated by GraphEvo

vertex with $deg^-(u) = 0$ is called a source; we call it an entry point as it is the origin of the outgoing calls and represents the entry point for an execution path. Similarly, a vertex with $deg^+(u) = 0$ is called a sink; we call it an exit point since it is the end of the incoming calls and represents the end of an execution path. A vertex with $deg^+(u) = 0$ AND $deg^-(u) = 0$ is called an isolated node. Isolated nodes are represented in the call graph. However, they are not included in any execution path and need further investigation by the developers since they represent unused functionality in the system.

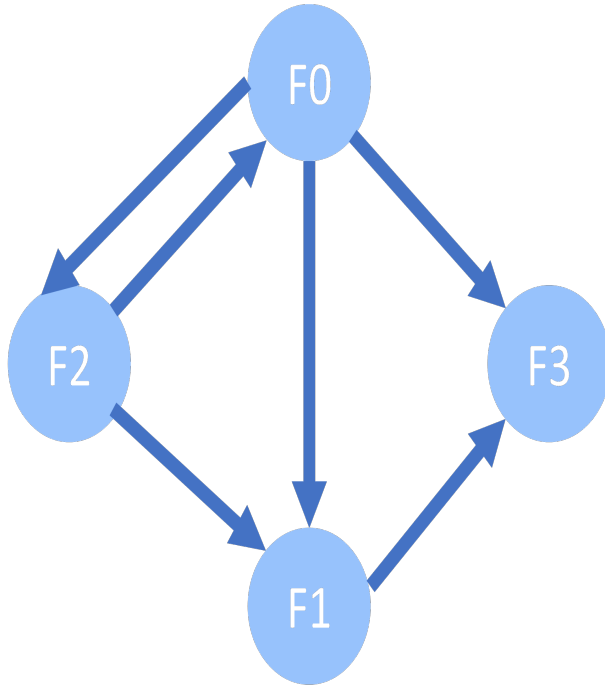
Nevertheless, visualizing a single software system's call graph has proven to facilitate understanding its behavior and functionality at the source-code level [86]. In order to visualize the call graph of a single system, we translate its edges and nodes from the OOP language that they were written with to a graph description language, *DOT* [41], which in return can be rendered using different tools, such as *Graphviz* [42], to actual graphs in *jpg*, *svg*, or *pdf* formats. Fig. 13 shows part of the call graph generated for our tool.

3.3.3 Characterizing Software Evolution

Network Portrait Divergence [66] was developed to compare networks using their portraits. Unlike the previous ad-hoc comparison measures, Network Portrait Divergence is based on information theory, which provides a reliable interpretation of the divergence measure. Thus, it can compare the networks based on their topology structures and not assume that they are defined on the same nodes. Moreover, unlike the current expensive node matching optimization methods, Network Portrait Divergence is a graph invariant and is relatively computationally efficient. Note that this approach can treat both directed and undirected graphs similarly.

3.3.3.1 Constructing Network Portraits

Network portrait is an efficient way to capture and visualize several structural properties of a given network (or a call graph in our case). We use the output of the previous step (i.e., step 1: extracting the call graph and its execution paths, an example shown in Fig. 14), to construct the network portraits. The network portrait B is defined as an array with (l, k) elements, such that $B_{l,k} \equiv$ *the number of nodes which have k nodes at distance l* for $0 \leq l \leq d$ and $0 \leq k \leq N - 1$, where the distance is the length of shortest path, and d is the graph diameter, and N is the number of nodes in the graph. Note that a distance $l = 0$ is admissible. It is also worth mentioning that network portraits are identical for the same graph despite the nodes' labels or orders. We illustrate the pseudocode to construct the network portraits in Algorithm 1, which results in a matrix that encodes several structural properties of the graph, including the number of nodes in



(a) A Call graph example

- F0 -> F1 -> F3
- F0 -> F2 -> F1 -> F3
- F0 -> F3
- F1 -> F3
- F2 -> F0 -> F1 -> F3
- F2 -> F0 -> F3
- F2 -> F1 -> F3

(b) A list of possible execution paths

Figure 14: Example of Call Graph with Execution Paths

the graph in the zeroth row, the degree distribution in the first row, and then the degree distributions of the next nearest neighbors and so forth. Also, the network portraits are graph invariant, i.e., they assign equal values to isomorphic graphs. The next step explains that network portraits are important for graph comparison.

3.3.3.2 Comparing Call Graphs using Network Portrait Divergence

Network Portrait Divergence between two graphs G and G' , $D_{JS}(G, G')$, is defined using Jensen-Shannon divergence in Equation 3.1.

$$D_{JS}(G, G') \equiv \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M) \tag{3.1}$$

Algorithm 1: Constructing Network Portraits

```
1 procedure NETWORK_PORTRAITS(paths)
2   counter  $\leftarrow$  count(paths)
3   while counter  $\neq$  0 do
4     path  $\leftarrow$  pathList(counter)
5     pathLength  $\leftarrow$  length(path)
6     startNode  $\leftarrow$  start(path)
7     lastNode  $\leftarrow$  last(path)
8     nodeEntry.StartNode  $\leftarrow$  startNode
9     nodeEntry.EndNode  $\leftarrow$  lastNode
10    nodeEntry.distance  $\leftarrow$  pathLength
11    portraitFreq.Add(nodeEntry)
12    counter  $\leftarrow$  counter - 1
13  end
14  networkPortrait  $\leftarrow$  PathsByLength(portraitFreq)
15  return networkPortrait
```

where $M = \frac{1}{2}(P + Q)$ is the mixture distribution of P and Q . Here, KL is defined in Equation 3.2 and P and Q are defined in Equation 3.3.

$$KL(P(k, l) || Q(k, l)) = \sum_{l=0}^{\max(d, d')} \sum_{k=0}^N P(k, l) \log \frac{P(k, l)}{Q(k, l)} \quad (3.2)$$

Where the log is base 2.

$$p(k, l) = p(k|l)P(l) = \frac{1}{N} B_{l,k} \frac{1}{(\sum_c n_c^2)} \sum_{k'=0}^N k' B_{l,k'} \quad (3.3)$$

Where n_c is the number of nodes within the connected component c , the sum $\sum_c n_c^2$ runs

over the number of connected components and the n_c satisfy $\sum_c n_c^2 = N$. Likewise for $Q(k, l)$ using B' instead of B . Selecting two nodes is random with replacement, and the probability that they are at a distance l from one another is given in Equation 3.4.

$$p(\text{distance } l) = \frac{\# \text{ paths of length } l}{\# \text{ paths}} = \frac{1}{(\sum_c n_c^2)} \sum_{k=0}^N k B_{l,k} \quad (3.4)$$

The Network Portrait Divergence $0 \leq D_{js} \leq 1$ quantifies the differences between two networks using a single value: the higher the value, the less similar the two networks are. Two identical networks have a Network Portrait Divergence of 0. Network Portrait Divergence receives the desirable qualities, including symmetric and normalized from Jensen-Shannon divergence. Furthermore, Network Portrait Divergence is applicable to both directed and undirected networks. In this chapter, we used the Network Portrait Divergence to measure the software evolution over time using its network portraits extracted from each release's call graph.

3.4 Results and Discussions

This section answers the research questions and discusses our experiments' results and findings. Before that, however, we explain the experiments and their setup. In particular, we introduce two experiments: one studies the significance of the mentioned software metrics and compares them, while the other uses these metrics to build models for defect prediction.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	VERSION	CLASS_NAME	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	Ce	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC	AVG_CC	MAX_CC	NEWONE	PORTRAIT	DEFECT_CNT	FAULTY	
2	ant-1.3	AntClassLoader	17	2	0	2	64	76	0	2	9	0.8792	713	0.6	0	0.8272	0.3393	1	3	40.0588	3	10	1	0.0199	2	Defective	
3	ant-1.3	BuildEvent	11	2	0	3	15	13	0	3	11	0.75	97	1	0	0.2	0.2208	0	0	7.2727	5	1	1	0.0041	0	NotDefective	
4	ant-1.3	BuildException	14	4	0	1	28	0	0	1	14	0.3846	153	1	0	0.7586	0.3333	1	2	9.7857	5	2	1	0.0137	0	NotDefective	
5	ant-1.3	BuildListener	7	1	0	1	7	21	0	1	7	2	7	0	0	0	1	0	0	0	12	1	0	0	0	NotDefective	
6	ant-1.3	BuildLogger	4	1	0	1	4	6	0	1	4	2	4	0	0	0	0.5	0	0	0	23	1	0	0	0	NotDefective	
7	ant-1.3	Constants	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	NotDefective	
8	ant-1.3	DefaultLogger	14	1	0	4	32	49	0	4	12	0.8352	257	1	0	0	0.3077	0	0	16.8571	8	6	1	0.0015	2	Defective	
9	ant-1.3	DesirableFilter	2	1	0	0	6	1	0	0	2	2	45	0	0	0	0.6667	0	0	21.5	63	8	0	0	0	NotDefective	
10	ant-1.3	DirectoryScanner	23	1	0	2	51	181	0	2	14	0.7314	1407	1	0	0	0.2909	0	0	59.6957	11	35	1	0.1543	0	NotDefective	
11	ant-1.3	FileScanner	13	1	0	0	13	78	0	0	13	2	13	0	0	0	0.3269	0	0	0	21	1	0	0	0	NotDefective	

Figure 15: Example Illustrating A Snippet of The Dataset Featuring Metrics from The Ant System, Version 1.3.

3.4.1 Experiment 1: software metrics comparison

Several features (software metrics) are essential for building an effective predictive model. The features' relevance can be evaluated individually through a univariate, fast, and straightforward approach. However, before building (i.e., training) a predictive model, we wanted first to study the usefulness of the software metrics listed in Table 5.

Here, we utilize the ANOVA (Analysis of variance) test that selects the features with the most substantial relationship to the output variable. ANOVA uses one or more categorical independent features and one continuous dependent feature. It provides a statistical test of whether the mean of several groups is equal.

Software predictive models mainly use supervised learning techniques, which require large amounts of data to train a reliable model. However, if there is no or insufficient past data for a particular software application, then a training set can be modeled using external projects with known defect information. This type of predictive model is called Cross-Project Defect Prediction (CPDP). Otherwise, if the software holds sufficient defect data from all of its releases and is used to form the training set, then this strategy is called Within-Project Defect Prediction (WPDP).

Fig. 15 shows the snippet of the training dataset where each row represents a class

of software and a list of class-level metrics as the independent variables. For the classification task, the target attribute is the column DEFECTIVE, the binary class, whether or not each class is defective. In the regression task, only the target variable is changed to DEFECT_CNT, the number of defects. Using a training set strategy and type of problems, we arrived at four scenarios and analyzed the metrics performances.

- **Classification problems on CPDP:** Building a classification model based on the training data, i.e., labeled data of external projects, and predicting the defect labels of unlabeled modules within the target project.
- **Regression problems on CPDP:** Building a regression model based on the training data, i.e., historical labeled software modules, and then predicting the count of defects for unlabeled modules within the target project.
- **Classification problems on WPDP:** Building a classification model based on the training data, i.e., historical labeled software modules, and then predicting the defect labels of unlabeled modules within the same project.
- **Regression problems on WPDP:** Building a regression model based on the training data, i.e., historical labeled software modules, and then predicting the count of defects for unlabeled modules within the same project.

3.4.2 Experiment 2: Training Predictive Deep Learning Models

We demonstrate through the training of two different deep learning models the usefulness of the software metrics that we studied—precisely the effect of network portrait

divergence on improving the performance of these models. We train two models for two tasks: defect classification and regression. Defect classification predicts whether or not a code entity contains a defect, i.e., a binary classification task. Defect prediction is a regression task that estimates the number of defects in a given code entity. We collected and processed the dataset for both tasks and then trained and evaluated the models. We briefly discuss each of the experiment steps in the following:

Data collection: First, we had to identify proper case studies (i.e., open-source applications), which we utilized the PROMISE dataset for this task [18, 19]. PROMISE includes lists of Java projects with some software metrics and defect information. For all case studies, we calculated network portraits at the class level. We also used our tool to extract the missing metrics. The number of features added up to 22 distinctive features. The data entries reached 4,796 (creating a table of size 4796x22). For simplicity, we have stored and organized these data in a *csv* file.

Data Processing: We normalized the data into smaller range values suitable for the neural network learning process. We utilized the Scikit Learn library, Standard Scaler function to standardize the features (subtracting off the mean and scaling to unit variance); for a vector input, x the standard scaler is determined as $z = (x - u)/s$, where u is the average of the training samples, and s is their standard deviation.

Neural network architecture. We experimented with various neural network structures (i.e., the different organization of dense layers) to find an optimal architecture that yielded the best performance results. Note that we used the same neural network architecture but changed the objective functions to train two separate models, one for

Table 6: Metric Values for The Selected Software Systems Based on Network Portraits

S/W	Release	Nodes	Edges	Paths	Avg-Degree	Cl-Coeff	Diameter	Modularity
Ant	1.3	591	965	8392	0.036	3.26	12	0.72
	1.4	824	1389	15835	0.032	3.37	13	0.7
	1.5	1347	2248	33616	0.028	3.34	15	0.73
	1.6	1838	3408	61638	0.025	3.71	17	0.69
	1.7	3445	5145	21381	0.018	2.99	17	0.84
JUnit	4.1	213	283	819	1.33	0.41	9	0.87
	4.3	370	573	1828	1.55	0.28	9	0.77
	4.5	361	490	1104	1.36	0.41	9	0.88
	4.7	414	573	1193	1.38	0.39	9	0.88
	4.9	439	611	1266	1.39	0.39	9	0.87
jMock	2.1	84	94	76	0.706	0	1	0.88
	2.2	85	95	76	0.702	0	1	0.88
	2.5	114	133	112	0.692	0	1	0.92
	2.6	127	117	118	0.673	0	1	0.93
	2.8	128	120	121	0.687	0	1	0.93
JGAP	3.5	1748	2336	4783	1.335	0.016	6	0.87
	3.6	1763	3218	5042	1.337	0.016	6	0.87
	3.6.1	1765	3219	5123	1.336	0.016	6	0.87
	3.6.2	1772	3242	5185	1.343	0.016	6	0.87
	3.6.3	1772	3242	5185	1.343	0.016	6	0.87
Guava	17.0	1981	2593	3604	1.098	0.043	4	0.94
	18.0	1987	2576	3466	1.092	0.042	4	0.94
	19.0	1975	2582	3466	1.1	0.042	4	0.94
	20.0	2218	2945	4277	1.094	0.041	5	0.95
	21.0	2288	3035	4416	1.097	0.04	5	0.94
SLF4J	1.6.2	11	32	31	1.091	0	1	0.12
	1.6.3	11	32	31	1.091	0	1	0.12
	1.6.6	11	32	31	1.091	0	1	0.12
	1.7.0	11	32	31	1.091	0	1	0.12
	1.7.25	13	37	36	1.154	0	1	0.15

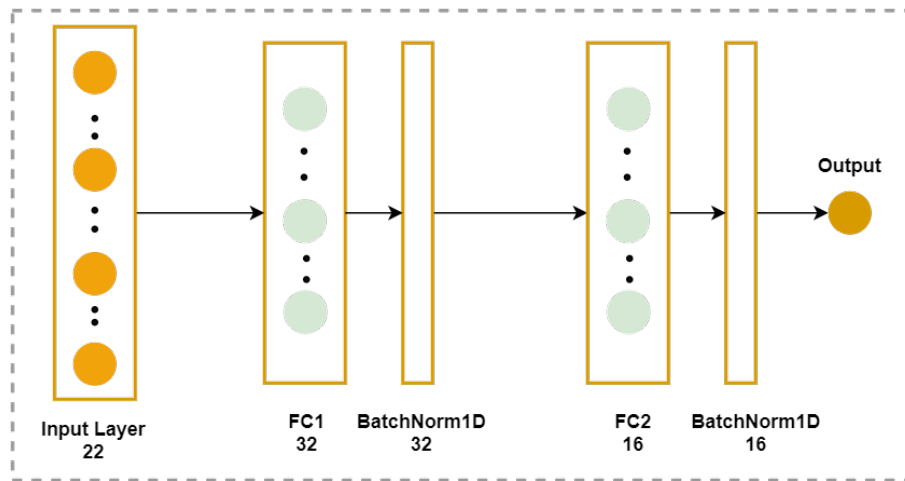


Figure 16: Neural Network Structure Overview

classification and one for regression. The architecture consists of a set of fully connected layers (FC) sorted in the following order, as shown in Fig. 16:

- **Input Layer:** dimension (layer size): 22 (equivalent to the number of features).
- **FC1:** A fully connected layer (aka., dense layer) with 32 neurons followed by a BatchNorm1D layer of size 32. Activation: ReLU (Rectified Linear Unit).
- **FC2:** A fully connected layer with 16 neurons followed by a BatchNorm1D layer of size 16. Activation: ReLU.
- **Dropout:** a dropout function with a probability of 0.3.
- **Output Layer:** one neuron.

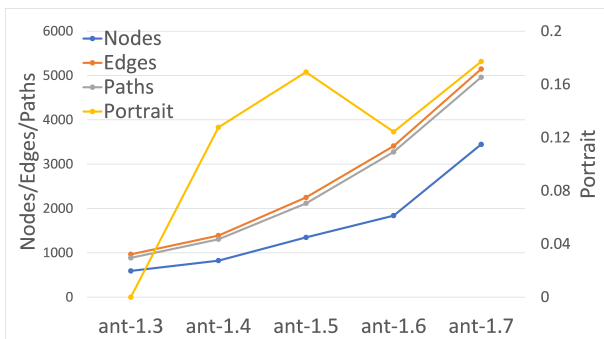
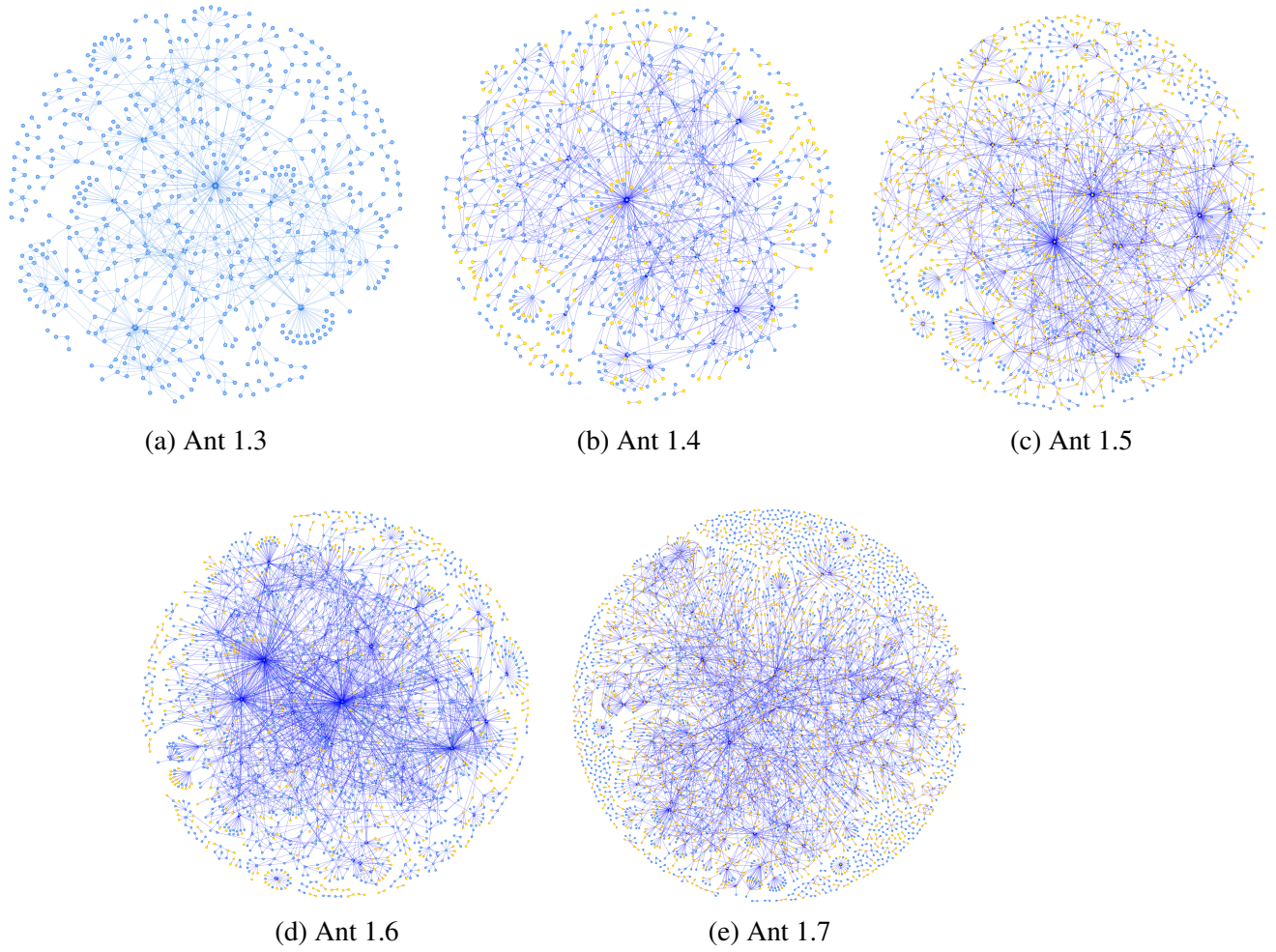
For the classification model, we used the Binary Cross-Entropy loss function with logits and the Adam optimizer with the default value for the learning rate, i.e., 1×10^{-3} . For the regression model, named GraphEvoDef, we used the loss function Smooth L1

that uses a squared term if the absolute element-wise error goes below 1 or an L1 term otherwise. It is less susceptible to outliers than MSE and avoids exploding gradients. We used Adam as the optimizer function and assigned it the default learning rate 1×10^{-3} .

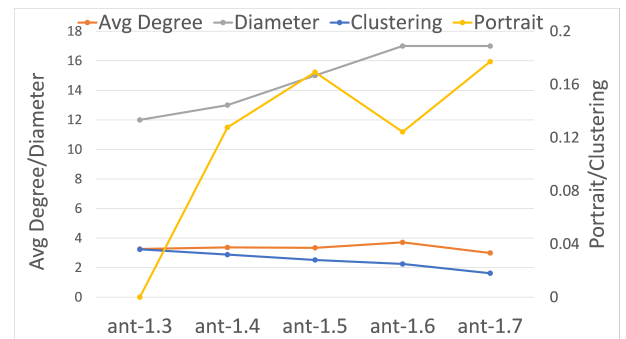
RQ 3.1 Can Network Portrait Divergence, like Other Graph-based Metrics, Detect Significant Events in Software Evolution?

This section answers this question by discussing the examined software systems' results. Specifically, we show how the changes in graph metrics over several releases can indicate *non-obvious events* in the software evolution, which could affect various software engineering concerns. We attempted to provide an answer at both software and class levels. When engineers want to see changes in the overall software, they typically look at metrics at the software level. Changes at the class level are more granular and precisely correspond to the engineer's concerns. For example, when working on a module with multiple classes, class changes are more important for engineers.

Software level. The metrics results for our case studies, including the number of nodes, the number of edges, the number of execution paths, the average degree of the graph, the clustering coefficient, the graph diameter, and the modularity ratio, are listed in Table 6. Note that the table includes the results for only five releases of six case studies due to space constraints. The complete results list is available on our website <https://vijaybw.github.io/graphevodef/>. Our tool can visualize these values in addition to the graph comparison and Network Portrait Divergence in a user-friendly web-based interface. Fig. 17 depicts the color-coded visualization of the call graph generated from



(f) Count of Nodes, Edges, Paths and NPD



(g) Avg Degree, Diameter, Clustering Coefficient and NPD

Figure 17: Color-Coded Visualization Of Ant Call Graph Evolution

5 versions of the software Ant as a case study. The nodes represent the functions, while edges represent the interactions among them. We also highlight the newly added functions with different colors, as shown in Fig. 17 with zoom in and zoom out features. It also outlines the software metrics plot with the Network Portrait divergence value on a line chart.

We first observed the increase in the number of nodes, edges, and execution paths as the software evolved—which is a natural behavior during software evolution. Some systems exhibit linear growth in these metrics: jMock (an Expressive Mock Object Library for Java) and JGAP (Java Genetic Algorithms Package). However, Guava witnessed the highest growth among the five systems in the releases 20.0 and 21.0, while its size decreased from 132.96 KSLOC to 106.85 KLOC. We believe this change happened due to code refactoring since the number of nodes and edges sharply grew in these two releases compared to the previous releases while the system’s overall size decreased. This implies that significant code changes had happened to increase system quality. It was also apparent that the most used releases of the SLF4J system were stable and did not grow over the first four releases compared to a slight increase in the last release.

Execution Time Overhead The experiments were conducted on a Windows 10 machine, 16 GB RAM, Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 GHz. The local server was implemented using Flask, and the tasks were implemented using Python 3.6. The user interface was developed in JavaScript. We measured the time cost for each software we used for metrics analysis.

