Using machine learning to detect architectural integrity violations associated with bugs

by

Alla Zakurdaeva

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Technology Innovation Management

Carleton University
Ottawa, Ontario

Abstract

Recent years have seen a surge of research into the impact architectural relations among files have on software maintainability and file bug-proneness. In particular, a set of rules for determining recurring design flaws associated with bugs has been proposed. In the present thesis we have investigated if machine learning can be used to advance the research on software architecture analysis and, specifically, on pinpointing architectural issues which may be the root causes of elevated bug- and change-proneness. In the case study of the Tiki open source project, we have been able to replicate three of the six known types of such architectural integrity violations and discover one new type, the Reverse Unstable Interface pattern. We have also demonstrated that one has to consider a mixture of local and global relationships in architectural bug detection and, contrary to common practice, should not disregard occasional co-changing of the two files as noise.

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Professor Michael Weiss, for his patience and tireless guidance throughout my research and study. The completion of this thesis would not have been possible without his continuous support and nurturing.

I would like to extend my sincere thanks to Professor Steven Muegge, who encouraged me and provided valuable comments at different stages of this work.

Table of contents

List of tables

List of figures

List of appendices

Glossary of terms

Architectural flaw	A software problem resulting from inappropriate design choices in early stages of software development (e.g., leading to incompatibility of the existing program architecture and its future view/current requirements), architecture decay, or a failure to abide by software design principles (a more broad term compared to an "architectural integrity violation")
Architectural integrity violation	An architectural flaw caused by a failure to abide by software design principles
Architectural space (software architecture)	A conglomerate of software files and architectural relationships (e.g., structural or logical dependencies) between them
Architecture decay	Degradation of software architecture caused by poorly thought-through design decisions
Area under curve (AUC)	ML model performance measure which expresses the probability that a classifier will be able to distinguish between positive and negative classes; the area under the curve created by plotting the recall against the false positive rate (FPR = FP/(FP + TN)) at various discrimination thresholds, which depicts relative trade-offs between true positives (benefits) and false positives (costs)
Classification accuracy (CA)	ML model performance measure which expresses the proportion of correct identifications among all identifications made, i.e. the number of correct predictions divided by the total number of cases examined: CA = (TP + TN)/(TP + TN + FP + FN)
Confusion matrix	A square matrix visualizing the performance of a machine learning classifier (typically, in a supervised setting) in terms of true positives, false positives, true negatives, and false negatives
Dependees *(in this research)*	The files which the studied file depends on
Dependers *(in this research)*	The files which depend on the studied file
Design Rule Hierarchy (DRH)	Clustering algorithm, which organizes software files into a hierarchical structure, such that the most influential files are attributed to the top layer of the hierarchy, files in the subsequent layers depend on files in the upper layers only, and files in the same layer of the hierarchy are decoupled into mutually independent nodules
Design Rule Space (DRSpace)	Architectural space, formed by one or more leading files and, if applicable, modules directly or indirectly dependent on them via structural and/or evolutionary

	relations, framed according to the DRH algorithm (see Section 2.2.1 for more details)
Design Structure Matrix (DSM)	A square matrix, which visualizes architectural relations between the files marking its rows and columns in the same order
Embedding-based machine learning model *(in this research)*	A model built on node embeddings as features learned by node2vec in an automated manner
Encapsulation (object-oriented programming)	The process of compartmentalizing the data with the methods that operate on that data into a single unit whose implementation is separated from the rest of the system through an interface, so that an external access to the said unit is restricted and a potential change to it is confined within (information hiding)
Evolutionary/logical dependency/coupling	Historical connection of the two software elements (classes, files, or packages); may be expressed as the number of co-changes, i.e. the number of times the two elements have been changed together in the project's version control system
F1 score	ML model performance measure which expresses the effectiveness of prediction based on precision and recall when they both are considered of the same importance, i.e. the harmonic mean of precision and recall: $F1 = 2 \times (Precision \times Recall)/(Precision + Recall) = 2TP/(2TP + FP + FN)$
False negatives (FN)	The number of positive instances which were misclassified by machine learning algorithm as negative (e.g., actual value: "bug-prone"; predicted/assigned value: "benign")
False positives (FN)	The number of negative instances which were misclassified by machine learning algorithm as positive (e.g., actual value: "benign"; predicted/assigned value: "bug-prone")
Heuristic-based machine learning model *(in this research)*	A model built on features manually engineered on the basis of formalization of known hotspot patterns (i.e., heuristics)
Hotspot pattern	A rule for pinpointing architectural integrity violations, associated with software bugs and incurring high maintenance costs
Inheritance (object-oriented programming)	The mechanism of basing a class on another class, such that a derived class (a "child") "inherits" properties and methods of a base class (a "parent"); provides the ability to build class hierarchies
Interface (object-oriented programming)	A collection of unimplemented method definitions and constants that serves as a class blueprint, so that objects can interact with a class implementing an interface

	without knowing its exact inner workings (see also "encapsulation")
Machine learning (ML) model	A mathematical representation of a specific process or phenomenon, inferred by the mathematical and statistical learning algorithm from the provided ("fed") data
Machine learning feature	An abstraction for the property that allows comparison between different observations (e.g., colour, size, image pixels, number of edges, presence or absence of some qualities)
Network representation learning (NRL)	A dimensionality reduction paradigm which aims to embed a graph into a latent vector space (usually, by representing its vertices as low-dimensional vectors; see "node embedding"), while preserving required properties of the original network
Node embedding	A continuous real-valued vector, representing a node in a network as learnt by an NRL algorithm, which may incorporate network topology structure, homophily, and vertex content, as well as (in semi-supervised NRL) vertex labels
Precision	ML model performance measure which expresses the proportion of positive identifications that were actually correct/relevant, i.e. the number of correct positive predictions divided by the total predicted positives: Precision = TP/(TP + FP)
Recall	ML model performance measure which expresses the proportion of actual positives that were identified correctly, i.e. the number of correct positive predictions divided by the total actual positives: Recall = TP/(TP+FN)
Refactoring	Improving the internal structure of an existing body of code without changing its external behavior
Software bug	An error, flaw or fault in a computer program or system that causes it to produce incorrect results or behave in an unintended way
Software modularity	Partitioning of a software system into independent, self-contained, interchangeable modules, each implementing a specific aspect of functionality and accessible to other components via a standardized interface
Software package	A collection of individual files that provides certain functionality or resources as part of a larger system; may contain source code files, executables, dynamically linked libraries, additional metadata
Structural dependency	Also known as "syntactic dependency," the directed relationship between software elements (classes, files, or packages) manifested in the source code such that a depender is not operational without a dependee (e.g.,

	would not compile); is exemplified by inheritance, implementation of an interface, or a call to another element's function
Technical debt	Implied cost of accumulated deficiencies in the code, technical documentation and development practices, deliberately incurred due to software project's time constrains
True negatives (TN)	The number of instances which were correctly classified by machine learning algorithm as negative (e.g., actual value: "benign"; predicted/assigned value: "benign")
True positives (TP)	The number of instances which were correctly classified by machine learning algorithm as positive (e.g., actual value: "bug-prone"; predicted/assigned value: "bug-prone")
Undersampling	A group of techniques designed to balance the class distribution in a dataset by removing samples from the majority class
White box machine learning models	ML models whose rationale behind the produced predictions can be output, examined, and interpreted

1. Introduction

Software architecture which represents the structure of a software system, i.e. its elements and relations among them, serves as a blueprint for the developing project, so its design decisions are costly to change once implemented. Unfortunately, these types of decisions are difficult to get right from the outset and often no single optimal solution for any given set of architecture design problems exists. Moreover, due to the continuous evolution of software the rationale behind particular architectural choices can easily be consigned to oblivion, making the system hard to understand and modify. In that case, and especially under time constraints, feature additions and bug fixes intended to increase the stability of the software may make the code less maintainable and introduce unforeseen vulnerabilities, resulting in gradual architecture decay. And while the appearance of bugs in the programs seems inevitable [22], architectural flaws are seen as one of the main sources of technical debt [9], because they propagate bug- and change-proneness among the elements (i.e. classes, files, or packages).

Although the impact of architectural relations on software bug-proneness, as well as the impact of patches to architecture, are far from being fully understood, early bug detection, or even prediction, seems vital to perform preventive maintenance and to avoid corrective maintenance, which is usually more costly and performed with greater time pressures [22]. By adopting architecture-driven methods, software developers and analysts can pinpoint potential vulnerabilities that code-based techniques alone may miss. Moreover, since the Pareto principle, as applied to software engineering, postulates that 80 percent of defects are concentrated in only 20 percent of the code, identification of

architectural flaws could confine further fine-grained (and more expensive) analyses, allowing for more efficient targeting of refactoring effort [18, 19].

In this thesis, we focus on a view on software architecture that describes the relations among the source files. In that regard, mining version control systems serves as a convenient and widely accepted means of retrieving information on changes associated with bug fixes, and, thus, pertinent to software component decay [4, 15, 37, 38]. One of such approaches was drawn on Baldwin and Clark's design rule theory[1] [2]. Having analyzed historical and structural data from many projects' repositories, a group of scholars proposed and empirically validated a suite of hotspot patterns, which are the rules for pinpointing recurring architectural problems that are caused by a failure to abide by software design principles, occur in complex software systems and incur high maintenance costs [18, 27, 28, 30]. Our research examines whether machine learning (ML) can be used for the semi-automatic discovery of such rules. If it can, it would give us a possibility to establish patterns that were not known before, determine combinations of variables that might be relevant but have not been explored before, and reveal overall regularities between various architectural concepts, not necessarily related to bug-proneness.

1.1 Objective

The objective of this thesis is to replicate and to advance the hotspot pattern approach to detecting architectural integrity violations in software through the use of machine learning.

---

[1] It is worth mentioning that the design rule theory itself did not deal with mining version control systems, but provided general guidelines for software modularization.

Treating known hotspot patterns, described in [18, 27, 28, 30], as the heuristics, we seek to answer the following research questions:

RQ1. Can machine learning identify the same hotspots as derived theoretically and from experience?

RQ2. Can machine learning reveal new patterns that have not been previously known?

Our initial method, utilized to address RQ1 and RQ2, did not take into account any potential locality of structural and evolutionary relations (as done through clustering in the hotspot pattern approach itself). At the same time, it has not been reliably established in the literature whether one should only focus on local dependencies, or what mixture of local and global impact of dependencies the heuristics should consider. To discover this, we have expanded our research by employing graph embeddings as a way of capturing the local/global relationships. Thereby, an additional research question has been formulated:

RQ3. Should just local structural and evolutionary dependencies, or longer-range, indirect dependencies as well, be included in the analysis of architectural flaws?

Furthermore, the definition of logical dependency adopted in the hotspot pattern approach was based on the threshold of co-change frequency, meaning that the two files were called evolutionary coupled only if they had been changed together (co-changed) in the commit history more than a certain number of times [27, 28, 30, 42]. We, in our turn, omitted co-change threshold in RQ1 and RQ2, taking into consideration all logical dependencies, regardless of their strength. Although co-change threshold can be found in many studies [5, 16, 36], no clear evidence has been provided as to whether occasional

co-changing of the two files is indeed noise and, as such, should be excluded from the analysis. In line with this, the following research question has been added in the transition to node embeddings:

RQ4. Should occasional co-changing be regarded as noise (and thus ignored) when determining the evolutionary coupling of two files?

1.2 Deliverables

Four deliverables are produced as a result of this study:

1) The algorithm for extracting structural and evolutionary dependency metrics (Appendix B).
2) The description of the process of interpreting the rules for determining architectural integrity violations learnt by white box heuristic-based machine learning models (Section 5.2.2), where heuristic-based ML models are those built on features manually engineered from the extracted architectural dependency metrics as per the formal definitions of the known hotspot patterns described in literature.
3) Results of a comparative analysis (structural and statistical) of the learnt rules to the heuristics (Sections 5.2.1 and 5.2.3).
4) Results of a comparative analysis of the machine learning models based on node embeddings in terms of the search parameters determining a network exploration strategy (Sections 6.2.5 and 6.2.6) and in terms of the co-change threshold determining a logical dependency graph composition[2] (Section 6.2.7).

[2] I.e. [determining] what logical dependencies are included.

1.3 Relevance

This research is relevant to at least three stakeholder groups. The first group consists of system analysts, software architects and developers. Architecture-level technical debt negatively affects long-term software evolution and maintenance and may eventually jeopardize the success of a software project [23]. Monitoring the appearance of architectural flaws is important for timely remediation, preventing the propagation of bugs to other system components [9, 11, 18, 27, 30]. Actionable knowledge about pinpointing violations of software design principles would help developers target their refactoring effort more efficiently [19] and would free up time for functional upgrades and feature additions [9]. The methods developed in this thesis could eventually also lead to the creation of tools for the automatic detection of architectural issues when code changes are about to be committed, similar to the lower-level compile-time checks for syntax errors.

The second group of stakeholders includes the management of IT companies and IT entrepreneurs. As has been mentioned earlier, an architectural perspective on bug detection would add one more layer to software quality tests, facilitating the identification of bugs before they propagate to other files or classes. In terms of product and project management, it would lead to software of higher quality, while reducing the cost of corrective maintenance [18, 22].

Finally, the content of this thesis is relevant to computer science researchers. First off, the interest in using the version control data to uncover actionable information on software systems and projects is increasing and has even led to the institution of the International Conference on Mining Software Repositories [22]. Secondly, our study is

directly pertinent to the hotspot pattern approach, the authors of which welcomed subsequent exploration into architectural anti patterns and offered a prospect of extending their tool to detect new types of hotspots should they be discovered [27]. Moreover, the concept of using white box machine learning models for pattern derivation may be found suitable in other scientific disciplines or areas of research.

1.4 Contribution

The work presented within this thesis makes the following contributions to the software engineering literature and practice:

1) It confirms, by replicating existing heuristics, that machine learning is capable of discovering patterns of recurring architectural flaws that are caused by violations of software design principles and are frequently associated with bugs;
2) It supplements the existing approach to determining architectural integrity violations in software with a new pattern, Reverse Unstable Interface (or, as we can call it to better convey its meaning, Unstable Coupled Client);
3) It explores the impact that the locality of structural and logical dependencies has on the quality of architectural flaw detection and demonstrates that one has to consider node neighborhoods beyond the immediate relationships, such that these neighborhoods convey both structural and homophilic properties of the studied node;
4) It ascertains that, contrary to common practice, occasional co-changes between the files should not be omitted from the software architecture analysis.

1.5 Overview of method

The research design for this thesis is an exploratory case study that is analyzed through qualitative and quantitative methods following a grounded theory approach [20]. The details on the open source project selected for this work are presented in Chapter 4.

Figure 1 provides a high-level overview of the research process. First, architectural and bug metrics are extracted from software version control system. Feature engineering is then performed on the obtained data, either manually, in accordance with the formal definitions of the known hotspot patterns, or in an automated manner (via node embeddings).

