Empirical Software Engineering (2022) 27:118
https://doi.org/10.1007/s10664-022-10147-0

Defect prediction using deep learning with Network Portrait Divergence for software evolution

Vijay Walunj[1] · Gharib Gharibi[1] · Rakan Alanazi[1,2] · Yugyung Lee[1]

Accepted: 28 February 2022

Abstract
Understanding software evolution is essential for software development tasks, including debugging, maintenance, and testing. As a software system evolves, it grows in size and becomes more complex, hindering its comprehension. Researchers proposed several approaches for software quality analysis based on software metrics. One of the primary practices 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 paper, 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*. Particularly, *GraphEvoDef* provides three major contributions: (1) detecting and visualizing significant events in software evolution using call graphs, (2) extracting metrics that are suitable for software comprehension, and (3) detecting and estimating the number of defects in a given code entity (e.g., class). One of our major findings is the usefulness of the Network Portrait Divergence metric, borrowed from the information theory domain, to aid the understanding of software evolution. To validate our approach, we examined 29 different open-source Java projects from GitHub and then demonstrated the proposed approach using 9 use cases with defect data from the the PROMISE dataset. We also trained and evaluated defect prediction models for both classification and regression tasks. Our proposed technique has an 18% reduction in the mean square error and a 48% increase in squared correlation coefficient over the state-of-the-art approaches in the defect prediction domain.

Keywords Static code analysis · Call graph · Program comprehension · Software evolution · Metric analysis · Defect prediction

Communicated by: Foutse Khomh, Gemma Catolino and Pasquale Salza

This article belongs to the Topical Collection: *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*

✉ Vijay Walunj
vbwgh6@umsystem.edu

Extended author information available on the last page of the article.

1 Introduction

Software comprehension is an imperative and indispensable prerequisite for software development activities such as maintenance, testing, and quality management (Krüger 2019; Xia et al. 2017). 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 evolution is responsible for 50%-90% of the total development cost (Ghezzi et al. 2002; Fernández-Sáez et al. 2018).

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. CISQ (2018) analysis has found that in 2018, low-quality software costed 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 also 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 the costs of re-producing, discovering, fixing, and verifying defects and their locations.

Allamanis et al. (2018) pointed out that detecting defects is a core challenge in software engineering because discovering defect patterns or defining appropriated measures to detect them is not a trivial due to the complexity of software features and the diversity of software structure. Because software defects are expensive, prediction of defects has been the subject of research for the last several decades, leading to many tools and predictive models (Song et al. 2011; Akiyama 1971).

These approaches help classify software artifacts (e.g., classes, modules, subsystems and files) as being defect-prone or not. There are even models that predict defect density (the number of defects / SLOC) or the likelihood of a software artifact to have 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 prioritized 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 (Menzies et al. 2010). Many of these techniques, such as statistics-based models, parametric models, and machine learning-based models, are used (Alsaeedi and Khan 2019; Rahman and Devanbu 2013). Defect prediction can improve software testing and has the potential to improve the overall software quality assurance process (Hall et al. 2011). 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 (Le Novere 2015), modeling the network's topology (Wang et al. 2014), and modeling software systems using complex graphs (Akoglu et al. 2015;

Bhattacharya et al. 2012). 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 paper, we aim at employing deep learning techniques on metrics and measurements extracted from software call graphs (Grove et al. 1997) 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 (Gharibi et al. 2018b). The call graph contains its logical workflows, execution pathways composed of caller-callee relationships that may accurately depict the constantly evolving state of the system. Figure 1 illustrates a code snippet with its corresponding call graph.

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:

- **RQ1:** Can Network Portrait Divergence, like other graph-based metrics, detect significant events in software evolution?
- **RQ2:** How does the proposed software class-level metric Network Portrait Divergence compare to other metrics for software defect prediction?
- **RQ3:** Can Network Portrait Divergence help to improve the prediction of software defects?

Our hypothesis is that network patterns and measures can help represent the software system as a graph, which 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 (2019), Cassandra (2019), Camel (2019), ZooKeeper (2019) 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 (Qiao et al. 2020; Ferenc et al. 2018) 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

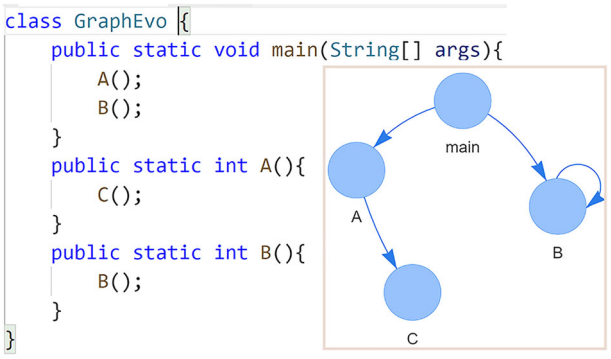


Fig. 1 An example of a Java code snippet and its corresponding call graph

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, 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 understanding software evolution and identifying code changes made 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 paper 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.

2 Related Work

The recent rise of network data across different scientific domains has led to new tools and methods (Gharibi et al. 2018b) 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.

2.1 Software Metrics

- *Code and Complexity Metrics* (Zimmermann et al. 2007; Zhang 2009): 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., very well-known Cyclomatic Complexity (CC) metric of Thomas McCabe), mostly calculated based on the source code. CC metric is the measure of independently testable paths that exist 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* (Zhou and Leung 2006; Subramanyam and Krishnan 2003): 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 of which 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* (Moser et al. 2008; Krishnan et al. 2011; Bell et al. 2011; Nagappan et al. 2010): 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 (Nagappan et al. 2010). The code delta metric computes the difference between two builds in terms of a specific metric, such as *code lines*.
- *Developer Metrics* (Matsumoto et al. 2010; Nagappan et al. 2010): As each developer contributes to the software, these metrics are recorded. Some of 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* (Zimmermann and Nagappan 2008, 2009; Premraj and Herzig 2011): 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 are set to signal a vast research area in software metrics selection. Better prediction models can be built when these subsets are combined appropriately.

2.2 Network Portrait Divergence

Portrait divergence (Bagrow and Bollt 2019) was recently developed to compare networks using their portraits. Unlike the previous ad-hoc comparison measures (see Section 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 do 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 therefore it is relatively computationally efficient. Note that this approach can treat both directed and undirected networks in the same way.

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 releases of the software system under investigation using their portraits.

Existing works considered all nodes to have the same weightage (Walunj et al. 2019). In this paper, we will be calculating Network Portrait Divergence for each class/module by iterating over classes and assigning weightage one by one. We will describe this in detail in Section 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 (Kamei and Shihab 2016). These focus on approaches that correlate with potentially defective code. Machine Learning techniques have serviced most of the defect prediction models. Those techniques derive several 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 (Huda et al. 2017; Li et al. 2018; Manjula and Florence 2019).