Our main goal is to measure the time needed to produce call graphs with the

Table 7: Time Cost for S/W in Seconds(Promise Dataset)

No	Project	Versions	Avg Time Per Class	Avg Time Per KLOC
1	ant	5	9	39
2	camel	2	5	62
3	ivy	2	7	43
4	lucene	3	6	29
5	pbeans	2	1	4
6	poi	3	34	121
7	velocity	3	4	11
8	xalan	4	7	20
9	xerces	2	4	14

software metrics calculation and UI elements. Table 7 shows that software poi took the longest time per class and per KLOC. This is because software poi has many classes with a smaller line of codes. This execution time consists of various steps such as uploading and reading Java jar files, calculating call graphs and their writing to UI elements, and calculating and storing software metrics. Table 7 also shows that it takes an average of 9 seconds to process a class and 38 seconds to process one thousand lines of code.

Network Portrait Divergence. The metrics we discussed above could exploit essential characteristics of the graph and its evolution. However, the previous metrics fail to detect such changes when changes are made at the code level without structural changes, i.e., the number of nodes did not change. Therefore, along with the previous metrics, we used the Network Portrait Divergence metric to compare two graphs despite their nodes' number and order. We observed that Network Portrait Divergence could quantify the change in software evolution and represent it in the form of network portraits, which subsequently provide insights into the execution path changes.

Table 8: Network Portrait Divergence Values for Ant

Version	1.3	1.4	1.5	1.6	1.7
NPD	NA	0.1277	0.1692	0.1243	0.1771

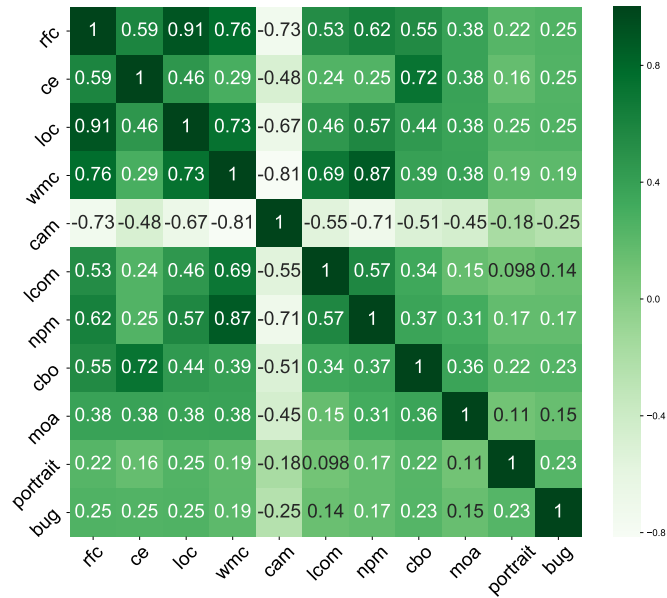
We found that Network Portrait Divergence can measure changes between software releases efficiently. For example, Table 8 shows the Ant release’s metrics and its color-coded Network Portrait Divergence. A higher number means that every two adjacent releases are more distinct. The zero value means that the two releases are the same. The highest portrait value is coded in red, the least in green, and the remainder of the numbers have a gradient color. We note that the two most different releases, i.e., the most changes, happened between releases 1.6 and 1.7; these are the ones that exhibited a sharp drop in the number of execution paths after the sharp growth in these values in releases 1.4 and 1.6. Not only can Network Portrait Divergence quantify code change, but it also identifies the changes’ locations on the function level, which can help understand system evolution.

The metrics indicate a range of structural changes over the releases of a single software system, but the structure was remarkably similar amongst the software systems examined. As seen in Table 6, one can observe the overall similarity across the most examined software systems and their evolution. In addition to many major releases, this table also shows some minor releases for the software JGAP and SLF4J, which have fewer or zero structural changes. Metrics typically change more for major releases due to the addition or modification of functionalities. Minor releases often contain defect fixes which are less likely to include structural changes. The Network Portrait Divergence

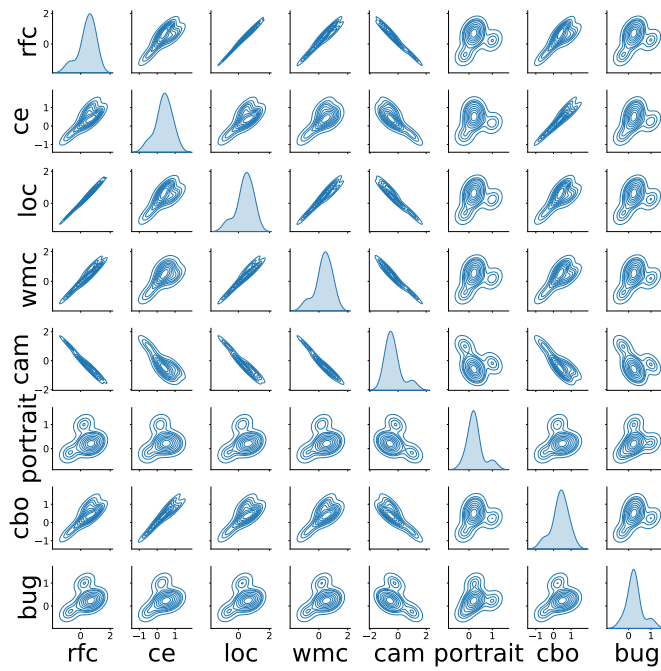
shifted in proportion as the Ant expanded in terms of functions and new paths. As Ant progressed from version 1.6 to 1.7, there were major improvements, such as the addition of 80% new nodes (functions) compared to the previous version, which was reflected by the significant change in the Network Portrait Divergence as well.

Class level. We examined the 22 software metrics at the class level and conducted Spearman's correlation analysis on all nine software systems. As illustrated in Fig. 18(a), the Network Portrait Divergence exhibits a weak correlation with *LOC*, *RFC*, and *CBO* but a marginal relationship with *NPM* and *CE*. Kernel Density Estimation (KDE) is a well-established statistical method for studying distributional data characteristics that yields a continuous function that estimates the data's density distribution [108, 109]. We can see the relationships between these metrics as shown in Fig. 18(b). Also, we infer that the *Network Portrait Divergence* is positively correlated to *RFC*, *CE*, *LOC*, and *CBO*. The *Network Portrait Divergence* also negatively correlates with *CAM*. Except for *LOC*, these five metrics are calculated by counting the length of one sub-paths in specific scenarios, bringing them closer to *Network Portrait Divergence*. We compared the *divergence of Network Portrait Divergence* to these five metrics and two additional graph-based metrics: *edges* and *execution paths*. The edges represent the number of function calls into and out of the class. The number of execution paths is the number of paths that pass through the class's functions.

Table 9 outlines the analysis of four classes from the *Ant* software, and the findings are discussed in this section. These classes range in size from small to large. Class



(a) Spearman's comparison of metrics



(b) KDE Plots of metrics

Figure 18: Pairwise Comparison of Top 10 Metrics

Table 9: Software Ant: Class-level Comparison with Network Portrait Divergence

Class: org.apache.tools.ant.taskdefs.AntStructure					
Version	Ant-1.3	Ant-1.4	Ant-1.5	Ant-1.6	Ant-1.7
DEFECTS	0	1	0	1	0
PORTRAIT	0	0.129	0.169	0.125	0.177
Edges	8	10	10	12	4
Execution Paths	8	11	11	15	6
CBO	6	7	7	7	7
RFC	55	60	61	61	40
NPM	3	3	3	3	4
Ce	6	7	7	7	7
LOC	643	748	758	749	871
Class: org.apache.tools.ant.taskdefs.Delete					
Version	Ant-1.3	Ant-1.4	Ant-1.5	Ant-1.6	Ant-1.7
DEFECTS	0	0	0	3	5
PORTRAIT	0	0.131	0.171	0.126	0.179
Edges	9	9	9	11	31
Execution Paths	63	67	97	125	78
CBO	8	10	11	28	37
RFC	49	52	56	98	134
NPM	16	17	19	38	40
Ce	8	10	11	28	37
LOC	579	699	715	890	1129
Class: org.apache.tools.ant.taskdefs.ExecuteOn					
Version	Ant-1.3	Ant-1.4	Ant-1.5	Ant-1.6	Ant-1.7
DEFECTS	0	1	0	0	2
PORTRAIT	0	0.134	0.176	0.132	0.183
Edges	16	21	21	43	59
Execution Paths	73	115	169	683	367
CBO	12	16	17	20	25
RFC	42	55	56	77	103
NPM	5	9	10	16	20
Ce	12	16	17	20	25
LOC	395	730	757	1164	1279
Class: org.apache.tools.ant.DirectoryScanner					
Version	Ant-1.3	Ant-1.4	Ant-1.5	Ant-1.6	Ant-1.7
DEFECTS	0	0	0	2	3
PORTRAIT	0.154	0.141	0.173	0.117	0.173
Edges	32	57	53	69	96
Execution Paths	601	1259	2779	2221	234
CBO	2	2	6	9	10
RFC	51	52	67	119	142
NPM	14	14	21	28	31
Ce	2	2	6	9	10
LOC	1407	1489	1171	1739	2382

AntStructure underwent significant refactoring in version 1.7; previously, it had been updated consistently. To be precise, *AntStructure* was split into three additional classes at version 1.7, and all of its connections to other classes were rewired. *RFC* and *LOC* reflect the appropriate metrics changes, whereas *Network Portrait Divergence* reflects the appropriate bump to account for the *execution path* changes.

Class *Delete*'s code-base and connections have steadily grown with each software release. It had become a component of numerous execution paths 97 and 125 in versions 1.5 and 1.6. And later, it was reduced to 78, roughly equivalent to version 1.4. Version 1.7 receives three times the number of function calls in or out but eliminates execution paths. This behavior indicates that the class is not doing too many activities. All metrics reflect the correct changes, and *Network Portrait Divergence* also reflects them.

Class *ExecuteOn* had a 47% increase in *execution path* changes, which *Network Portrait Divergence* can detect despite no change in other metrics. Additionally, the number of *execution paths* was significantly reduced in version 1.7, while the number of edges was increased. This behavior could indicate that the class has been extended with new functions, and all existing *call-in/call-out* have been rearranged. *RFC*, in collaboration with *Network Portrait Divergence*, has correctly identified the changes.

Class *DirectoryScanner* is a large class with numerous execution paths. From versions 1.3 to 1.6, we can see that all of the metrics increased steadily. The *execution paths* were significantly reduced in version 1.7, while the *edges* were increased. All metrics constantly changed, including *Network Portrait Divergence*. The *portrait number* is the same in versions 1.5 and 1.7; this could be because *execution paths* in version 1.5

Table 10: Software Lucene: Class-level Comparison with Network Portrait Divergence

Class: org.apache.lucene.analysis.TokenFilter							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2	7	4	1	1	13	0.0001	0
2.2	8	4	1	1	13	0.1628	2
2.4	9	6	1	2	20	0.1465	3
Class: org.apache.lucene.analysis.TokenStream							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2	19	4	1	3	7	0.0167	1
2.2	24	5	1	4	9	0.1422	4
2.4	28	17	2	5	72	0.151	7
Class: org.apache.lucene.analysis.TopDocCollector							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2	7	13	5	4	110	0.0038	0
2.2	7	13	5	4	110	0.1629	2
2.4	7	13	5	4	117	0.147	1

were increased by 120%, while *execution paths* in version 1.7 were reduced by 90%, and *edges* were increased by 39%.

Table 10 outlines the analysis of four classes from the *Lucene* software, and the findings are discussed in this section. Class *TokenFilter*'s code-base and connections have slightly increased with each software release. In versions 2.2 and 2.4, its coupling between other classes was changed. But, we see little change in metrics other than CBO, RFC, and NPD. We also see that adding an extra public method in version 2.4 NPD can capture the structural change.

Class *TokenStream* had a steady increase in a number of public methods changes, which *Network Portrait Divergence* can detect despite the slight change in other metrics. Additionally, the coupling of class with other classes is also increased significantly and correctly captured by NPD.

Class *TopDocCollector* is an average class with typical coupling between other classes. From versions 2 to 2.4, we can see that only NPD showed almost no change while other metrics remain the same. NPD has identified the structural changes that happen within the software. Within the code-base, we identified that surrounding classes underwent significant changes, affected the execution paths, and resulted in changes in NPD.

The *Network Portrait Divergence* can attain new variations in some scenarios where typical class-level metrics may show little changes. Also, we may see different results in some scenarios where changes were made to the class and significant changes to its connections. *Network Portrait Divergence* can provide new insights that can help identify significant events in software evolution and help understand some simple software evolution tasks such as relating features to code changes and the rationale behind refactorings.

Table 11 outlines the analysis of four classes from the *Xalan* software, and the findings are discussed in this section. Class *QueryParameter*'s code-base and connections were modified in version 2.6 and followed by a minor update in version 2.7. But, we see that the NPD did update to a more significant value even with no changes in other software metrics, which means that the other software classes have contributed to execution path changes. We also find a similar pattern in the case of class *ProcessorDecimalFormat*.

Class *ElemExsltFuncResult* is a small class with typical coupling between other classes. From versions 2.4 to 2.7, we can see that only NPD showed almost no change while other metrics remain the same. NPD has recognized the structural alterations that

Table 11: Software Xalan: Class-level Comparison with Network Portrait Divergence

Class: org.apache.xalan.lib.sql.QueryParameter							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2.4.0	1	6	0	5	30	0.0004	0
2.5.0	1	6	0	5	30	0.1107	1
2.6.0	3	21	0	12	344	0.0541	0
2.7.0	3	21	0	12	344	0.0452	1
Class: org.apache.xalan.processor.ProcessorDecimalFormat							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2.4.0	6	9	5	1	27	0.0038	0
2.5.0	6	9	5	1	27	0.1108	2
2.6.0	7	13	6	1	35	0.0555	0
2.7.0	7	13	6	1	36	0.0452	1
Class: org.apache.xalan.templates.ElemExsltFuncResult							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2.4.0	10	15	7	5	62	0.0032	1
2.5.0	9	13	6	5	54	0.1122	2
2.6.0	9	15	6	4	49	0.0542	1
2.7.0	9	16	6	4	64	0.0459	1
Class: org.apache.xalan.templates.ElemWhen							
VERSION	CBO	RFC	CE	NPM	LOC	PORTRAIT	DEFECTS
2.4.0	10	16	9	6	55	0.0003	0
2.5.0	9	16	8	6	55	0.1104	1
2.6.0	9	16	8	6	55	0.0532	0
2.7.0	9	16	8	6	56	0.045	1

occur within the program. We discovered that adjacent classes received considerable modifications, affecting execution paths and resulting in changes in NPD.

Class *ElemWhen*'s code-base and connections have hardly changed from release 2.4 to 2.7. NPD showed almost no change while other metrics remained the same. We found that changes to neighboring classes had a significant impact on execution paths and hence on NPD.

Additionally, the number of *execution paths* was significantly reduced in version 1.7, while the number of edges was increased. This behavior could indicate that the class has been extended with new functions, and all existing *call-in/call-out* have been rearranged. *RFC*, in collaboration with *Network Portrait Divergence*, has correctly identified the changes.

It had become a component of numerous execution paths 97 and 125 in versions 1.5 and 1.6. And later, it was reduced to 78, roughly equivalent to version 1.4. Version 1.7 receives three times the number of function calls in or out but eliminates execution paths. This behavior indicates that the class is not doing too many activities. All metrics reflect the correct changes, and *Network Portrait Divergence* also reflects them.

RQ 3.2: *How does the proposed software class-level metric Network Portrait Divergence compare to other metrics for software defect prediction?*

Building a predictive model requires several features. As discussed in Section 3.3, we did experiments to identify top-performing metrics for software defect prediction. The first experiment was to find the top 10 software metrics. The ANOVA test selects the features with the most significant relationship to the output variable. ANOVA F-score

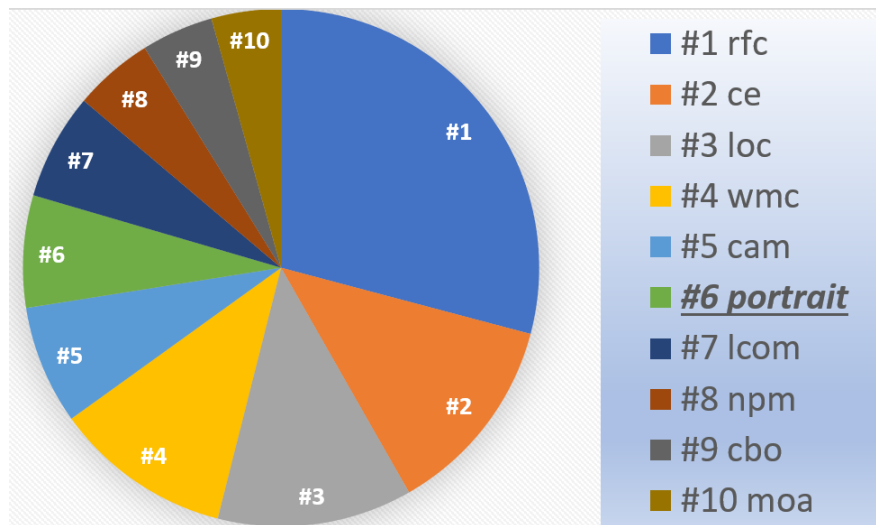


Figure 19: Top 10 Metrics

is calculated separately for all investigated projects for the classification and regression tasks. Fig. 19 illustrates the used features, while Fig. 20 depicts the analysis results of the selected features. According to cumulative performance ranking Fig. 19, all metrics (RFC, CE, LOC, WMC, CAM, PORTRAIT) are highly correlated with the number of defects for both CPDP and WPDP. As shown in Fig. 19, we have found that the Network Portrait Divergence metric exists in the top six important metrics referring to the association with the number of defects.

Secondly, we wanted to find the influential metrics in 22 software metrics. We performed the univariate and features importance analysis for all investigated projects for the tasks classification and regression separately. Fig. 20 shows graphs outlining the metric analysis for CPDP and WPDP scenarios. CAM, LOC, and RFC are the most influential metrics. We also could see that Network Portrait Divergence is performing well for the classifications scenario.

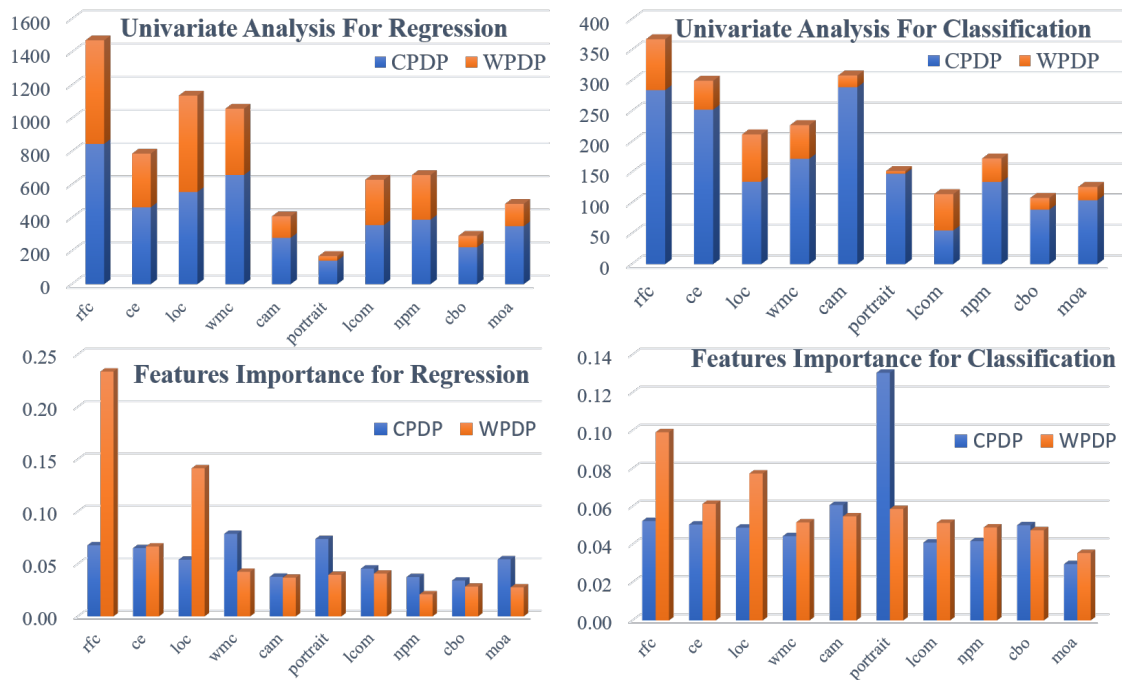


Figure 20: Univariate and Features Importance Analysis

[110] provides details on which software metrics were used in software defect prediction models. Each constructed model used only several metrics (usually five to ten) in the regression task. Nine or more defect prediction models have used the metrics (*LOC*, *RFC*, *CBO*, *AMC*, *CA*, *LCOM*). In comparison, the least used metrics were *CMB*, *DAM*, and *WMC* and were only used for one defect prediction model. *LOC*, *RFC*, and *CAM* are also weakly correlated with *Network Portrait Divergence*, demonstrating the overall significance of *Network Portrait Divergence* usage.

RQ 3.3: Can Network Portrait Divergence help to improve the prediction of software defects?

As discussed in the answers to RQ 3.1 and RQ 3.2, we could determine the usefulness of Network Portrait Divergence for predicting software defects. We built deep learning models for the classification and regression tasks described in subsection *Experiment 2*.

To assess the performance of our models, we used the measure $F1$ for the classification task and MSE and R^2 for the regression task. The $F1$ score is calculated as Equation 3.5, and it transmits the balance between the precision and the recall of the model. Mathematically, precision is the number of true positives divided by the number of true positives plus false positives. The recall is the number of true positives divided by the number of true positives plus the number of false negatives. For example, given a highly-imbalanced dataset where the number of bug-free classes is much higher than the number of bug classes, a model can predict only class 0 (i.e., no bug) and still achieve very high accuracy, sometimes even in the 90% range. Therefore, the $F1$ score can provide a much more accurate evaluation of our prediction model.

$$F1 = \frac{2 \times (precision \times recall)}{(precision + recall)} \quad (3.5)$$

We used MSE and R^2 for the regression task. The MSE score calculates an average squared discrepancy between the predicted results \hat{y} and the actual true value y as shown in Equation 3.6, where n is the number of total data samples.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.6)$$

Using an input function of an independent variable, the proportion of the variance

of the dependent variable is determined by R^2 . It is determined by applying the association between the real y and the predicted \hat{y} values in a regression model. In other words, the percentage of the response variable variation is explained by a linear model, as seen in Equation 3.7.

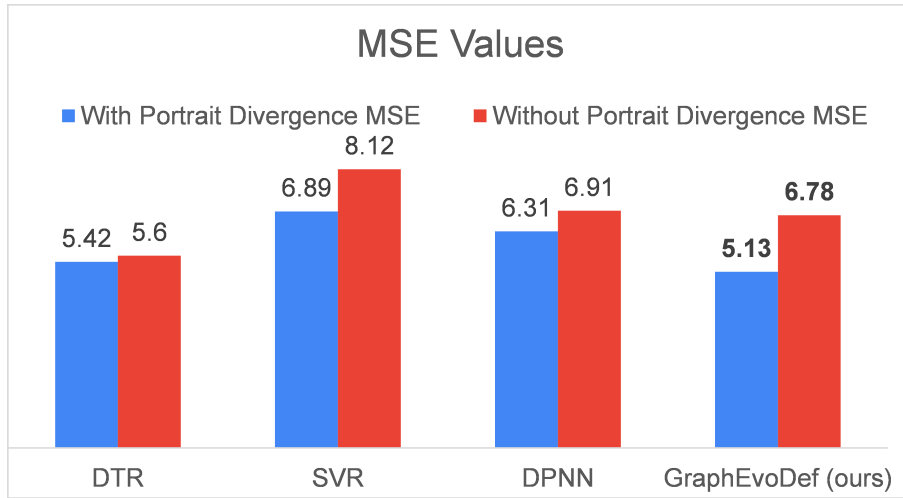
$$R^2 = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad (3.7)$$

Our defect classification model evaluation produced an overall accuracy of 94.48% with an F1 score of 0.76. The confusion matrix is shown in Table 12, actual and predicted values for the model. We consider these results acceptable, given the fewer features considered in this study, which prove informative. We argue that increasing the dataset size would increase our performance significantly. These results are also better than the models produced in similar studies, as we show later.