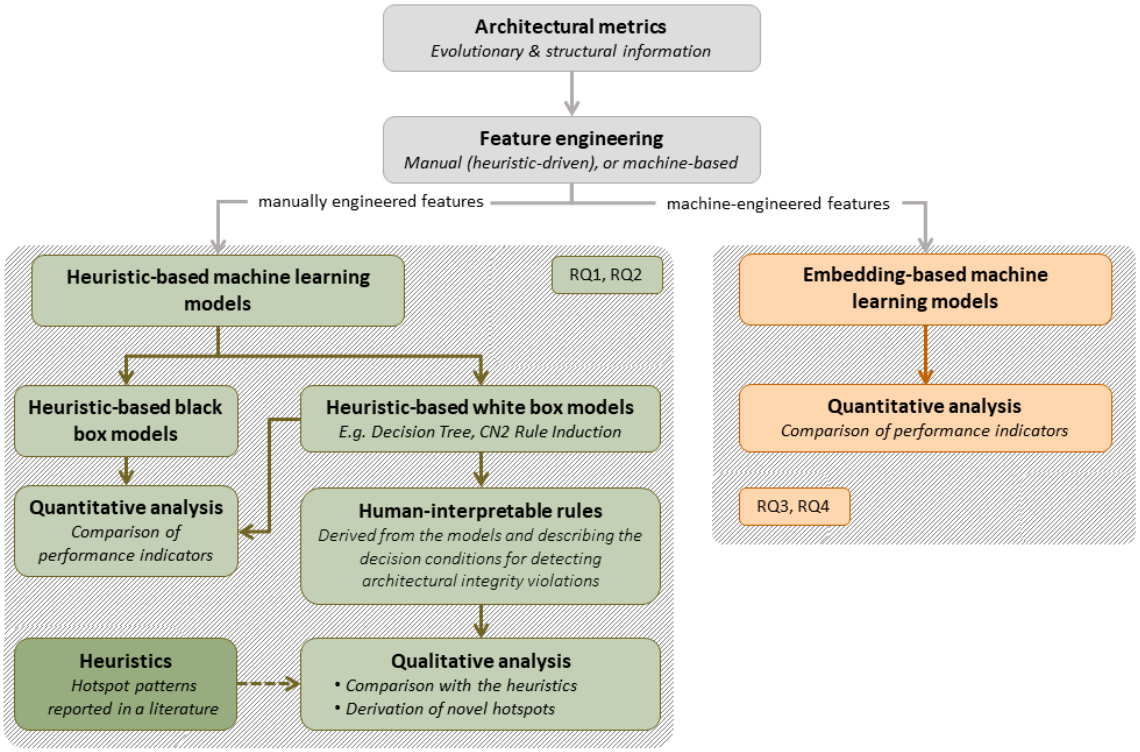


Figure 1 Research method overview

In order to answer RQ1 and RQ2, heuristic-based machine learning models are scrutinized. Particular attention is given to white box models, whose rationale behind the classification can be output, examined, and interpreted. The derived rule sets representing

the decision conditions are structurally and statistically compared to the heuristics, and novel rules, which have not been documented before, may be discovered. The empirical validity of the identified patterns is verified, inter alia, by comparing the performance of white box and black box heuristic-based models.

To accommodate the expansion of the research to RQ3 and RQ4, several machine learning models based on node embeddings are also developed. They are evaluated with regard to the impact that the parameters, determining the locality of architectural dependency network exploration, and the co-change threshold have on the quality of architectural bug detection.

1.6 Organization of the document

This thesis is organized as seven chapters, each structured into sections, accompanied by six appendices. Chapter 2 summarizes the related work on architecture-based software quality analysis, on hotspot pattern approach to pinpointing design flaws associated with bugs, as well as on feature engineering, machine learning model training and evaluation. Chapter 3 outlines the research method. Chapter 4 acquaints with the open source project studied and presents the particulars of structural and evolutionary data extraction from its version control system.

The four research questions are divided into two sets, each being addressed in a separate chapter encompassing specific method used to tackle the questions, analysis results and discussion which includes answers to research questions (Sections 5.2.5 and 6.2.9), in addition to a description of possible limitations. Thus, Chapter 5 examines RQ1 and RQ2 detailing the hotspot pattern derivation through machine learning, while Chapter 6 pertains to the remaining two research questions, and is devoted to the exploration into

node embeddings as machine-learned features preserving the network structure of a Design Rule Space, a notion which hotspot patterns are defined upon (more details on that in the upcoming literature review).

Finally, Chapter 7 discusses the implications of our research, while Chapter 8 concludes the study, followed by the list of references.

Six appendices are provided at the end of the document. Appendix A contains the visual representation of architectural relations between the files in our dataset. The next four appendices detail the process described in Chapters 4, 5, and 6, by offering step-by-step algorithm and source code for the scripts. Appendices B and C accompany Chapter 4, elaborating on architectural and bug data extraction. Appendix D supplements Chapter 5, expounding data post-processing for heuristic-based models. Appendix E extends Chapter 6, explaining the particulars of data post-processing for embedding-based models. Finally, Appendix F presents the table with performance indicators for embedding-based models, which was moved to the appendix section due to the size.

2. Literature review

This chapter reviews the salient findings from the scholarly and practitioner literature in three streams: (1) software architecture and quality analysis, (2) hotspot pattern approach to detecting architectural integrity violations associated with bugs, and (3) machine learning techniques. Table 1 summarizes the highlights of the three streams with citations to the key sources. Each literature stream is being particularized in its own section below, followed by the summary of the key lessons that are most relevant to this research.

Table 1 Highlights of the scholarly and practitioner literature

Stream	Key highlights of the stream	Citations to key sources
Software architecture and quality analysis	• Discusses different architectural dependency types, their impact on system bug-proneness, and interplay • Provides architecture-based metrics for bug localization • Suggests visualization approaches	[1, 7, 8, 14, 16, 19]
Hotspot pattern approach to detecting architectural integrity violations associated with bugs	• Research project led by Yuanfang Cai at Drexel University, Philadelphia • Proposes the set of rules for determining recurring architectural flaws in a software (hotspot patterns) • Combines information on structural dependencies between the files and file co-change history (what files were changed together in a version control system and how many times) and presents it in a matrix • Stems from Baldwin and Clark's design rule theory	[2, 18, 27, 28, 30]
Machine learning techniques	• Examines different machine learning algorithms, their advantages and drawbacks • Describes approaches and performance metrics for machine learning model evaluation • Provides information on various feature engineering methods	[12, 21, 33, 46]

2.1 Software architecture and quality analysis

Architectural relations are most commonly expressed in terms of structural and evolutionary dependencies between files. Structural dependencies are exemplified by inheritance, implementation of an interface, or a call to another file's function.

Evolutionary dependencies, which are sometimes referred to as logical dependencies or logical/evolutionary coupling, denote the joint evolution of the two files during the course of a project. It may be defined in different ways, but most often is expressed as the number of co-changes between these two files, i.e. the number of times they have been changed together in the commit history.

The importance of historical information for software quality analysis and error-prone file identification cannot be overestimated. Abundant empirical evidence has been provided that evolutionary coupling measures not only correlate with software defects [38], but correlate better than more elaborate complexity metrics [15]. This correlation is somewhat expected as source files with high activity rate seem more likely to generate bugs than those with low activity [38]. It is noteworthy that logical coupling measures appear to be particularly good at spotting more severe bugs (major/high-priority bugs), and that prioritizing newer commits over older ones may not yield anticipated results: it was shown that time-based algorithms giving more weight to recent changes correlate with defects less than simpler ones based on unweighted/untimed metrics [15]. The possible explanation is that the assumption that the code gets better over time is not always valid. If refactoring never happens, the code keeps on accumulating responsibilities and coupling [40].

The history of project evolution may be visualized as an animated storyboard composed of a sequence of dynamic panels exhibiting co-change relations. Although analysis of a static graph may reveal symptoms of decays, the storyboard could reveal that these symptoms have existed over the years since the beginning of the project and are most probably design decisions rather than decay symptoms [7]. Inclusion of the

historical information is especially valuable for multi-language software projects, because the dependencies between artifacts written in different programming languages cannot be detected with syntax-based analyses [8].

Benkoczi et al. [6] reemphasized the importance of logical coupling for software quality analysis as a way of reflecting on system evolution. They added, however, that "the nature of the interplay between static dependencies and co-changes is not completely understood although reasonably strong correlation exists between the two" [6]. The empirical evidence for this correlation had been provided in the study of 35 large Java projects, which backed the previously expressed in scientific community assumptions that co-changes propagate through a path of static dependencies [19]. The research also showed that the role of dependencies as propagators of change diminishes as the path length increases. Moreover, the dependency structure alone was proven to be insufficient for passing judgment on software evolvability, which means that the dependency view and the co-change view complement each other and should both be taken into account when evaluating software architecture [19].

In a similar vein, Oliveira et al. [16] also established that the potential ripple effect that might occur during maintenance tasks cannot be completely explained by static dependencies. Organizing the architecture according to a decomposition based on co-change clusters, however, might not improve software stability compared to a decomposition based on static structural relations. So again, both the change characteristics and the dependency structure of a system should be considered [16]. The more recent study of 79 open-source software projects confirms that the co-changing of software classes is partly brought about by source code dependencies. However, while a

directional relationship exists between the system architecture and the co-evolution of its elements, the co-change behaviour is hardly a mirror image of the dependency structure, so one needs to draw insights from both types of dependencies [1].

Apart from reaffirming that structural and evolutionary dependencies present significantly different architectural views, Cui et al. [14] showed that their interaction with defective files is also drastically different. The syntactic structure captures the largest number of defective files but with the lowest accuracy. By contrast, the history structure covers the least number of defective files but with high accuracy. Although the authors also examined the semantic relation, or semantic similarity of the two files, derived from identifiers and comments, the results of its analysis were in-between and didn't impact the overall conclusion: using just one type of relations to conduct bug prediction/localization is insufficient. The union of different dependency types can cover most of the defective files, and their intersection can capture files with most severe problems [14].

Both structural and evolutionary dependencies are being extracted from the version control system and then used to identify software components with elevated bug- and change-proneness caused by architecture decay. The analysis can be done at the package or at the file level. The former approach involves aggregating structural dependencies and co-changes between any modules in, for example, tiki/lib/accounting and tiki/lib/blog subsystems. After the inter-package relationships have been examined, one would recursively step into the most problematic subsystem(s) and repeat the assessment. This is the general idea presented by Tornhill [40], who used co-changes alone, without structural dependencies, to pinpoint potential candidates for refactoring.

At the same time, at the file level, all files are considered at once. This is the approach we follow in our research.

Figure 2 depicts architectural relations between the source files in the library package of the Tiki project (more on the project in Chapter 4). Structural and evolutionary dependencies represent the state of the system as of March 3rd, 2019, 03:09:25 UTC, with logical dependencies having been accumulated since the beginning of the project. Full version of the figure is provided in Appendix A, but at the moment let us zoom in on the file `tikilib.php`. Already from the visualization itself we may assume that the file is in high risk group and may require more thorough attention, because many files structurally depend on it and/or are evolutionary coupled with it, so the error probability is higher. What is more, if something happens to any of its neighbors, bugs will likely propagate through architectural relations among the whole community, thus making the problem difficult to eradicate.

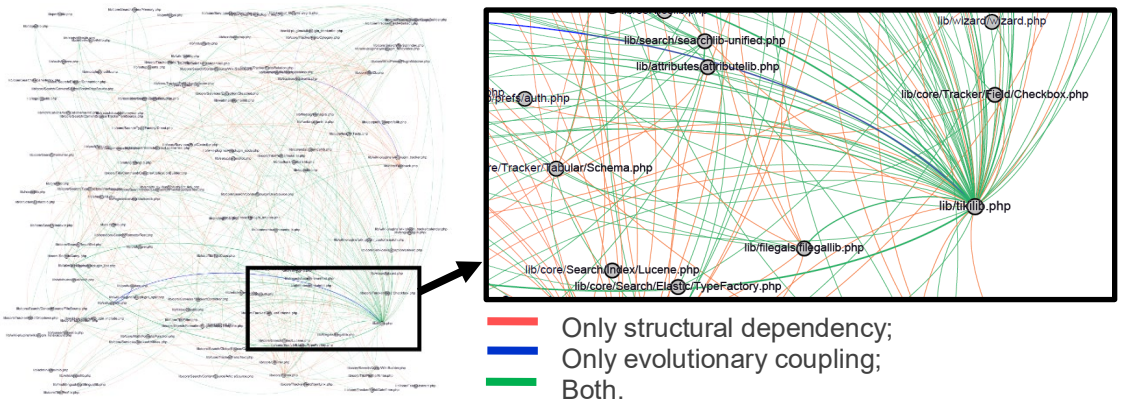


Figure 2 Visualization of architectural relations

Next we will discuss the hotspot pattern approach to identifying architectural flaws, which is based on Baldwin and Clark's design rule theory.

2.2 Hotspot pattern approach to detecting architectural integrity violations associated with bugs

According to Baldwin and Clark's theory, design rules are the key architectural decisions that decouple a software system into independent modules which can be implemented, revised, or even replaced without influencing other components [2]. Building upon this concept, there was developed an architecture model called Design Rule Space, which presents software architecture as a set of design rules and modules framed by them [42]. Having examined error-prone Design Rule Spaces over many commercial and open source projects, the researchers noticed that some types of architectural issues occur repeatedly and may be amalgamated into a suite of what they called hotspot patterns, which are the definitions and at the same time the rules for detecting these design flaws [28]. It was found that compared with average files in a project, the files involved in these patterns have significantly higher bug and change rates, and the more hotspot patterns a file is involved in, the more error- and change-prone it is [27, 28, 30]. Moreover, the authors of the hotspot pattern approach also found out that hotspots strongly correlate with high rates of security bugs [18].

An advantage of the proposed approach is that patterns not only help identify the most error- and change-prone files and the underlying architectural design flaws, but also provide guidance in analyzing and fixing these flaws [18]. This is especially important since bug-prone files are likely to be architecturally connected, propagating bugs among each other [11]. In the subsequent subsections we will discuss the notion of a Design Rule Space, the hotspot pattern definitions, and how they are currently being detected in software projects in more details.