Chen and Ma (2015) 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, Rathore and Kumar (2016) were able to forecast the number of faults for intra- and inter-release situations. Additionally, several empirical studies (Yao et al. 2020; Hammouri et al. 2018; Rathore and Kumar 2017; Chen and Ma 2015) 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 (Vapnik et al. 1997). Many defect prediction empirical software engineering research has utilized the SVR approach (Zhang et al. 2018; Cao et al. 2018). Deep learning is currently becoming more widespread in the field of software engineering. Zhao and Huang (2018) 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 (Guo et al. 2019), while some researchers have considered multi-class classification (Arshad et al. 2019).

A novel deep neural network named Code-Description Embedding Neural Network (CODEnn) was proposed to help developers perform code search (Gu et al. 2018). 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 (White et al. 2016). The deep learning-based defect prediction model (i.e., DPNN model) introduced in Qiao et al. (2020), first obtained 11 metrics in the MIS dataset (Lyu MR and et al 1996), and 21 metric variables from the KC2 dataset from the PROMISE repository (Shirabad and Menzies 2005), 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.

2.4 Call Graphs

Call graphs have been extensively studied and used to show the software system's inner-interactions in terms of function calls. Some existing studies either focus on generating and analyzing the call graphs of one system at a time (Gharibi et al. 2018a, c; Alanazi et al. 2021) or focus on the power law of the node degree distribution. For example, the researchers in Vasa et al. (2007) built a tool to detect and visualize the static interactions between the classes of software systems written in Java; the work in Wang et al. (2009) studied the call graphs of 223 releases of the Linux kernel to identify similar graph structures; while Bhattacharya et al. (2012) 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 software systems were assessed based on design quality principles (Savić et al. 2019). Code2Vec (Alon et al. 2019) 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.

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 re-wired, resulting in exhibiting small-world and scale-free network properties (Myers 2003). When comparing such networks, the methods may be divided into two categories: Known Node-Correspondence (KNC) and Unknown Node-Correspondence (UNC) (Tantardini et al. 2019).

- KNC—Both networks have a common node-set (or at least a portion), and their pairwise connection is known. As a result, in general, 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 re-wired.
- UNC—Any two networks, regardless of their scale, density, or application sector, can be compared. Most often, these techniques employ one more statistics, 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 few software versions undergo 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. Tantardini et al. (2019) compared real-world datasets using the distances mentioned in Table 1 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

Table 1 Classification of network distances

Network distance name	Type	Description
Euclidean	KNC	It is the shortest distance between two points in N-dimensional space.
Manhattan (Hamming 1950)	KNC	It is the sum of the lengths of the line segment projections from the points onto the coordinate axes.
Canberra (Lance and Williams 1966)	KNC	It is a numerical measure of the distance between two points in a vector space.
Jaccard (Jaccard 1901)	KNC	It measures dissimilarity between sample sets.
DeltaCon (Koutra et al. 2016)	KNC	It is a distance calculated using the fast belief-propagation method for determining node affinities.
Clustering Coefficient	UNC	It's a metric for how closely nodes in a graph tend to cluster together.
Diameter	UNC	It is the maximum distance between the pair of vertices.
DGCD-129 (Sarajlić et al. 2016)	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 (Kuchaiev et al. 2010; Kuchaiev and Pržulj 2011)	UNC	It's 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 (Bagrow and Bollt 2019)	UNC	It is an information-theoretic measure for comparing networks by constructing graph-invariant distributions from the information contained in portraits.

because they provide information on the amount and significance of changes in graph structures. Also, Hartle et al. (2020) systematically compared graph distances using package netrd (McCabe et al. 2020) and found that the Network Portrait Divergence is well suited for real-world network comparisons.

Most of the current approaches in this field examine the structural properties of a single software system at a time, including model dependency (Rahman et al. 2019; Concas et al. 2007), class collaboration graphs (Rahman et al. 2019; Valverde and Solé 2003), and inheritance graphs (Savić et al. 2017). Other related works have discussed motifs (Stone et al. 2019; Russo 2018). 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 (Gao et al. 2010) 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 also limited to capturing the network topology only. 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 (Milo et al. 2002). 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 (Ikehara and Clauset 2017). 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 that can facilitate this study's reproduction using other software systems as test cases.

3 Proposed Work

Our study explores the use of Network Portrait Divergence for identifying significant events in software evolution, comparing it to other software metrics, and applying 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, and it is illustrated in Fig. 2. 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

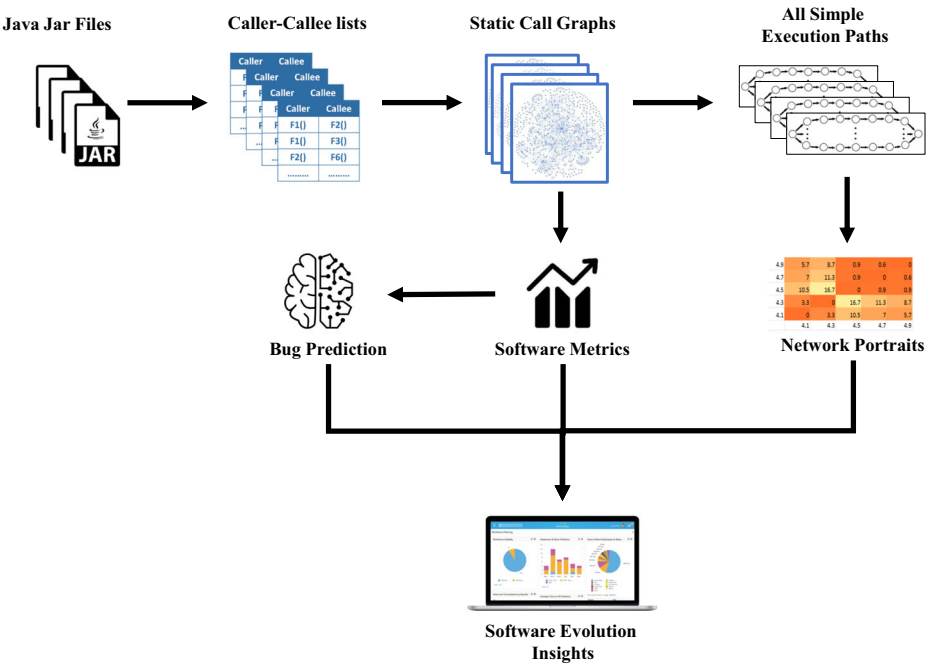


Fig. 2 Overview of our approach

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 next subsection.

3.1 Study Subjects

The analysis is focused on 384 releases of 29 open-source Java software systems, including software libraries and applications, namely JUnit (2019), Cassandra (2019), Camel (2019), Zookeeper (2019), and more. A list of the study subject applications and their number of versions used in our study are listed in Fig. 3. These software systems have been chosen on the basis of 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 BigQuery (2018). Figure 3 also lists the applications code-base size, which can be small, medium, and large based on KSLOC. Small applications are those with less than 100 KSLOC while 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 (Shirabad and Menzies 2005). Projects contain more than 4546+ classes and 1,251,619+ lines of code. The projects' descriptive statistics are shown in Table 2. #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 3 lists the metric names and their description (Spinellis 2018). These metrics are used as dependent variables and are used to analyze in the metric selection section.