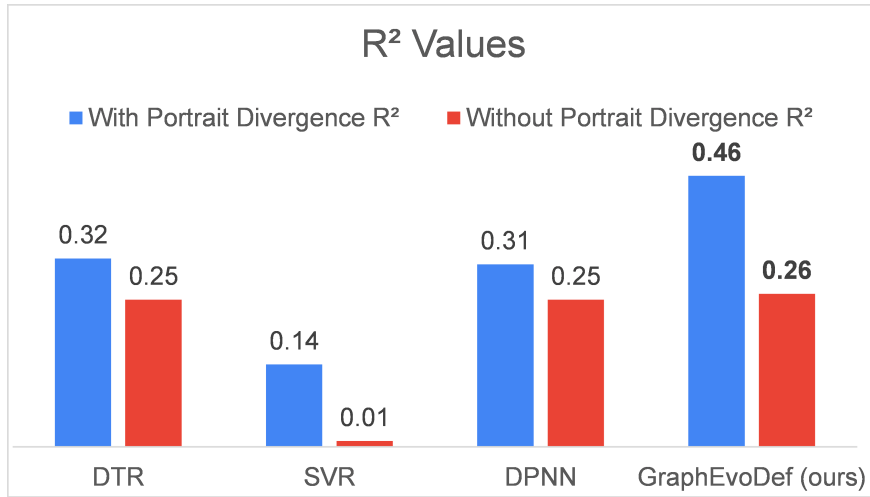
Table 12: Classification Confusion Matrix

Actual	Predicted	
	Non-defective	Defective
Non-Defective	1835	234
Defective	434	1094

We used the MSE and R^2 measures on the predicted results to measure the defect regression model’s overall performance. We also compared our neural network results to the results produced by the models: DTR, SVR, and DPNN. These three models were chosen for the following reasons. First, these state-of-the-art methods give automated predictions of defects in software modules. Second, to the best of our knowledge, these methods are more accurate than other comparable approaches [53, 74, 111, 75, 76]. We tested all the aforementioned models on the dataset described above. All experiments



(a) MSE Values



(b) R² Values

Figure 21: Defect Prediction Evaluation Results

were performed using NVIDIA Titan RTX GPU with 64G RAM. We implemented our models using PyTorch and monitored the experiments using an internal model management tool called ModelKB [112, 113, 114, 115].

The evaluation results are shown in Fig. 21. When the Network Portrait Divergence metric is considered, the proposed approach achieves better results than DTR, SVR, and DPNN in MSE . In contrast to selecting a metrics set, selecting a modeling technique appears to have less impact on the MSE values of the model. Furthermore, our approach outperforms other approaches in terms of R^2 by a substantial margin. Compared to the other models, our model provides a better fit to the data that has been observed. As shown in Fig. 21, the Network Portrait Divergence leads to better results in both MSE and R^2 values.

Network Portrait Divergence has contributed to the model is a better fit, proving its usefulness for Software defect prediction. Low MSE and high R^2 are typically desirable for the deep learning models. Our method improves on the state-of-the-art approaches by 18% in terms of MSE and 48% in terms of the R^2 . Based on the findings, we can conclude that the proposed approach is accurate and outperforms the current approaches significantly.

3.5 Tooling

Here is the screenshot of the visualization we used as part of the class-level evaluation as shown in Fig. 22, Fig. 23, and Fig. 24.

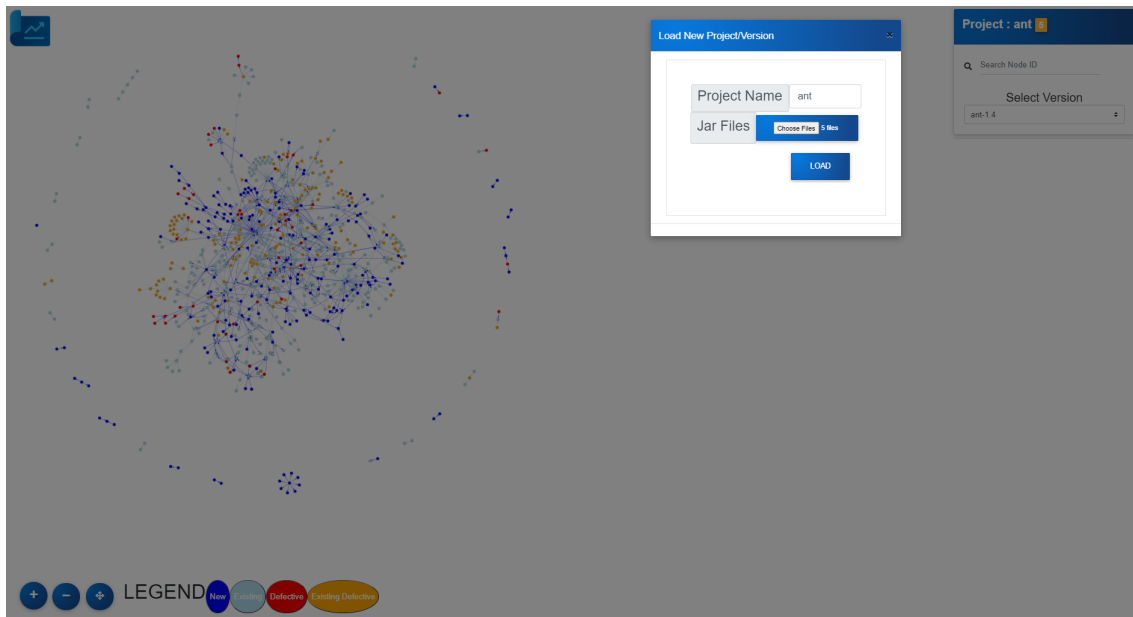


Figure 22: Class-level: Creating New Project Through UI And Uploading Jar Files

3.6 Threats to Validity

Our research methodology is comprised of various steps, including the identification of software versions, the construction of call graphs, the comparison of metrics, the creation of models, and the analysis of relevant studies [116]. We briefly describe the threats to validity in this section. We begin by discussing construct validity, followed by the internal and external validity threats.

3.6.1 Construct Validity

Construct validity of a study assesses if the conclusions are likely erroneous due to incorrect concept engagement, incorrect modeling, or misleading data.

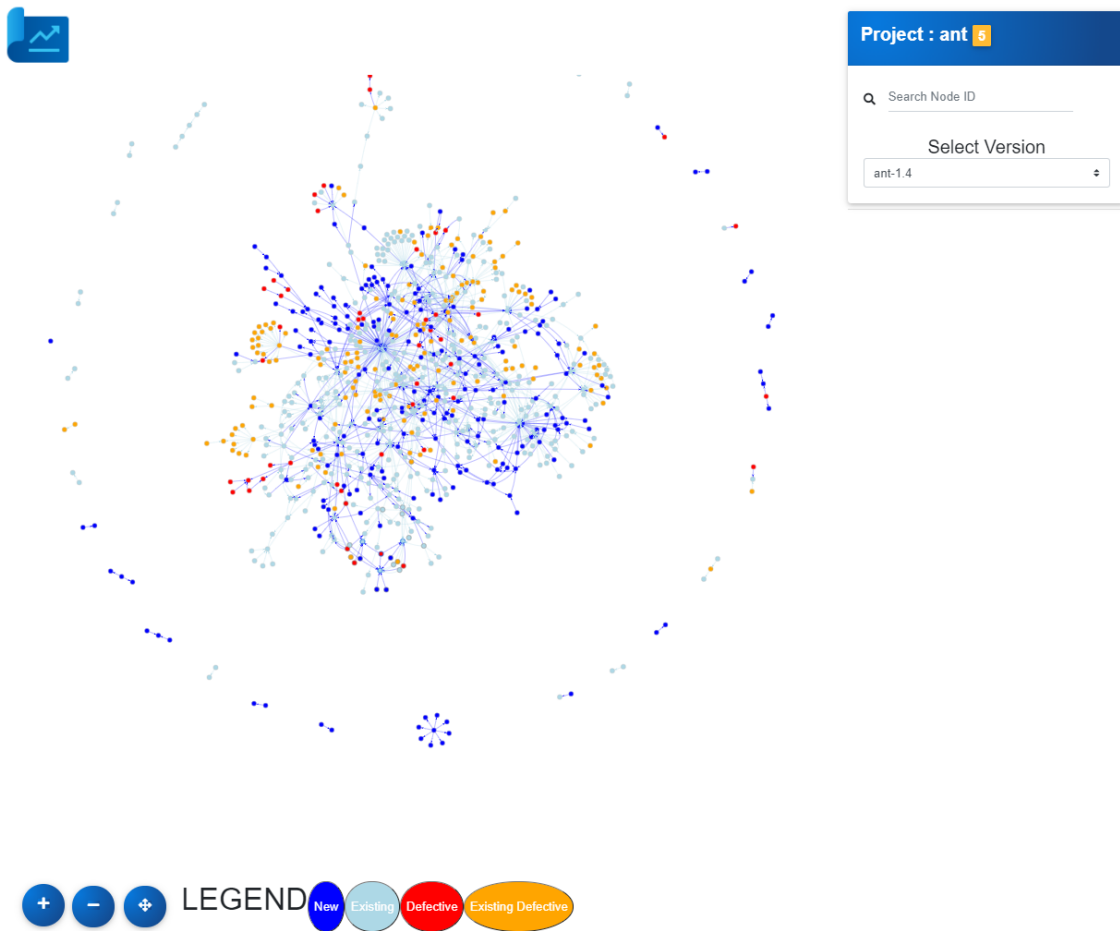


Figure 23: Class-level: Viewing Complete Call Graph

Our approach does not account for code changes made within a function without affecting the overall function calls since we study software evolution using the software structure using call graphs. Such code changes might introduce new defects that are not detectable using our tool and software metrics. If certain defects occurred due to incorrect logic within a given method and were fixed without affecting its connectivity to other methods, the Network Portrait Divergence would not change. However, this specific challenge is out of the scope of our study. It can be addressed in future work

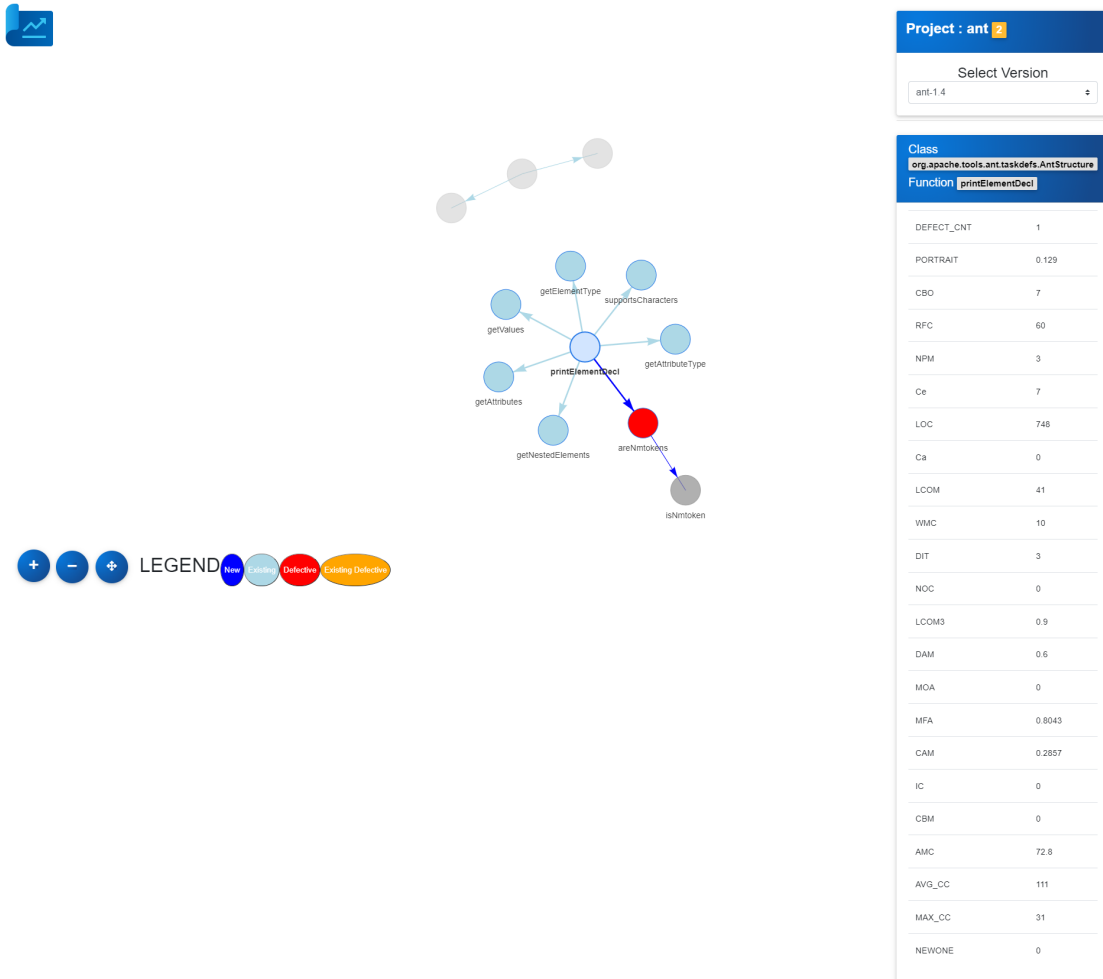


Figure 24: Class-level: Viewing Class Metrics

by incorporating dynamic call graphs that require running the actual application with particular test cases.

3.6.2 Internal Validity

Threats to internal validity concern the causal relationship between study design and execution artifacts. As a result, it may include uncontrolled or unmeasured variables

and those introduced during the study’s execution.

The use cases collected and studied in this chapter were not gathered from a single location; hence, their metrics and defect data were inconsistent. We did not validate these collected datasets for correctness (e.g., Was the number of defects associated with a software release correct?). However, after extensive manual efforts, we completed these datasets and ensured they all included the features needed to train the deep learning models. Moreover, we developed GraphEvoDef to automate this process in the future.

3.6.3 External Validity

As part of this study, we identify two potentially similar external threats. One threat is that the study subjects do not entirely cover all software development areas. The number of product versions and defects discovered will vary by domain and language. Nevertheless, our work aimed to show that applying network comparison advancements to Java-based systems is feasible and helpful. Another threat is that the granularity will undermine the validity of our findings. Our experiments drew on two distinct data sources, each containing metrics at the function and class levels. As a result, it is impossible to claim that network metrics collected at the function level are adequate indicators of defective classes in large and complex systems. However, we argue that the number and diversity of the studied subjects in this work were sufficient to prove the overall usefulness and meet the study targets, as explained in Section 3.4.

3.7 Conclusions and Future Work

We discussed in this chapter the use of Network Portrait Divergence to identify significant events in software evolution, how it compares to other software metrics, and how it can be used to predict software defects. Particularly, we implemented (1) a framework to construct call graphs and calculate their Network portrait divergence, (2) evaluated several metrics to detect discrepancies between several software system releases, and (3) we presented a semi-automated tool named GraphEvoDef, for assisting software engineers in predicting defects and improving software quality.

We studied 384 software releases of 29 open-source Java systems. The study found that graph metrics would take advantage of similarities and disparities in software system structure for evolution comprehension. We also studied the importance of software metrics. We found that the Network Portrait Divergence metric is helpful for identifying significant events in software evolution and its application to defect prediction. Following that, we built defect prediction models. Compared to existing techniques, our models achieved an 18% reduction in the mean square error and a 48% increase in the squared correlation coefficient. The review findings indicate that the solution suggested with GraphEvoDef is accurate and can enhance state-of-the-art approaches. We have also implemented an application that can replicate our study with two or more releases through any Java open source project. The rest of the study results and charts are listed on the tool's website: <https://vijaybw.github.io/grapevodef>.

Our future work will extend in several dimensions: we will focus on building a complete dataset with a more significant number of software systems and release the

dataset for further applications in this domain. We also plan to investigate the impact of class-based metrics on the tests accompanying the code. Another possible direction is to investigate the code pull requests made in a software version control system to collect more metrics about the software and its evolution.

CHAPTER 4

NPD-BASED TOOLING, EXTENDIBLE DEFECT DATASET AND ITS ASSESSMENT

4.1 Introduction

Every software has just enough defined and implied customer requirements or expectations in the program. Software typically consists of specification, design and implementation, validation, and evolution. Software evolution involves building, maintaining, and upgrading software for requirements, defects, technical debt, and compatibility. Software evolution influences all current software processes. These expectations shift with new features and defect fixes in every release. Additionally, each software change can introduce new defects. The consequences of defects on business are enormous because of the glorified or ruined company's brand image. It is possible to overrun costs, delays, and higher maintenance expenses [6]. Predicting defects early in the release cycles can help promote efficient management of testing resources and improve overall software quality [117, 118].

Open-source software has gained more and more popularity worldwide in the last decade. GitHub has emerged as a winner in hosting millions of such software and creating abundant data. Researchers have been probing through GitHub and coming to some insightful conclusions. PROMISE [18, 19] and Metrics Data Program (MDP) [20] from NASA are the popular publicly available datasets used for Software Defect Prediction

(SDP). The researchers investigated different approaches to SDP and determined whether a new approach improves current ones [21, 22, 23, 24]. Only a few datasets have been the foundation for further research from too many datasets.

Additionally, some of the work done on the dataset is not publicly available, and researchers must collect and process such data. Some of the datasets built on top of the popular datasets would lack repeatability [21]. To build a dataset, we chose 19 Java projects as our subject systems, which differed in various ways (size, domain, reported errors). We then collected defect details and downloaded the Java binary files for different versions from GitHub. Then, we ramped up our custom-coded utility GraphEvoProcessor; we processed these applications and generated the 21 class-level software metrics to create GraphEvo Dataset. Walunj et al.[67] introduced the Network portrait divergence (NPD) metric, which makes capturing and visualizing many structural features of a call graph-based network convenient [119, 88]. These 21 software metrics include the NPD metric, which helps measure path-based changes that happen to the class[66, 89]. We also made this utility available on Github so researchers can onboard new java projects. Researchers would need to configure the utility with java jar files and class-level defect mapping.

We also ran experiments to see if the GraphEvo dataset suited SDP. We gathered software metrics at the class level for each piece of software and its version. Following the dataset's creation, we assessed its value using several machine learning techniques. At the class level, the best algorithms generated F-score above 0.7. Additionally, we compared GraphEvo's defect predictability to other publicly accessible datasets such as BugHunter [120] and Github Bug dataset [121] and discovered that it is comparable with them in

terms of defect predictability. Our research tried to answer the following two research questions:

- **RQ 4.1:** *Is the GraphEvo dataset generated through NPD-based tooling intended for SDP? Which families of algorithms or algorithms are best suited for SDP?*
- **RQ 4.2:** *Is the GraphEvo DataSet powerful? How does the GraphEvo dataset perform compared to the GitHub Bug and the Bug Hunter datasets?*

The rest of the chapter is organized as follows: Section 4.2 discusses the latest state-of-the-art research articles in datasets and techniques for SDP. The network portrait divergence metric and the GraphEvoProcessor utility will be discussed in Section 4.3. The process of constructing the dataset and the data’s semantics are explained in Section 4.4. Section 4.5 exhibits the GraphEvo Dataset’s evaluation through the applied machine learning algorithm results. Finally, we mention threat of validity in Section 4.6 and wrap up the chapter in Section 4.7.

4.2 Related Work

Detecting and removing defects is critical for software quality maintenance. Most SDP techniques are based on an approach that constructs a predictive model based on previous errors. Typically, the methods identify the potential locations of defect-prone areas within the codebase and are known as classification or regression models [67, 122]. Many of these researchers either create their datasets for specific experiments or use existing ones [121, 120]. Different SDP methods need different sources of information as

predictors, resulting in new defect datasets [123]. These public datasets are negligible compared to the number of studies devoted to defect prediction [124].

Many researchers keep their datasets private, presumably due to confidentiality agreements. In addition, several researchers are working on combining several datasets into one for improved potential use [123]. We compiled a wide range of software listings from [18, 121] and collected Java binary files for all software versions having defect information at the class level. We wrote a utility to generate different software metrics, including object-oriented metrics such as CK Suite network metrics. We ran the utility for software listing and generated a dataset with class-level defects count and software metrics.

Various software metrics are used, including object-oriented, shift or process-related, developer-specific, network-based, and so on [125, 126]. These types and the underlying software metrics signal a large research field of software metrics selection [71, 127]. Combining the different subsets yields improved prediction models. The existing graph metrics can use the critical properties of the graph and its changes. However, when modifications are made to a software system while the topology remains constant, i.e., the number of nodes remains constant, the preceding metrics fail to identify such changes. As a result, in addition to the preceding measures, we utilized the Network portrait divergence metric, which can compare two graphs regardless of the number and order of their nodes [66]. It can quantify changes in software evolution and portray them as network portraits, giving insights on execution path modifications to help identify significant changes. This measure is based on first creating graph portraits utilizing the call

graph execution paths, graph invariant. Our dataset has used a new metric called Network portrait divergence, which was introduced in our previous work [67].

The most popular datasets for SDP for Java-based codebase are PROMISE, Eclipse Bug Dataset, BugsCatcher, GitHub Bug Dataset, and BugHunter. Zimmermann et al. [128] gathered bug alert stats from publicly available Bugzilla bug reports. They also created structural metrics based on AST for file-level faults to detect pre and post-release problems. PROMISE is one of the largest software engineering research data repositories with KC2 and KC3, Java-based. D’Ambros et al. [129] investigated pre and post-release defects using CVS, SVN, Bugzilla, and Jira, resulting in 71 software metrics from five Java applications. Hall et al. [130] showed that such code smells are only associated with defects under particular conditions and that their influence on faults is minimal. The Bugcatchers collected information from Bugzilla and Jira.

Table 13: Related Work

Research work	Advantages	Disadvantages
BugHunter dataset [120]	Publicly available dataset Java-based dataset SZZ based mapping	Some projects results are not repeatable No tooling available No software metrics
Github public dataset [121]	Publicly available dataset Java-based dataset SZZ variant based mapping	Some projects results are not repeatable No tooling available No software metrics
Promise, MDP, NASA dataset [18, 19, 20]	Publicly available	No tooling available No software metrics

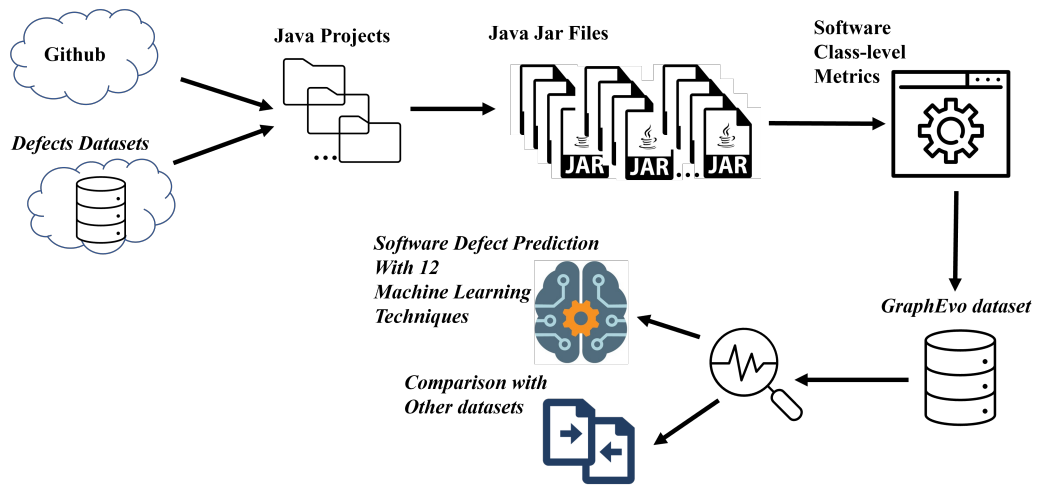
The GitHub dataset is constructed from Java open source projects hosted on GitHub. This dataset contains software from different domains with various class-level metrics [121, 19]. Numerous other articles extracted additional data from bug databases, but

these databases were never released. Among these databases are I Bugs [131], Mozilla [132], and Eclipse [133]. We also highlight the advantages and disadvantages of the existing datasets in Table 13.