2.2.1 Design Rule Space (DRSpace)

A Design Rule Space (DRSpace) is a graph-like representation of an architectural space formed by a particular design rule. It is composed of a set of files and structural or evolutionary relationships between them, with the following features [42]:

- Edges of a DRSpace describe types of relationships between the files (vertices), such as structural dependencies (e.g. inheritance, implementation of another file's method, or a function call) and evolutionary couplings, which are being derived from project's repository and may, for instance, be modeled as the number of co-changes, i.e. the number of times the two files have been changed together in a single commit as recorded in a project's revision history.
- It must have one or more leading files, which are usually the design rules of the space. If a DRSpace has more than one layer, then the files within the first layer are the leading files of a DRSpace. If a DRSpace has only one layer, then all the files are considered as leading ones.
- It has to be clustered using the design rule hierarchy (DRH) [10, 41] algorithm which aggregates the design rules to the top layers of the hierarchy, manifesting their dominating position and their independence from other subsystems. Subsequent layers of the hierarchy contain mutually independent modules that depend on the files in the upper layers only.

A DRSpace is visualized as a DRH-clustered design structure matrix (DSM) [10, 41]. A DSM is a square matrix whose rows and columns exhibit source files listed in the same order. A cell along the diagonal represents self-dependency, and a non-empty off-diagonal cell captures some structural and/or evolutionary relation between the file in the

row to the file in the column. The files are clustered into modules, which in a DSM are illustrated as shaded blocks along the diagonal. File names are typically provided in rows to the right of the file index.

Figure 3 depicts an exemplary layout of a DRSpace DSM with 10 source files. "x" in an off-diagonal cell *cell(rx; cy)* in row x, column y, $x \neq y$, signifies a structural dependency of the file in row x on the file in column y. For instance, *cell(r3; c1)* is labeled with "x", which indicates that *file3* structurally depends on *file1*. Number in a cell *cell(rx; cy)*, $x \neq y$, exhibits evolutionary coupling of files x and y, which hereinafter is expressed as the number of co-changes shared between these two files in a given time period. For example, *cell(r2; c6)* is marked with ",9", meaning that *file2*, although not structurally dependent on *file6*, has been committed together with it 9 times as per the version control system. Accordingly, if a cell *cell(rx; cy)*, $x \neq y$, contains both an "x" and a number, then the file x is both structurally related to file y and evolutionary coupled with it. See *cell(r4; c3)* in Figure 3 – it is annotated with "x,7", which signifies that *file4* not only structurally depends on *file3*, but has also been changed together with it in a single commit 7 times as evidenced by the project's revision history. It is noticeable that the DRH-clustered DSM presented in Figure 3 aggregates the source files into 2 layers: L1: (rc1-rc2) and L2: (rc3-rc10). Within L2, there exist 4 modules: *M1: (rc3-rc5)*, *M2: (rc6-rc7)*, *M3: (rc8)*, and *M4: (rc9-rc10)*.

		1	2	3	4	5	6	7	8	9	10
1	file1	(1)			,2	,4					,2
2	file2		(2)				,9	,5	,6	,4	
3	file3	x		(3)	,7	,1		,2			,3
4	file4	,2		x,7	(4)			,8		,7	,3
5	file5	x,4		x,1		(5)	,5		,8		
6	file6		,9			x,5	(6)	,6	,2	,9	
7	file7		x,5	,2	x,8		x,6	(7)	,1		
8	file8	x	x,6			x,8	,2	x,1	(8)	,7	,2
9	file9		x,4		,7		,9		x,7	(9)	,5
10	file10	x,2		x,3	,3		x	x	x,2	x,5	(10)

Figure 3 DRSpace exemplary layout

In the recent years, it has been demonstrated that DRSpaces serve as a powerful tool for software defect detection. Common architectural integrity violations contravening software design principles that can be detected through the DRSpace analysis have been identified, described and amalgamated into the suite of hotspot patterns.

2.2.2 Hotspot pattern definitions

2.2.1 Overview. Hotspot patterns are defined as the rules for determining recurring architectural flaws that occur in most complex software systems, are associated with bugs, and thus incur high maintenance costs [18, 27, 28, 30]. Table 2 presents the list of patterns proposed by the authors of the approach based on the issues they had repeatedly found in the projects they had studied along with the brief descriptions. In our research, we regard these known architectural hotspots as the heuristics.

Table 2 Summary of known architectural hotspots

Pattern name	Description
Unstable Interface	too frequent co-changing of highly influential (e.g., leading) files
Unhealthy Inheritance Hierarchy	mutual dependence of parent and child, or client of the hierarchy, that depends on both parent and its child
Modularity Violation Group	the minimal number of file sets that cover all file pairs that are evolutionary coupled but have no structural dependencies
Crossing	"God file" which has both high fan-in and fan-out and is changed frequently with both files which depend on it and the files that it depends on
Clique	a set of files that are closely coupled with each other through one or more structural dependency cycles
Package Cycle	cyclic structural dependency of different packages

2.2.2 Formal definitions. Let us introduce some notions for hotspots formalization [27, 28, 30]: $StructImpact_{thr}$ is the threshold of the structural impact scope of a file, $HistoryImpact_{thr}$ is the threshold of the number of files evolutionary coupled with a file, $cochange_{thr}$ is the threshold for the frequency of files changing together for them to be considered as evolutionarily coupled (the threshold is meant to exclude the files which are only occasionally changed together from the analysis), $crossing_{thr}$ is the threshold of the number of dependers and dependees of a file, where dependers are the files which depend on the file, and dependees are the files that the file in question depends on. For mathematical formalization we will also use the following notions: F – a set of all files: $F = \{f_i \mid i \in \mathrm{N}\}$, $depend(x, y)$ – x structurally depends on y, $\#cochange(x, y)$ – the number of times x was committed together with y in a given time period as per the revision history, and $SRelation(x, y)$ – structural relation from file x to file y [27, 30].

On the basis of the concepts described above we provide formal definitions for each type of hotspot patterns.

Unstable Interface. There are more than $StructImpact_{thr}$ files that structurally depend on the file under study (which represents a highly influential file) and more than $HistoryImpact_{thr}$ of these files have been changed together with it more than $cochange_{thr}$ times (denotes the high number of evolutionary couplings with its dependers). As Geipel & Schweitzer [19] note, a highly connected class in the dependency network does not necessarily indicate flawed design but distinguishes the class as very important, as the one with many responsibilities. Only if this involvement goes along with change propagation does the design deserve a critical look.

Mathematical formalization: $\exists f_1 \in F \mid \{F_{set\_S}\colon \forall f_i \in F_{set\_S} \mid SRelation(f_i, f_1)\} \wedge \{F_{set\_H}\colon \forall f_j \in F_{set\_H} \mid \#cochange(f_j, f_1) > cochange_{thr}\} \wedge (|F_{Set}| > StructImpact_{thr}) \wedge \left(\left|F_{set\_S} \cap F_{set\_H}\right| > HistoryImpact_{thr}\right)$, where $i \in \{1, 2, 3, \dots, n\}$, n is the number of files in F_{set_S}, $j \in \{1, 2, 3, \dots, m\}$, m is the number of files in F_{set_H} [27, 30].

Figure 4 presents the exemplary layout of Unstable Interface hotspot. We can see that multiple files structurally depend on *file1* and have changed together with it frequently as recorded in the project's commit history.

		1	2	3	4	5	6	7	8	9	10
1	**file1**	**(1)**	x,3	,5	,5	,12	,7	,5	,10	,9	,4
2	file2	**x,3**	(2)					,3	,3	,7	
3	file3	**x,5**		(3)	,4				,7	,4	
4	file4	**x,5**	x	x,4	(4)	,2	,4				,6
5	file5	**x,12**		x	x,2	(5)	,5			,8	
6	file6	**x,7**			x,4	x,5	(6)	,3	,6	,9	
7	file7	**x,5**	x,3				x,3	(7)	,8	x,3	,5
8	file8	**x,10**	,3	x,7	x		,6	,8	(8)	,4	,3
9	file9	**x,9**	x,7	,4		x,8	,9	x,3	x,4	(9)	,3
10	file10	**x,4**			,6		x	,5	x,3	x,3	(10)

Figure 4 Unstable Interface exemplary layout

Unhealthy Inheritance Hierarchy. This pattern is described by two cases of design rule theory and Liskov Substitution Principle violations (Liskov Substitution Principle postulates that functions which use pointers to parents must be able to use objects of derived children without knowing it):

1) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a child file f_i that f_{parent} depends on.
2) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a client f_j of the hierarchy that depends on both the parent and any of its children.

Mathematical formalization: $\exists f_{parent}, F_{child} \in F \wedge \exists f_i \in F_{child} \mid depend(f_{parent}, f_i) \vee [\exists f_j \in F \mid depend(f_j, f_{parent}) \wedge \exists f_i \in F_{child} \mid depend(f_j, f_i)]$, where $i, j \in [1, 2, 3, \ldots, n]$, $f_j \notin F_{child}$ and $f_j \neq f_{parent}$ [27, 30].

Figure 5 depicts two mock-up instances of Unhealthy Inheritance Hierarchy:

1) The parent file, *file1*, depends on its child, *file2*.
2) The client file, *file6*, depends on the parent, *file3*, and all of its children.

Please note that here all structural dependencies have been divided into two types: extend (a child inheriting from a parent) and depend (all other dependencies), which, instead of a uniform notation "x", are referenced as "ex" and "dp", respectively.

		1	2	3	4	5	6
1	**file1**	**(1)**	**dp,9**	,2		,5	
2	**file2**	**ex,9**	(2)		,8		
3	**file3**	dp,2		(3)	,4	,3	,7
4	**file4**		,8	**ex,4**	(4)	,11	,4
5	**file5**	,5		**ex,3**	,11	(5)	,5
6	**file6**			**dp,7**	**dp,4**	**dp,5**	(6)

ex - extend
dp - depend

Figure 5 Unhealthy Inheritance Hierarchy exemplary layout

Modularity Violation Group. There exists a set of files which are not structurally related to the file under scrutiny, but have been changed together with it more than $cochange_{thr}$ times. The rationale behind this pattern is that if two structurally independent modules (or files, if we consider each file as the smallest module in a project) are shown to change together frequently in the revision history, it means that they are not truly independent from each other and may share implicit assumptions: "if there is no syntactic relationship, a logical coupling might indicate possible or even necessary refactorings" [38].

Mathematical formalization: $MVG_1 \cup MVG_2 \cup MVG_3 \cup \ldots \cup MVG_n = P \mid f_{core} \in MVG_i,\ \forall f_j \in MVG_i,\ f_j \neq f_{core} \mid \neg SRelation(f_{core},\ f_j) \wedge \neg SRelation(f_j,\ f_{core}) \wedge (\#cochange(f_{core},\ f_j) > cochange_{thr})$, where $i \in \{1, 2, 3, \ldots, n\}$, n is the minimal number of modularity violation groups (MVGs) whose union covers all violated file pairs. P is the set of all violated file pairs. f_j is a file which is structurally independent from f_{core}, but frequently changes together with it [27, 30].

Figure 6 exemplifies a Modularity Violation Group wherein *file8* is the core file, since it is structurally unrelated to other elements of the DRSpace, but has been changed frequently with them.

		1	2	3	4	5	6	7	8
1	file1	(1)	,2	,3				,4	**,32**
2	file2	x,2	(2)			,7			**,16**
3	file3	x,3		(3)	,3		,2		**,15**
4	file4			x,3	(4)			,3	**,4**
5	file5		x,7		x	(5)			**,14**
6	file6			x,2		x	(6)	,6	**,12**
7	file7	x,4			,3	x	x,6	(7)	**,6**
8	**file8**	**,32**	**,16**	**,15**	**,4**	**,14**	**,12**	**,6**	**(8)**

Figure 6 Modularity Violation Group exemplary layout

Crossing. The file structurally depends on more than $crossing_{thr}$ files evolutionary coupled with it, and more than $crossing_{thr}$ files evolutionary coupled with it structurally depend on it. Crossing indicates a violation of the information hiding design principle: encapsulate what is likely to change.

Mathematical formalization: $\exists f_i, f_j, f_c \in F \mid (|SRelation(f_i,\ f_1)| > crossing_{thr} \wedge |\#cochange(f_i,\ f_c) > cochange_{thr}| > crossing_{thr}) \wedge (|SRelation(f_c,\ f_j)| > crossing_{thr} \wedge |\#cochange(f_c,\ f_j) > cochange_{thr}| > crossing_{thr})$, where $i, j \in \{2, 3, 4, \dots, n\}$, n is the number of files in a project's DSM [27, 30].

The pattern is represented in a DSM as the center of a cross (see *file8* in Figure 7 – being the center file of a Crossing, it changes frequently with both its dependers and dependees).

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	file1	(1)			,11			,5	**x,4**					,9	
2	file2		(2)			,2			**x,6**		,2	,2	,2		
3	file3			(3)				,8	**x,2**						
4	file4	,11	x		(4)		x,7		**,7**	,7			x,5	x,11	
5	file5		,2			(5)			**,1**			,14			
6	file6	x			,7		(6)	,3	**x,3**		,1	,1		x,2	
7	file7	,5		,8			,3	(7)	**x,3**			x			x,1
8	**file8**	**,4**	**,6**	**,2**	**x,7**	**x,1**	**,3**	**,3**	**(8)**	**,7**	**,2**	**x,14**	**,9**	**,4**	**x,6**
9	file9				,7				**x,7**	(9)	,5				
10	file10		x,2				,1		**x,2**	x,5	(10)		,7	,2	,9
11	file11		,2			,14	,1		**,14**			(11)	,3		
12	file12		,2	x	,5				**x,9**		,7	,3	(12)		,1
13	file13	,9	x		,11		,2		**x,4**		,2			(13)	,4
14	file14						x	,1	**,6**		,9		,1	,4	(14)

Figure 7 Crossing exemplary layout

Clique. A group of files that form a strongly connected graph-like component based on cyclic structural dependencies between them. The reasoning behind Clique is that tight coupling of files deteriorates maintainability and propagates bug-proneness across the connected components of a system, because changes to one file will often require changes to other, connected files. Related to this, circular dependencies have long

been abhorred by software engineers: “The dependency structure between packages must be a directed acyclic graph. That is, there must be no cycles in the dependency structure” [19].

Mathematical formalization: $\forall f_i \in F_{cq}, \forall f_j \in F_{cq}, i \neq j \mid [depend(f_i,\ f_j) \wedge depend(f_j,\ f_i)] \cup [\exists f_1, \dots, f_n \in F_{cq}, n \geq 1 \mid depend(f_i,\ f_1) \wedge depend(f_1,\ f_2) \wedge \dots \wedge depend(f_n,\ f_j)]$, where $i, j \notin [1, 2, 3, \dots, n]$, F_{cq} is the set of files in a clique [27, 30].

Figure 8 illustrates a mock-up of a Clique pattern. Although all files in a set are highly coupled through multiple dependency cycles (thus constituting a Clique) we have highlighted only one of them as an example: *file1* → *file9* → *file2* → *file1*.