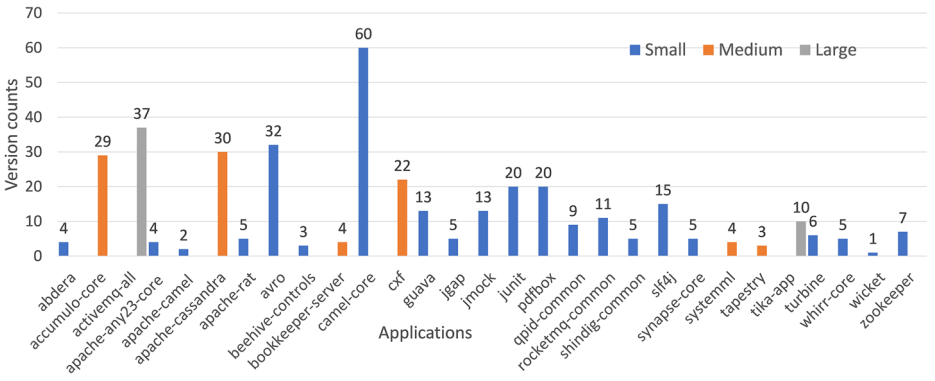


Fig. 3 The study subjects: applications and their versions

Table 2 Details of Java projects for metric analysis (promise dataset)

No	Project	Versions	# Instances/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

Table 3 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 same class
	MAX_CC	Maximum values of methods in the same class
Others (3)	LOC	Lines of code
	Network Portrait Divergence	Measures the software complexity changes
	BUG	Non-buggy or buggy

3.2 Constructing and Visualizing Call Graphs

A call graph is defined as a directed graph, $\overrightarrow{G} = (V, E)$, where V is the set of vertices, i.e., nodes of the graph, and E is the set of directed edges of the graph. In a software call graph, the nodes represent functions, and the edges represent function 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 functions, 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 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 they 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 (Alanazi et al. 2021). 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* (Gansner and North 2000), which in return can be rendered using different tools, such as *Graphviz* (Ellson et al. 2001), to actual graphs in *jpg*, *svg*, or *pdf* formats. Figure 4 shows part of the call graph generated for our tool.

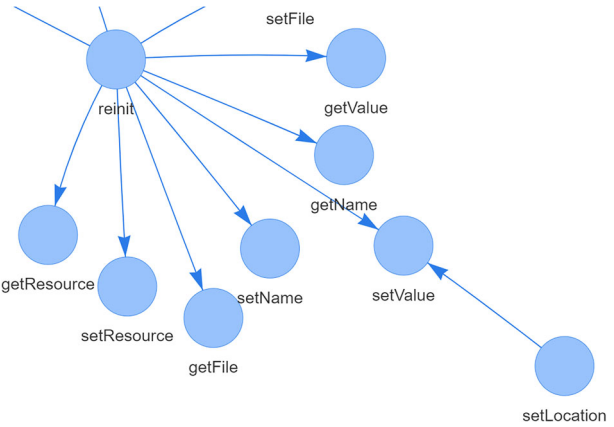


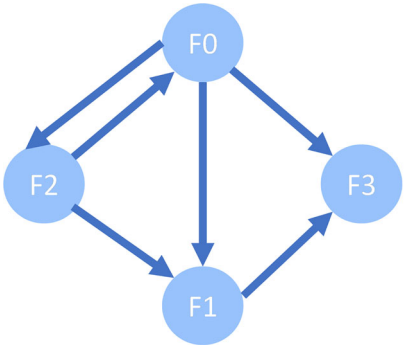
Fig. 4 Snippet of the call graph generated by GraphEvo

3.3 Characterizing Software Evolution

Network Portrait Divergence (Bagrow and Bollt 2019) 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 do 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 therefore it is relatively computationally efficient. Note that this approach can treat both directed and undirected graphs in the same way.

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 uyse the output of the previous step (i.e., step 1: extracting the call graph and its execution paths, an example is shown in Fig. 5), 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 always 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 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.



(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

Fig. 5 Example of call graph with execution paths

Algorithm 1 Constructing network portraits.

```
procedure NETWORK_PORTRAITS(paths)
counter ← count(paths)
while counter ≠ 0 do
    path ← pathList(counter)
    pathLength ← length(path)
    startNode ← start(path)
    lastNode ← last(path)
    nodeEntry.StartNode ← startNode
    nodeEntry.EndNode ← lastNode
    nodeEntry.distance ← pathLength
    portraitFreq.Add(nodeEntry)
    counter ← counter − 1
end
networkPortrait ← PathsByLength(portraitFreq)
return networkPortrait
```

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 (1).

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

where $M = \frac{1}{2}(P + Q)$ is the mixture distribution of P and Q. Here, KL is defined in (2) and P and Q are defined in (3).

$$KL(P(k,l)||Q(k,l)) = \sum_{l=0}^{max(d,d')} \sum_{k=0}^{N} P(k,l) log \frac{P(k,l)}{Q(k,l)} \tag{2}$$

Where the log is base 2.

$$p(k,l) = p(k|l)P(l) = \frac{1}{N} B_{l,k} \frac{1}{(\sum_c n_c^2)} \sum_{k'=0}^{N} k' B_{l,k'} \tag{3}$$

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 (4).

$$p(distance\ l) = \frac{\#\ paths\ of\ length\ l}{\#paths} = \frac{1}{(\sum_c n_c^2)} \sum_{k=0}^{N} k B_{l,k} \tag{4}$$

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 paper, we used the Network Portrait Divergence to measure the software evolution over time using its network portraits extracted from each release's call graph.

4 Results and Discussion

In this section, we answer the research questions and discuss 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.

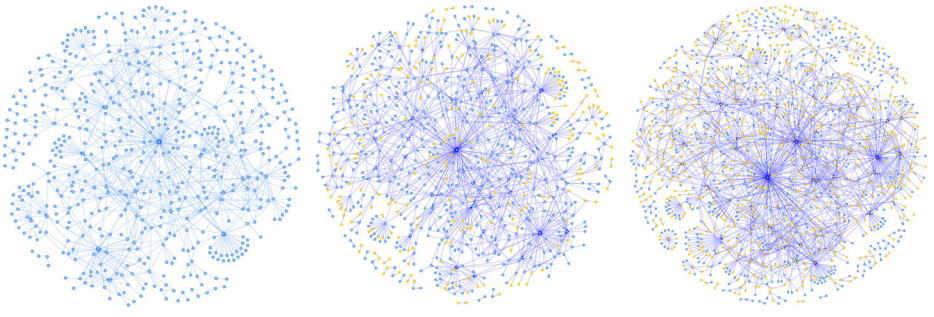
4.1 Experiment 1: Software Metrics Comparison

Several features (software metrics) are important for building an effective predictive model. The features' relevance can be evaluated individually through 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 3. Here, we utilize the ANOVA (Analysis of variance) test that selects the features with the most substantial relationship to the output variable. ANOVA operates using one or more categorical independent features and one continuous dependent feature. It provides a statistical test of whether the mean of several groups are equal or not (Fig. 6).