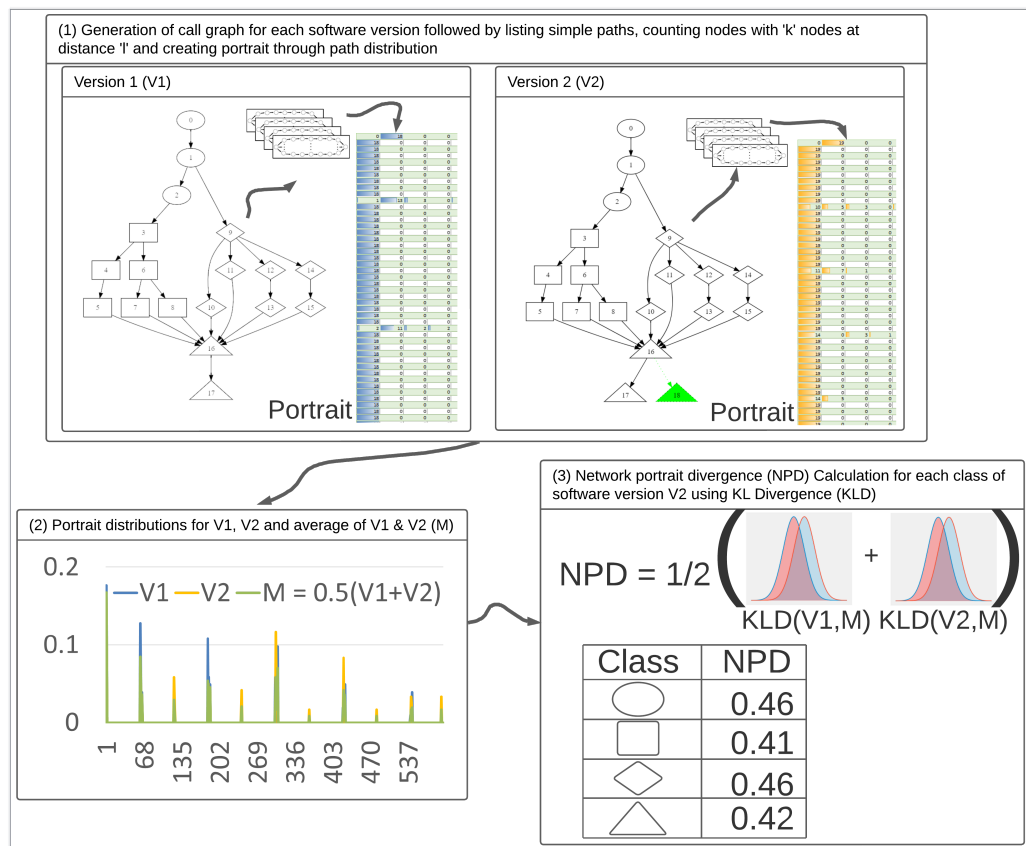
This chapter proposes a method for computing NPD, a new information theory-based metric, and existing Java-based software metrics. We built a dataset using the utility and supplied the Java binary files used to calculate these projects so they can be replicated. Finally, the defect dataset is simple to grow and adds additional open-source projects.

4.3 Proposed Work

Our method is divided into two major phases, as shown in Fig. 25(a). First, we create a call graph for each version of the software system under consideration. Second, we compute software metrics as detailed in Table 14 over several system releases to characterize the development of the software. Additional steps are included in calculating network portrait divergence, such as measuring simple execution paths and calculating divergence as shown in 25(b). All 21 metrics are then saved to the GraphEvo dataset, one row per software version class. Using static analysis tools, the GraphEvoProcessor utility accepts Java binary files as input and generates a call graph with caller-callee representation. The list of caller-callee pairs is now used to generate static call graphs. We detect possible execution pathways after building static call graphs. We begin calculating NPD as shown in Fig. 25(b). We first construct a portrait by determining the number of nodes that have k nodes at distance l . The portrait ensures that the software version is structurally captured, which is the major goal of utilizing it and is discussed in depth by



(a) Outline of the GraphEvoProcessor utility to calculate software metrics and build the dataset



(b) Calculation of Network Portrait divergence (NPD) for each class of the software version (1 - maximum dissimilarity between two software networks(versions), 0 - identical software networks(versions))

Figure 25: Overview of Our GraphEvo Dataset Approach

Walunj et al.[67]. Following that, we compute the portrait distribution for each version and the average of the two currently processed versions. After this step, we are ready to apply KL Divergence to each combination of two software version distributions and the average of the two versions. If we have more than two versions, they are processed one by one until the most recent version is reached. Averaging two KL divergences yield the NPD used to identify structural changes to a certain class. In the example given in Fig. 25(b), we can see that adding a function to the triangle class has resulted in the greatest structural change to the oval and diamond classes. As we can see, the oval and diamond functions are involved in most execution paths, as opposed to the functions of the rectangle and triangle, which have changed. NPD can measure the impact of software changes on classes relating to execution pathways or software structure.

4.3.1 Network Portraits

Network portrait is an efficient way to capture several structural properties of a given network (or a call graph in our case). We use the previous step outputs, i.e., simple execution paths Fig. 26, to construct the network portraits. The network portrait B is defined as an array with (l, k) elements, such that $B_{l,k} \equiv$ *the number of nodes which have k nodes at distance l* for $0 \leq l \leq d$ and $0 \leq k \leq N - 1$, where the distance is the length of shortest path, and d is the graph diameter, and N is the number of nodes in the graph. Note that a distance $l = 0$ is admissible. It is also worth mentioning that network portraits are identical for the same graph despite the nodes' labels or orders. We illustrate the pseudocode to construct the network portraits in Algorithm 2, which results in a matrix

Table 14: Static Code Metrics

Metric Suite	#Metrics	Metric Acronym	Metric Full Name
CK suite	6	WMC	Weighted method per class
		DIT	Depth of inheritance tree
		LCOM	Lack of cohesion in methods
		RFC	Response for a class
		CBO	Coupling between object classes
		NOC	Number of children
Martins metrics	2	CA	Afferent couplings
		CE	Efferent couplings
QMOOM suite	5	DAM	Data access metric
		NPM	Number of public methods
		MFA	Measure of functional abstraction
		CAM	Cohesion among methods
		MOA	Measure of aggregation
Extended CK suite	4	IC	Inheritance coupling
		CBM	Coupling between methods
		AMC	Average method complexity
		LCOM3	Normalized version of LCOM
McCabe's CC	2	AVG_CC	Mean values of methods within the same class
		MAX_CC	Maximum values of methods in the same class
Others	3	LOC	Lines of code
		Network Portrait	Measures the software complexity changes
		BUG	Non-buggy or buggy

that encodes several structural properties of the graph, including the number of nodes in the graph in the zeroth row, the degree distribution in the first row, and then the degree distributions of the next nearest neighbors and so forth. Also, the network portraits are graph invariant, i.e., they assign equal values to isomorphic graphs. Network portraits are important for graph comparison, as we explain in the next step.

Network Portrait Divergence between two graphs G and G' , $D_{JS}(G, G')$, is defined using Jensen-Shannon divergence in Equation 4.1.

$$D_{JS}(G, G') \equiv \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M) \quad (4.1)$$

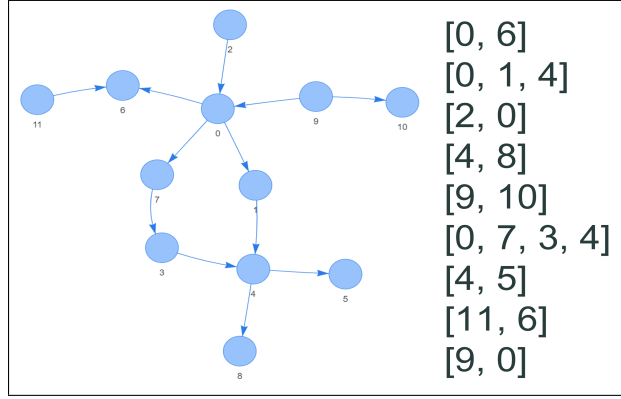


Figure 26: A Snippet of The Call Graph with Functions Represented As Numbers

where $M = \frac{1}{2}(P + Q)$ is the mixture distribution of P and Q . Here, KL is defined in Equation 4.2 and P and Q are defined in Equation 4.3.

$$KL(P(k, l) || Q(k, l)) = \sum_{l=0}^{\max(d, d')} \sum_{k=0}^N P(k, l) \log \frac{P(k, l)}{Q(k, l)} \quad (4.2)$$

Where the log is base 2.

$$p(k, l) = p(k|l)P(l) = \frac{1}{N} B_{l,k} \frac{1}{(\sum_c n_c^2)} \sum_{k'=0}^N k' B_{l,k'} \quad (4.3)$$

Where n_c is the number of nodes within the connected component c , the sum $\sum_c n_c^2$ runs over the number of connected components and the n_c satisfy $\sum_c n_c^2 = N$. Likewise for $Q(k, l)$ using B' instead of B . Selecting two nodes is random with replacement and the probability that they are at a distance l from one another is given in Equation 4.4.

Algorithm 2: Constructing Network Portraits

```
1 procedure NETWORK_PORTRAITS(paths)
2   counter  $\leftarrow$  count(paths)
3   while counter  $\neq$  0 do
4     path  $\leftarrow$  pathList(counter)
5     pathLength  $\leftarrow$  length(path)
6     startNode  $\leftarrow$  start(path)
7     lastNode  $\leftarrow$  last(path)
8     nodeEntry.StartNode  $\leftarrow$  startNode
9     nodeEntry.EndNode  $\leftarrow$  lastNode
10    nodeEntry.distance  $\leftarrow$  pathLength
11    portraitFreq.Add(nodeEntry)
12    counter  $\leftarrow$  counter - 1
13 end
14 networkPortrait  $\leftarrow$  PathsByLength(portraitFreq)
15 return networkPortrait
```

$$p(\text{distance } l) = \frac{\# \text{ paths of length } l}{\# \text{ paths}} = \frac{1}{(\sum_c n_c^2)} \sum_{k=0}^N k B_{l,k} \quad (4.4)$$

The NPD $0 \leq D_{js} \leq 1$ measures the distance between two networks by comparing their portraits. NPD between two identical networks is 0. Jensen-Shannon divergence provides NPD with desirable properties such as symmetry and normalization. Furthermore, NPD works for both directed and undirected networks. This research used NPD to track software change over time by extracting network portraits from each release’s call graph.

The GraphEvoProcessor utility is fully implemented in Python and operated locally using command line arguments. For the backend portion of the development, we utilized several Java-based open source projects for tasks such as decompilation of source code [134, 86], metrics calculation [135, 136], network portrait calculations [67], etc.

Furthermore, we chose Python to rapidly expand on UI in the future, facilitating collaboration and visual debugging.

Table 15: Details of Java Projects in GraphEvo Dataset

Project	Domain	Versions	#Classes	kLOC	# Defects
universal-I.L.	Android library	6	475	61	92
ant	Build tool	5	2078	474	767
antlr	Language processing	5	1696	382	45
broadleafcommerce	E-commerce framework	8	2972	248	106
camel	Enterprise Integration	2	623	52	463
hazelcast	Computing platform	7	10279	107	2781
ivy	Dependency Manager	2	564	91	91
junit	Test framework	8	1054	62	61
lucene	Language processing	3	640	137	849
mapdb	Database engine	6	970	223	174
mcMMO	Game	4	861	196	270
netty	Networking framework	8	4062	676	1027
orientdb	Database engine	3	2369	328	1015
pbeans	Language processing	2	82	17	56
poi	Java Library	3	525	146	380
titan	Database engine	6	2997	318	203
velocity	Template Engine	3	428	125	92
xalan	XSLT processor	4	2752	980	1441
xerces	XML manipulation	2	659	184	284

4.4 Dataset Creation

We selected the 19 software applications with defect information at the class level. A list of the study subject applications and the number of versions used in our study are listed along with some statistics in Table 15. As observed in existing datasets, such as PROMISE [18], Eclipse [128, 133] and GitHub Bug dataset [121] and Bughunter dataset

[120], we found that the following criteria could help pick software versions while constructing the dataset [137]. These software systems were chosen using the following criteria: (1) They must be open-source Java systems with defect information, (2) They must have a track record of several versions, (3) The codebase should be extensive and complex (thousands of lines of source code - KLOC), and (4) They must be well known and have a wide range of users must be available.

Table 15 provides a list of software applications and information about their domain and size. The first column has the software's name and the link to their GitHub repositories. Some software had more than two links, as the repository paths were updated as part of some releases. The second column has the primary domain of the software. The last four columns denote the number of versions, the number of classes, the number of lines of code (k), and the number of defects. The range of domains and software versions bolsters the dataset's generality. Xalan has the most significant code base and moderate defects, whereas the hazelcast application has the most defects. Nonetheless, we will present conclusions for each project separately, based on our evaluation of the best machine learning algorithms for various release versions. The larger the project and the greater the number of defect reports received, the more extensive the database.

We manually downloaded the Java binary files from the project's GitHub accounts list. We had defect information for some programs, such as neo4j, mct, and elasticsearch, but we could not discover specific files. Therefore we did not include them in the research. Furthermore, we could not discover specific software versions for numerous applications used in the study, such as hazelcast and orientdb, and were forced to omit some versions

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	VERSION	CLASS_NAME	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	Ce	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC	AVG_CC	MAX_CC	NEWONE	PORTRAIT	DEFECT_CNT	FAULTY	
2	ant-1.3	AntClassLoader	17	2	0	2	64	76	0	2	9	0.8792	713	0.6	0	0.8272	0.3393	1	3	40.0588	3	10	1	0.0199	2	Defective	
3	ant-1.3	BuildEvent	11	2	0	3	15	13	0	3	11	0.75	97	1	0	0.2	0.2208	0	0	7.2727	5	1	1	0.0041	0	NotDefective	
4	ant-1.3	BuildException	14	4	0	1	28	0	0	1	14	0.3846	153	1	0	0.7586	0.3333	1	2	9.7857	5	2	1	0.0137	0	NotDefective	
5	ant-1.3	BuildListener	7	1	0	1	7	21	0	1	7	2	7	0	0	0	1	0	0	0	12	1	0	0	0	NotDefective	
6	ant-1.3	BuildLogger	4	1	0	1	4	6	0	1	4	2	4	0	0	0	0.5	0	0	0	23	1	0	0	0	NotDefective	
7	ant-1.3	Constants	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	NotDefective	
8	ant-1.3	DefaultLogger	14	1	0	4	32	49	0	4	12	0.8352	257	1	0	0	0.3077	0	0	16.8571	8	6	1	0.0015	2	Defective	
9	ant-1.3	DesirableFilter	2	1	0	0	6	1	0	0	2	2	45	0	0	0	0.6667	0	0	21.5	63	8	0	0	0	NotDefective	
10	ant-1.3	DirectoryScanner	23	1	0	2	51	181	0	2	14	0.7314	1407	1	0	0	0.2909	0	0	59.6957	11	35	1	0.1543	0	NotDefective	
11	ant-1.3	FileScanner	13	1	0	0	13	78	0	0	13	2	13	0	0	0	0.3269	0	0	0	21	1	0	0	0	NotDefective	

Figure 27: Software Ant Dataset Snippet

even if the defect information was available.

We used GraphEvoProcessor as described in Section 4.3 to generate software metrics and defect mapping to classes. As detailed in Table 14, we calculated 21 software metrics. Also, we have added the screenshot of the data generated for software ant Fig. 27. We created different files for each software. These documents are in CSV format (comma-separated values). The first row of the CSV includes header details such as software name, version identifier, class name, software metrics, and defect count. Furthermore, the rest of the lines would row for the unique combination of the classes and version identifier. We chose 87 release versions for each software’s 19 projects and datasets. Table 15 shows the number of entries built for each project.

4.5 Evaluation

We conclude this section by summarizing our final results and accomplishments.

RQ 4.1: Is the GraphEvo dataset generated through NPD-based tooling intended for SDP? Which families of algorithms or algorithms are best suited for SDP?

We used machine learning techniques to analyze our dataset on all data generated

by all software versions. Table 15 shows that the number of defects varies from program to program. We labeled the classes in the software version as defective if it has any defects and non-defective otherwise.

A software version's ratio of non-defective to defective elements will show more non-defective components. Since we wish to use machine learning approaches, the results may be biased because of fewer non-defective classes than defective ones. To solve this issue, we employed random under-sampling to balance the learning corpus [11],[21]. We randomly picked components from the non-buggy class to match the buggy category's size. Consequently, we created a training set with equal positive and negative examples. We repeated this learning process ten times and averaged the results. For training, we used 10-fold cross-validation and compared the results using precision, recall, F-measure, and AUC metrics, which are described as follows:

$$Precision = \frac{TP}{TP + FP} \quad (4.5)$$

$$Recall = Sensitivity = True\ positive\ rate = \frac{TP}{TP + FN} \quad (4.6)$$

$$False\ positive\ rate = \frac{FP}{FP + TN} \quad (4.7)$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4.8)$$

Where TP (True Positive) is the number of classes predicted as defective and found to be defective, FP (False Positive) is the number of classes predicted as defective but found to be non-defective, and FN (False Negative) is the number of classes predicted as non-defective but found to be defective. The confusion matrix and the receiver operating characteristic (ROC) curve determine the diagnostic ability of classifiers. A ROC is one of the fundamental tools for diagnostic test evaluation. It is created by plotting the true positive rate (RECALL) against the false positive rate at various threshold settings. The area under the ROC curve (AUC) is also commonly used to determine the predictability of a classifier. A higher AUC value represents the superiority of a classifier and vice versa. To execute the training, we used the famous machine learning library named Weka [138]. Algorithms of many kinds, such as Bayesian approaches, support vector machines, and decision trees, are included.

We used the following algorithms:

- Naive Bayes - It is a probabilistic classifier that employs Bayes' theorem. It is assumed that the value of features is independent of the value of other features and that all characteristics are equally important. It predicts membership probabilities for each class, such as the likelihood that a given record or data point belongs to a specific class.

- Naive Bayes Multinomial - It is a variant of Naive Bayes optimized for classifying using discrete features. It is advantageous to model feature vectors with values according to the number of occurrences of a phrase or its relative frequency.
- Logistic - It is a classification approach used when the value of the target variable is categorical. Logistic regression is the most commonly used approach when the data has a binary output.
- SGD - Stochastic Gradient Descent (SGD) is a simple and efficient method for fitting linear classifiers and regressors to convex loss functions. This optimization strategy does not belong to a separate family of machine learning models. SGD reduces computation costs and allows faster training iterations for a lower convergence rate, particularly in large-scale optimization problems.
- Simple Logistic - It uses the LogitBoost technique to fit a multinomial logistic regression model. Each cycle adds one SimpleLinearRegression model per class to the logistic regression model. SimpleLogistic has the advantage of built-in attribute selection.
- SMO - Sequential minimal optimization (SMO) is a technique for resolving the quadratic programming (QP) problem that emerges during support-vector machine training (SVM). A multivariable optimization problem is divided into subproblems; each optimizes an objective function specified by a small number of variables. In contrast, the remaining variables are treated as constants in the subproblem.
- Voted Perceptron - This technique is based on Rosenblatt and Frank's perceptron

algorithm. The method uses data that can be separated linearly with high margins. It makes use of data that is linearly separable with large margins. Every weight vector seen casts a vote on a prediction. The number of votes given to a weight vector may indicate the weight vector's correctness.

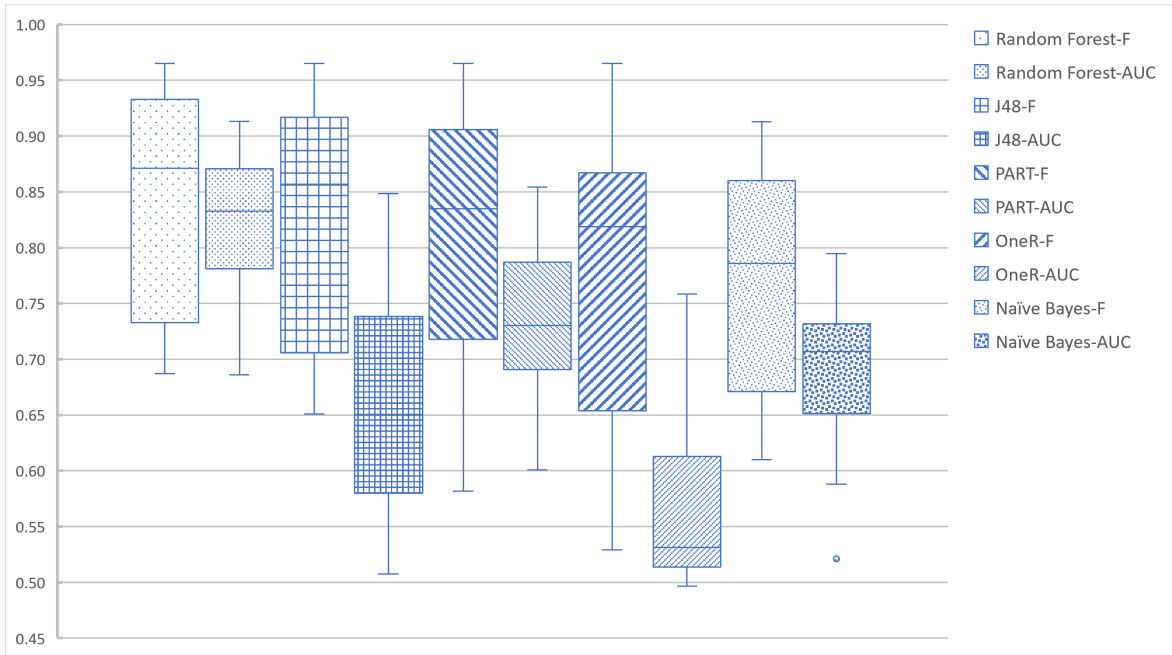
- Decision Table - This is a reliable approach for predicting numerical values from decision trees. It is a collection of ordered If-Then rules that can be more compact and comprehensible than decision trees. The technique used to examine decision tables is more straightforward and less computationally expensive than the decision-tree-based approach.
- OneR - This classifier generates one rule for each predictor in the data and then selects the rule with the lowest total error as its single rule. It creates a frequency table for each predictor vs. the objective to develop a rule for a predictor. It was straightforward for people to understand.
- PART - It is a rule-based classifier that combines the separate and conquers method with the divide and conquers strategy. With the records that are accessible, this categorization technique constructs an incomplete tree. The tree is then used to generate a rule. Once the rule has been applied, the procedure iteratively constructs the partial decision tree.
- J48 - (also known as C4.5) employs a top-down, recursive, divide-and-conquer method to choose which characteristic to split on at each stage. The data is partitioned at each node, resulting in smaller partitions. Internal nodes in this decision

tree represent the different qualities, while branches connect nodes that signify possible values for those attributes.

- Random Forest - It is part of the ensemble learning method used for classification, regression, and other tasks. Random forests include numerous decision trees. Decision trees classified each input as “yes” or “no” (in the case of binary classification). Then once all the trees have been categorized as “yes” or “no” the dominant value is output. Additionally, this strategy works on significant inputs, helping identify which features are critical.

Following that, we determined whether the class datasets were adequate for SDP. We employed 10-fold cross-validation for training and recorded the outcomes such as precision, recall, F-measure parameters, and AUC. Then, we assessed the results and picked the top five algorithms from twelve machine learning techniques. Fig. 28 summarizes the results.