		1	2	3	4	5	6	7	8	9	10
1	**file1**	(1)	,5	,2	x,2	,2	,10	,3	,3	**x,7**	,8
2	**file2**	**x,5**	(2)		,3	,6	,3	,6	,4	,9	
3	**file3**	x,2		(3)	,7	,2		x,7	,5		,5
4	**file4**	,2	x,3	,7	(4)	,5	,8		,2		x,11
5	**file5**	x,2	,6	x,2	x,5	(5)		,7			
6	**file6**	,10	,3		,8		(6)		x,3	,4	,7
7	**file7**	,3	x,6	,7		x,7		(7)	,4		,2
8	**file8**	x,3	,4	,5	x,2		x,3	x,4	(8)	,9	,2
9	**file9**	,7	**x,9**			x	,4		x,9	(9)	x,3
10	**file10**	x,8		x,5	,11		x,7	,2	,2	,3	(10)

Figure 8 Clique exemplary layout

Package Cycle. The file f_1 from the package P_a depends on some file from the package P_b, and the file f_2 from the package P_b depends on some file from the package P_a. Thus, the software package structure does not form a hierarchy, which reduces the understandability and maintainability of a system.

Mathematical formalization: $\exists f_1, f_i \in P_a \wedge \exists f_2, f_j \in P_b \mid depend(f_1,\ f_j) \wedge depend(f_2,\ f_i)$, where P_a, P_b are the packages of the system, $i, j \in [1, 2, 3, \dots, n]$, n is the number of files in the system [27, 30].

Figure 9 shows an instance of Package Cycle: *file1* in *Package1* depends on *file1* in *Package2* and *file2* in *Package2* depends on *file2* in *Package1*, forming a dependency cycle between two packages.

		1	2	3	4
1	**Package1.file1**	(1)	,10	**x,2**	
2	**Package1.file2**	,10	(2)		
3	**Package2.file1**	,2		(3)	,5
4	**Package2.file2**		**x**	,5	(4)

Figure 9 Package Cycle exemplary layout

2.2.3 Existing (non-ML-based) approach to detection of hotspot patterns

The authors of the hotspot pattern approach developed a process for automated detection of hotspot patterns. First, a tool called Titan [43] generates an SDSM (StructureDSM) file from the structural dependency report produced by a reserve engineering application, e.g. SciTools Understand[3]. In a similar fashion, Titan then generates an HDSM file (HistoryDSM), which records the co-change frequencies between files, from the revision history logs (e.g., SVN or Git log) and an SDSM. Finally, another tool called the Architecture Flaw Detector, developed by the authors, clusters the acquired information based on the .clsx file which represents the clustering (decomposition) of system elements and automatically detects hotspot pattern instances. To detect a Package Cycle, for example, a .clsx file should contain the package structure of the files, while in order to detect other types of architectural integrity violations, a .clsx file corresponding to the DRH clustering can be used. The Architecture Flaw Detector tool outputs a summary of all found architectural integrity violations, the files involved in each violation, and DSMs depicting the DRSpaces with these violations [27].

[3] https://www.scitools.com/.

Since the existing algorithm relies on static analysis and the rules detectable by the tools are hard-coded, some of the instances which are based off patterns not incorporated in the Architecture Flaw Detector may remain unnoticed – in other words, existing patterns may pose a limitation. Machine learning, not dependent that much on a theory (except maybe for feature engineering), could prove to be more efficient in this regard.

2.3 Machine learning techniques

Machine learning is a method of data analysis which enables a computer system to "reason" in a similar way a human would do: a system learns by distilling concepts from a body of complex information and performs specific tasks without being explicitly programmed. In a nutshell, machine learning is a collection of mathematical and statistical learning techniques (algorithms) that dissect a dataset and create generalizable abstractions (models) which facilitate pattern discovery and information mining. Inferences drawn from the previous computations (during the training stage) are used to adapt to new information (testing stage) with minimal human intervention. Algorithm selection is determined by computational complexity (i.e. training/testing time), mathematical complexity (decision boundary of the data), and explainability (black box models – hard to understand how the different features interact – and white box models – easy to interpret) [12, 25].

There are two main approaches to machine learning model evaluation: conventional validation (a dataset is split into non-overlapping training and test sets so that a model is tested on different data than it was trained on) and cross-validation (multiple dataset splits are generated repeatedly and validated individually, like in

conventional method, with final result derived as the average across different splits). Cross-validation is considered to be more rigorous and robust, because it "avoids the pitfall of information loss from a single train/test split that might not adequately capture the statistical properties of the data" [12]. The most common cross-validation technique is *k*-fold cross-validation, where *k* is a number specified by a user. The idea is that a dataset is partitioned into *k* subsets of equal size, called folds. Based on these folds, *k* models are created, such that each time, one of the *k* subsets (in turn) is used as the test set and the other *k-1* subsets constitute a training set [31].

Many metrics and tools have been developed to quantify model performance. Since our task, as will be shown in the following chapters, is to classify labelled data points (i.e. source files) into "bug-prone" and "benign" categories, we will discuss metrics pertaining to supervised classification. The most common evaluation metric in this respect is classification accuracy, which is the total proportion of correctly assigned (predicted) labels. It works well, however, only if the number of data points in each class is the same. For instance, if 98% of elements belong to class *A* and 2% – to class *B*, then a model that attributes every single element to class *A* would have a classification accuracy of 98%, which, nevertheless, is misleading and does not mean a model is suitable for prediction purposes. Therefore, using only accuracy to measure performance is insufficient. Instead, other metrics, such as area under curve (AUC), precision, recall, and F1 score, which are based on confusion matrix values (true positives, true negatives, false positives, and false negatives), should be considered in combination to provide a more comprehensive picture of how the model performs [12]. It is also important to understand the context before prioritizing particular metrics because each machine

learning model tries to solve a problem with a different objective using a different dataset [31].

Since our original method did not consider architectural relationships beyond the immediate, one-hop, neighbors, network representation learning (NRL) has been examined as a way of capturing higher-order dependencies through learning continuous vector-valued embeddings of each node in an automated manner. As opposed to recovering network topology via explicit, pre-defined, featurization of nodes (i.e., by handpicking suitable properties, e.g., node degree) this method does not require domain expertise and manual inspection of the graph structure, and therefore is more robust [17].

Zhang et al. [46] define network representation learning as a paradigm for embedding a network into a latent space that preserves network topology structure, vertex content, and other side information, such that the original vertices of the network can be represented as low-dimensional vectors. Vertices similar or close to each other in the original structure space retain their affinity in the learned vector space. Therefore, NRL takes both local and global relationships into consideration and relays network architecture in all its complexity. Vertex attribute content can also be imported to supplement edge information and render more informative node representations which would allow measuring attribute-level similarity between vertices [46].

A variety of network analytic tasks can be performed using learned embeddings: node classification [17, 21], link prediction (where a goal is to determine the existence of an edge) [21, 33], similarity search [17, 33], and clustering [17, 21]. Different NRL methods may be divided into two main categories: unsupervised (no labeled vertices are provided) and semi-supervised (labeled vertices directly benefit the learning of

informative representations). The former group is of particular interest to us since we will use NRL just to engineer features while the classification itself will be carried out via other machine learning algorithms.

From the methodology perspective, there are two approaches to NRL suitable for our problem: matrix factorization based techniques and random walk based techniques. The first class of methods represents the connections between network vertices in the form of a matrix, which is then factored to obtain the embeddings [46]. An example of a feature learning algorithm adopting a factorization strategy is GraphWave [17]. It embeds each node's network neighborhood into a low-dimensional space by leveraging spectral graph wavelet diffusion patterns: the vertex emits a unit of energy which propagates through edges over the graph, so that the shape of the heat wavelet diffusion characterizes node's neighborhood structure [17, 46]. It therefore follows that GraphWave recovers network topology based on structural similarity of different vertices, meaning that nodes with similar structural roles within their local neighborhoods will have similar embeddings [17, 46]. The algorithm captures higher-order relationships without requiring the explicit hand-labeling of features and is robust to noise in the edge structure [17].

The second class of techniques exploits random walks to learn the embeddings. Random walk is a randomized procedure that samples many different neighborhoods of a certain source node. By repeating the process for each given vertex, the overall network is transformed into a collection of node sequences which are then aggregated together. An example of an NRL algorithm following random walk methodology is node2vec [21]. It applies to any (un)directed, (un)weighted graph, smoothly interpolating between two extreme network exploration strategies: Breadth-first Sampling and Depth-first Sampling

[21, 46] (more details on node2vec search bias in Section 6.1). Being based on proximity, the algorithm integrates structural context and homophily to learn effective representations that preserve local neighborhoods of nodes: vertices from the same network community are embedded closely together, while nodes that share similar structural roles have similar embeddings. Although node2vec does not incorporate node properties or edge properties, amongst other desirable pieces of information, it is robust to perturbations in the form of noisy or missing edges. Robustness to missing edges in the network is especially important in cases where the graphs are evolving over time, or where network construction is expensive [21]. Since one of the most challenging problems in NRL is to predict the representations of newly added, "out-of-sample", vertices, Ren et al. [33] proposed a node2vec-based model for incremental updating of embeddings that focuses on computation time and performance. It is worth mentioning, though, that the authors worked on the task of financial news recommendation, which is essentially a link prediction task (as opposed to node classification).

2.4 Summary of key findings from the literature

Seven insights from across the scholarly and practitioner literature are particularly pertinent to this research:

1) Software architecture may be pictured as a graph whose nodes represent system elements (classes, files, or packages), and whose edges denote any type of relation between these elements [1, 14, 18, 19, 27, 28, 30, 42].
2) Structural and evolutionary dependencies are the two most common architectural relation types, which, however, present different architectural views and interact with software defects differently. Therefore, these two dependency types should

be considered in combination when performing software architecture analysis: their union covers most of the bug-prone files, while their intersection captures files with most severe problems [1, 14, 16, 19].

3) Change coupling (i.e. co-changing) correlates with defects more than complexity metrics, but less than the number of changes. However, for major/high priority bugs change coupling measures are always better than software metrics and, in many cases, than the number of changes [15].
4) Prioritizing more recent commits over older ones when identifying defective files is not always a good idea – software code does not necessarily improve over time, so all changes should be considered of equal importance [15, 40].
5) A literature has recently proposed a suite of six hotspot patterns for pinpointing architectural integrity violations which may be a root cause of elevated software bug-proneness [18, 27, 28, 30].
6) White box machine learning models are an effective tool for discovering patterns in data, because they produce rule sets describing the decision conditions which are explicit, interpretable and can easily be related to the formalization of known rules [12, 25].
7) Network representation learning is a paradigm for network topology recovery wherein, for each node, a real-valued vector embedding, which, inter alia, captures this node's local neighborhood, is learned in an automated manner. Thus, by converting a graph into a vector space, we can represent not only immediate, but higher-order relations between its vertices [17, 21, 46].

3. Research method

Table 3 summarizes the steps of the research method and outcomes of each step.

Table 3 Research method summary

Step	Description of activity undertaken	Outcome of the activity
Preparatory		
1	Conduction of a literature review on the detection of software design flaws, the interplay between different architectural dependency types and their impact on system bug-proneness, feature engineering, machine learning model training and evaluation	Lists of heuristics, features, suitable machine learning techniques
2	Computing of architectural and bug metrics based on records in the selected project's version control system	Algorithm for metrics extraction, e.g. raw input creation for machine learning models (Appendices B and C)
Part 1. Hotspot pattern derivation through machine learning (RQ1 & RQ2)		
3	Construction of several feature sets from the collected metrics according to the hotspot pattern formalization, implementation of several white box machine learning models, interpretation of model outputs and comparison of the derived rule sets with the heuristics	Process of interpreting the rules for determining architectural integrity violations (Section 5.2.2); results of structural and statistical comparison of the learnt rules with the heuristics (Sections 5.2.1 and 5.2.3)
Part 2. Exploration into node embeddings as machine-engineered features preserving the network structure of a DRSpace (RQ3 & RQ4)		
4	Development of several machine learning models based on node embeddings; evaluation of the impact the model parameters, determining the network exploration strategy, and co-change threshold have on the quality of detection	Results of the embedding-based model analysis in terms of the impact of the locality of the dependency network exploration (Sections 6.2.5 and 6.2.6) and in terms of the co-change threshold impact (Section 6.2.7)
Drawing insights & reporting		
5	Results summarization	Report of the analysis results, suggestions for future investigation (Sections 5.2.5, 5.3, 6.2.9, 6.3, Chapter 7)

The literature review (Step 1) has been previously elaborated in Chapter 2. Step 2 is particularized in the following chapter, along with the details on the selected open source project. The details on methods, utilized for Part 1 and Part 2 are provided in Sections 5.1 and 6.1, respectively. The answers to research questions are given in

Sections 5.2.5 (RQ1 and RQ2) and 6.2.9 (RQ3 and RQ4). Limitations of each part of this study and avenues for future research are presented in Sections 5.3 and 6.3, while Chapter 7 sums up and discusses the overall results.

4. Case study: Tiki open source web-application

In order to answer our research questions, we have conducted a case study of the Tiki open source project. Tiki is a web-application, written primarily in PHP, which encompasses the capabilities of a wiki platform, a content management system, and groupware. Unlike larger projects that are usually being researched (e.g., the Eclipse JDT or Apache Cassandra), Tiki is an example of a medium-sized project, and existing research has not tested whether the heuristics (i.e., the proposed hotspot patterns) apply to small and medium-sized systems as well. In addition to the smaller size, Tiki stands out for its duration. The project was started around 20 years ago and is still active, which means that there is a long history of commits. Moreover, the Technology Innovation Management program at Carleton University has been involved with the Tiki project since 2005, and has access to its developers for direct feedback on potential practical implications (however, this discussion remains part of the future work).

In our study, we have examined the library (lib) package of the Tiki project. The focus on the library package has been prompted by the distribution of unique files involved in bug fixes among different project packages. It has been found that the library is ranked first in both the number of the unique files involved in security bug fixes (45.83% of the total number across the whole application) and the unique files involved in bug fixes in general (32.91%). The other reason is that the common tools for extracting structural dependencies require code to be object-oriented, whereas PHP applications often also contain code that is not in an object-oriented form. The library package is written in the form of classes, while other parts of the system mainly consist of scripts or template code.

Tiki Git repository is publicly available on GitLab[4]. The commits are being diligently labeled. A label (in this particular project) is a predefined term in square brackets placed in the commit message to categorize the changes: e.g., [FIX] for "bug fix," [ENH] for "enhancement," [MRG] for "branch merge," [LANG] or [TRA] for "language/translation." It is noteworthy that labels in Tiki project have been applied systematically for more than 13 years – developers started using them in 2004, but became truly consistent in labeling in late 2006 when labels got standardized and used ubiquitously.

The revision history over the period from October 7th, 2002, to March 3rd, 2019, 03:09:25 UTC, has been extracted from the Tiki version control system, skipping unrelated commits that contained no other labels than [BRANCH], [DOC], [LANG], [MRG] (and similar, e.g., [MERGE]), [OOPS], [TEST], and [TRA]. If there were no labels in a commit, it was included. Following [1, 5, 15, 16, 42, 47], we have also filtered out commits with more than 20 files, since they tend to represent minor automatic modifications (e.g. bulk changes to files to update project information), which may introduce substantial noise in data.