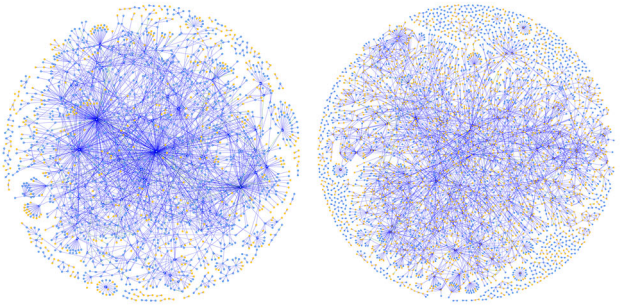
Software predictive models mainly use supervised learning techniques, which require large amounts of data to train a reliable model. However, if there is no or insufficient past data for a particular software application, then a training set can be modeled using external projects with known defect information. This type of predictive models is referred to as 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).

Figure 7 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. By the use of a combination of training set strategy and type of problems, we arrived at four scenarios and analyzed the metrics performances.

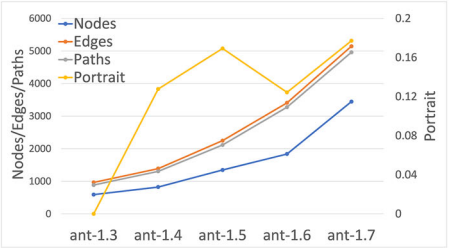
- **Classification problems on CPDP:** Building a classification model based on the training data, i.e., labeled data of external projects, and predicting the defect labels of unlabeled modules within the target project.
- **Regression problems on CPDP:** Building a regression model based on the training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



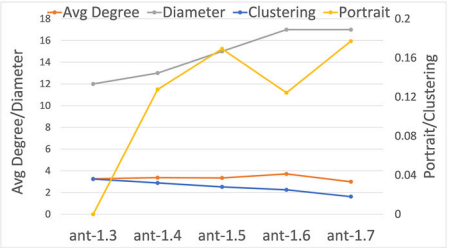
(a) Ant 1.3 (b) Ant 1.4 (c) Ant 1.5



(d) Ant 1.6 (e) Ant 1.7



(f) Count of Nodes, Edges, Paths and Network Portrait Divergence



(g) Avg Degree, Diameter, Clustering Coefficient and Network Portrait Divergence

Fig. 6 Color-coded visualization of ant call graph evolution

– **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.

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
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
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
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
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
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
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
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
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
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
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

Fig. 7 Example illustrating a snippet of the dataset featuring metrics from the Ant system, version 1.3

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–specifically the effect of network portrait divergence on improving the performance of these models. Particularly, we train two models for two tasks: defect classification and defect 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. For both tasks, we first collected and processed the dataset 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 (Shirabad and Menzies 2005; Ferenc et al. 2018). 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 number of data entries reached 4,796 (creating a table of size 4796×22). We have stored and organized these data in a csv file for simplicity.
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 of 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., different organization of dense layers) in order 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 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. 8:

– Input Layer: dimension (layer size): 22 (equivalent to the number of features).

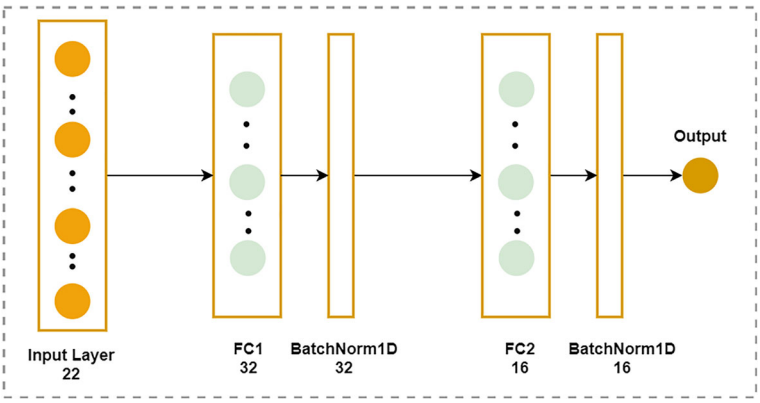


Fig. 8 Neural network structure overview (layers are not scaled to reflect their size)

- 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}.

RQ1 ***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, class changes are more important for engineers when working on a module with multiple classes.

Software Level The metrics results for our case studies, including the number of the 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 4. Note that the table includes the results for only five releases of six case studies due to space constraints. The complete list of results is available on our website https://vijaybw.github.io/graphevodef/. Our tool can visualize all of these values in addition to the graph comparison and Network Portrait Divergence in a user-friendly web-based interface. Figure 6 depicts the color-coded visualization of the call graph generated from 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. 6 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 of the systems exhibit linear growth in these metrics, namely 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 KLOSC. We believe that this change happened due to code re-factoring 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.

Table 4 Metric values for the selected software systems based on network portraits

Software	Release	Nodes	Edges	Paths	Avg-Degree	Clustering-Coef	Diameter	Modularity
Ant	1.3	591	965	8392	0.036	3.26	12	0.716
	1.4	824	1389	15835	0.032	3.37	13	0.703
	1.5	1347	2248	33616	0.028	3.34	15	0.731
	1.6	1838	3408	61638	0.025	3.71	17	0.692
	1.7	3445	5145	21381	0.018	2.99	17	0.837
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

Network Portrait Divergence The metrics we discussed above could exploit important characteristics of the graph and its evolution. However, when changes are made at the code level without any structural changes, 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 Network Portrait Divergence metric to compare two graphs despite their nodes' number and order. We observed that Network Portrait Divergence can quantify the change in software evolution and represent it in the form of network portraits, which subsequently provide insights on the execution path changes.

We found that Network Portrait Divergence can measure changes between software releases efficiently. For example, Table 5 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

Table 5 Network portrait divergence values for ant

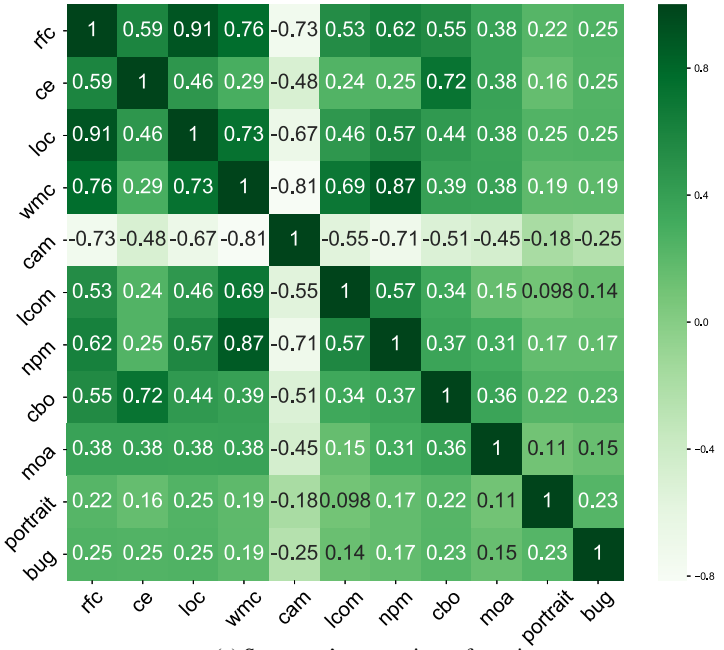
Version	1.3	1.4	1.5	1.6	1.7
Portrait	NA	0.1277	0.1692	0.1243	0.1771

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.