F-measure values vary greatly among projects for several reasons (number of versions, codebase size, defects). Take, for example, the Ant and Broadleaf Commerce projects. The ant is one of the largest, while Broadleaf is in the middle. Ant contains 2078 class-level entries, but Broadleaf has 2972 entries, making it more suitable for usage as a training corpus. Nonetheless, a closer look at the data indicates that the best F-measure has been observed with projects JUnit or titan having more than five versions, showing that we cannot generalize this assumption to be correct; however, further study is needed to confirm this. Different projects have varying prediction capacities (e.g., camel,



(a) Distribution of the performance of ML techniques in terms of F-Measure(F) and AUC

Application	Random Forest		J48		PART		OneR		Naive Bayes	
	F	AUC	F	AUC	F	AUC	F	AUC	F	AUC
antlr	0.964	0.8329	0.965	0.5259	0.967	0.6007	0.965	0.4994	0.911	0.7253
universal-I.L.	0.871	0.8655	0.865	0.618	0.856	0.7022	0.836	0.5137	0.695	0.729
ant	0.84	0.8612	0.834	0.7383	0.793	0.7774	0.819	0.6523	0.818	0.7539
BroadleafCommerce	0.958	0.7879	0.959	0.5873	0.961	0.7488	0.955	0.4995	0.913	0.7947
camel	0.691	0.6874	0.651	0.6368	0.582	0.6462	0.629	0.5456	0.612	0.5212
hazelcast	0.91	0.9032	0.898	0.7931	0.884	0.8293	0.863	0.5238	0.86	0.6498
ivy	0.879	0.8078	0.876	0.6499	0.881	0.7103	0.85	0.5137	0.858	0.6206
junit	0.965	0.8919	0.953	0.714	0.952	0.7707	0.935	0.5188	0.896	0.665
lucene	0.731	0.8142	0.704	0.7304	0.709	0.7453	0.632	0.6203	0.636	0.6976
mapdb	0.878	0.8649	0.856	0.5799	0.859	0.7304	0.847	0.4966	0.847	0.6752
mcMMO	0.811	0.7058	0.819	0.5075	0.823	0.6102	0.809	0.4998	0.809	0.7319
netty	0.933	0.9131	0.917	0.7392	0.906	0.8479	0.867	0.5315	0.72	0.7305
orientdb	0.886	0.9120	0.868	0.8478	0.835	0.8545	0.793	0.6130	0.736	0.7485
pbeans	0.733	0.7813	0.672	0.6534	0.674	0.6908	0.804	0.7586	0.672	0.7893
poi	0.687	0.7699	0.706	0.6449	0.718	0.7015	0.625	0.5581	0.671	0.7070
titan	0.938	0.8578	0.933	0.5280	0.933	0.787	0.938	0.5322	0.91	0.6759
velocity	0.689	0.6862	0.661	0.6879	0.643	0.702	0.529	0.5294	0.61	0.6515
xalan	0.776	0.8709	0.802	0.8483	0.758	0.8472	0.654	0.6166	0.665	0.7114
xerces	0.826	0.8005	0.813	0.533	0.806	0.6018	0.807	0.5588	0.786	0.5879
AVG	0.838	0.838	0.83	0.6612	0.817	0.7318	0.798	0.5569	0.762	0.6929

(b) F-Measure(F) and AUC values of ML Techniques

Figure 28: F-Measure(F) and AUC Analysis of Software Versions in GraphEvo Dataset Analysis

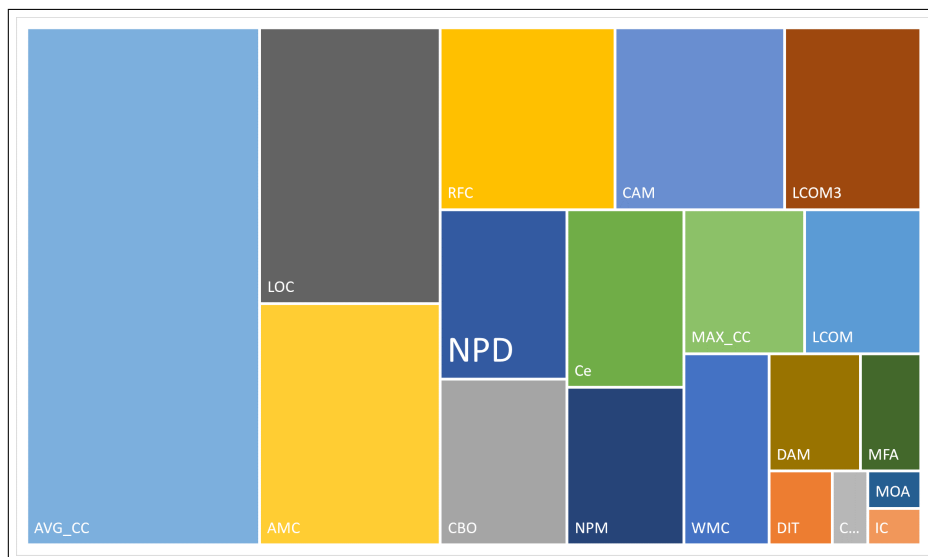


Figure 29: Feature Importance in GraphEvo Dataset

velocity, poi). In the best situation, the F-measure is 0.823. The AUC ranges from extremely low (0.49) to very good (0.91), demonstrating the difficulty of creating reliable models when few samples and defects are considered. The number of classes available in the software has also been a critical factor. Fig. 29 shows the feature importance observed within the Graphevo dataset. NPD metric exists in the top 10 critical metrics referring to the association with the defects. Walunj et al.[67] evaluated the usefulness of NPD concerning SDP; our results also confirm that NPD positively affects the machine learning method's performance.

Answering RQ 4.1: Based on the F-measure and AUC values for the selected software versions, we can conclude that such datasets are relevant for SDP using machine learning approaches to create prediction models. Random Forest outperformed function and rule-based machine learning approaches. Thus one should consider these first when creating prediction models with our datasets.

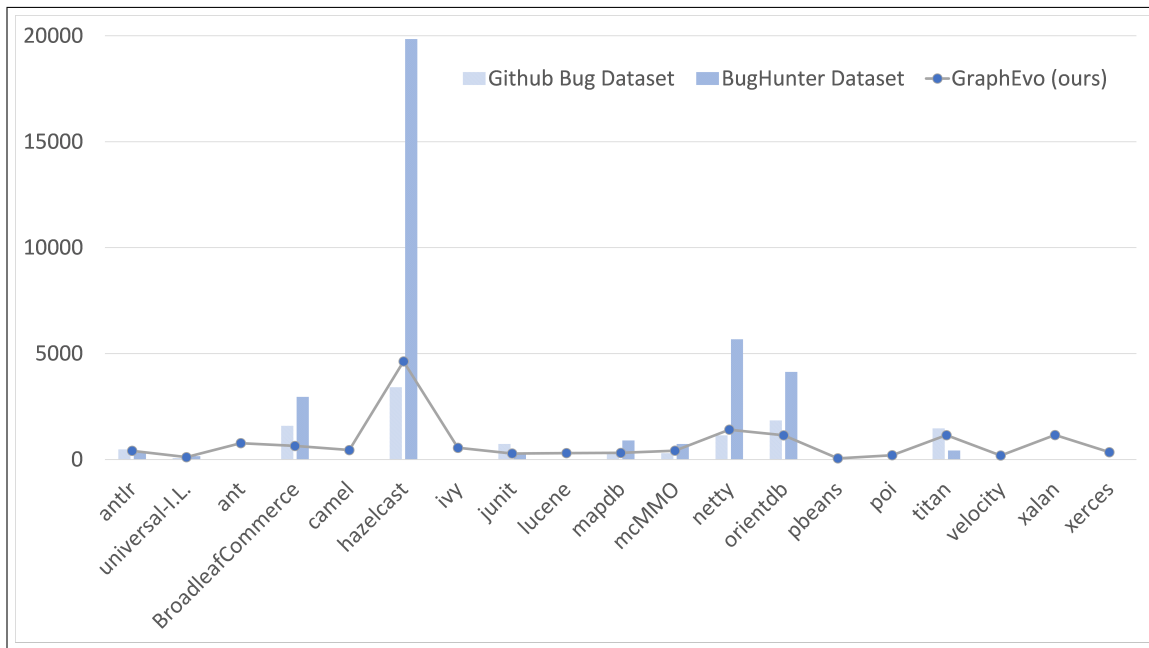


Figure 30: Comparison of The Size of The Datasets

RQ 4.2: Is the GraphEvo DataSet powerful? How does the GraphEvo dataset perform compared to the GitHub Bug and the Bug Hunter datasets?

Each dataset was created with a specific aim in mind. Additionally, the selected projects may differ in their domains, size, and version selection. Therefore, comparing various datasets might be a complex task. We have generated the GraphEvo dataset, which comprises data from 19 software applications. We will compare it to two other datasets based on 15 projects, ten of which are part of the GraphEvo dataset.

First, we compare dataset sizes regarding the number of entries in the dataset,

such as classes. The number of classes considered varies from dataset to dataset, as indicated in Fig. 30. GraphEvo has the most classes in 9 projects, followed by BugHunter [120](7 projects) and Github Bug dataset [121](3 projects). [120] calculated the rate value by comparing the number of classes in different datasets that affect the same dataset. The BugHunter dataset [120] has the highest rate compared to the Github Bug dataset [121] and GraphEvo. We could not find entire Java binary files to compare against other projects, such as BroadleafCommerce and hazelcast, so we had to select only the project's common-utility component and, in some cases, omit some versions, resulting in fewer classes than BugHunter [120]. Furthermore, we could not locate the files for several projects, resulting in a missing item in Fig. 30 for comparison. Furthermore, project lengths in the GitHub Bug dataset were limited to six months, but BugHunter and GraphEvo provide data from project conception to extended durations.

The number of entries in each dataset varies, but they all include a sufficient number of entries to develop a prediction model for each dataset. Table 16 compares the predictive capabilities of the three datasets. We have shown the averages of the F-measure from the top 5 performing machine learning algorithms. BugHunter dataset restricted the software versions to versions with high defects, while GitHub Bug Dataset and GraphEvo used all software versions irrespective of defect counts. Machine learning algorithms have performed better with GraphEvo and GitHub Bug datasets by achieving better F-measure. BugHunter dataset is more precise as it covers the uncertainty around the defect origin. The defect might have been introduced in different versions, and the exact time of its occurrence can not be determined as we only know when it was reported.

Table 16: Predictive Capabilities in Datasets (F-Measure)

Dataset	Average	Minimum	Maximum
GraphEvo (ours)	0.7773	0.6992	0.8290
BugHunter Dataset[120]	0.5685	0.3572	0.7400
GitHub Bug Dataset [121]	0.7710	0.3446	0.8331

Answering RQ 4.2: If the training data is not properly labeled, machine learning algorithms might not learn as well as they should. We can not pinpoint precisely when the issue emerged because we only know when a defect was reported. These characteristics make it difficult to say which dataset is superior for SDP. Compared to the other two datasets, we find that GraphEvo performs favorably.

4.6 Threats to Validation

Our research methodology comprises various steps, including identifying software versions, the construction of call graphs and metrics, and evaluating defect prediction performance and its comparison. We briefly describe the threats to validity in this section.

It would be beneficial to use multiple static source code analyzers in conjunction with one another to mitigate the risks to internal validity posed by measuring source code element characteristics with a single tool. Identifying which tool accurately measures a given metric based on the interpretation of conceptual definitions would require significant efforts. Even once this was determined, additional manual validations would be required. Also, coupling static and dynamic analyzers can enrich our approach’s performance.

The size of the dataset, which may limit SDP capabilities because we only have

19 software, is a threat to external validity. To improve generality, we need to increase the number of software. Extracting software information from additional project hosting services, such as BitBucket, Sourceforge, and Gitlib can increase the count of projects. Also, we could not download precise Java binary file versions for some versions when building the dataset from a list of software from the GitHub bug dataset. As a result, we may have the closest version for a few software instead of the correct version, posing a construct validity threat.

4.7 Conclusions and Future Work

We proposed using Network Portrait Divergence-based tooling to augment datasets for defect prediction at the class level. Using the tooling, we build a GraphEvo dataset for SDP. Our contributions to this research can be summarized as follows: (1) built tooling for building call graphs, computing their Network portrait divergence, and also making tooling available to add new projects, (2) evaluated the GraphEvo dataset using 12 machine learning techniques, and (3) compared GraphEvo's performance to that of two other datasets. We conclude that the GraphEvo dataset is relevant for SDP utilizing machine learning approaches to create prediction models based on the F-measure values. The evaluation found that the proposed approach for analyzing the GraphEvo dataset is accurate and can improve existing methodologies. We also built a tool to replicate our study using open-source Java projects. The tool's website includes the following study findings and figures: <https://vijaybw.github.io/grapevodataset>.

Our future work will extend in several dimensions: We plan to investigate the impact of class-based metrics related to the tests accompanying the code. Another possible direction is to investigate the code pull requests made in a software version control system to collect more metrics about the software and its evolution.

CHAPTER 5

REVIEWING PULL REQUESTS WITH PATH-BASED NPD AND TESTS

5.1 Introduction

Pull requests make it easy for developers to contribute to projects, and many major projects, both open-source and commercial, use them to examine the contributions of different developers. As illustrated in Fig. 31, contributors fork and clone projects to their local repositories, modify the code and tests, and then submit their modifications via pull requests. Following submission, reviewers are responsible for evaluating pull requests and determining whether the modifications should be accepted or rejected.

Researchers have focused their attention on pull request processes, researching many features such as the review process, how pull requests are assigned to different reviewers, and under what conditions they are accepted [25, 26]. Pull requests are inherently social. Change requests are sent to the project reviewers and integrators, who hold a conversation and ultimately decide whether the changes are merged into the main branch. Researchers have also found that the developer's reputation in sending the pull request and implementing new features appears to be a more critical acceptance consideration than any other criterion, including quality [27].

To our knowledge, no studies have investigated pull requests from structural changes and their impact on test cases. Therefore, to understand whether structural code changes can be one of the drivers for accepting pull requests, we designed and conducted a case

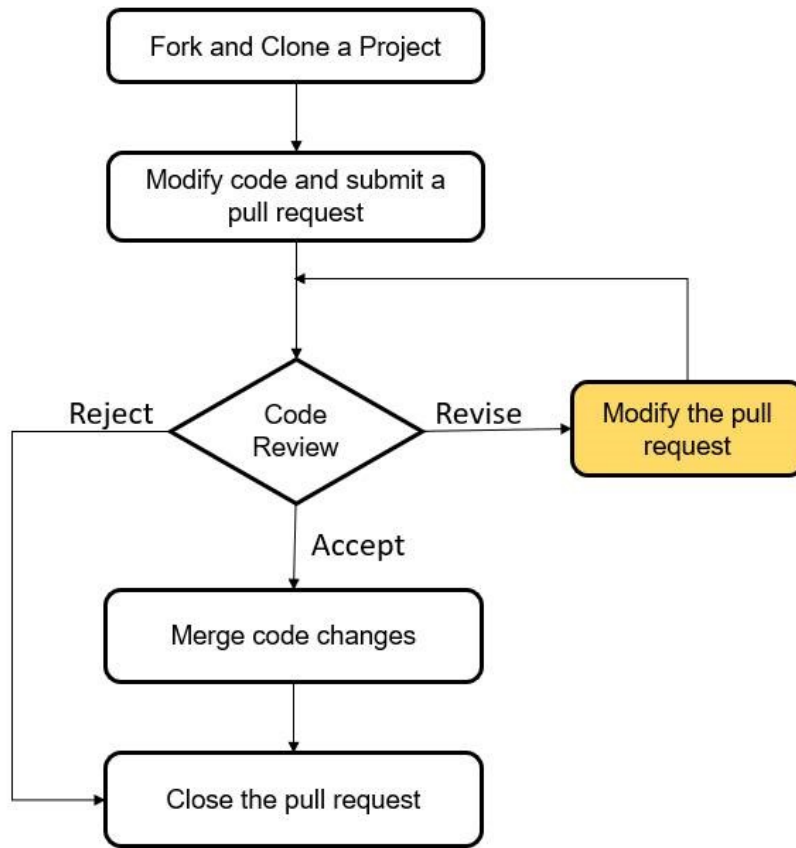


Figure 31: Code Review Process

study involving well-known Java projects [139]. The case study utilized 14 Java open-source projects and analyzed thousands of lines of code quality issues in a total of 627 pull requests. Two of the 14 software have already been processed, and the chapter is written per current findings. We calculated the class-level metrics, including network portrait divergence for each Pull request with and without change. In addition, we counted the number of test cases that were modified or added for each pull request. Furthermore, correlations between class-level metrics were investigated through manual inspection.

Previous studies have reported that pull request review is crucial for software development and that reviewers do not spend more time on test files than code files. At the same time, code reviewers are concerned that the tests accompanying the code modifications are adequate and cover all possible paths. When accepting or rejecting pull requests, project maintainers can use network portrait divergence metrics and decide if they can pay more attention to test files of pull requests [140]. The manual inspection shows that the network portrait divergence does influence the failure of tests in a pull request. Therefore, we decided to manually inspect the pull requests and analyze the impact of changes in class-level metrics on corresponding test failures. We also provided the GraphEvoPR tool to visualize the pull request changes [141].

The remainder of this work is structured as follows. Section 5.2 summarizes related work. Section 5.3 provides a high-level overview of GraphEvoPR and its implementation. Then, in Section 5.4, we will briefly review its implementation. Section 5.5 evaluates and discusses our system concerning open-source java software. Finally, we conclude the chapter in Section 5.7 per our findings.

5.2 Related Work

5.2.1 Software Quality Of Pull requests

Pull requests have been researched from several perspectives, including pull-based development, reviewer assignment, and acceptance. Another issue regarding pull requests that have been investigated is latency. The amount of the change, comments, perceived

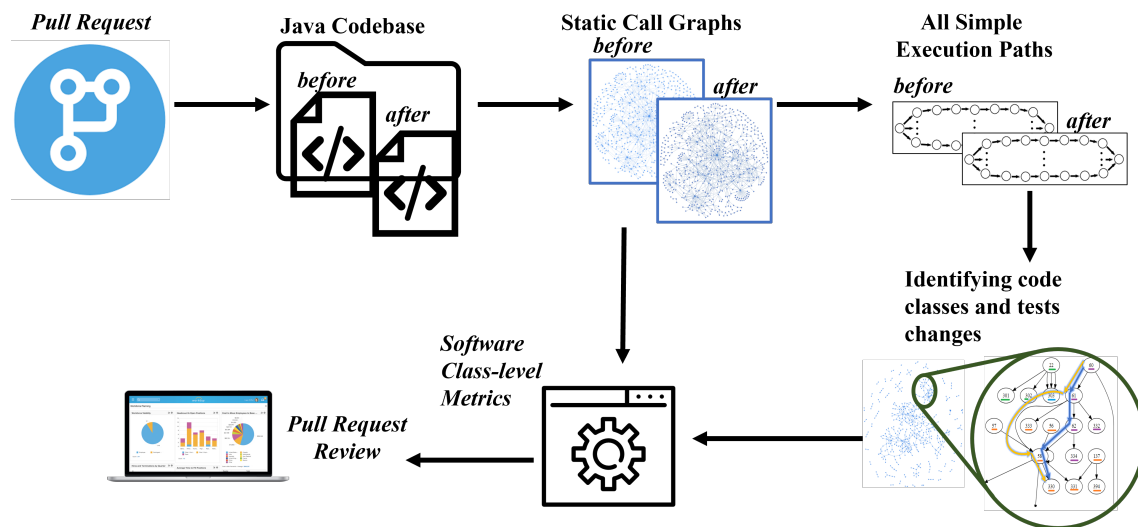


Figure 32: System Overview of GraphEvoPR

quality, and context all play a role in accepting pull requests. Integrators evaluate contributions based on quality, style, documentation, granularity, and conformance to project norms. Furthermore, testing has a crucial impact in accepting a pull request, positively influencing the majority of approved contributions (85 percent) [25]. Rejection of pull requests may increase if technical issues are not adequately resolved and the number of forks grows. Not familiarity with pull requests, the complexity of contributions, location of changed artifacts, and policy contribution are other reasons for rejection [142]. [143] explored open-source static analysis tools like PMD. They used defect density as a proxy to examine the impact of PMD rules on software quality. Data with and without PMD showed a statistically significant variation in defect density. But they only commented on accepted pull requests' code quality, not tests.

[25] evaluated a commercial project's pull request acceptance process, focusing on developers' impressions of review quality. They used data mining on the project's

Table 17: Static Code Metrics

Metric Suite (Number of Metrics)	Metric Acronym	Metric Full Name
CK suite (6)	WMC	Weighted method per class
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion in methods
	RFC	Response for a class
	CBO	Coupling between object classes
	NOC	Number of children
Martins metrics (2)	CA	Afferent couplings
	CE	Efferent couplings
QMOOM suite (5)	DAM	Data access metric
	NPM	Number of public methods
	MFA	Measure of functional abstraction
	CAM	Cohesion among methods
	MOA	Measure of aggregation
Extended CK suite (4)	IC	Inheritance coupling
	CBM	Coupling between methods
	AMC	Average method complexity
	LCOM3	Normalized version of LCOM
McCabe's CC (2)	AVG_CC	Mean values of methods within the class
	MAX_CC	Maximum values of methods in the class
Others (2)	LOC	Lines of code
	NPD	Measures the software complexity changes

GitHub repository to understand the merge, then manually inspected the pull requests. They looked into things, including the size of the pull request and the number of persons interested in the discussion. These studies examined the software quality of pull requests based on developer experience and affiliation. We also highlight the advantages and disadvantages of the existing work in Table 18.

Our research adds to this by measuring the code quality of pull requests in popular open-source projects, taking structural changes to the code and their influence on tests into account as quality criteria. Furthermore, we can see these changes in the tool and other

Table 18: Related Work

Research work	Advantages	Disadvantages
ChangeViz [144]	Replication package available Function level change	No call graph visualization No software metrics
MSPRComplTime [145]	No replication package C# based software PR-level non-code based metrics	Not publicly available No Software metrics used No software metrics
PullBasedDataset [146]	Publicly available PR-level data used More than one language	No Software metrics used
SysGraph4AJ, Reacher, Code2Graph, Codex, BeyondTheCodeItself, Microarray, CI-STUDY [147, 148, 149, 150, 151, 152, 153]	Publicly available	No Software metrics

class-level metrics.

5.2.2 Call graph and Network portrait divergence for comparisons

Call graphs have been extensively explored and utilized to depict the inner functions of a software system via function calls [154, 87]. Some existing studies concentrate on producing and evaluating call graphs for a single system, while others concentrate on collaborative software graphs. The software components were examined as software networks by Code2Vec [88]. It was evaluated using design quality criteria and employed Abstract Syntax Tree to extract all software routes as vectors, which were then aggregated to construct the Code2Vec neural model.

Portrait divergence is a novel approach to comparing networks based on their portraits. Unlike earlier ad hoc comparison measures, Network Portrait Divergence is

founded on information theory, allowing consistent interpretation of the divergence measure [66, 89]. As a result, it may compare them based on their topology structures rather than assuming that networks are defined on the same nodes. GraphEvo used the metric to compare software versions and identify critical events in program evolution [141].

5.3 System Overview

GraphEvoPR aims to exploit current network comparison developments to identify structural changes in code within pull requests and aid in the pull request review process. We provide an open-source tool that calculates class-level metrics, quantifies structural changes, and highlights differences in execution pathways to help stakeholders make better decisions.

GraphEvoPR's approach comprises two steps, as shown in Fig. 32. To begin, it creates a call graph for the feature branch before and after the changes. While the call graph can be used to comprehend the functionality and behavior of a single software system at a time, the objective here is to use it to characterize the pull request. Second, it computes a list of class-level metrics as per Table 17 that compares both versions, including network portrait divergence [155]. Additionally, the application includes visual analysis to assist engineers in comprehending path-level modifications.

5.4 Implementation

The implementation tasks were divided into three steps. First, we extended and integrated an existing tool, GraphEvo, which extracts a list of the caller-callee functions

for a given Java project. Second, the GraphEvo utility calculates class-level metrics and network portrait divergence using the caller-callee relationships in two graph data structures. Third, we can visualize the pull request as a color-coded call graph, view class-level metrics, and identify structural changes to the software.

GraphEvoPR is fully implemented in Python and operated locally using a web-based interface. We used HTML and CSS for the visuals and Javascript for client-side requests to data retrieval from the server. We utilized Flask, a well-known open-source micro web framework for backend server development. The tool uses multiple open-source utilities to calculate 21 class-level metrics, including network portrait divergence. The count of failed tests is manually added to the system, which we hope to automate in future work.

The GraphEvoPR landing page is depicted in Fig. 33. Users can first set up the project by providing the project name, Github URL, default master branch, and authorization key. The user can then select the project and enter the pull request URL. GraphEvoPR will use the Github API to download the feature branch in the pull request. The code is then processed, generating class-level metrics, including NPD.

Defects4J is a large, peer-reviewed, structured dataset of real-world Java bugs [156]. Defects4J includes a test suite and at least one failing test case for each defect. We brought each bug item as a pull request and reviewed it for pull request analysis.

We analyzed the GraphEvo dataset and clustered the NPD values using the Weka tool. The strategy used for clustering was Simple EM (expectation maximization). EM

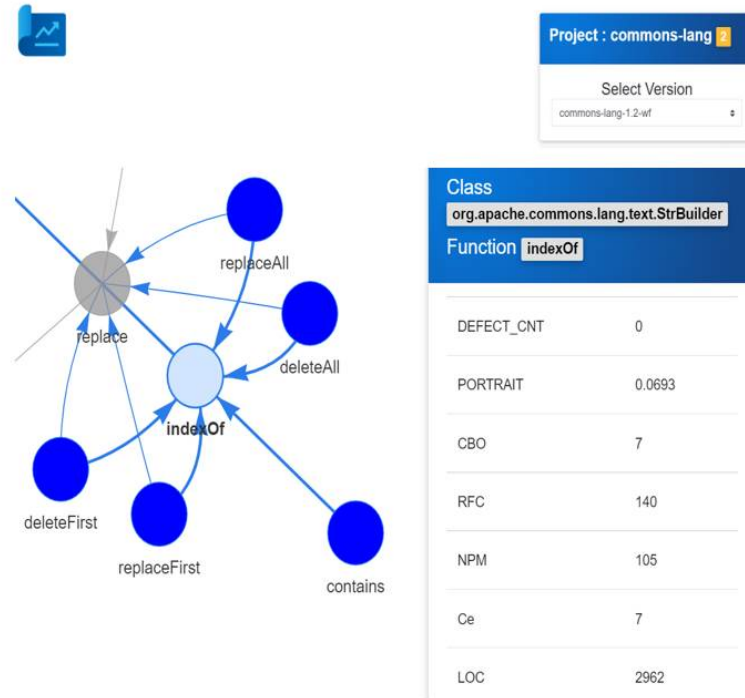


Figure 33: Screenshot of Software commons-lang Call Graph in GraphEvoPR Tool

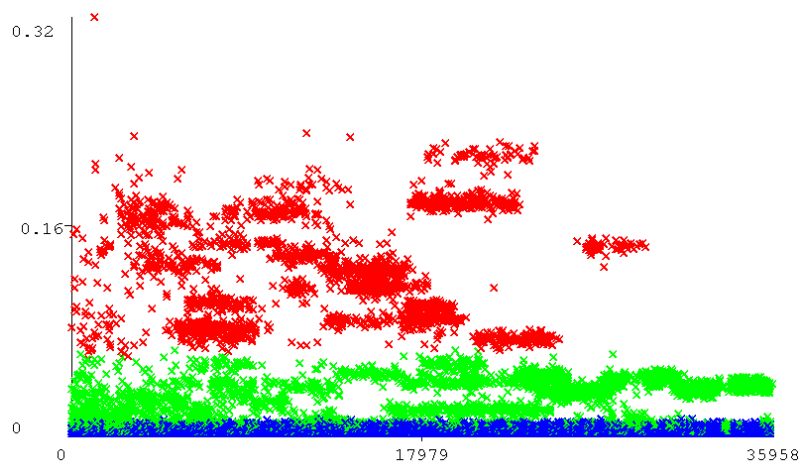


Figure 34: NPD Values as Observed in GraphEvo Dataset

assigns a probability distribution to each instance, indicating the probability of it belonging to each cluster. EM can decide how many clusters to create by cross-validation, or you may specify how many clusters to generate apriori. We observed 3 clusters from 0 to 0.02, 0.03 to 0.1, 0.1 to 0.3 and more than 0.3, as shown in Fig. 34.