The structure of the latest version of the Tiki library package (as of March 3rd, 2019, 03:09:25 UTC, which pertains to release 19) has been recovered using the dePHPend[5] open source tool designed to detect dependencies in object-oriented PHP code.

Table 4 summarizes the highlights of the data collection.

---

[4] https://gitlab.com/tikiwiki/tiki.
[5] https://github.com/mihaeu/dephpend.

Table 4 Data acquisition overview

Characteristic	Value
Evolutionary data collected for the period from	October 7th, 2002 (the outset of the project)
Evolutionary data collected for the period until/ structural data collected as of	March 3rd, 2019, 03:09:25 UTC (release 19)
# of commits in the cloned Tiki repository (as per `git rev-list --count master`, main branch)	51,074
# of commits after filtration during revision history extraction (as per `dataforprocessing.log` produced by `git_commits_extraction.sh`, see Appendix B.2)	46,046
# of structural dependencies detected by dePHPend in the library package	5,287
# of unique files (as per `merged.csv` produced by `visualization.py`, see Appendix B.2)	3,001

The obtained structural and historical information has then been pre-processed and merged together. See Appendix B for more details on architectural data acquisition and preparation, i.e. on its collection and transformation up to the moment when we are able to recreate Figure 2 (full version of which can be found in Appendix A). Please note that evolutionary coupling in the present research has been defined through co-changing of the two files *in the same commit* (similar to [6, 11, 19, 27, 30, 42]). Please refer to Appendix C for the description of bug metric (bug frequency) extraction process. Bug frequency is defined in our research as the number of times a file has been involved in bug-fixing commits (i.e. the number of times it has been included in commits labeled in Tiki Git repository with [FIX], [BUG], or [SEC], where the latter term stands for security bug fixes).

5. Part 1: Hotspot pattern derivation through machine learning

5.1 Details of the method

Based on the formal definitions of hotspot patterns presented in Section 2.2.2, the following features have been manually engineered: CCN (i.e. the co-change number) – the total number of times a file has been committed together with each other file of a system, Dependers – the number of files which depend on a file, Dependees – the number of files a file depends on, DependersCCN – the number of co-changes with a file's dependers, and DependeesCCN – the number of co-changes with a file's dependees. Along with the file name as the meta variable and the file bug-proneness (described in more detail later) as the target, these parameters constitute the complete feature set for hotspot derivation through machine learning. In addition, a reduced feature set has been constructed, consisting of just two features: total CCN and total number of Dependencies (that, in essence, is a sum of Dependers and Dependees). The latter feature set could not be used for pattern identification itself (at least consolidated features would not have allowed for structural comparison with the heuristics), but instead has been built to match the performance of the machine learning models based on more and less comprehensive feature sets and check if a higher granularity is worthwhile from a practical perspective. Table 5 provides a summary of different parameters included in each set of features.

Table 5 Feature sets constructed

Feature	Role	Complete feature set	Reduced feature set
FileName	meta	x	x
CCN	feature	x	x
Dependers	feature	x	
Dependees	feature	x	
Dependencies	feature		x
DependersCCN	feature	x	
DependeesCCN	feature	x	
BugProneness	target	x	x

In our research, bug-proneness is based on the notion of bug frequency, which is the number of bug-fixing commits a file has been included in, i.e. commits labeled with [FIX], [BUG], or [SEC] ("security bug fix"). In order to compare machine learning models' performance with regard to the granularity of the target variable, two settings of bug-proneness have been adopted: binary and categorical ones. In the binary setting, *BugProneness* = 0 if the file had never been a part of a bug-fixing commit (*BugFrequency* = 0) and *BugProneness* = 1 if had been at least once (*BugFrequency* > 0). In the categorical setting, bug frequency has been projected onto a scale of None/Low/Medium/High bug-proneness in accordance with the values exhibited in Table 6: files which had never been involved in bug-fixing commits (*BugFrequency* = 0) have been mapped to category *BugProneness* = *None*, files which had been involved in 1 to 9 bug-fixing commits (*BugFrequency* = $\overline{1,9}$) have been mapped to category *BugProneness* = *Low*, 10 to 19 bug-fixing commits (*BugFrequency* = $\overline{10,19}$) – to category *BugProneness* = *Medium*, 20 and more bug-fixing commits (*BugFrequency* ≥ 20) – to category *BugProneness* = *High*. The rationale behind choosing this mapping is that the bug frequency, as per the resulting dataset `mlinput.csv` which serves as an input to heuristic-based machine learning models (see Appendix D for more details), ranges from 0 to 535 with only 4.0% of files having *BugFrequency* ≥ 20, and 6.3% of files with *BugFrequency* = $\overline{10,19}$ (see Table 6 for the distribution of files across different categories of bug-proneness).

All numerical values have been normalized using unity-based (min-max) normalization formula in order to increase the applicability of our models to other projects and to handle class imbalance:

$g_{norm} = \frac{g - \min(g)}{\max(g) - \min(g)}$, where g denotes a feature or bug frequency, which was also normalized.

Since in all cases $\min(g) = 0$, the formula is brought down to:

$g_{norm} = \frac{g}{\max(g)}$.

The maximum values for each feature, as well as for bug frequency, are presented in Table 7.

Table 6 Bug frequency mapping to categorical bug-proneness

Bug frequency	Normalized bug frequency	Bug-proneness	Number of files	Percentage of files
0	0	None	1012	33.7%
1-9	0.0019-0.0168	Low	1679	56.0%
10-19	0.0187-0.0355	Medium	190	6.3%
20+	0.0374-1	High	120	4.0%

Table 7 Maximum feature values and maximum bug frequency value

g	max(*g*)
CCN	1291
Dependers	645
Dependees	97
Dependencies	663
DependersCCN	653
DependeesCCN	149
BugFrequency	535

The two feature sets and the two levels of granularity of the target variable described above amalgamate into four combinations: complete feature set with the binary target variable, complete feature set with the categorical target variable, reduced feature set with the binary target variable, and reduced feature set with the categorical target variable. On the basis of these combinations the following three analyses have been carried out:

1) Rule (hotspot pattern) derivation through Decision Tree classification and CN2 Rule Induction (only for combinations based on complete feature set). Decision Tree and CN2 Rule Induction models belong to a class of so-called white box machine learning models, which means that we are able to not just interpret them, but to structurally compare the rule sets, guiding their decision making, with the heuristics. The main difference between Decision Tree and CN2 Rule Induction algorithms is that CN2 Rule Induction produces explicit rules itself, while in case of Decision Tree they have to be manually derived from the model. We will discuss the process in more detail in Section 5.2.2.
2) Performance comparison of the Decision Tree classifier and the CN2 Rule Inducer with Naive Bayes, Random Forest, and AdaBoost (only for combinations based on complete feature set).
3) Performance comparison of the Decision Tree classifiers based on complete and reduced feature sets (to evaluate the impact of feature granularity on model performance; both target variables examined).

All the analyses have been conducted using the Orange[6] open source toolkit for interactive machine learning model creation and data mining. Table 8 summarizes the parameters having been used for each machine learning algorithm under study (wherever Orange provides the opportunity to fine-tune the model). To maximize the accuracy, some parameters for Random Forest and AdaBoost differ for binary and categorical versions of the target variable.

[6] https://orange.biolab.si/.

The hotspot patterns revealed by the Decision Tree and the CN2 Rule Induction classifiers have been derived by manually examining the generated tree and the rule output in the corresponding viewers. The derivation and the interpretation of the produced rules will be detailed further in Section 5.2.2. The performance comparison of different algorithms was based on the analysis of such measures, as the area under curve (AUC), classification accuracy (CA), F1 score, precision and recall, obtained through the stratified 10-fold cross validation, as well as on the inspection of corresponding confusion matrices.

Table 8 Machine learning algorithm parameters in heuristic-based models

Parameter	**Target variable setting**	
	Binary	**Categorical**
Decision Tree		
Induce binary tree	yes	
Min. number of instances in leaves	2	
Do not split subsets smaller than	5	
Limit the maximal tree depth to	100	
Stop when majority reaches	100%	
CN2 Rule Induction		
Rule ordering	Unordered	
Covering algorithm	Exclusive	
Evaluation measure	Laplace accuracy	
Beam width	5	
Minimum rule coverage	1	
Maximum rule length	5	
Statistical significance (default α)	unchosen/unchecked	
Relative significance (parent α)	unchosen/unchecked	
Random Forest		
Number of trees	10	
Number of attributes considered at each split	unchosen/unchecked	
Replicable training	yes	no
Limit depth of individual trees	unchosen/unchecked	
Do not split subsets smaller than	5	10
AdaBoost		
Base estimator	Tree	
Number of estimators	50	
Learning rate	0.90	
Fixed seed for random generator	unchosen/unchecked	
Classification algorithm	SAMME.R	SAMME
Regression loss function	Linear	

5.2 Results and discussion

5.2.1 Heuristic verification

The analysis of the Decision Tree and the CN2 Rule Induction models based on complete feature set has revealed all of the heuristics which could be identified in our research setting with the features we engineered: three of the six hotspot patterns that were proposed previously [18, 27, 28, 30] have indeed been reproduced by machine learning. Table 9 exhibits the feature correspondence to the known hotspots (in accordance with the complete feature set). Mapping of the features to hotspot patterns has been based on hotspot pattern description, mathematical formalization and exemplary DSMs (see Section 2.2.2). We postulate that the Crossing pattern can be brought down to one of the pairs {high Dependers, high DependeesCCN}, or {high Dependees, high DependersCCN}, because, as follows from its visual representation (see [27, 30], or Figure 7 with exemplary layout), this hotspot is symmetrical: if file x constitutes a Crossing, then for each file $i \neq x$ in the DSM one of the cells *cell(rx; ci)* or *cell(ri; cx)* (depending on the direction of dependency) contains structural dependency and a co-change number, and the other one indicates only co-change. Due to this symmetry, the pair {high Dependers, high DependeesCCN}, for example, which in the DSM is depicted as a filled column, implies the presence of a filled row (which is represented by {high Dependees, high DependersCCN}). The inverse relationship is based on the similar reasoning; however, the symmetry will only work if DependeesCCN or DependersCCN are high.

Due to aggregating co-changes and structural dependencies as total numbers for each individual file, we have been unable to capture the remaining three heuristics. As

follows from the definitions presented in Section 2.2.2, in order to detect Unhealthy Inheritance Hierarchy, Clique, or Package Cycle, we need to know which files structurally depend on which exactly files. Moreover, with regard to the Package Cycle hotspot, we have only examined source files within a single package of the Tiki project - the library, so, being a package-level pattern, Package Cycle at all fell outside the scope of our analysis.

Table 9 Correspondence of the complete feature set to the heuristics

Pattern name	**Machine learning model features**
Unstable Interface	high Dependers, high DependersCCN
Unhealthy Inheritance Hierarchy	N/A (structural dependencies have been aggregated for each file separately)
Modularity Violation Group	low Dependers, low Dependees, high CCN
Crossing	{high Dependers + high DependeesCCN}, or {high Dependees + high DependersCCN}, or the combination of these
Clique	N/A (structural dependencies have been aggregated for each file separately)
Package Cycle	N/A (single package has been examined)

5.2.2 Interpretation of the rule sets generated by white box heuristic-based models

Table 10 exemplifies some of the rules attributed to each of the three heuristics we have been able to cover, as generated by the CN2 Rule Induction model with the binary target variable. Rule quality is scored as the relative frequency of the class corrected by the Laplacian error estimate [13] (see Table 8). As is shown in the Figure 10, the CN2 Rule Viewer in Orange does not aggregate the conditions for each feature (see line 60 in the Figure 10, for example: all three conditions listed are related to just one feature – CCN), so we have manually combined the conditions together. We have also combined the rules produced by the algorithm: for instance, rule presented in the Table

10, line 4 (Modularity Violation Group), has been compounded of the following two rules:

1) $0.0333 \leq CCN$, $Dependers \leq 0.0016$, $Dependees \leq 0.0103$, $DependeesCCN \leq 0.0134$ (Quality 0.982; 107 bug-prone cases).

2) $0.0279 \leq CCN$, $Dependers \leq 0.0016$, $Dependees \leq 0.0103$, $DependeesCCN \leq 0.0134$ (Quality 0.913; 62 bug-prone cases).

The second column of the Table 10 presents the human-readable interpretation of the corresponding rules with non-normalized feature values specified. To denormalize the features, we used values from Table 7, reversing the unity-based normalization formula as $g = g_{norm} \cdot \max(g)$. Then we "translated" the produced rules into the human-readable form by explaining the meaning of each feature: e.g., CCN denotes the number of times a file has been changed together with other files of the system, Dependers define the number of files which depend on a file, etc.

Although some of the rules may contain noise, overall, the output of the CN2 Rule Induction model is explicit and can easily be interpreted. Therefore, we assert that the rules generated by it can somewhat be regarded as the formalization of architectural integrity violations, and thus, among other things, we can relate them to the formal definitions of known hotspot patterns. To confirm that the produced rules indeed reflect the sought heuristics, three files, one for each identified hotspot pattern (see Table 10 for exact rules), have been picked, and DSMs of their architectural relations with one-hop neighbors have been constructed. As seen from Figures 11 to 13, representative files in fact pertain to the found heuristics (compare the manually created DSMs in Figures 11 to 13 with exemplary layouts in Figures 4, 6, and 7).

Table 10 Examples of the rules produced by the CN2 Rule Inducer based on the binary target variable, and their correspondence to the heuristics

Generated rule	Interpretation	Heuristic
0.0139 ≤ CCN, 0.0109 ≤ Dependers, 0.0031 ≤ DependersCCN ≤ 0.0092 (Quality 0.800; 3 bug-prone cases)	A file has at least 7 files that structurally depend on it and has been changed together with them 2 to 6 times. The total number of co-changes for this file constitute 18 or more.	Unstable Interface
CCN ≤ 0.0178, 0.0233 ≤ Dependers, 0.0168 ≤ DependersCCN ≤ 0.0184 (Quality 0.667; 1 bug-prone case)*	A file has at least 15 files that structurally depend on it and has been changed together with them 11 or 12 times. The total number of co-changes for this file constitute 23 or less.	
0.0101 ≤ CCN ≤ 0.0263, Dependers ≤ 0.0016, Dependees ≤ 0.0103, DependersCCN ≤ 0.0015 (Quality 0.789; 518 bug-prone cases)	A file has been changed together with other files of the system 13 to 34 times. It is structurally dependent on no more than 1 file, and no more than 1 file structurally depends on it. A file has been changed together with its depender (if it exists) no more than 1 time.	Modularity Violation Group
0.0279 ≤ CCN, Dependers ≤ 0.0016, Dependees ≤ 0.0103, DependeesCCN ≤ 0.0134 (Quality 0.948; 169 bug-prone cases)**	A file has been changed together with other files of the system 36 times or more. It is structurally dependent on no more than 1 file, and no more than 1 file structurally depends on it. A file has been changed together with its dependee (if it exists) no more than 2 times.	
0.0116 ≤ CCN ≤ 0.0434, Dependers ≤ 0.0124, 0.0092 ≤ DependersCCN, 0.0309 ≤ Dependees (Quality 0.867; 12 bug-prone cases)	A file has been changed together with other files of the system 15 to 56 times. It structurally depends on at least 3 files, and 1 to 8 files structurally depend on it. A file has been changed together with its dependers at least 6 times.	Crossing
0.0199 ≤ DependersCCN, 0.0206 ≤ Dependees (Quality 0.833; 4 bug-prone cases)***	A file structurally depends on at least 2 other files and have been changed together with the files that structurally depend on it at least 13 times.	