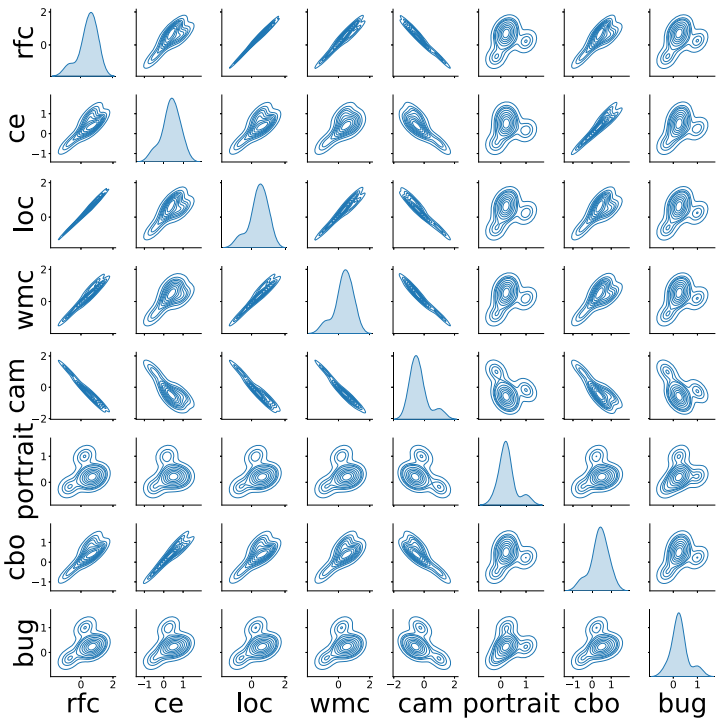
The metrics indicate a range of structural changes over the releases of a single software system, but the structure was remarkably similar amongst the software systems examined. As seen in Table 4, one can observe the overall similarity across the most examined software systems and their evolution. In 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 apparent 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. 9(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 (Scott 2015; Silverman 2018). We can see the relationships between these metrics as shown in Fig.9(b). Also, we infer that the *Network Portrait Divergence* is positively correlated to *RFC*, *CE*, *LOC*, and *CBO*. The *Network Portrait Divergence* also has a negative correlation 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 6 outlines the analysis of four classes from the *Ant* software and findings are discussed in this section. These classes range in size from small to large. Class *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.



(a) Spearman's comparison of metrics



(b) KDE Plots of metrics

Fig. 9 Pairwise comparison of top 10 metrics

Table 6 Class level comparison with Network Portrait Divergence (Promise dataset)

Version	Ant-1.3	Ant-1.4	Ant-1.5	Ant-1.6	Ant-1.7
Class: org.apache.tools.ant.taskdefs.AntStructure					
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					
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					
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					
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

Class *Delete*'s code-base and connections have been steadily growing 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, which is 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 of the metrics reflect the correct changes, and *Network Portrait Divergence* reflects them as well.

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* were significantly reduced in version 1.7, while the number of *edges* were 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. Now, all of the metrics, including *Network Portrait Divergence* were constantly changing. The *portrait number* is the same in versions 1.5 and 1.7; this could be because *execution paths* in version 1.5 were increased by 120%, while *execution paths* in version 1.7 were reduced by 90%, and *edges* were increased by 39%.

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 major 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 feature to code changes and understanding rationale behind refactorings.

RQ2: ***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, 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 is calculated for all investigated projects for both the tasks classification and regression separately. Figure 10 illustrates the used features while Fig. 11 depicts the analysis results of the

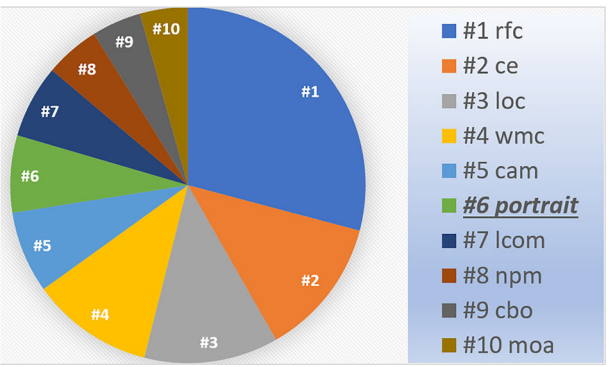


Fig. 10 Top 10 metrics

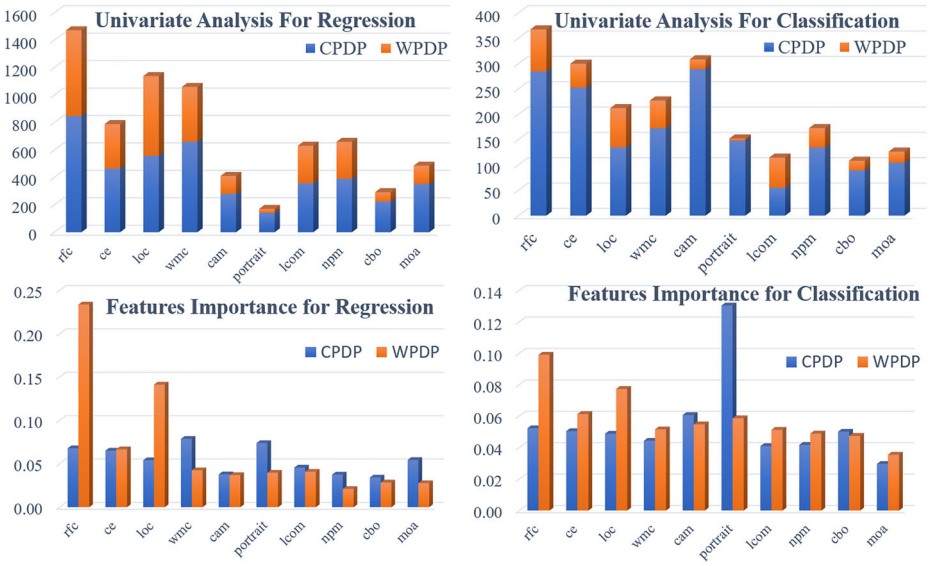


Fig. 11 Univariate and features importance analysis

selected features. According to cumulative performance ranking Fig. 10, 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. 10, 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 analysis and features importance analysis for all investigated projects for the tasks classification and regression separately. Figure 11 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.

Rathore and Kumar (2019) provides details on which software metrics were used in software defect prediction models. Each of the constructed models 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.