We went through such samples and figured we could name the clusters as small, medium, and large. We also apply such clustering for individual java projects and found that the NPD range will change because of the domain and size of the software. We found that some small software exhibited a different range in terms of changes. This is evident that if the software has less number of classes and functions, each change will exhibit a more significant value. This type of range will be used as a starting point for the java application under study, which needs to be modified based on new observations.

The Table 19 below shows the level of complexity and the range of values for the respective software metrics network portrait divergence. To determine the pull request criteria, this was categorized.

Table 19: Code Complexity and Software Metric Classification

Complexity	Network Portrait Divergence Range	Assumption
Low	0.00 - 0.02	Small degree of change
Medium	0.02 - 0.10	Considerable degree of change
High	0.10 - 0.30	Significant degree of change
Very High	0.30 - 1.00	Severe change

Decisions -

- Accept - If software structure change depicts the degree of change and it is in an acceptable range concerning code and test

- Reject - If software structure change is not in an acceptable range concerning code and test change

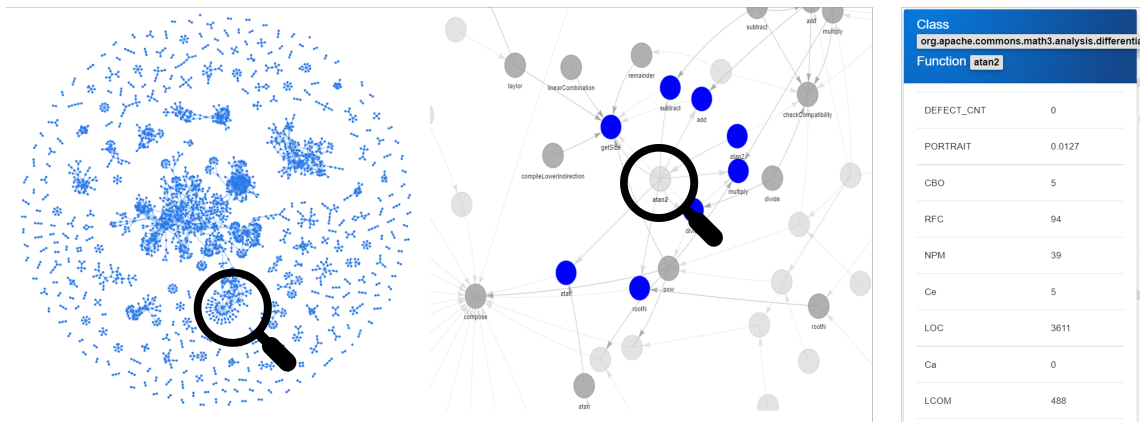
5.5 Results

Classes from the application Apache Math [139], and their findings are discussed in the section. A detailed list of class data can be found on <https://vijaybw.github.io/graphbevopr>. These are preliminary results and are based on two of 14 software. These classes are with a varying range of lines of codes. Classes from respective feature branches of projects before and after the change. To be precise, a significant code change to fix the test directly impacts the code's structure, indicating that the connection for changed classes is rewired with associated classes. Network Portrait Divergence resonates with the changes applied in the broken test fix. At the same time, RFC and LOC also indicate the appropriate metrics changes.

Class DSCompiler in PR of math (math-10) has one method change that includes two repair actions and patterns to fix the broken test in Fig. 35. The changes indicate that it, directly and indirectly, impacts 43 execution paths. The execution paths in which this class is used/ called. All the metrics reflect the correct changes. Precisely, the change in values from 0 to 0.0203 for Network Portrait Divergence reflects the change in the structure of the code and how it captures the impacted execution paths. We also see one test failing, typical for such a small structural change.

Class StrBuilder in PR of lang (lang-61) had a 14.66% impact on the execution path compared to overall execution paths (absolute execution paths for before the change

Math 10 - org.apache.commons.math3.analysis.differentiation.DSCompiler



(a) Source code change in Class DSCompiler

```

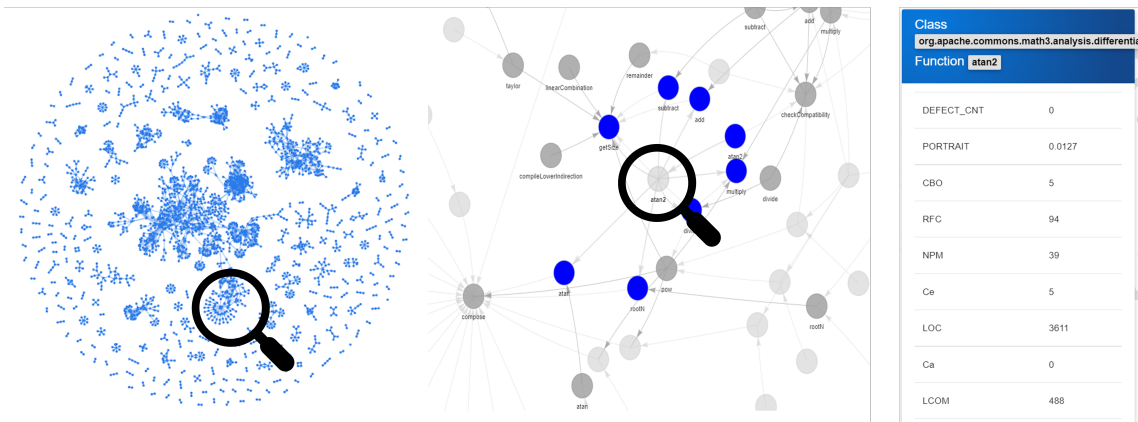
src/main/java/org/apache/commons/math3/analysis/differentiation/DSCompiler.java CHANGED
@@ -1416,6 +1416,7 @@ public void atan2(final double[] y, final int yOffset,
1416 1416     }
1417 1417
1418 1418     // fix value to take special cases (+0/+0, +0/-0, -0/+0, -0/-0, +/-infinity) correctly
1419 + result[resultOffset] = FastMath.atan2(y[yOffset], x[xOffset]);
1419 1420
1420 1421     }
1421 1422

```

(b) Screenshot of GraphEvoPR

Figure 35: Class DSCompiler in PR of math (math-10)

Math 10 - org.apache.commons.math3.analysis.differentiation.DSCompiler



(a) Source code change in Class StrBuilder

```

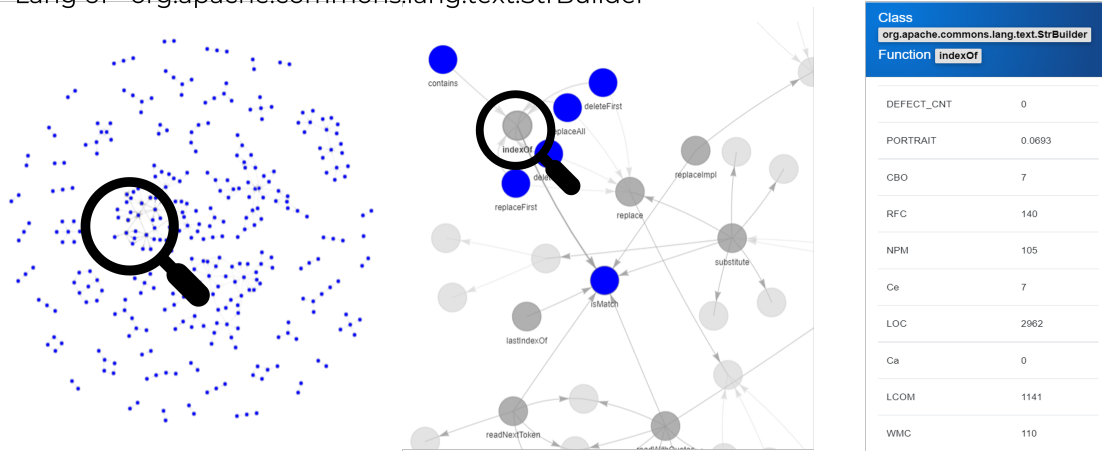
src/main/java/org/apache/commons/math3/analysis/differentiation/DSCompiler.java CHANGED
@@ -1416,6 +1416,7 @@ public void atan2(final double[] y, final int yOffset,
1416 1416     }
1417 1417
1418 1418     // fix value to take special cases (+0/+0, +0/-0, -0/+0, -0/-0, +/-infinity) correctly
1419 + result[resultOffset] = FastMath.atan2(y[yOffset], x[xOffset]);
1419 1420
1420 1421     }
1421 1422

```

(b) Screenshot of GraphEvoPR

Figure 36: Class StrBuilder in PR of lang (lang-61)

Lang 61 - org.apache.commons.lang.text.StrBuilder



(a) Source code change in Class EqualsBuilder

```

src/java/org/apache/commons/lang/text/StrBuilder.java [CHANGED]
@@ -1773,7 +1773,7 @@ public int indexOf(String str, int startIndex) {
1773 1773     return -1;
1774 1774 }
1775 1775 char[] thisBuf = buffer;
1776 - int len = thisBuf.length - strLen;
1776 + int len = size - strLen + 1;
1777 1777 outer:
1778 1778 for (int i = startIndex; i < len; i++) {
1779 1779     for (int j = 0; j < strLen; j++) {

```

(b) Screenshot of GraphEvoPR

Figure 37: Class StrBuilder in PR of lang (lang-48)

after the change are 709, respectively, and StrBuilder class appeared in 104 execution paths for each version) in Fig. 36. This indicates that it has a significant impact on code quality and coverage. The Network Portrait Divergence correctly identified these changes along with RFC and LOC. We also see the two tests failing, reflecting the more significant change than class DSCompiler.

Class EqualsBuilder, Entities, and FastDateFormat in PR of lang (lang-48) are large classes with numerous execution paths. The number of execution paths impacted

was significant in Fig. 36. The Network Portrait Divergence values are in a similar range for these classes, indicating that code fixes in these classes had an equal impact on execution paths. We also see the one test failing, reflecting a small change structurally.

We have also seen that the top class-level metrics changed along with portrait divergence: AMC, LOC, RFC, and LCOM. Portrait divergence does not change when changes are limited to code lines within function or class constructors. On the other hand, portrait divergence can capture additional changes when a class is refactored, such as changes in class interaction. The number of code-based tests is affected by structural changes in metrics.

5.6 User Study

User studies are conducted with the primary purpose of gaining an understanding of the various practical uses of a specific technology. In a manner comparable to this, the purpose of our user study is to determine the usefulness of our strategy and tool based on the information provided by user experience. In addition, it may assist in assessing the benefits and drawbacks of our visualization tool, and it can guide future efforts to improve upon the already in place approaches [145].

In this section, we test our technique and tool by doing user research to determine how applicable and usable they are. Out of more than 100 persons who were invited, 24 people took part in the user study and completed the series of activities required before responding to the questions on the questionnaire. The questionnaire, as shown in Fig. 38 contained ten questions that were meant to be answered to evaluate how useful the tool

was [157].

5.6.1 Participants

As part of the user study, we asked software development managers, engineers, and testers with experience levels ranging from 1 to 10 to test our solution and then fill out a questionnaire about their thoughts on it. Our team contacted over one hundred software development managers, engineers, and testers. Twenty-four individuals decided to take part in the study. 23 out of 24 individuals acknowledged their prior involvement in the sector. More than ten international companies worldwide are represented here, including Amazon, Google, Cummins, and others. Nevertheless, we did not want to include incomplete statistics in the study because our participants did not provide adequate information about their age and gender. As shown in Fig. 39, most of our participants had prior experience (69 % had more than ten years, and 25% had between 3 and 10 years).

5.6.2 Procedure

Before the research began, the participants were given a few simple tasks to complete. They were required to provide their email address as a prerequisite for consent. Second, they were asked to answer questions designed to collect demographic information, such as their name, experience level, and thoughts on the study topic. Third, a brief video demonstrating the tool was shown, followed by instructions on examining sample instances using software metrics and visualization.

Participants in this study were shown a demonstration video, activities, and a questionnaire to evaluate the tool. Participants were required to gather information about the

Task ID	Description
T1	Name two classes that have a high value of network portrait divergence
T2	Name two classes where you see a significant change in respective test cases
T3	Name two classes where you see no code changes; however, they impact the path or network portrait value
T4	Impact of a code change on test coverage
T5	Code and test change analysis using path-based visualization

(a) User study tasks

Question	Description	Type
Q1	The tool's interface is intuitive and user-friendly	Likert Scale
Q2	I would recommend software developers to use this tool	Likert Scale
Q3	This tool helps review the pull request rather than doing it manually	Likert Scale
Q4	I believe that using this tool can save time and effort in the pull request review process	Likert Scale
Q5	Using this tool, you can identify test coverage	Likert Scale
Q6	The call graph visualization is helpful to identify the overall impact on software structure before accepting or rejecting the pull request	Likert Scale
Q7	Network portrait value from software metric helps in pull request review to identify low, medium, or high impact	Likert Scale
Q8	Overall, the tool is helpful in the pull request review process before merging the code	Likert Scale
Q9	Please list any reasons for your answer to previous questions	Open Ended
Q10	Would you like to add any other suggestions? Feedback? Or limitations?	Open Ended

(b) User study questions

Figure 38: Details of User Study

Count of Year of experience in the industry?

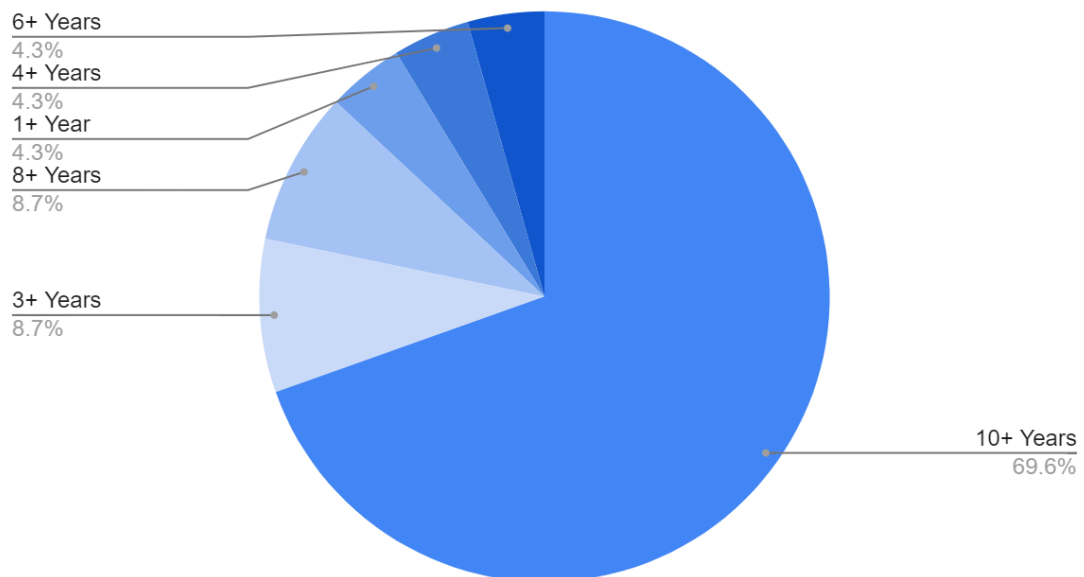


Figure 39: Survey Participants Industry Experience

usability and utility of the GrapshEVOPR tool to complete the tasks. Fig. 38(a) depicts the roles and responsibilities (a). The participants completed the questionnaire using the tool's demonstration film and sample examples. Google Forms was used to compile the responses from the participants.

5.6.3 Questionnaire

Our questionnaire includes two types of questions, including (1) Likert scale questions to evaluate the usability and usefulness of the tool and (2) open-ended questions to gain feedback on the design and improvements of the tool. The list of questions and their types are listed in Fig. 38(b)

5.6.4 Results and Discussion

We first report the results of usability & use-fullness questions, Q1 to Q8, respectively, and then discuss the strengths and weaknesses of the tool from user feedback in Q9 and Q10.

To evaluate the usability and usefulness of the tool, the participant was asked to perform the set of small tasks, watch the video clip and review the sample examples with visualization and software metrics of a different pull request from this study.

Q1: Participants are questioned on their thoughts on the tool's user interface. More than 90% of the participants (22 out of 24) agreed that the tool's user interface is simple and easy to use.

Q2: We asked participants if they could reduce the time and effort spent on pull

request review. More than 85% of the participants agreed with most of the positive comments. And 8% of those polled said it would be beneficial.

Q3: We asked participants if they would recommend this tool for pull request review to other developers and engineers, and we received a very positive response. This received 91% support from participants.

Q4: The goal of this question⁴ was to see if experienced developers, the majority of whom had more than three years of experience in the field, would prefer a manual review of a pull request to GraphEVOPR, which displays software metrics and call graphs alongside code changes. According to the results, 75% of participants voted favorably, while 25% thought it might help with the review of pull requests. We learned from speaking with a small number of participants from the 25% who may have voted that they would still prefer to review the code directly for upkeep and bug-fixing duties. This investigation aimed to discover more about software architecture rather than debugging. The responses to Q2 back this up.

Q5: We asked the participant if it could help with the problem of detecting test coverage while reviewing a pull request. More than 95% of respondents strongly agreed with this.

Q6: We asked the participant if a visual representation of classes, methods, and their related pathways would be helpful in the pull request review process. One person disagreed, but the other 23 participants strongly agreed.

Q7: We were curious whether the Network Portrait Divergence value could help pull request reviewers determine how a code change would affect the software's structure.

Will this reduce the need for knowledgeable resources in a group setting when reviewing pull requests? Over 95% of participants responded positively and agreed with this.

Q8: In this question, participants were asked to vote on the general usability and usefulness of the GraphEVOPR tool. The final score shows that all participants agreed to it.

5.6.5 Participant's Feedback

Two open-ended questions are included in the questionnaire. The participants requested that they rate the tool as a whole, offer us input that could assist us in enhancing the tool, and contribute valuable information regarding the challenges encountered during the analysis. Following is a summary as well as a synthesis of the remarks made by the participants:

Java version and other languages: One participant specifically asked, “*Will this tool help with different versions of java?*”. Yes, GraphEVOPR can support different versions of java. However, it needs further enhancements to support other languages like python, C sharp, etc.

Lack of Information: Along with Network Portrait Divergence, we used basic software metrics such as AMC, LOC, fan in, fanout, etc., to evaluate the impact on software structure and how it can help in a pull request review process. However, 2 participants suggested including more data points to support this. One participant commented “*More data to back up this analysis*”, and another participant asked “*Can also include*

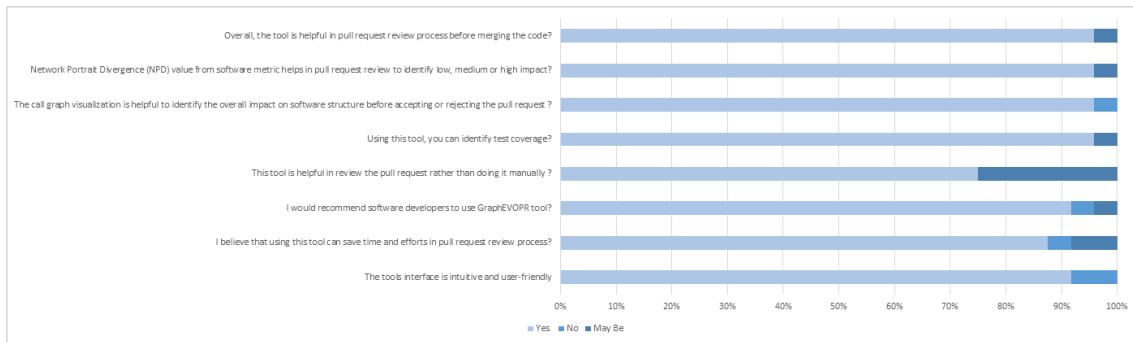


Figure 40: Likert Scale Representation of Survey Responses

statement coverage, branch coverage, etc. like what Testing framework like a jest, jasmine, Junit, etc.. do.”

5.7 Conclusions and Future Work

We studied the use of changes in software class-level metrics, such as Network portrait divergence, for evaluating and visualizing pull requests in this study. In specifically, (1) we collected class-level metrics for code in pull requests before and after modifications, (2) we reviewed a set of metrics and identified associated changes to test failures, and (3) we provided tool support for our technique, GraphEvoPR. Using 627 Pull requests from 14 popular Java open-source systems, we evaluated the performance of class-level metrics. The study demonstrated how network portrait divergence measures identify the structural changes. We will concentrate on growing the amount of software and pull requests in the near future. Furthermore, we intend to use machine learning models (Decision Trees, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, and XGBoost) to estimate code quality in terms of a set of tests that may be required to pass a pull request.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The dissertation primarily aimed to aid in software evolution using metrics and visualization. Network portrait divergence, a statistic based on information theory, was used for the software domain. The NPD metric could identify significant events at the class and software levels. In addition, our model trained with open-source java programs performed well compared to state-of-the-art models. The NPD-based dataset was proven advantageous compared with similar datasets for Software defect prediction. Finally, our research showed that software metrics and call graph visualization could help improve the review of pull requests. Software evolution can be driven by several factors, including changes in the hardware or operating system environment, changes in user requirements, or the need to improve the software itself. At each step of the software version, software metrics will play the role of guardian and help track the process better. We conclude that our research will improve software evolution.

Here are the four significant research contributions of the work described in this dissertation:(1) Software-level characterization and analysis (2) Class-level analysis and building model for SDP (3) Building NPD-based utility and construct dataset and its comparison to other SDP datasets (4) Pull request review analysis and correlation of code changes and tests.

- **Summary of Software level characterization and analysis**

We investigated the use of information theory for analyzing and visualizing software evolution. We evaluated the GraphEvo approach using 66 software releases of five popular Java open-source systems. The study illustrated that graph metrics can exploit the similarities and differences in the structure and evolution of a software system. GraphEvo can be used to re-produce our study using any open-source Java project with two or more releases.

- **Summary of Class-level analysis and building model for SDP**

We discussed in this research the use of Network Portrait Divergence to identify significant events in software evolution, how it compares to other software metrics, and how it can be used to predict software defects. We utilized the 384 software releases of 29 open-source Java systems. The prediction model achieved an 18% reduction in the mean square error and a 48% increase in the squared correlation coefficient. The approach is accurate and can enhance state-of-the-art approaches for SDP.

- **Summary of Building NPD-based tooling and constructing dataset and its comparison to other SDP datasets**

We built NPD-based tooling and constructed a defect dataset by mining 19 diverse Java projects. Following that, we assessed its SDP capability using 12 machine learning techniques. Lastly, we compared the dataset with other defect datasets and found that the dataset is relevant for SDP and that the dataset can improve existing methodologies.

- **Summary of Pull request review analysis and correlation of code changes and tests.**

We developed a utility that can aid the PR review process. The utility downloads the feature branch within the PR and calculates class-level metrics. We calculated the class-level metrics for 627 Pull requests from 14 popular Java open-source systems. The study demonstrated how network portrait divergence measures identify the structural code changes and the correlation between code changes and tests.

Our future work will concentrate on utilizing recent Graph Neural Networks (GNN) development. GNN is effective in a variety of practical applications. According to recent research, GNN can be used to learn source code properties. We plan to evaluate GNN's effectiveness in terms of software evolution. We also hope to expand our software analysis to popular programming languages such as Python, Go, and C#. Finally, we intend to use cloud infrastructure to process software metrics and reduce the execution time spent on analysis tasks.

APPENDIX

GREAPHEVO ADDITIONAL DATA

In this chapter, we present the additional data that was used in this research.

- **Network Portrait Divergence Calculation Example:** Here is an example of a processing software versions to calculate class-level Network portrait divergence values as shown in Fig. 41, 42, 43, and 44. The names of the function were replaced with numbers for readability. Following that, the portraits are calculated for each software version, and then the calculation is done for each class.
- **GraphEvo Dataset Machine Learning Results:** Section 4.5 mentioned the result of the top 5 machine learning algorithms. Here, we list all the results for reference in Table 20, 21, 22, 23,24, 25, and 26. The same results can also be found on our Github <https://vijaybw.github.io/graphevodataset> in CSV format.