* Representative file: `lib/core/Services/Utilities.php`

** Representative file: `lib/core/Feed/ForwardLink.php`

*** Representative file: `lib/core/Tracker/Field/Abstract.php`

CN2 Rule Viewer

	IF conditions		THEN class	Distribution	Probabilities [%]	Quality	Length
60	CCN≥0.027885360185902403 AND CCN≥0.05034856700232378 AND CCN≥0.07978311386522076	→	BinaryBugProneness=1	[0, 74]	1 : 99	0.987	3
61	Dependees≥0.07216494845360824 AND CCN≥0.020914020139426802 AND CCN≥0.030209140201394268	→	BinaryBugProneness=1	[0, 40]	2 : 98	0.976	3
62	CCN≥0.027110766847405113 AND DependersCCN≤0.009188361408882083 AND CCN≥0.047250193648334625 AND Dependees≥0.010309278350515464	→	BinaryBugProneness=1	[0, 40]	2 : 98	0.976	4
63	CCN≥0.027110766847405113 AND Dependers≤0.0015503875968992248 AND Dependees≤0.010309278350515464 AND DependeesCCN≤0.013422818791946308 AND CCN≥0.033307513555383424	→	BinaryBugProneness=1	[1, 107]	2 : 98	0.982	5
64	CCN≥0.010069713400464756 AND Dependees≥0.10309278350515463	→	BinaryBugProneness=1	[0, 12]	7 : 93	0.929	2
65	CCN≥0.010069713400464756 AND DependeesCCN≤0.020134228187919462 AND CCN≥0.027885360185902403 AND Dependees≥0.020618556701030927 AND DependeesCCN≥0.006711409395973154	→	BinaryBugProneness=1	[1, 40]	5 : 95	0.953	5

Figure 10 An example of the CN2 Rule Induction output in Orange toolkit

		1	2	3	4	5	6	7	8
1	**lib/core/Services/Utilities.php**	**(1)**	,2	,1	,4	,3	,2		
2	lib/core/Services/Edit/PluginController.php	**x,2**	(2)		,2			,3	,2
3	lib/core/Services/Forum/Controller.php	**x,1**		(3)	,2				
4	lib/core/Services/User/Controller.php	**x,4**	,2	,2	(4)	,1	,8	,3	,2
5	lib/core/Services/Group/Controller.php	**x,3**			,1	(5)	,1		
6	lib/userslib.php	**x,2**			,8	,1	(6)		
7	lib/core/Services/Tracker/Controller.php	**x**	,3		,3			(7)	,3
8	lib/core/Services/File/Controller.php	**x**	,2		,2			,3	(8)

Figure 11 Detected instance of Unstable Interface. Excerpt of the Design Structure Matrix for lib/core/Services/Utilities.php (one-hop neighbors)

		1	2	3	4	5	6	7	8	9	10
1	**lib/core/Feed/ForwardLink.php**	**(1)**	**,19**	**,10**	**,10**	**,8**	**,11**	**,9**	**,9**	**,8**	**,8**
2	lib/wiki-plugins/wikiplugin_textlink.php	**,19**	(2)	,8	,5	,2	,5	,8	,4	,5	,1
3	lib/core/Feed/Abstract.php	**,10**	,8	(3)	,1	,3	,3	,5	,1	,2	
4	lib/core/Feed/ForwardLink/Metadata.php	**,10**	,5	,1	(4)	,2	,3	,5			
5	lib/core/Feed/ForwardLink/Receive.php	**,8**	,2	,3	,2	(5)	,2	,6			
6	lib/core/Feed/ForwardLink/Search.php	**,11**	,5	,3	,3	,2	(6)	,7			,2
7	lib/core/Feed/ForwardLink/Send.php	**,9**	,8	,5	,5	,6	,7	(7)			
8	lib/core/Feed/Remote/ForwardLink.php	**,9**	,4	,1					(8)	,6	
9	lib/core/Feed/Remote/ForwardLink/Contribution.php	**,8**	,5	,2					,6	(9)	
10	lib/core/JisonParser/Phraser/Handler.php	**,8**	,1				,2				(10)

Figure 12 Detected instance of Modularity Violation Group. Excerpt of the Design Structure Matrix for lib/core/Feed/ForwardLink.php (one-hop neighbors)

		1	2	3	4	5	6	7	8	9	10
1	lib/core/Tracker/Field/Interface.php	(1)				**,3**	,2	,1	,2		,1
2	lib/core/Tracker/Field/Indexable.php	x	(2)			**,2**		,1	,2	,1	,1
3	lib/tikilib.php			(3)	,2	**,2**					
4	lib/core/Tracker/Item.php			x,2	(4)	**,2**		,1	,1	,1	,1
5	**lib/core/Tracker/Field/Abstract.php**	**x,3**	**x,2**	**x,2**	**x,2**	**(5)**	**,2**	**,5**	**,6**	**,2**	**,3**
6	lib/core/Tracker/Field/Category.php	,2	x	x		**x,2**	(6)	,4	,1	,2	,4
7	lib/core/Tracker/Field/ItemLink.php	,1	,1	x	,1	**x,5**	,4	(7)	,6	,5	,6
8	lib/core/Tracker/Field/Text.php	,2	,2	x	,1	**x,6**	,1	,6	(8)	,1	,8
9	lib/core/Tracker/Field/ItemsList.php		,1	x	x,1	**x,2**	,2	,5	,1	(9)	,2
10	lib/core/Tracker/Field/Dropdown.php	,1	,1		,1	**x,3**	,4	,6	,8	,2	(10)

Figure 13 Detected instance of Crossing. Excerpt of the Design Structure Matrix for lib/core/Tracker/Field/Abstract.php (one-hop neighbors)

Analysis of the Decision Tree output has been performed in a similar fashion. The only difference is that the rules had to be manually derived from the tree structure itself. Figure 14 exemplifies what the generated tree looks like in the Tree Viewer in Orange. The presented fragment belongs to the model based on the categorical target variable. The graph-like representation allows us to follow the logic easily: for example, taking the very right branch of the tree - if a normalized number of Dependees is more than 0.0309, and a co-change number is greater than 0.0759, a file is most likely to have a low level of bug-proneness (more precisely, it with a 50/50 chance has either low or medium bug-proneness). In the same manner, each leaf of a tree corresponds to a particular rule, or a set of conditions. As with CN2 Rule Induction, feature values in these conditions can be renormalized and a human-readable interpretation of the corresponding rules can be composed. The process is roughly visualized in Figure 14 as well.

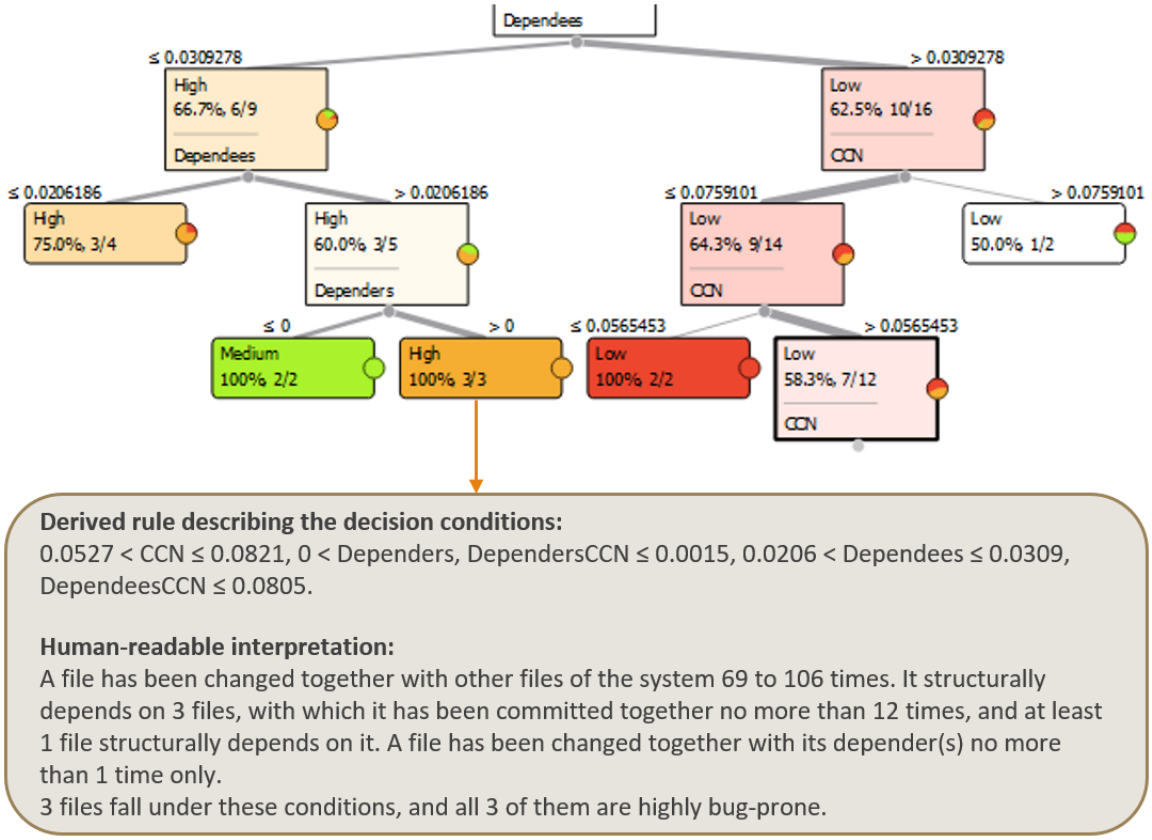


Figure 14 An example of the Decision Tree output in Orange toolkit

5.2.3 Analysis of the rule sets generated by white box heuristic-based models

Despite our research is a case study and thus the analysis results should not be hastily generalized or freely used to infer any statistics, the following outcomes have been produced by examining Decision Tree and CN2 Rule Induction models:

- CCN is the greatest determinant of file bug-proneness (the most important feature). It is confirmed by various attribute scoring methods (e.g. information gain ratio, Gini decrease, or ReliefF) and concurs with the findings presented in [15, 40].
- If a file has always been committed to the repository alone (its *CCN* = 0), it is likely to be non-bug-prone. If a file has low CCN and some dependencies, it is also usually non-bug-prone. However, if a file has low or medium CCN and no

dependencies at all, it very likely has a medium level of bug-proneness (presumably due to some shared logic which hasn't been manifested in an explicit connection, i.e. as a part of a Modularity Violation Group).

- Bug-prone cases have been found when a file with a moderate number of structural dependencies exhibits a fairly large number of co-changes, none of which, however, pertain to the files which structurally relate to it (*DependersCCN* = 0, *DependeesCCN* = 0).
- Amongst the patterns defined by [18, 27, 28, 30], Modularity Violation Group appears as the most common hotspot. Taking into consideration complete feature set only, in both of Decision Tree models and CN2 Rule Induction model with the binary target variable, Modularity Violation Group has comprised the majority of bug-prone cases with an adequately good CA (≈ 88% for Decision Tree and ≈ 83% for CN2 Rule Induction). This finding is in line with the study of 19 projects reported in [27, 30], where of all three patterns that we have been able to capture in our research Modularity Violation Group had the highest total number of distinct files involved in it. However, we would like to note that for CN2 Rule Induction model with the categorical target setting Crossing turned out to be the most widespread hotspot.
- Unstable Interface seems to be the most sound indicator of file bug-proneness as, for a complete feature set and, in case of the categorical target variable setting, with regard to medium or highly bug-prone instances, it has the CA of 100%, the best among other patterns. This has been the case for both of the Decision Tree models and the CN2 Rule Induction model with the binary target variable. No

single case of Unstable Interface has been found for the CN2 Rule Induction model with the categorical option (i.e. Unstable Interface still has the best accuracy of 100% when detected). This inference complies with the literature, which showed that Unstable Interface contributes the most to file change- and error-proneness (in terms of security bugs [18], as well as bugs in general [28]), while Crossing was ranked the second with respect to the severity of its impact [27, 30] (the latter statement, however, does not hold true for the Decision Tree model with the categorical target).

- There was a clearly detectable novel pattern of Reverse Unstable Interface, or, as we can call it to better convey its meaning, Unstable Coupled Client. As opposed to Unstable Interface which corresponds to the feature pair {high Dependers, high DependersCCN}, Reverse Unstable Interface is based on {high Dependees, high DependeesCCN} and denotes an unstable coupled client (in contrast to an interface) that structurally depends on a large number of files and is changed frequently with them as recorded in project's revision history. Figure 15 shows a fragment of the DSM composed manually for one of the source files attributed to this pattern. The file belongs to the exact same rule revealed by the Decision Tree model based on complete feature set with the categorical target variable which is depicted in Figure 14 with human-readable interpretation (*BugProneness = High*, 100% CA, 3/3 cases). As seen from its relationships with one-hop neighbors, the file indeed depends on many files and changes frequently with them. Overall, Reverse Unstable Interface was spotted in each white box model built on

complete feature set and, in general, appeared to be pretty common – many its instances had been found.

		1	2	3	4	5	6	7	8	9	10
1	lib/core/Search/Expr/Or.php	(1)	,5			,5	,4		,2		,1
2	lib/core/Search/Expr/Not.php	,5	(2)			,5	,4		,2		,1
3	lib/core/Search/Expr/MoreLikeThis.php			(3)						,2	,3
4	lib/core/Search/Expr/Distance.php				(4)			,1		,1	,2
5	lib/core/Search/Expr/And.php	,5	,5			(5)	,4		,2	,1	,1
6	lib/core/Search/Expr/Token.php	,4	,4			,4	(6)	,1	,2	,1	,2
7	lib/core/Search/Elastic/TypeFactory.php				,1		,1	(7)		,10	,2
8	lib/core/Search/Expr/Range.php	,2	,2			,2	x,2		(8)		,2
9	lib/core/Search/Elastic/Index.php			,2	,1	,1	,1	x,10		(9)	x,12
10	**lib/core/Search/Elastic/QueryBuilder.php**	**x,1**	**x,1**	**x,3**	**x,2**	**x,1**	**x,2**	**x,2**	**x,2**	**x,12**	**(10)**

Figure 15 An example of Reverse Unstable Interface (alternatively termed as Unstable Coupled Client) pattern. Excerpt of the Design Structure Matrix for lib/core/Search/Elastic/QueryBuilder.php (one-hop neighbors)

- Examination of the CN2 Rule Induction output has led to identification of two more potential hotspots, which, in our opinion, require additional confirmation by inspection of other software projects or alternative white box models based on the same set of features:

    1) {high DependersCCN, high DependeesCCN};
    2) {high Dependers, high Dependees}.