RQ3: ***Can Network Portrait Divergence help to improve the prediction of software defects?***

As discussed in the answers to RQ1 and RQ2, we were able to determine the usefulness of Network Portrait Divergence for the prediction of software defects. For the classification and regression tasks, we built deep learning models as described 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 (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 accurate evaluation for our prediction model.

$$F1 = \frac{2 \times (precision \times recall)}{(precision + recall)} \tag{5}$$

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 (6), where n is the number of total data samples.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6}$$

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 putting in a regression model, the association between the real y and the predicted $\hat{y}$ values. In other words, the percentage of the response variable variation that is explained by a linear model, as seen in (7).

$$R^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{7}$$

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 7, 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.

To measure defect regression model's overall performance, we used the MSE and R^2 measures on the predicted results. 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, all of 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 (Qiao et al. 2020; Yao et al. 2020; Alsolai and

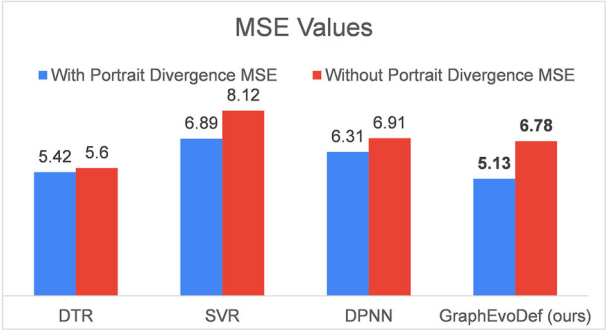
Table 7 Classification confusion matrix

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

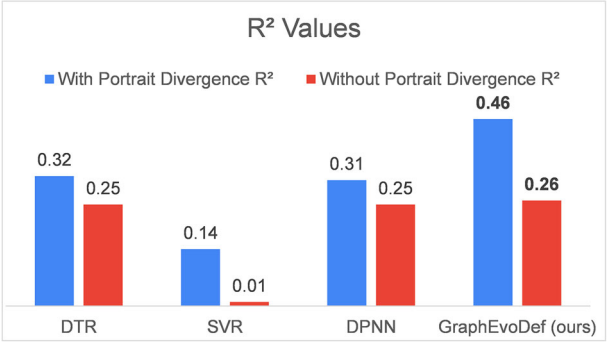
Roper 2020; Hammouri et al. 2018; Rathore and Kumar 2017). We tested all the aforementioned models on the dataset described above. All experiments were performed using NVIDIA Titan RTX GPU with 64G RAM. We implemented our models using PyTorch and monitored the experiments using an internal model management tool, called ModelKB (Gharibi et al. 2019, 2021).

The evaluation results are shown in Fig. 12. The proposed approach achieves better results compared to DTR, SVR, and DPNN in terms of MSE when Network Portrait Divergence metric is considered. 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. 12, the Network Portrait Divergence leads to better result in both MSE and R^2 values.

Network Portrait Divergence has contributed to the model being a better fit, proving its usefulness for Software defect prediction. Typically low MSE and high R^2 are 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.



(a) MSE Values



(b) R^2 *Values*

Fig. 12 Defect prediction evaluation results

5 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 (Wohlin et al. 2012).

We briefly describe the threats to validity in this section. We begin by discussing construct validity followed by the internal and external validity threats.

5.1 Construct Validity

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

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 itself using call graphs. Such code changes might lead to introducing 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 in our study and can be addressed in a future work by incorporating dynamic call graphs that require running the actual application with particular test cases.

5.2 Internal Validity

Threats to internal validity concern the causal relationship resulting from 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 paper were not gathered from a single location; and, hence, their metrics and defect data were not consistent. We did not validate these collected datasets for correctness (e.g., Was the number of defects associated with a software release actually correct?). However, after extensive manual efforts, we were able to complete these datasets and ensure they all included the features needed to train the deep learning models. Moreover, we developed the GraphEvoDef to automate this process in the future. We aim to polish and then publish the dataset used in this work in a future paper.

5.3 External Validity

As part of this study, we identify two potentially similar external threats. One threat is that the study subjects do not cover all areas of software development in their entirety. The number of product versions and defects discovered will almost certainly vary by domain and language. Nevertheless, our work aimed to show that applying network comparison advancements to Java-based systems is feasible and actually 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 was sufficient to prove the overall usefulness and meet the study targets, as explained in Section 4.

6 Conclusions

We discussed in this paper 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 and found that the Network Portrait Divergence metric is useful for identifying significant events in software evolution and its application to defect prediction. Following that, we built two defect prediction models. In comparison to existing techniques, our models achieved an 18% reduction in the mean square error and a 48% increase in the squared correlation coefficient. The findings of the review 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/graphevodef/.

Our future work will extend in several dimensions: we will focus on building a complete dataset with a larger 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 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.

Acknowledgements We would like to thank Duy Ho for his help in some implementation parts in the early versions of GraphEvo. We also thank the anonymous reviewers for their time and effort reviewing this work. The first author thanks Mattia Tantardini for sharing the network comparisons study code-base. The co-author, Yugyung Lee, would like to acknowledge the partial support of the NSF, USA Grant No. 1747751, 1935076, and 1951971.

Declarations

Conflict of Interest The authors whose names are listed immediately below certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers? bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

References

Akiyama F. (1971) An example of software system debugging. In: IFIP congress (1), vol 71, pp 353–359

Akoglu L, Tong H, Koutra D (2015) Graph based anomaly detection and description: a survey. Data Min Knowl Discov 29(3):626–688

Alanazi R, Gharibi G, Lee Y (2021) Facilitating program comprehension with call graph multilevel hierarchical abstractions. J Syst Softw 176:110945. https://doi.org/10.1016/j.jss.2021.110945. https://www.sciencedirect.com/science/article/pii/S016412122100042X

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv (CSUR) 51(4):1–37

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. Proc ACM Program Lang 3(POPL):1–29

Alsaeedi A, Khan MZ (2019) Software defect prediction using supervised machine learning and ensemble techniques: a comparative study. J Softw Eng Appl 12(5):85–100

Alsolai H, Roper M (2020) A systematic literature review of machine learning techniques for software maintainability prediction. Inf Softw Technol 119:106214. https://doi.org/10.1016/j.infsof.2019.106214. https://www.sciencedirect.com/science/article/pii/S0950584919302228

Arshad A, Riaz S, Jiao L (2019) Semi-supervised deep fuzzy c-mean clustering for imbalanced multi-class classification. IEEE Access 7:28100–28112

Bagrow JP, Bollt EM (2019) An information-theoretic, all-scales approach to comparing networks. Appl Netw Sci 4(1):1–15

Bell RM, Ostrand TJ, Weyuker E. J. (2011) Does measuring code change improve fault prediction? In: Proceedings of the 7th international conference on predictive models in software engineering, association for computing machinery Promise '11. New York. https://doi.org/10.1145/2020390.2020392

Bhattacharya P, Iliofotou M, Neamtiu I, Faloutsos M. (2012) Graph-based analysis and prediction for software evolution. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, pp 419–429. http://dl.acm.org/citation.cfm?id=2337223.2337273