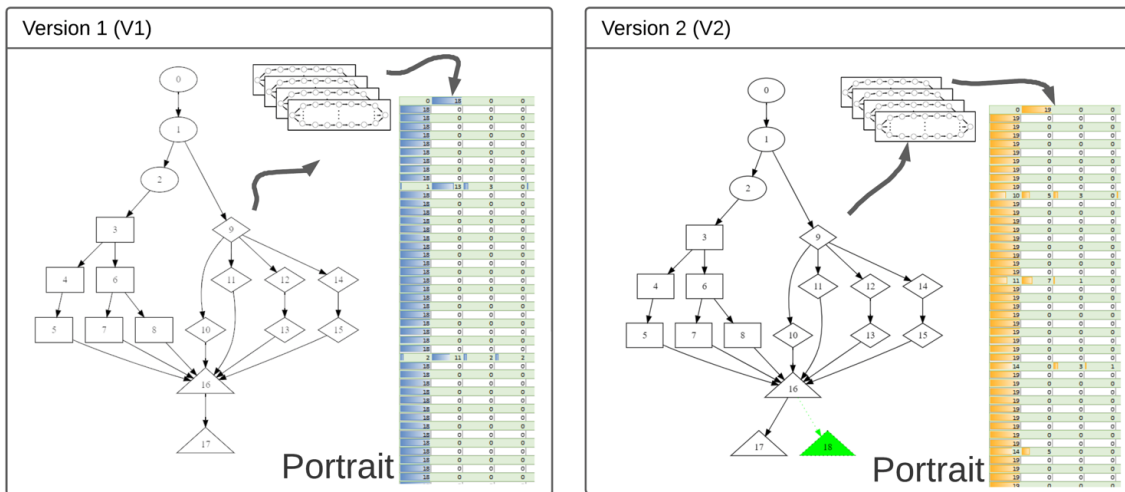


Figure 41: Network Portrait Divergence Calculation Example 1 Of 4

Version 1		
Caller	Callee	Weightage
0	1	1
1	2	1
1	9	1
2	3	1
3	4	1
3	6	1
4	5	1
5	16	1
6	7	1
6	8	1
7	16	1
8	16	1
9	10	1
9	11	1
9	12	1
9	14	1
10	16	1
11	16	1
12	13	1
13	16	1
14	15	1
15	16	1
16	17	1

Version 2		
Caller	Callee	Weightage
0	1	1
1	2	1
1	9	1
2	3	1
3	4	1
3	6	1
4	5	1
5	16	1
6	7	1
6	8	1
7	16	1
8	16	1
9	10	1
9	11	1
9	12	1
9	14	1
10	16	1
11	16	1
12	13	1
13	16	1
14	15	1
15	16	1
16	17	1
16	18	1

Figure 42: Network Portrait Divergence Calculation Example 2 Of 4

Triangle			Square			Diamond			Oval		
Caller	Callee	Weightage	Caller	Callee	Weightage	Caller	Callee	Weightage	Caller	Callee	Weightage
0	1	1	0	1	1	0	1	1	0	1	1.5
1	2	1	1	2	1	1	2	1	1	2	1.5
1	9	1	1	9	1	1	9	1.5	1	9	1.5
2	3	1	2	3	1.5	2	3	1	2	3	1
3	4	1	3	4	1.5	3	4	1	3	4	1
3	6	1	3	6	1.5	3	6	1	3	6	1
4	5	1	4	5	1.5	4	5	1	4	5	1
5	16	1.5	5	16	1.5	5	16	1	5	16	1
6	7	1	6	7	1.5	6	7	1	6	7	1
6	8	1	6	8	1.5	6	8	1	6	8	1
7	16	1.5	7	16	1.5	7	16	1	7	16	1
8	16	1.5	8	16	1.5	8	16	1	8	16	1
9	10	1	9	10	1.5	9	10	1.5	9	10	1
9	11	1	9	11	1	9	11	1.5	9	11	1
9	12	1	9	12	1	9	12	1.5	9	12	1
9	14	1	9	14	1	9	14	1.5	9	14	1
10	16	1.5	10	16	1	10	16	1.5	10	16	1
11	16	1.5	11	16	1	11	16	1.5	11	16	1
12	13	1	12	13	1	12	13	1.5	12	13	1
13	16	1.5	13	16	1	13	16	1.5	13	16	1
14	15	1	14	15	1	14	15	1.5	14	15	1
15	16	1.5	15	16	1	15	16	1.5	15	16	1
16	17	1.5	16	17	1	16	17	1	16	17	1
16	18	1.5	16	18	1	16	18	1	16	18	1

Figure 43: Network Portrait Divergence Calculation Example 3 Of 4

Version	V1
Type	Digraph
Number of nodes	18
Number of edges	23
Average in degree	1.2778
Average out degree	1.2778
Version	V2
Type	Digraph
Number of nodes	19
Number of edges	24
Average in degree	1.2632
Average out degree	1.2632
Network Portrait Divergence (V1,V2)	0.1864
Network Portrait Divergence (V1,V2-Oval)	0.42
Network Portrait Divergence (V1,V2-Square)	0.41
Network Portrait Divergence (V1,V2-Diamond)	0.46
Network Portrait Divergence (V1,V2-Triangle)	0.42

Figure 44: Network Portrait Divergence Calculation Example 4 Of 4

Table 20: GraphEvo Dataset Machine Learning Results 1 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
universal-I.L.	RandomForest	0.141	0.2897	68.5693	0.871	0.8655
universal-I.L.	NaiveBayesMultinomial	0.3622	0.595	176.1831	0.705	
universal-I.L.	Logistic	0.1643	0.2958	79.9403	0.864	
universal-I.L.	SGD	0.1175	0.3428	57.1713	0.837	
universal-I.L.	SimpleLogistic	0.1755	0.2945	85.3643	0.859	
universal-I.L.	SMO	0.1154	0.3397	56.1318	?	
universal-I.L.	VotedPerceptron	0.1345	0.3666	65.4378	0.821	
universal-I.L.	DecisonTable	0.2012	0.3158	97.8654	0.843	
universal-I.L.	PART	0.1575	0.3207	76.6419	0.856	0.7022
universal-I.L.	OneR	0.1197	0.3459	58.2108	0.836	0.5137
universal-I.L.	NaiveBayes	0.3758	0.5343	182.7944	0.695	0.729
universal-I.L.	J48	0.164	0.3235	79.7785	0.865	0.618
ant	RandomForest	0.1992	0.321	65.1497	0.84	0.8612
ant	NaiveBayesMultinomial	0.2987	0.5434	97.6643	0.729	
ant	Logistic	0.2415	0.3502	78.9704	0.813	
ant	SGD	0.1675	0.4092	54.7649	0.793	
ant	SimpleLogistic	0.2455	0.3506	80.2858	0.805	
ant	SMO	0.1752	0.4186	57.2901	0.771	
ant	VotedPerceptron	0.194	0.4404	63.4405	0.79	
ant	DecisonTable	0.2395	0.3477	78.3251	0.828	
ant	PART	0.223	0.3587	72.9169	0.793	0.7774
ant	OneR	0.1631	0.4039	53.3445	0.819	0.6523
ant	NaiveBayes	0.1873	0.4121	61.264	0.818	0.7539
ant	J48	0.207	0.3636	67.6881	0.834	0.7383
antlr	RandomForest	0.0387	0.1549	84.5379	0.964	0.8329
antlr	NaiveBayesMultinomial	0.3377	0.5802	738.2967	0.777	
antlr	Logistic	0.0424	0.1492	92.7671	0.965	
antlr	SGD	0.0231	0.152	50.5129	?	
antlr	SimpleLogistic	0.5	0.5	1093.1518	?	
antlr	SMO	0.0231	0.152	50.5129	?	
antlr	VotedPerceptron	0.0255	0.1596	55.6938	0.964	
antlr	DecisonTable	0.0458	0.1502	100.0326	?	
antlr	PART	0.0416	0.1659	90.9713	0.965	0.6007
antlr	OneR	0.0243	0.1558	53.1033	0.965	0.4994
antlr	NaiveBayes	0.1338	0.3465	292.4636	0.911	0.7253
antlr	J48	0.0448	0.152	97.9896	0.965	0.5259

Table 21: GraphEvo Dataset Machine Learning Results 2 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
BroadleafCommerce	RandomForest	0.0474	0.1662	81.1338	0.958	0.7879
BroadleafCommerce	NaiveBayesMultinomial	0.3213	0.5652	549.9751	0.783	
BroadleafCommerce	Logistic	0.0543	0.166	92.9766	?	
BroadleafCommerce	SGD	0.0299	0.173	51.2521	?	
BroadleafCommerce	SimpleLogistic	0.1181	0.2261	202.1261	?	
BroadleafCommerce	SMO	0.0299	0.173	51.2521	?	
BroadleafCommerce	VotedPerceptron	0.0299	0.173	51.2521	?	
BroadleafCommerce	DecisonTable	0.0583	0.1705	99.77	?	
BroadleafCommerce	PART	0.0484	0.1736	82.9087	0.961	0.7488
BroadleafCommerce	OneR	0.031	0.1759	52.9797	0.955	0.4995
BroadleafCommerce	NaiveBayes	0.1223	0.3337	209.2696	0.913	0.7947
BroadleafCommerce	J48	0.0523	0.167	89.5038	0.959	0.5873
camel	RandomForest	0.3555	0.441	82.8442	0.691	0.6874
camel	NaiveBayesMultinomial	0.4571	0.663	106.5314	0.557	
camel	Logistic	0.4146	0.4642	96.622	0.577	
camel	SGD	0.313	0.5595	72.9408	0.561	
camel	SimpleLogistic	0.488	0.4954	113.7151	0.561	
camel	SMO	0.3114	0.558	72.5667	?	
camel	VotedPerceptron	0.329	0.5735	76.6638	0.573	
camel	DecisonTable	0.4292	0.4631	100.0175	?	
camel	PART	0.4043	0.4667	94.2157	0.582	0.6462
camel	OneR	0.3419	0.5847	79.6738	0.629	0.5456
camel	NaiveBayes	0.3924	0.5521	91.4543	0.612	0.5212
camel	J48	0.3834	0.4901	89.3454	0.651	0.6368
hazelcast	RandomForest	0.1104	0.2461	61.8969	0.91	0.9032
hazelcast	PART	0.1329	0.2701	74.517	0.884	0.8293
hazelcast	NaiveBayesMultinomial	0.342	0.5803	191.7297	0.727	
hazelcast	Logistic	0.1644	0.2878	92.1527	0.862	
hazelcast	SGD	0.0989	0.3146	55.4691	?	
hazelcast	SimpleLogistic	0.1785	0.2905	100.0663	0.861	
hazelcast	SMO	0.0989	0.3146	55.4691	?	
hazelcast	VotedPerceptron	0.1113	0.3336	62.396	0.855	
hazelcast	DecisonTable	0.167	0.2854	93.6372	0.873	
hazelcast	OneR	0.1011	0.3179	56.6691	0.863	0.5238
hazelcast	NaiveBayes	0.1377	0.354	77.1845	0.86	0.6498
hazelcast	J48	0.1267	0.2808	71.0382	0.898	0.7931

Table 22: GraphEvo Dataset Machine Learning Results 3 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
ivy	RandomForest	0.1488	0.2808	81.2908	0.879	0.8078
ivy	PART	0.1381	0.3091	75.4592	0.881	0.7103
ivy	NaiveBayesMultinomial	0.297	0.5425	162.2915	0.76	
ivy	Logistic	0.1658	0.3048	90.6159	0.861	
ivy	SGD	0.1047	0.3236	57.2053	0.849	
ivy	SimpleLogistic	0.276	0.3696	150.8064	0.856	
ivy	SMO	0.1029	0.3208	56.219	0.85	
ivy	VotedPerceptron	0.1101	0.3318	60.1643	0.858	
ivy	DecisonTable	0.1737	0.3057	94.8912	0.852	
ivy	OneR	0.1191	0.3452	65.0957	0.85	0.5137
ivy	NaiveBayes	0.1472	0.3735	80.4429	0.858	0.6206
ivy	J48	0.1461	0.3121	79.8338	0.876	0.6499
junit	RandomForest	0.0487	0.1614	56.5594	0.965	0.8919
junit	PART	0.0634	0.2027	73.6153	0.952	0.7707
junit	NaiveBayesMultinomial	0.261	0.5002	302.9793	0.815	
junit	Logistic	0.0817	0.2038	94.8389	?	
junit	SGD	0.0446	0.2113	51.8035	?	
junit	SimpleLogistic	0.5	0.5	580.3097	?	
junit	SMO	0.0446	0.2113	51.8035	?	
junit	VotedPerceptron	0.0446	0.2113	51.8035	?	
junit	DecisonTable	0.0861	0.2065	99.9392	?	
junit	OneR	0.0475	0.2179	55.1101	0.935	0.5188
junit	NaiveBayes	0.1609	0.3355	186.7695	0.896	0.665
junit	J48	0.0655	0.1881	76.0436	0.953	0.714
lucene	RandomForest	0.3138	0.4159	63.9943	0.731	0.8142
lucene	PART	0.329	0.4714	67.0917	0.709	0.7453
lucene	NaiveBayesMultinomial	0.3767	0.6039	76.8159	0.611	
lucene	Logistic	0.4051	0.4604	82.6257	0.684	
lucene	SGD	0.3497	0.5913	71.3141	0.625	
lucene	SimpleLogistic	0.4086	0.4576	83.3282	0.664	
lucene	SMO	0.3418	0.5846	69.7007	0.633	
lucene	VotedPerceptron	0.3284	0.5726	66.9769	0.658	
lucene	DecisonTable	0.3903	0.4464	79.6049	0.71	
lucene	OneR	0.3639	0.6033	74.2183	0.632	0.6203
lucene	NaiveBayes	0.3427	0.5636	69.8842	0.636	0.6976
lucene	J48	0.3428	0.4867	69.9072	0.704	0.7304

Table 23: GraphEvo Dataset Machine Learning Results 4 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
mapdb	RandomForest	0.1269	0.2739	70.1248	0.878	0.8649
mapdb	PART	0.1524	0.3169	84.2714	0.859	0.7304
mapdb	NaiveBayesMultinomial	0.3306	0.5739	182.7524	0.732	
mapdb	Logistic	0.158	0.2877	87.3221	0.873	
mapdb	SGD	0.1011	0.318	55.9064	0.867	
mapdb	SimpleLogistic	0.188	0.2976	103.9168	0.858	
mapdb	SMO	0.1042	0.3228	57.6178	0.861	
mapdb	VotedPerceptron	0.1011	0.318	55.9064	0.852	
mapdb	DecisonTable	0.1742	0.2974	96.2823	0.857	
mapdb	OneR	0.1146	0.3385	63.3226	0.847	0.4966
mapdb	NaiveBayes	0.1579	0.3864	87.2615	0.847	0.6752
mapdb	J48	0.1643	0.3154	90.7972	0.856	0.5799
mcMMO	RandomForest	0.1966	0.3525	87.8319	0.811	0.7058
mcMMO	PART	0.2119	0.3339	94.6918	0.823	0.6102
mcMMO	OneR	0.1419	0.3766	63.3851	0.809	0.4998
mcMMO	NaiveBayesMultinomial	0.3773	0.6127	168.5635	0.684	
mcMMO	Logistic	0.2032	0.3276	90.7766	0.823	
mcMMO	SGD	0.1279	0.3576	57.1505	?	
mcMMO	SimpleLogistic	0.3766	0.4244	168.2574	0.817	
mcMMO	SMO	0.1279	0.3576	57.1505	?	
mcMMO	VotedPerceptron	0.1372	0.3704	61.3069	0.81	
mcMMO	DecisonTable	0.2238	0.334	100.0124	?	
mcMMO	NaiveBayes	0.2231	0.4183	99.6925	0.809	0.7319
mcMMO	J48	0.2172	0.3394	97.0372	0.819	0.5075
netty	RandomForest	0.0905	0.2226	51.2589	0.933	0.9131
netty	PART	0.1147	0.2674	64.9385	0.906	0.8479
netty	OneR	0.1	0.3162	56.6224	0.867	0.5315
netty	NaiveBayesMultinomial	0.22	0.4627	124.5763	0.799	
netty	Logistic	0.151	0.2751	85.5407	0.858	
netty	SGD	0.0977	0.3126	55.361	?	
netty	SimpleLogistic	0.1881	0.3051	106.5348	0.857	
netty	SMO	0.0977	0.3126	55.361	?	
netty	VotedPerceptron	0.1057	0.3251	59.8459	0.852	
netty	DecisonTable	0.1679	0.2873	95.1061	0.859	
netty	NaiveBayes	0.3683	0.4891	208.5871	0.72	0.7305
netty	J48	0.111	0.2579	62.8828	0.917	0.7392

Table 24: GraphEvo Dataset Machine Learning Results 5 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
orientdb	RandomForest	0.1488	0.2878	47.4166	0.886	0.9120
orientdb	PART	0.188	0.3297	59.9135	0.835	0.8545
orientdb	OneR	0.1798	0.424	57.3024	0.793	0.6130
orientdb	NaiveBayesMultinomial	0.4113	0.6202	131.0874	0.63	
orientdb	Logistic	0.2414	0.3509	76.9351	0.783	
orientdb	SGD	0.1947	0.4413	62.055	?	
orientdb	SimpleLogistic	0.262	0.3568	83.5094	0.757	
orientdb	SMO	0.1947	0.4413	62.055	?	
orientdb	VotedPerceptron	0.1951	0.4417	62.1908	0.718	
orientdb	DecisonTable	0.247	0.3396	78.7225	0.838	
orientdb	NaiveBayes	0.3559	0.4592	113.4096	0.736	0.7485
orientdb	J48	0.1646	0.316	52.4642	0.868	0.8478
pbeans	RandomForest	0.3199	0.4139	71.1922	0.733	0.7813
pbeans	PART	0.3219	0.505	71.6416	0.674	0.6908
pbeans	OneR	0.1875	0.433	41.7293	0.804	0.7586
pbeans	NaiveBayesMultinomial	0.2749	0.5226	61.1792	0.715	
pbeans	Logistic	0.2998	0.4826	66.7139	0.711	
pbeans	SGD	0.3125	0.559	69.5489	0.662	
pbeans	SimpleLogistic	0.347	0.4596	77.2271	0.679	
pbeans	SMO	0.3	0.5477	66.7669	0.657	
pbeans	VotedPerceptron	0.2798	0.5247	62.2621	0.714	
pbeans	DecisonTable	0.3148	0.4074	70.0681	0.765	
pbeans	NaiveBayes	0.2994	0.5262	66.6234	0.672	0.7893
pbeans	J48	0.3413	0.5045	75.9679	0.672	0.6534
poi	RandomForest	0.3281	0.4357	73.7728	0.687	0.7699
poi	PART	0.337	0.4751	75.7907	0.718	0.7015
poi	OneR	0.345	0.5873	77.5728	0.625	0.5581
poi	NaiveBayesMultinomial	0.3279	0.569	73.7368	0.678	
poi	Logistic	0.3559	0.4336	80.0438	0.712	
poi	SGD	0.2752	0.5246	61.8839	0.71	
poi	SimpleLogistic	0.3542	0.4244	79.6443	0.723	
poi	SMO	0.2791	0.5283	62.7555	0.705	
poi	VotedPerceptron	0.3314	0.5746	74.5148	0.663	
poi	DecisonTable	0.3725	0.4473	83.7705	0.687	
poi	NaiveBayes	0.2907	0.5261	65.3782	0.671	0.7070
poi	J48	0.3387	0.5019	76.162	0.706	0.6449

Table 25: GraphEvo Dataset Machine Learning Results 6 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
titan	RandomForest	0.0695	0.2021	79.6121	0.938	0.8578
titan	PART	0.0789	0.2066	90.3996	0.933	0.7870
titan	OneR	0.0442	0.2103	50.666	0.938	0.5322
titan	NaiveBayes	0.1061	0.3046	121.5577	0.91	0.6759
titan	NaiveBayesMultinomial	0.3958	0.6217	453.5881	0.72	
titan	Logistic	0.0839	0.2073	96.1685	0.932	
titan	SGD	0.0455	0.2134	52.2014	?	
titan	SimpleLogistic	0.5	0.5	573.0637	?	
titan	SMO	0.0455	0.2134	52.2014	?	
titan	VotedPerceptron	0.0479	0.2188	54.8882	0.931	
titan	DecisonTable	0.0873	0.2085	100.076	?	
titan	J48	0.0865	0.2092	99.1435	0.933	0.5280
velocity	RandomForest	0.3832	0.4735	76.6797	0.689	0.6862
velocity	PART	0.3867	0.475	77.3648	0.643	0.7020
velocity	OneR	0.4706	0.686	94.1568	0.529	0.5294
velocity	NaiveBayes	0.3767	0.5658	75.3796	0.61	0.6515
velocity	NaiveBayesMultinomial	0.418	0.633	83.6288	0.57	
velocity	Logistic	0.3857	0.4541	77.1661	0.668	
velocity	SGD	0.3176	0.5636	63.5559	0.682	
velocity	SimpleLogistic	0.4057	0.4503	81.1726	0.703	
velocity	SMO	0.2941	0.5423	58.848	0.705	
velocity	VotedPerceptron	0.4314	0.6554	86.3162	0.569	
velocity	DecisonTable	0.4224	0.4675	84.5178	0.675	
velocity	J48	0.3824	0.4745	76.5074	0.661	0.6879
xalan	RandomForest	0.2542	0.3821	54.3382	0.776	0.8709
xalan	PART	0.2641	0.3866	56.4509	0.758	0.8472
xalan	OneR	0.3302	0.5746	70.5674	0.654	0.6166
xalan	NaiveBayes	0.3732	0.4806	79.7703	0.665	0.7114
xalan	NaiveBayesMultinomial	0.3782	0.6129	80.8362	0.624	
xalan	Logistic	0.3718	0.4314	79.4591	0.7	
xalan	SGD	0.2718	0.5213	58.0914	0.698	
xalan	SimpleLogistic	0.3774	0.4329	80.6555	0.699	
xalan	SMO	0.2729	0.5224	58.3253	0.695	
xalan	VotedPerceptron	0.355	0.5957	75.8683	0.63	
xalan	DecisonTable	0.3457	0.4171	73.8953	0.712	
xalan	J48	0.2307	0.3874	49.3105	0.802	0.8483

Table 26: GraphEvo Dataset Machine Learning Results 7 Of 7

Application	MLAlgorithm	MAE	RMSE	RAE	F-Measure	AUC
xerces	RandomForest	0.1902	0.3278	73.1824	0.826	0.8005
xerces	PART	0.22	0.3694	84.6511	0.806	0.6018
xerces	OneR	0.1575	0.3969	60.6049	0.807	0.5588
xerces	NaiveBayes	0.1997	0.433	76.8415	0.786	0.5879
xerces	NaiveBayesMultinomial	0.3846	0.6143	147.995	0.669	
xerces	Logistic	0.2468	0.3594	94.9647	0.79	
xerces	SGD	0.159	0.3988	61.1933	0.774	
xerces	SimpleLogistic	0.2695	0.3595	103.7007	0.785	
xerces	SMO	0.1529	0.391	58.8397	?	
xerces	VotedPerceptron	0.1667	0.4082	64.1353	0.773	
xerces	DecisonTable	0.019	0.912	0.775	0.775	
xerces	J48	0.227	0.3752	87.3508	0.813	0.5330

Bibliography

- [1] Jacob Krüger. “Tackling knowledge needs during software evolution”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 1244–1246.
- [2] Xin Xia et al. “Measuring program comprehension: A large-scale field study with professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (2017), pp. 951–976.
- [3] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.
- [4] Ana M Fernández-Sáez, Michel RV Chaudron, and Marcela Genero. “An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles”. In: *Empirical Software Engineering* (2018), pp. 1–65.
- [5] Herb Krasner CISQ. *The Cost of Poor Quality Software in the US: A 2018 Report*. 2018. URL: <https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf>.
- [6] Miltiadis Allamanis et al. “A survey of machine learning for big code and naturalness”. In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–37.
- [7] Q. Song et al. “A general software defect-proneness prediction framework”. In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 356–370.
- [8] Fumio Akiyama. “An example of software system debugging.” In: *IFIP Congress (1)*. Vol. 71. 1971, pp. 353–359.
- [9] Tim Menzies et al. “Defect prediction from static code features: current results, limitations, new approaches”. In: *Automated Software Engineering* 17.4 (2010), pp. 375–407.