  The interpretation of the first pattern is that a file, whenever it's being changed, is changed together with the files it depends on and/or the files which depend on it. However, it may just signify that a co-change number (whenever it is a total number or the one pertaining to file's dependers/dependees) serves as an important indicator of file bug-proneness, which would support our first finding, but would not denote a novel hotspot. The second pattern describes a file which is highly coupled with other components of a system, forming some sort of a

structural hub. Table 11 presents the distribution of files which fell under the aforementioned patterns with the calculated average accuracy.

Table 11 The distribution of files in possible hotspot patterns (previously unknown) detected by the CN2 Rule Induction

Pattern number	Target variable	Bug-prone files[7]	Files total	Accuracy
1	Binary	17	18	94%
	Categorical	2	2	100%
2	Binary	23	27	85%
	Categorical	4	4	100%

5.2.4 Performance comparison of different algorithms (heuristic-based models)

Our experiments show that when the target variable is defined as binary, the Decision Tree performs the worst, while CN2 Rule Induction appears as the most effective option. However, when the target variable is defined as categorical (and we are targeting only cases with medium or high levels of bug-proneness), the situation changes the other way around: CN2 Rule Induction becomes one of the weakest algorithms (along with Naive Bayes), while Random Forest takes the lead, with Decision Tree following very closely. In particular, Random Forest appears to be the most successful in detecting highly bug-prone files, while Decision Tree excels at identifying files of medium bug-proneness. This may mean that the CN2 Rule Induction outperforms its counterparts at predicting low bug-prone files, whereas, when the goal is to expose severe cases, the Decision Tree or the Random Forest has to be utilized.

Table 12 exhibits the performance indicators for machine learning models based on complete feature set with respect to the malicious target class (*BugProneness* = 1 for

[7] For the categorical setting, files with *BugProneness* = *Medium* or *High* have been regarded as "bug-prone," and files with *BugProneness* = *None* or *Low* – as "benign."

the binary setting of target variable, and *BugProneness* = *Medium* or *High* for the categorical one). The best values are highlighted in bold. As our analysis involved the examination of the confusion matrices, we include the information on false negatives, which have been regarded as the most important misclassification parameter (due to the application of our research). The summary is provided in Table 13. When the target is set as binary, CN2 Rule Induction performs better than other algorithms, having misclassified 186 buggy files as benign (compared to 274 for Naïve Bayes, 293 for Random Forest, 306 for AdaBoost, and 327 for Decision Tree). When the categorical setting is adopted, Naive Bayes at first sight leads with 4 files of medium and high bug-proneness misclassified as non-bug-prone (CN2 Rule Induction misclassified 6, Random Forest – 16, AdaBoost – 21, and Decision Tree – 22). However, if we include files of medium or high bug-proneness misclassified as low bug-prone, the total numbers would come to 238 for Naive Bayes, 245 for CN2 Rule Induction, 213 for Random Forest, 191 for AdaBoost, and 205 for Decision Tree (all values are provided based on complete feature set).

Comparing the performance of Decision Tree models based on complete and reduced feature sets with respect to the malicious target class, the reduced feature set has been found to have a bit higher AUC, however, in terms of CA, F1 score, precision, and recall, the complete feature set has slightly outperformed the simplified version (please refer to Table 14 for exact numbers). With regard to false negatives, the complete feature set has also performed better. When the target variable was defined according to the binary setting, it has produced 327 false negatives compared to 335 for the reduced feature set. At the same time, when the target was set as a categorical variable, the model

based on complete feature set has misclassified 22 files of medium and high bug-proneness as non-bug-prone and 183 files as low bug-prone versus 33 and 182 files for the reduced set, respectively (see fragments of confusion matrixes in Figure 16). Nevertheless, this difference appears to be insignificant from a practical perspective, which suggests that system analysts can use the consolidated information on structural dependencies and evolutionary couplings with relatively similar results.

Table 12 Performance indicators for different heuristic-based models built on complete feature set

Model name	AUC	CA	F1	Precision	Recall
Binary option (Target class = 1)					
Decision Tree	0.692	0.695	0.784	0.738	0.836
CN2 Rule Induction	0.721	0.694	**0.797**	0.711	**0.906**
Random Forest	**0.729**	0.701	0.791	0.738	0.853
Naïve Bayes	0.712	0.700	0.792	0.733	0.862
AdaBoost	0.701	**0.705**	0.792	**0.744**	0.846
Categorical option (Target class = Medium)					
Decision Tree	0.670	0.920	0.143	0.225	0.105
CN2 Rule Induction	0.778	0.930	0.054	0.188	0.032
Random Forest	0.770	0.925	0.111	0.226	0.074
Naïve Bayes	**0.779**	**0.936**	0.010	**0.250**	0.005
AdaBoost	0.595	0.914	**0.174**	0.223	**0.142**
Categorical option (Target class = High)					
Decision Tree	0.745	0.961	0.410	0.512	0.342
CN2 Rule Induction	**0.925**	0.961	0.356	0.533	0.267
Random Forest	0.883	**0.964**	**0.443**	**0.581**	**0.358**
Naïve Bayes	0.873	0.922	0.239	0.196	0.308
AdaBoost	0.733	0.956	0.385	0.441	0.342

In terms of execution time, almost all machine learning algorithms have yielded comparable results, measured in milliseconds. Table 15 records the exact training and testing time for binary and categorical complete feature sets. Testing time has been evaluated on a slightly modified original dataset (`mlinput.csv`, see Appendix D) as if we were trying to detect architectural integrity violations after a new commit had been added to the revision history. The only algorithm that stands out is the CN2 Rule Induction

which has had the highest training time among all (almost 9 seconds for the binary feature set and 18 seconds for the categorical one), although its testing time, once trained, has been on par with other machine learning methods.

Table 13 False negative values for heuristic-based machine learning models built on complete feature set

Model name	FN
Binary target variable setting: files with *BugProneness* = 1 misclassified as those with *BugProneness* = 0	
Decision Tree	327
CN2 Rule Induction	**186**
Random Forest	293
Naïve Bayes	274
AdaBoost	306
Categorical target variable setting: files with *BugProneness = Medium* or *High* misclassified as those with *BugProneness = None*	
Decision Tree	22
CN2 Rule Induction	6
Random Forest	16
Naïve Bayes	**4**
AdaBoost	21
Categorical target variable setting: files with *BugProneness = Medium* or *High* misclassified as those with *BugProneness = None* or *Low*	
Decision Tree	205
CN2 Rule Induction	245
Random Forest	213
Naïve Bayes	238
AdaBoost	**191**

Table 14 Performance indicators for heuristic-based Decision Tree classifiers built on complete and reduced feature sets

Feature set	AUC	CA	F1	Precision	Recall
Binary option (Target class = 1)					
Complete	0.692	**0.695**	**0.784**	**0.738**	**0.836**
Reduced	**0.711**	0.688	0.779	0.733	0.832
Categorical option (Target class = Medium)					
Complete	0.670	**0.920**	**0.143**	**0.225**	**0.105**
Reduced	**0.689**	**0.920**	0.131	0.212	0.095
Categorical option (Target class = High)					
Complete	0.745	**0.961**	**0.410**	**0.512**	**0.342**
Reduced	**0.757**	0.960	0.388	0.500	0.317

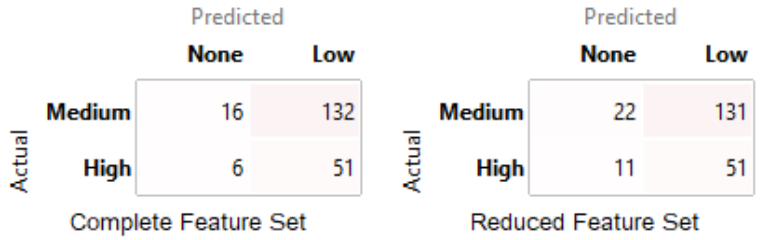


Figure 16 False negative values for heuristic-based Decision Tree models built on the categorical target variable

Table 15 Execution time of different heuristic-based models built on complete feature set

	Binary option		*Categorical option*	
ML algorithm	**Train**	**Test**	**Train**	**Test**
Decision Tree	00s : 41ms	00s : 30ms	00s : 32ms	00s : 25ms
CN2 Rule Induction	08s : 63ms	00s : 40ms	18s : 09ms	00s : 32ms
Random Forest	00s : 18ms	00s : 11ms	00s : 20ms	00s : 18ms
Naïve Bayes	00s : 16ms	00s : 10ms	00s : 14ms	00s : 12ms
AdaBoost	00s : 20ms	00s : 18ms	00s : 39ms	00s : 30ms

The difference in execution time between Decision Tree models built on complete and reduced datasets has been insignificant, as can be seen from Table 16.

Table 16 Execution time of heuristic-based Decision Tree classifiers built on complete and reduced feature sets

	Binary option		*Categorical option*	
Feature set	**Train**	**Test**	**Train**	**Test**
Complete	00s : 41ms	00s : 30ms	00s : 32ms	00s : 25ms
Reduced	00s : 21ms	00s : 16ms	00s : 30ms	00s : 24ms

The experiments were conducted on a Windows 10 machine with Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz processor and 4GB of RAM.

5.2.5 Summary

Based on the results of our analysis, we can answer the research questions RQ1 and RQ2 positively: machine learning is capable of replicating hotspot patterns proposed by existing research (RQ1) and can even discover novel rules for pinpointing architectural integrity violations (RQ2).

5.3 Limitations

In this section, we discuss the limitations of the Part 1 of our study and propose the directions for future research.

First, our definition of bug frequency (and, consequently, of bug-proneness) has been based on file participation in bug-fixing commits. However, although bug fixes may provide a good proxy for bug-proneness, we could instead focus on bug-introducing commits [37] and architectural relations that induce subsequent bug fixes.

Second, we have regarded a co-change as two files being changed together in a single commit [6, 11, 19, 27, 30, 42]. Alternatively, a co-change could be defined as two files being committed together in a given time period. Such approach would allow us to work with timestamps rather than require knowledge of particular commits. Depending on the developer commit style, changes associated with a specific task may spread over multiple records of the version control system [40]. The method of a "sliding window" described above takes account of modifications that span multiple commits [37, 47]. Graph-based techniques can also be used to identify files which are part of the same history, but never of the same commit, i.e., in this case, are coupled through co-changing with other mediatory files [38]. It would introduce a "higher-order logical coupling" into analysis; however, even aggregating co-changes within a particular time frame should partly answer the purpose.

Third, the hotspot pattern approach defines evolutionary coupling through a co-change threshold, i.e. the two files are considered evolutionary coupled only if the number of times they have changed together in the commit history is greater than a certain value [27, 28, 30, 42]. If the number of co-changes does not exceed the chosen

threshold, the logical dependency is removed from the graph of architectural relations. Although occasional co-changing may be deemed as noise [5, 16, 36], no co-change threshold was set in our research, and thus all logical dependencies were examined, without exception.

Fourth, as noted previously, since co-changes and structural dependencies were aggregated as total numbers for each individual node, and thus our features did not reflect the actual graph-like nature of architectural relations between the files, we have been unable to observe Unhealthy Inheritance Hierarchy and Clique patterns[8], or to reveal other hotspots which are conditioned by these graph-like relations. In this regard, future investigation should give due consideration to graph-based machine learning models, especially white box ones, e.g. Decision Tree Chunkingless Graph-Based Induction [32]. Moreover, one could experiment with completely new features, not pre-determined by the existing approach (or defined differently). For instance, number of lines of code added/deleted could be included in the feature set to take into account the magnitude of change [40], structural dependency could be quantified as the number of variable or method calls [1] to manifest a strength of the relationship as opposed to just binary state "exists/not exists," or, instead of co-changes, the evolutionary coupling between the files could be modeled based on (co-)change probability [4, 14, 30, 44]. Exploration into machine engineered features (like those derived by NRL algorithms, such as node2vec or GraphWave [17]) may yield particularly interesting results. Although the interpretation of such features may be more complicated, they preclude logical assumptions being made when looking for patterns (and thus being incorporated into a feature set), substituting the

[8] Package Cycle was anyway outside the scope of our research setting.

usual abductive reasoning *from a hypothesis to a feature* with an inductive one – *from a feature to a hypothesis*.

Fifth, unity-based normalization has been used in our research to prevent the skewness of machine learning models towards features with the broader range of values. Since bug and change rates correlate with file size (i.e. the larger the file, the more often it is being changed and the greater the chance that it contains a bug), one could instead normalize variables by lines of code to ensure that high values are not merely subject to finding large files [11].

Sixth, another threat comes from the class imbalance problem, which is a common problem in machine learning and defect prediction. However, our objective is to discover patterns in architectural data, not to propose a new defect prediction/localization algorithm [14].

Seventh, as mentioned earlier, the validity of some less certain patterns should be reinforced by further analysis. For example, in terms of [3], the potential pattern 2 that we have identified ({high Dependers, high Dependees}) may denote core components which are members of the largest cyclic group and have the same visibility fan-in (the number of components that directly or indirectly depend on them) and visibility fan-out (the number of components that they directly or indirectly depend on). Please note that the visibility fan-in and fan-out include indirect dependencies as well, so they are not the same as dependers and dependees in our research. To examine this hypothesis, one needs to check which actual files are involved. Could we consider those files core files in the Tiki project, or are these high numbers an architectural flaw?

Eighth, despite there was one study conducted on hotspot pattern correlation with security bugs [18], more research on patterns of architectural integrity violations and their impact on software security is required, especially regarding the presence of any currently unknown hotspots, specific solely to security bugs.

Lastly, DRSpaces can be used for other analytical tasks accomplishable with architectural data, such as pinpointing system components with increasing complexity [27, 29, 30]. Therefore, it would be interesting to examine if machine learning is capable of discovering other suites of patterns, for example, related to software maintenance or maintainability.