BigQuery G (2018) https://cloud.google.com/bigquery/

Camel A (2019) https://camel.apache.org/

Cao Y, Ding Z, Xue F, Rong X (2018) An improved twin support vector machine based on multi-objective cuckoo search for software defect prediction. Int J Bio-Inspired Comput 11(4):282–291

Cassandra A (2019) https://junit.org/junit4/index.html

Chen M, Ma Y. (2015) An empirical study on predicting defect numbers. In: SEKE, pp 397–402

CISQ HK (2018) The cost of poor quality software in the us: a 2018 report. 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

Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Trans Softw Eng 33(10):687–708

Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. (2001) Graphviz—open source graph drawing tools. In: International symposium on graph drawing. Springer, pp 483–484

Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T. (2018) 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. Association for Computing Machinery, New York, pp 12–21. https://doi.org/10.1145/3273934.3273936

Fernández-Sáez AM, Chaudron MR, Genero M (2018) An industrial case study on the use of uml in software maintenance and its perceived benefits and hurdles. Empir Softw Eng 1–65

Gansner ER, North SC (2000) An open graph visualization system and its applications to software engineering. Softw: Pract Exp 30(11):1203–1233

Gao X, Xiao B, Tao D, Li X (2010) A survey of graph edit distance. Pattern Anal Appl 13(1):113–129

Gharibi G, Alanazi R, Lee Y. (2018a) Automatic hierarchical clustering of static call graphs for program comprehension. In: 2018 IEEE international conference on big data (big data). IEEE, pp 4016–4025

Gharibi G, Tripathi R, Lee Y. (2018b) 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. Association for Computing Machinery, New York, pp 880–883. https://doi.org/10.1145/3238147.3240484

Gharibi G, Tripathi R, Lee Y. (2018c) Code2graph: automatic generation of static call graphs for python source code. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 880–883

Gharibi G, Walunj V, Rella S, Lee Y. (2019) 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, pp 28–34

Gharibi G, Walunj V, Nekadi R, Marri R, Lee Y (2021) Automated end-to-end management of the modeling lifecycle in deep learning. Empir Softw Eng 26(2):1–33

Ghezzi C, Jazayeri M, Mandrioli D (2002) Fundamentals of software engineering. Prentice Hall PTR

Grove D, DeFouw G, Dean J, Chambers C (1997) Call graph construction in object-oriented languages. ACM SIGPLAN Not 32(10):108–124
Gu X, Zhang H, Kim S. (2018) Deep code search. In: 2018 IEEE/ACM 40th international conference on software engineering (ICSE), pp 933–944
Guo L, Lei Y, Xing S, Yan T, Li N (2019) Deep convolutional transfer learning network: a new method for intelligent fault diagnosis of machines with unlabeled data. IEEE Trans Ind Electron 66(9):7316–7325
Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 38(6):1276–1304
Hamming RW (1950) Error detecting and error correcting codes. Bell Syst Tech J 29(2):147–160
Hammouri A, Hammad M, Alnabhan M, Alsarayrah F (2018) Software bug prediction using machine learning approach. Int J Adv Comput Sci Appl 9(2):78–83
Hartle H, Klein B, McCabe S, Daniels A, St-Onge G, Murphy C, Hébert-Dufresne L (2020) Network comparison and the within-ensemble graph distance. Proc R Soc A 476(2243):20190744
Huda S, Alyahya S, Ali MM, Ahmad S, Abawajy J, Al-Dossari H, Yearwood J (2017) A framework for software defect prediction and metric selection. IEEE Access 6:2844–2858
Ikehara K, Clauset A (2017) Characterizing the structural diversity of complex networks across domains. arXiv:171011304
Jaccard P (1901) Etude de la distribution florale dans une portion des alpes et du jura. Bull Soc Vaudoise Sci Nat 37:547–579. https://doi.org/10.5169/seals-266450
JUnit (2019) https://junit.org/junit4/index.html
Kamei Y, Shihab E. (2016) Defect prediction: accomplishments and future challenges. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 5, pp 33–45
Koutra D, Shah N, Vogelstein JT, Gallagher B, Faloutsos C (2016) Deltacon: principled massive-graph similarity function with attribution. ACM Trans Knowl Discov Data (TKDD) 10(3):1–43
Krishnan S, Strasburg C, Lutz RR, Goševa-Popstojanova K. (2011) 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. Association for Computing Machinery, New York. https://doi.org/10.1145/2020390.2020397
Krüger J. (2019) 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, pp 1244–1246
Kuchaiev O, Pržulj N (2011) Integrative network alignment reveals large regions of global network similarity in yeast and human. Bioinformatics 27(10):1390–1396
Kuchaiev O, Milenković T, Memišević V, Hayes W, Pržulj N (2010) Topological network alignment uncovers biological function and phylogeny. J R Soc Interface 7(50):1341–1354
Lance GN, Williams WT (1966) Computer programs for hierarchical polythetic classification ("similarity analyses"). Comput J 9(1):60–64
Le Novere N (2015) Quantitative and logic modelling of molecular and gene networks. Nat Rev Genet 16(3):146
Li Z, Jing XY, Zhu X (2018) Progress on approaches to software defect prediction. IET Softw 12(3):161–175
Lyu MR et al (1996) Handbook of software reliability engineering, vol 222. IEEE Computer Society Press, California
Manjula C, Florence L (2019) Deep neural network based hybrid approach for software defect prediction using software metrics. Clust Comput 22(4):9847–9863
Matsumoto S, Kamei Y, Monden A, Matsumoto K, Nakamura M (2010) An analysis of developer metrics for fault prediction. In: Proceedings of the 6th international conference on predictive models in software engineering, PROMISE '10. Association for Computing Machinery, New York. https://doi.org/10.1145/1868328.1868356
McCabe S, Torres L, LaRock T, Haque SA, Yang CH, Hartle H, Klein B (2020) netrd: a library for network reconstruction and graph distances. arXiv:201016019
Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17(4):375–407
Milo R, Shen-Orr S, Itzkovitz S, Kashtan N, Chklovskii D, Alon U (2002) Network motifs: simple building blocks of complex networks. Science 298(5594):824–827
Moser R, Pedrycz W, Succi G. (2008) 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, pp 181–190
Myers CR (2003) Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys Rev E 68(4):046116

Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B. (2010) Change bursts as defect predictors. In: 2010 IEEE 21st international symposium on software reliability engineering, pp 309–318
Premraj R, Herzig K. (2011) Network versus code metrics to predict defects: a replication study. In: 2011 International symposium on empirical software engineering and measurement, pp 215–224
Qiao L, Li X, Umer Q, Guo P (2020) Deep learning based software defect prediction. Neurocomputing 385:100–110
Rahman F, Devanbu P. (2013) How, and why, process metrics are better. In: 2013 35th International conference on software engineering (ICSE). IEEE, pp 432–441
Rahman M, Palani D, Rigby P. C. (2019) Natural software revisited. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 37–48
Rathore SS, Kumar S (2016) A decision tree regression based approach for the number of software faults prediction. ACM SIGSOFT Softw Eng Notes 41(1):1–6
Rathore SS, Kumar S (2017) An empirical study of some software fault prediction techniques for the number of faults prediction. Soft Comput 21(24):7417–7434
Rathore SS, Kumar S (2019) A study on software fault prediction techniques. Artif Intell Rev 51(2):255–327
Russo B. (2018) Profiling call changes via motif mining. In: Proceedings of the 15th international conference on mining software repositories, MSR '18. Association for Computing Machinery, New York, pp 203–214. https://doi.org/10.1145/3196398.3196426
Sarajlić A, Malod-Dognin N, Yaveroğlu ÖN, Pržulj N (2016) Graphlet-based characterization of directed networks. Sci Rep 6(1):1–14
Savić M, Ivanović M, Radovanović M (2017) Analysis of high structural class coupling in object-oriented software systems. Computing 99(11):1055–1079
Savić M, Ivanović M, Jain L. C. (2019) Analysis of software networks. In: Complex networks in software, knowledge, and social systems. Springer, pp 59–141
Scott DW (2015) Multivariate density estimation: theory, practice, and visualization. Wiley, New York
Shirabad JS, Menzies TJ (2005) The promise repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada 24
Silverman BW (2018) Density estimation for statistics and data analysis. Routledge
Song Q, Jia Z, Shepperd M, Ying S, Liu J (2011) A general software defect-proneness prediction framework. IEEE Trans Softw Eng 37(3):356–370
Spinellis DJM (2018) http://gromit.iiar.pwr.wroc.pl/pinf/ckjm/metric.html
Stone L, Simberloff D, Artzy-Randrup Y (2019) Network motifs and their origins. PLoS Comput Biol 15(4):e1006749
Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Trans Softw Eng 29(4):297–310
Tantardini M, Ieva F, Tajoli L, Piccardi C (2019) Comparing methods for comparing networks. Sci Rep 9(1):1–19
Valverde S, Solé RV (2003) Hierarchical small worlds in software architecture. arXiv:cond-mat/0307278
Vapnik V, Golowich SE, Smola A et al (1997) Support vector method for function approximation, regression estimation, and signal processing. Advances in neural information processing systems, pp 281–287
Vasa R, Schneider JG, Nierstrasz O. (2007) The inevitable stability of software change. In: IEEE international conference on software maintenance, 2007. ICSM 2007. IEEE, pp 4–13
Walunj V, Gharibi G, Ho DH, Lee Y. (2019) Graphevo: characterizing and understanding software evolution using call graphs. In: 2019 IEEE international conference on big data (big data), pp 4799–4807
Wang L, Wang Z, Yang C, Zhang L, Ye Q. (2009) Linux kernels as complex networks: a novel method to study evolution. In: IEEE international conference on software maintenance, 2009. ICSM 2009. IEEE, pp 41–50
Wang Y, Wen S, Xiang Y, Zhou W (2014) Modeling the propagation of worms in networks: a survey. IEEE Commun Surv Tutor 16(2):942–960
White M, Tufano M, Vendome C, Poshyvanyk D. (2016) Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016. Association for Computing Machinery, New York, pp 87–98. https://doi.org/10.1145/2970276.2970326
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media
Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2017) Measuring program comprehension: a large-scale field study with professionals. IEEE Trans Softw Eng 44(10):951–976

Yao Y, Liu Y, Huang S, Chen H, Liu J, Yang F. (2020) Cross-project dynamic defect prediction model for crowdsourced test. In: 2020 IEEE 20th international conference on software quality, reliability and security (QRS), pp 223–230. https://doi.org/10.1109/QRS51102.2020.00040
Zhou Y., Leung H. (2006) Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans Softw Eng 32(10):771–789
Zhang H. (2009) An investigation of the relationships between lines of code and defects. In: 2009 IEEE international conference on software maintenance, pp 274–283
Zhang W, Du Y, Yoshida T, Wang Q, Li X (2018) Samen-svr: using sample entropy and support vector regression for bug number prediction. IET Softw 12(3):183–189
Zhao G, Huang J. (2018) 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. Association for Computing Machinery, New York, pp 141–151. https://doi.org/10.1145/3236024.3236068
Zimmermann T, Nagappan N. (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on software engineering, ICSE '08. Association for Computing Machinery, New York, pp 531–540. https://doi.org/10.1145/1368088.1368161
Zimmermann T, Nagappan N (2009) Predicting defects with program dependencies. In: ESEM '09. IEEE Computer Society, pp 435–438. https://doi.org/10.1109/ESEM.2009.5316024
Zimmermann T, Premraj R, Zeller A. (2007) Predicting defects for eclipse. In: Third international workshop on predictor models in software engineering (PROMISE'07: ICSE workshops 2007), pp 9–9
Zookeeper A (2019) https://zookeeper.apache.org/

Vijay Walunj is a Ph.D. student and Tech Lead at Teladoc Health. He is researching at the intersection of Software Engineering and AI. He is applying AI to various software engineering tasks to increase effectiveness and efficiency and developing tools/frameworks for better AI Software Development. He holds a master's degree in Computer Science from the University of Missouri Kansas City and a Bachelor's degree from the University Of Mumbai. Vijay holds extensive experience in architecting and implementing large-scale applications.

Dr. Gharib Gharibi is an Applied Research Scientist at TripleBlind leading the efforts of advancing the methods and tools of privacy-preserving AI and analytics. Before joining TripleBlind, he taught at the University of Missouri-Kansas City for over 4 four years while pursuing his PhD degree. He's published several papers in well-known journals and conferences and have several patents in the domain of secure computation.

Rakan Alanazi received the Ph.D. Degree in Computer Science from University of Missouri - Kansas City. He is currently an Assistant Professor of Faculty of Computing and Information Technology at Northern Border University in Saudi Arabia. He has published several conference, and journal papers. His research interests include Software Analytics, Software Engineering, Machine Learning, Software Visualization, and Program Comprehension.

Yugyung Lee is the Professor of Computer Science Electrical Engineering at University of Missouri - Kansas City. She is a co-Director of Center for Big Learning (CBL) at UMKC for National Science Foundation. Her research interests include AI, Data Science, Deep Learning, Large Scale Software Systems, and Biomedical Applications. Dr. Lee has led numerous projects funded by NSF, NIH, the Missouri Life Sciences Research Board, the Mid America Heart Institute, and Children's Mercy Hospital, etc. Dr. Lee has published over 180 refereed research papers.

Affiliations

Vijay Walunj[1] · Gharib Gharibi[1] · Rakan Alanazi[1,2] · Yugyung Lee[1]

Gharib Gharibi
ggk89@umsystem.edu

Rakan Alanazi
rakan.nalenezi@nbu.edu.sa; rna9r3@umsystem.edu

Yugyung Lee
LeeYu@umkc.edu

[1] School of Computing and Engineering, University of Missouri-Kansas City, Kansas City, MO, USA

[2] Faculty of Computing and Information Technology, Northern Border University, Rafha, Saudi Arabia