- [10] Abdullah Alsaeedi and Mohammad Zubair Khan. “Software defect prediction using supervised machine learning and ensemble techniques: A comparative study”. In: *Journal of Software Engineering and Applications* 12.5 (2019), pp. 85–100.
- [11] Foyzur Rahman and Premkumar Devanbu. “How, and why, process metrics are better”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.
- [12] Tracy Hall et al. “A systematic literature review on fault prediction performance in software engineering”. In: *IEEE Transactions on Software Engineering* 38.6 (2011), pp. 1276–1304.
- [13] Nicolas Le Novere. “Quantitative and logic modelling of molecular and gene networks”. In: *Nature Reviews Genetics* 16.3 (2015), p. 146.
- [14] Yini Wang et al. “Modeling the propagation of worms in networks: A survey”. In: *IEEE Communications Surveys & Tutorials* 16.2 (2014), pp. 942–960.
- [15] Leman Akoglu, Hanghang Tong, and Danai Koutra. “Graph based anomaly detection and description: a survey”. In: *Data mining and knowledge discovery* 29.3 (2015), pp. 626–688.
- [16] Pamela Bhattacharya et al. “Graph-based analysis and prediction for software evolution”. In: *Proceedings of the 34th International Conference on Software Engineering. ICSE '12*. Zurich, Switzerland: IEEE Press, 2012, pp. 419–429. ISBN: 978-1-4673-1067-3.
- [17] David Grove et al. “Call graph construction in object-oriented languages”. In: *ACM SIGPLAN Notices* 32.10 (1997), pp. 108–124.
- [18] J Sayyad Shirabad and Tim J Menzies. “The PROMISE repository of software engineering databases”. In: *School of Information Technology and Engineering, University of Ottawa, Canada* 24 (2005).

- [19] Rudolf Ferenc et al. “A public unified bug dataset for java”. In: *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE’18. Oulu, Finland: Association for Computing Machinery, 2018, 12â21. ISBN: 9781450365932. DOI: 10.1145/3273934.3273936. URL: <https://doi.org/10.1145/3273934.3273936>.
- [20] David Gray et al. “The misuse of the NASA metrics data program data sets for automated software defect prediction”. In: *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*. IET. 2011, pp. 96–103.
- [21] Elena N Akimova et al. “A survey on software defect prediction using deep learning”. In: *Mathematics* 9.11 (2021), p. 1180.
- [22] Song Wang, Taiyue Liu, and Lin Tan. “Automatically learning semantic features for defect prediction”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 297–308.
- [23] Safa Omri and Carsten Sinz. “Deep learning for software defect prediction: a survey”. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020, pp. 209–214.
- [24] Yanming Yang et al. “A survey on deep learning for software engineering”. In: *ACM Computing Surveys (CSUR)* (2020).
- [25] Georgios Gousios, Martin Pinzger, and Arie van Deursen. “An exploratory study of the pull-based software development model”. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 345–355.
- [26] Oleksii Kononenko et al. “Studying pull request merges: a case study of shopify’s active merchant”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 2018, pp. 124–133.
- [27] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. “A preliminary analysis on the effects of propensity to trust in distributed software development”. In: *2017*

- IEEE 12th international conference on global software engineering (ICGSE)*. IEEE. 2017, pp. 56–60.
- [28] Jacob Krüger et al. “Effects of explicit feature traceability on program comprehension”. In: *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019.
- [29] Roger S Pressman. *Software engineering: a practitioner’s approach*. Palgrave Macmillan, 2005.
- [30] Keith H Bennett and Václav T Rajlich. “Software maintenance and evolution: a roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 73–87.
- [31] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. “The inevitable stability of software change”. In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE. 2007, pp. 4–13.
- [32] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. “Code2graph: automatic generation of static call graphs for Python source code”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 880–883.
- [33] Gharib Gharibi, Rakan Alanazi, and Yugyung Lee. “Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 4016–4025.
- [34] Lei Wang et al. “Linux kernels as complex networks: A novel method to study evolution”. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE. 2009, pp. 41–50.
- [35] JUnit. 2019. URL: <https://junit.org/junit4/index.html>.
- [36] jMock. 2019. URL: <http://jmock.org/>.
- [37] Jgap. 2019. URL: <https://sourceforge.net/p/jgap/wiki/Home/>.

- [38] Guava. 2019. URL: <https://github.com/google/guava>.
- [39] SLF4J. 2019. URL: <https://www.slf4j.org/>.
- [40] Java CallGraph Constructor. 2018. URL: <https://github.com/gousiosg/java-callgraph>.
- [41] Emden R Gansner and Stephen C North. “An open graph visualization system and its applications to software engineering”. In: *Software: practice and experience* 30.11 (2000), pp. 1203–1233.
- [42] John Ellson et al. “Graphviz—open source graph drawing tools”. In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 483–484.
- [43] James P Bagrow and Erik M Bollt. “An information-theoretic, all-scales approach to comparing networks”. In: *arXiv preprint arXiv:1804.03665* (2018).
- [44] Christopher R Myers. “Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs”. In: *Physical Review E* 68.4 (2003), p. 046116.
- [45] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. “Power laws in software”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18.1 (2008), p. 2.
- [46] Ron Milo et al. “Network motifs: simple building blocks of complex networks”. In: *Science* 298.5594 (2002), pp. 824–827.
- [47] Google Big Query. 2018. URL: <https://cloud.google.com/bigquery/>.
- [48] Priya Mahadevan et al. “Orbis: rescaling degree correlations to generate annotated internet topologies”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. 4. ACM. 2007, pp. 325–336.

- [49] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. “Code2graph: Automatic Generation of Static Call Graphs for Python Source Code”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, 880â883. ISBN: 9781450359375. DOI: 10.1145/3238147.3240484. URL: <https://doi.org/10.1145/3238147.3240484>.
- [50] Apache Cassandra. 2019. URL: <https://junit.org/junit4/index.html>.
- [51] Apache Camel. 2019. URL: <https://camel.apache.org/>.
- [52] Apache Zookeeper. 2019. URL: <https://zookeeper.apache.org/>.
- [53] Lei Qiao et al. “Deep learning based software defect prediction”. In: *Neurocomputing* 385 (2020), pp. 100–110.
- [54] T. Zimmermann, R. Premraj, and A. Zeller. “Predicting defects for eclipse”. In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. 2007, pp. 9–9.
- [55] H. Zhang. “An investigation of the relationships between lines of code and defects”. In: *2009 IEEE International Conference on Software Maintenance*. 2009, pp. 274–283.
- [56] Yuming Zhou and Hareton Leung. “Empirical analysis of object-oriented design metrics for predicting high and low severity faults”. In: *IEEE Transactions on Software Engineering* 32.10 (2006), pp. 771–789.
- [57] R. Subramanyam and M. S. Krishnan. “Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects”. In: *IEEE Transactions on Software Engineering* 29.4 (2003), pp. 297–310.
- [58] R. Moser, W. Pedrycz, and G. Succi. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 181–190.

- [59] Sandeep Krishnan et al. “Are Change Metrics Good Predictors for an Evolving Software Product Line?” In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. Promise '11. Banff, Alberta, Canada: Association for Computing Machinery, 2011. ISBN: 9781450307093. DOI: 10.1145/2020390.2020397. URL: <https://doi.org/10.1145/2020390.2020397>.
- [60] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. “Does measuring code change improve fault prediction?” In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. Promise '11. Banff, Alberta, Canada: Association for Computing Machinery, 2011. ISBN: 9781450307093. DOI: 10.1145/2020390.2020392. URL: <https://doi.org/10.1145/2020390.2020392>.
- [61] N. Nagappan et al. “Change bursts as defect predictors”. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 2010, pp. 309–318.
- [62] Shinsuke Matsumoto et al. “An analysis of developer metrics for fault prediction”. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. PROMISE '10. Timișoara, Romania: Association for Computing Machinery, 2010. ISBN: 9781450304047. DOI: 10.1145/1868328.1868356. URL: <https://doi.org/10.1145/1868328.1868356>.
- [63] Thomas Zimmermann and Nachiappan Nagappan. “Predicting defects using network analysis on dependency graphs”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, 531â540. ISBN: 9781605580791. DOI: 10.1145/1368088.1368161. URL: <https://doi.org/10.1145/1368088.1368161>.
- [64] R. Premraj and K. Herzig. “Network versus code metrics to predict defects: A replication study”. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. 2011, pp. 215–224.
- [65] Thomas Zimmermann and Nachiappan Nagappan. “Predicting defects with program dependencies”. In: *ESEM '09*. USA: IEEE Computer Society, 2009, 435â438.

ISBN: 9781424448425. DOI: 10.1109/ESEM.2009.5316024. URL: <https://doi.org/10.1109/ESEM.2009.5316024>.

- [66] James P Bagrow and Erik M Bollt. “An information-theoretic, all-scales approach to comparing networks”. In: *Applied Network Science* 4.1 (2019), pp. 1–15.
- [67] V. Walunj et al. “Graphevo: Characterizing and understanding software evolution using call graphs”. In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 4799–4807.
- [68] Y. Kamei and E. Shihab. “Defect prediction: Accomplishments and future challenges”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. 2016, pp. 33–45.
- [69] Shamsul Huda et al. “A framework for software defect prediction and metric selection”. In: *IEEE access* 6 (2017), pp. 2844–2858.
- [70] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. “Progress on approaches to software defect prediction”. In: *IET Software* 12.3 (2018), pp. 161–175.
- [71] C Manjula and Lilly Florence. “Deep neural network based hybrid approach for software defect prediction using software metrics”. In: *Cluster Computing* 22.4 (2019), pp. 9847–9863.
- [72] Mingming Chen and Yutao Ma. “An empirical study on predicting defect numbers.” In: *SEKE*. 2015, pp. 397–402.
- [73] Santosh Singh Rathore and Sandeep Kumar. “A decision tree regression based approach for the number of software faults prediction”. In: *ACM SIGSOFT Software Engineering Notes* 41.1 (2016), pp. 1–6.
- [74] Yi Yao et al. “Cross-project dynamic defect prediction model for crowdsourced test”. In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 2020, pp. 223–230. DOI: 10.1109/QRS51102.2020.00040.

- [75] Awni Hammouri et al. “Software bug prediction using machine learning approach”. In: *International journal of advanced computer science and applications* 9.2 (2018), pp. 78–83.
- [76] Santosh S Rathore and Sandeep Kumar. “An empirical study of some software fault prediction techniques for the number of faults prediction”. In: *Soft Computing* 21.24 (2017), pp. 7417–7434.
- [77] Vladimir Vapnik, Steven E Golowich, Alex Smola, et al. “Support vector method for function approximation, regression estimation, and signal processing”. In: *Advances in neural information processing systems* (1997), pp. 281–287.
- [78] Wen Zhang et al. “SamEn-SVR: using sample entropy and support vector regression for bug number prediction”. In: *IET Software* 12.3 (2018), pp. 183–189.
- [79] Yang Cao et al. “An improved twin support vector machine based on multi-objective cuckoo search for software defect prediction”. In: *International Journal of Bio-Inspired Computation* 11.4 (2018), pp. 282–291.
- [80] Gang Zhao and Jeff Huang. “Deepsim: Deep learning code functional similarity”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018*. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 141â151. ISBN: 9781450355735. DOI: 10.1145/3236024.3236068. URL: <https://doi.org/10.1145/3236024.3236068>.
- [81] L. Guo et al. “Deep convolutional transfer learning network: A new method for intelligent fault diagnosis of machines with unlabeled data”. In: *IEEE Transactions on Industrial Electronics* 66.9 (2019), pp. 7316–7325.
- [82] Ali Arshad, Saman Riaz, and Licheng Jiao. “Semi-supervised deep fuzzy c-mean clustering for imbalanced multi-class classification”. In: *IEEE Access* 7 (2019), pp. 28100–28112.

- [83] X. Gu, H. Zhang, and S. Kim. “Deep code search”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 933–944.
- [84] Martin White et al. “Deep learning code fragments for code clone detection”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, 87â98. ISBN: 9781450338455. DOI: 10.1145/2970276.2970326. URL: <https://doi.org/10.1145/2970276.2970326>.
- [85] Michael R Lyu et al. *Handbook of software reliability engineering*. Vol. 222. IEEE computer society press CA, 1996.
- [86] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. “Facilitating program comprehension with call graph multilevel hierarchical abstractions”. In: *Journal of Systems and Software* 176 (2021), p. 110945. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2021.110945>. URL: <https://www.sciencedirect.com/science/article/pii/S016412122100042X>.
- [87] Miloš Savić, Mirjana Ivanović, and Lakhmi C Jain. “Analysis of software networks”. In: *Complex Networks in Software, Knowledge, and Social Systems*. Springer, 2019, pp. 59–141.
- [88] Uri Alon et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.
- [89] Mattia Tantardini et al. “Comparing methods for comparing networks”. In: *Scientific reports* 9.1 (2019), pp. 1–19.
- [90] Harrison Hartle et al. “Network comparison and the within-ensemble graph distance”. In: *Proceedings of the Royal Society A* 476.2243 (2020), p. 20190744.
- [91] Stefan McCabe et al. “netrd: A library for network reconstruction and graph distances”. In: *arXiv preprint arXiv:2010.16019* (2020).
- [92] Richard W Hamming. “Error detecting and error correcting codes”. In: *The Bell system technical journal* 29.2 (1950), pp. 147–160.

- [93] Godfrey N Lance and William T Williams. “Computer programs for hierarchical polythetic classification (âsimilarity analysesâ)”. In: *The Computer Journal* 9.1 (1966), pp. 60–64.
- [94] Paul Jaccard. “Etude de la distribution florale dans une portion des Alpes et du Jura”. In: *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (Jan. 1901), pp. 547–579. DOI: 10.5169/seals-266450.
- [95] Danai Koutra et al. “Deltacon: Principled massive-graph similarity function with attribution”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10.3 (2016), pp. 1–43.
- [96] Anida Sarajlić et al. “Graphlet-based characterization of directed networks”. In: *Scientific reports* 6.1 (2016), pp. 1–14.
- [97] Oleksii Kuchaiev et al. “Topological network alignment uncovers biological function and phylogeny”. In: *Journal of the Royal Society Interface* 7.50 (2010), pp. 1341–1354.
- [98] Oleksii Kuchaiev and Nataša Pržulj. “Integrative network alignment reveals large regions of global network similarity in yeast and human”. In: *Bioinformatics* 27.10 (2011), pp. 1390–1396.
- [99] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. “Natural software revisited”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 37–48.
- [100] Giulio Concas et al. “Power-laws in a large object-oriented software system”. In: *IEEE Transactions on Software Engineering* 33.10 (2007), pp. 687–708.
- [101] Sergi Valverde and Ricard V Solé. “Hierarchical small worlds in software architecture”. In: *arXiv preprint cond-mat/0307278* (2003).
- [102] Miloš Savić, Mirjana Ivanović, and Miloš Radovanović. “Analysis of high structural class coupling in object-oriented software systems”. In: *Computing* 99.11 (2017), pp. 1055–1079.

- [103] Lewi Stone, Daniel Simberloff, and Yael Artzy-Randrup. “Network motifs and their origins”. In: *PLoS computational biology* 15.4 (2019), e1006749.
- [104] Barbara Russo. “Profiling call changes via motif mining”. In: *Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18*. Gothenburg, Sweden: Association for Computing Machinery, 2018, 203â214. ISBN: 9781450357166. DOI: 10.1145/3196398.3196426. URL: <https://doi.org/10.1145/3196398.3196426>.
- [105] Xinbo Gao et al. “A survey of graph edit distance”. In: *Pattern Analysis and applications* 13.1 (2010), pp. 113–129.
- [106] Kansuke Ikehara and Aaron Clauset. “Characterizing the structural diversity of complex networks across domains”. In: *arXiv preprint arXiv:1710.11304* (2017).
- [107] Jureczko M. Spinellis D. 2018. URL: <http://gromit.iar.pwr.wroc.pl/p%20inf/ckjm/metric.html>.
- [108] David W Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.
- [109] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.
- [110] Santosh S Rathore and Sandeep Kumar. “A study on software fault prediction techniques”. In: *Artificial Intelligence Review* 51.2 (2019), pp. 255–327.
- [111] Hadeel Alsolai and Marc Roper. “A systematic literature review of machine learning techniques for software maintainability prediction”. In: *Information and Software Technology* 119 (2020), p. 106214. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2019.106214>. URL: <https://www.sciencedirect.com/science/article/pii/S0950584919302228>.

- [112] Gharib Gharibi et al. “ModelKB: towards automated management of the modeling lifecycle in deep learning”. In: *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE. 2019, pp. 28–34.
- [113] Gharib Gharibi et al. “Automated end-to-end management of the modeling lifecycle in deep learning”. In: *Empirical Software Engineering* 26.2 (2021), pp. 1–33.
- [114] Gharib Gharibi et al. “Automated Management of Deep Learning Experiments”. In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning. DEEM’19*. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450367974. DOI: 10.1145/3329486.3329495. URL: <https://doi.org/10.1145/3329486.3329495>.
- [115] Gharib Gharibi et al. “ModelKB: Towards Automated Management of the Modeling Lifecycle in Deep Learning”. In: *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. 2019, pp. 28–34. DOI: 10.1109/RAISE.2019.00013.
- [116] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [117] Hamza Turabieh, Majdi Mafarja, and Xiaodong Li. “Iterated feature selection algorithms with layered recurrent neural network for software fault prediction”. In: *Expert systems with applications* 122 (2019), pp. 27–42.
- [118] Zimin Chen and Martin Monperrus. “A literature study of embeddings on source code”. In: *arXiv preprint arXiv:1904.03061* (2019).
- [119] Jiaxi Xu, Fei Wang, and Jun Ai. “Defect prediction with semantics and context features of codes based on graph representation learning”. In: *IEEE Transactions on Reliability* 70.2 (2020), pp. 613–625.

- [120] Rudolf Ferenc et al. “An automatically created novel bug dataset and its validation in bug prediction”. In: *Journal of Systems and Software* 169 (2020), p. 110691. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2020.110691>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121220301436>.
- [121] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. “A public bug database of github projects and its application in bug prediction”. In: *International Conference on Computational Science and Its Applications*. Springer. 2016, pp. 625–638.
- [122] Yuanxun Shao et al. “A novel software defect prediction based on atomic class-association rule mining”. In: *Expert Systems with Applications* 114 (2018), pp. 237–254.
- [123] Rudolf Ferenc et al. “A public unified bug dataset for Java”. In: *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2018, pp. 12–21.
- [124] Samuel Benton, Ali Ghanbari, and Lingming Zhang. “Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 47–50.
- [125] Lech Madeyski and Marian Jureczko. “Which process metrics can significantly improve defect prediction models? An empirical study”. In: *Software Quality Journal* 23.3 (2015), pp. 393–422.
- [126] Meiliana et al. “Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset”. In: *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*. 2017, pp. 19–23. DOI: 10.1109/CYBERNETICSCOM.2017.8311708.
- [127] Peng He et al. “An empirical study on software defect prediction with a simplified metric set”. In: *Information and Software Technology* 59 (2015), pp. 170–190.

- [128] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. “Predicting defects for eclipse”. In: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. PROMISE '07. USA: IEEE Computer Society, 2007, p. 9. ISBN: 0769529542. DOI: 10.1109/PROMISE.2007.10. URL: <https://doi.org/10.1109/PROMISE.2007.10>.
- [129] Marco D’Ambros, Michele Lanza, and Romain Robbes. “An extensive comparison of bug prediction approaches”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 31–41. DOI: 10.1109/MSR.2010.5463279.
- [130] Tracy Hall et al. “Some code smells have a significant but small effect on faults”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (2014), pp. 1–39.
- [131] Valentin Dallmeier and Thomas Zimmermann. “Extraction of bug localization benchmarks from history”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 433–436.
- [132] Tibor Gyimóthy, Rudolf Ferenc, and Istvan Siket. “Empirical validation of object-oriented metrics on open source software for fault prediction”. In: *IEEE Transactions on Software engineering* 31.10 (2005), pp. 897–910.
- [133] Phiradet Bangcharoensap et al. “Locating source code to be fixed based on initial bug reports-a case study on the eclipse project”. In: *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. IEEE. 2012, pp. 10–15.
- [134] IntelliJ IDEA. *IntelliJ IDEA Java Decompiler*. Github [java-decompiler/engine](https://github.com/java-decompiler/engine). 2021.
- [135] Maurício Aniche. *Java code metrics calculator (CK)*. Github: [mauricioaniche/ck/](https://github.com/mauricioaniche/ck/). 2015.

- [136] Marian Jureczko. *Calculating Chidamber and Kemerer Java Metrics (and many other metrics)*. Available in <https://github.com/mjureczko/CKJM-extended>. 2020.
- [137] Steffen Herbold et al. “A fine-grained data set and analysis of tangling in bug fixing commits”. In: *Empirical Software Engineering* 27.6 (2022), pp. 1–49.
- [138] G. Holmes, A. Donkin, and I.H. Witten. “WEKA: a machine learning workbench”. In: *Proceedings of ANZIIS '94 - Australian New Zealand Intelligent Information Systems Conference*. 1994, pp. 357–361. DOI: 10.1109/ANZIIS.1994.396988.
- [139] Victor Sobreira et al. “Dissection of a bug dataset: Anatomy of 395 patches from defects4j”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 130–140.
- [140] Davide Spadini et al. “When testing meets code review: Why and how developers review tests”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 677–687.
- [141] Vijay Walunj et al. “Graphevo: Characterizing and understanding software evolution using call graphs”. In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 4799–4807.
- [142] Daricélio Moreira Soares et al. “Rejection factors of pull requests filed by core team developers in software projects with high acceptance rates”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2015, pp. 960–965.
- [143] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. “A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects”. In: *Empirical Software Engineering* 25.6 (2020), pp. 5137–5192.

- [144] Lorenzo Gasparini et al. “ChangeViz: Enhancing the GitHub Pull Request Interface with Method Call Information”. In: *2021 Working Conference on Software Visualization (VISSOFT)*. 2021, pp. 115–119. DOI: 10.1109/VISSOFT52517.2021.00022.
- [145] R. Kosara et al. “Visualization viewpoints”. In: *IEEE Computer Graphics and Applications* 23.4 (2003), pp. 20–25. DOI: 10.1109/MCG.2003.1210860.
- [146] Xunhui Zhang, Ayushi Rastogi, and Yue Yu. “On the Shoulders of Giants: A New Dataset for Pull-Based Development Research”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 543â547. ISBN: 9781450375177. DOI: 10.1145/3379597.3387489. URL: <https://doi.org/10.1145/3379597.3387489>.
- [147] Otávio Augusto Lazzarini Lemos et al. “Visualization, Analysis, and Testing of Java and AspectJ Programs with Multi-level System Graphs”. In: *2013 27th Brazilian Symposium on Software Engineering*. 2013, pp. 49–58. DOI: 10.1109/SBES.2013.7.
- [148] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. “Code2graph: Automatic Generation of Static Call Graphs for Python Source Code”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 880–883. DOI: 10.1145/3238147.3240484.
- [149] Thomas D. LaToza and Brad A. Myers. “Visualizing call graphs”. In: *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2011, pp. 117–124. DOI: 10.1109/VLHCC.2011.6070388.
- [150] Denae Ford et al. “Beyond the Code Itself: How Programmers Really Look at Pull Requests”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. 2019, pp. 51–60. DOI: 10.1109/ICSE-SEIS.2019.00014.

- [151] Michael D. Shah and Samuel Z. Guyer. “An Interactive Microarray Call-Graph Visualization”. In: *2016 IEEE Working Conference on Software Visualization (VIS-SOFT)*. 2016, pp. 86–90. DOI: 10.1109/VISSOFT.2016.14.
- [152] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. “Facilitating program comprehension with call graph multilevel hierarchical abstractions”. In: *Journal of Systems and Software* 176 (2021), p. 110945.
- [153] Rakan Alanazi. *Software Analytics for Improving Program Comprehension*. University of Missouri-Kansas City, 2021.
- [154] Pamela Bhattacharya et al. “Graph-based analysis and prediction for software evolution”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE press. 2012, pp. 419–429.
- [155] J. Sayyad Shirabad and T.J. Menzies. *The PROMISE Repository of Software Engineering Databases*. School of Information Technology and Engineering, University of Ottawa, Canada. 2005. URL: <http://promise.site.uottawa.ca/SERepository>.
- [156] Matias Martinez et al. “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset”. In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964.
- [157] Md Alamgir Kabir, Muaan Ur Rehman, and Shariful Islam Majumdar. “An analytical and comparative study of software usability quality factors”. In: *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. 2016, pp. 800–803. DOI: 10.1109/ICSESS.2016.7883188.

VITA

Vijay Walunj is a Teladoc Health R&D engineering leader and currently building a messaging platform for enterprise-wide communications at Teladoc. He earned a Master's degree in 2013 from UMKC and a Bachelor's of Computer Engineering in 2008 from the University of Mumbai. He has also completed Harvard University's data science certificate courses. Among his research interests are software engineering, analytics, artificial intelligence, microservices, 5G, network slicing, and security engineering. In his research, he applies AI to improve the effectiveness of various software engineering tasks and develop tools/frameworks for better AI software development. He has published three journal papers and eight conference papers in his research. He has also worked in the software industry for over a decade and written thousands of lines of code.