6. Part 2: Exploration into node embeddings as machine-engineered features preserving the network structure of a DRSpace

6.1 Details of the method

If we circle back to the software architecture model the hotspot patterns are defined upon, DRSpace, we would notice that it has the form of a layered hierarchy. Whereas in RQ1 and RQ2 only file's immediate architectural dependencies have been considered, DRSpace is not limited to 1-hop neighbors, directly related to the studied file, but also may comprise n-hop neighbors such that $n > 1$. These indirect dependencies are important because a change to one element may cause a cascade of subsequent changes traveling through the dependency network [19], which is why some research on software quality has been specifically devoted to the analysis of mediated, transitive architectural relationships [26, 38, 39]. Nonetheless, it is yet unclear what degree of network exploration is appropriate – yes, DRSpaces reflect locality of structural and logical dependencies in a system, but do we actually need to consider local vs. global in the first place?

Another deviation of our original methodology from the hotspot pattern approach, as raised in the previous section, pertains to the co-change threshold in the definition of evolutionary coupling (see Section 2.2.2 – the two files are called evolutionary coupled if the number of co-changes they have exceeded a certain $cochange_{thr}$). Authors of the hotspot pattern approach use $cochange_{thr} = 2$ in the latest studies [27, 30][9]. We, in RQ1

[9] Previously they had used $cochange_{thr} = 4$ [28] and $cochange_{thr} = 10$ [42].

and RQ2, took into account all logical dependencies, regardless of their "strength," i.e. regardless of the number of co-changes these files shared, so our $cochange_{thr} = 0$.

In order to examine the impact of the locality of architectural relations and the composition of a logical dependency subgraph on the quality of bug detection, we have extended our research to network representation learning as a methodology for automated feature engineering preserving the network structure of a given space (in our case – architectural space). When we ignored indirect dependencies in RQ1 and RQ2, we assumed everything was local, but it does not reflect what a DRSpace is. Node embeddings produced by NRL algorithms will let us play with local vs global by setting different local/global ratios. Although we are not representing DRSpaces clearly in that case, to some degree we are.

How do we combine structural and evolutionary dependencies for network representation learning? Previous research proposes to consider the union of the two different dependency (sub)graphs [7], which would cover most of the defective files - unlike intersection, which would only capture files with most severe problems [14]. For the sake of simplicity, we will regard our union of two graphs as undirected. While evolutionary coupling is undirected[10] per se, some studies show that treating structural dependencies as undirected, instead of directed, relationships does not qualitatively impact the network recovery [19].

[10] Unless we purposefully make it directed through weighting, e.g., if we try to take into account the value, the relative importance of a given evolutionary relation to a specific node and weigh a directed edge by dividing the number of co-changes shared between the two files by the total number of co-changes the file has had.

Discussion about weights is a difficult one. Even assigning weights to edges in a logical dependency graph alone is already problematic. If we just weigh each edge with the number of co-changes shared between the two files, a change transaction of *n* artifacts would increase the weights of $\frac{1}{2}n(n-1)$ edges by 1, which is an increase of $\frac{1}{2}(n-1)$ per changed artifact. So, a change of an artifact in a small commit would be less important than a change in a large commit, and the results would be skewed towards large transactions. To prevent this bias, the weighting function for evolutionary coupling should monotonically decrease with the size of the commit [8].

Bringing structural dependencies into the picture poses other challenges. We know that there is no significant correlation between the strengths of the two architectural dependency types [1], and should we actually quantify the structural coupling as the strength, e.g., as the number of variable or method calls from the depender to the dependee? Or should we set a common, "default" weight for all structural relationships instead (binary weighting: 0 if the files are structurally independent, and a set *x* if a dependency exists)? And how to balance structural and evolutionary coupling? Our own attempt to do that in `visualization.py` is far from perfect – we tried to compute the default weight for the structural dependency subgraph as *DefaultWeightForDependency* = max(*CCN*)/2 (see Appendix B), which in the case of our data (see Chapter 4) equaled to 32. As per `cochanges19.csv` which contains evolutionary information then to be merged with structural dependencies (see Appendix B), the share of file pairs with 32 or more co-changes was only 0.022%[11a], so the results produced by such formula are largely

[11] When information on evolutionary couplings is merged with the structural dependency data, the value decreases accordingly, because of the file pairs with *Status* = 1 (see Appendix B): a. to 0.021%, b. to 1.41.

skewed towards structural dependency impact (if different dependency subgraphs are considered equally important, then they should have approximately the same edge weights in the combination [7]). If we try to modify the weighting function to a mean of all co-change numbers instead, the structural dependency weight will amount to 1.48[11b] (or, rounded, 1), which seems insufficient. Should we take a median? Since there are no clear guidelines in the scholarly literature on adequate tackling of this problem, we decided to lay the foundation for the future research by producing a baseline model based on unweighted edges.

The resulting undirected unweighted union of logical and structural dependency (sub)graphs needs to be represented as a set of vertex embeddings. We have chosen node2vec [21] as an NRL algorithm to accomplish this task. node2vec exhibits fairly good performance [46] and, being based on random walk with stochastic gradient descent optimization, is much more efficient than matrix factorization based methods (e.g., GraphWave [17], mentioned in Section 2.1) that are solved by eigen decomposition and alternative optimization. What is, however, the most important is that node2vec, with its two hyperparameters shaping the behavior of a random walk, allows us to balance between local/global network exploration strategies, calibrating the locality of topology being recovered. The return parameter p determines the likelihood of a random walk immediately revisiting a node it has just come from (i.e., whether it would backtrack to the node preceding its current state), while the in-out parameter q conditions the inward vs outward walk orientation manifested in two extreme sampling strategies: Breadth-first Sampling (BFS) and Depth-first Sampling (DFS). The difference between both search strategies is illustrated in Figure 17: BFS goes around the source node, generating

neighborhoods from its immediate neighbors, whereas DFS samples vertices at increasing distances from the source, moving further away in the network.

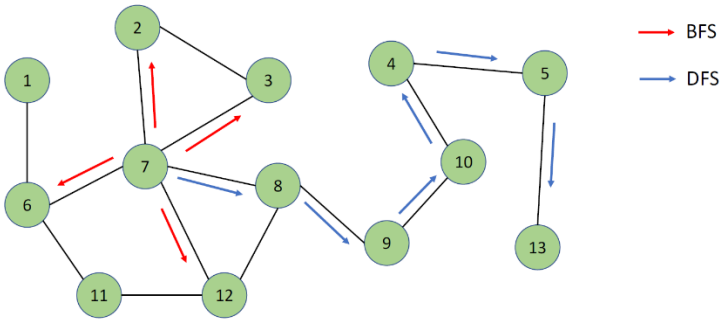


Figure 17 Neighborhood sampling strategies considered by node2vec: BFS and DFS (adopted from [46])

Consider a random walk that just traversed edge (t, v) and now resides at node v (see Figure 18) [21]. To decide what vertex to move to next, the algorithm evaluates the transition probability $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ on all edges (v, x) leading from v, where $\alpha_{pq}(t, x)$ is the search bias and w_{vx} is the static edge weight [21] (in our case $w_{vx} = 1$ for all edges, since the graph is unweighted). As we can see from Figure 18, the search parameters impact the random walk transition probability in the following way:

- $p < \min(q,1)$ makes the walk "local," close to the starting node;
- $p > \max(q,1)$ decreases the probability of revisiting the previous node, encourages moderate exploration and avoids 2-hop redundancy in sampling;
- $q < 1$ enforces Depth-first Sampling (inferring communities based on homophily) – the walk is more inclined to visit nodes which are further away from the preceding node;

- $q > 1$ prioritizes Breadth-first Sampling (learning embeddings that reflect structural equivalence) – the random walk is biased towards nodes close to the preceding node.

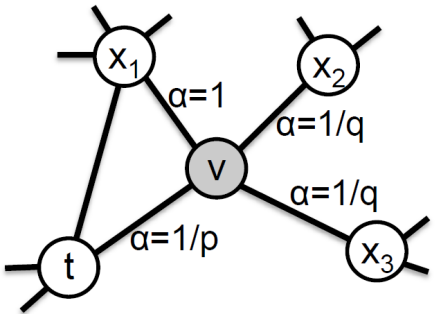


Figure 18 node2vec random walk transition probability bias (adopted from [21])

This helps us adjust the degree of exploration necessary for feature learning [21]. We will examine 9 possible cases:

- Case #1 – $p = 0.25$, $q = 0.5$ (local DFS)
- Case #2 – $p = 1$, $q = 0.5$ (balanced DFS)
- Case #3 – $p = 2$, $q = 0.5$ (global DFS)
- Case #4 – $p = 0.5$, $q = 1$ (local balanced walk)
- Case #5 – $p = 1$, $q = 1$ (default values)
- Case #6 – $p = 2$, $q = 1$ (global balanced walk)
- Case #7 – $p = 0.5$, $q = 2$ (local BFS)
- Case #8 – $p = 1$, $q = 2$ (balanced BFS)
- Case #9 – $p = 4$, $q = 2$ (global BFS)

In addition to the 9 network exploration alternatives realized in the above cases, we will also consider two different thresholds for the number of times the two files should be committed together to be considered evolutionary coupled. Thus we introduce

a notion of CCN Threshold, i.e. "co-change (number) threshold." For the sake of brevity, the word "number" will be omitted in the verbal reference to this parameter.

If the threshold is set to 1, all the co-changing pairs are being included in the analysis, regardless of the number of co-changes, or regardless of the "strength" of a logical coupling. As mentioned before, two thresholds have been examined: *CCNThreshold* = 1 (all co-changing pairs are included) and *CCNThreshold* = 3 (only pairs which have co-changed 3 times or more are included). The latter approach allows us to omit file pairs which are only occasionally changed together, as suggested by the authors of hotspot pattern approach [27, 28, 30], who state that occasional co-changing is noise. We will check if this understanding is correct. With the number of co-changes spanning from 0 to 65, only 8.7% of file pairs in our research have $CCNThreshold \geq 3$. The number of individual files under study at *CCNThreshold* = 3 has reduced from 3001 to 1924 (as per `file-node_mapping_th3.csv` produced by `node2vec_input.py`, see Appendix E.2).

Please note that in the hotspot pattern approach the two files are considered evolutionary coupled if the number of co-changes they share is strictly greater than the set threshold of co-change frequency ($CCN > cochange_{thr}$). In our research CCN Threshold is a minimum number of co-changes the two files should share to be considered to have changed together "frequently," i.e. the threshold value is included ($CCN \geq CCNThreshold$). Therefore, our *CCNThreshold* = 3 corresponds to $cochange_{thr} = 2$, which the authors of hotspot pattern approach have fixed on [27, 30].

The above brings us to 18 combinations of p, q, and CCN Threshold overall. The target variable has been (re)defined as a category: files having $BugFrequency \leq 9$ are considered benign, and files with $BugFrequency \geq 10$ are considered bug-prone. This

mapping corresponds to the one used for heuristic-based models with a categorical target variable, when only cases pertinent to Medium or High categories of bug-proneness were investigated. This time, however, the extracted data has been undersampled to tackle the class imbalance (310 bug-prone files out of 3001 files total for *CCNThreshold* = 1 combinations and 278 out of 1924 total for *CCNThreshold* = 3). Especially since the features in embedding-based models are machine-engineered, we need to ensure good performance. As opposed to Part 1, where our objective was to discover architectural patterns and thus we needed all the data to identify as many hotspots as we could, now it is reasonable that we omit some instances. So, we have balanced the data by randomly undersampling the majority class without replacement (so the values in the produced sample have all been unique).

As one more distinction from the method utilized for heuristic-based models, in the embedding-driven analysis the number of machine learning algorithms has increased to 7 – Logistic Regression and Neural Network were added to the same list of 5. Neural Network algorithm in Orange belongs to multi-layer perceptrons, a class of feedforward artificial neural network [34, 35]. We used the default Orange parameters for the algorithms (please see Table 17 for exact settings). The performance of the resulting models has been assessed based on stratified 10-fold cross validation.

Please refer to Appendix E for more details on data post-processing for machine learning models built on node representations.

Table 17 Machine learning algorithm parameters in embedding-based models

Parameter	Value
Decision Tree	
Induce binary tree	yes
Min. number of instances in leaves	2
Do not split subsets smaller than	5
Limit the maximal tree depth to	100
Stop when majority reaches	95%
CN2 Rule Induction	
Rule ordering	Ordered
Covering algorithm	Exclusive
Evaluation measure	Entropy
Beam width	5
Minimum rule coverage	1
Maximum rule length	5
Statistical significance (default α)	unchosen/unchecked
Relative significance (parent α)	unchosen/unchecked
Random Forest	
Number of trees	10
Number of attributes considered at each split	unchosen/unchecked
Replicable training	no
Limit depth of individual trees	unchosen/unchecked
Do not split subsets smaller than	5
AdaBoost	
Base estimator	Tree
Number of estimators	50
Learning rate	1.0
Fixed seed for random generator	unchosen/unchecked
Classification algorithm	SAMME.R
Regression loss function	Linear
Logistic Regression	
Regularization type	Ridge (L2)
Strength	C = 1
Neural Network	
Neurons in hidden layers	100,
Activation	ReLu
Solver	Adam
Regularization	α = 0.0001
Maximal number of iterations	200
Replicable training	yes

6.2 Results and discussion

The performance indicators and some values of the confusion matrix (specifically, false negatives, false positives, and true positives) are presented in Appendix F for each embedding-based model.

6.2.1 Best machine learning algorithm (embedding-based models)

Naïve Bayes, Neural Network and Logistic Regression are three sure leaders in terms of performance (we regard recall as the most important indicator). Naïve Bayes performs the best when all co-changes are being considered (*CCNThreshold* = 1), but, when CCN Threshold is set at 3, it is always surpassed by Neural Network and Logistic Regression. However, if we scrutinize the best-performing models at each of the 18 combinations of p, q, and CCN Threshold, we would notice that on average the top models for *CCNThreshold* = 1 across 9 p/q cases (all belonging to Naïve Bayes) achieve *recall* ≈ 82%, whereas the top models for *CCNThreshold* = 3 (all belonging to Neural Network) only attain *recall* ≈ 76%. Moreover, Naïve Bayes models with CCN Threshold = 1 in general have the highest recall among all others, including those built on *CCNThreshold* = 3. The only exception is the Naïve Bayes model at case #2 – $p = 1$, $q = 0.5$ (balanced DFS), which is outperformed by the best model at *CCNThreshold* = 3 – Neural Network at case #8 – $p = 1$, $q = 2$ (balanced BFS). Therefore, we would still call Naïve Bayes the best algorithm, although this superiority comes at a cost of ~10-30% more false positives than Neural Network and Logistic Regression with *CCNThreshold* = 1 produce.

6.2.2 Best embedding-based machine learning model

Naïve Bayes at case #5 – $p = 1$, $q = 1$ (default values) with *CCNThreshold* = 1 has been the best machine learning model, showing the highest recall amongst all others. The number of false positives (*FP* = 101) is pretty much comparable to what have been produced by the next two alternatives within the same combination – Logistic Regression (*FP* = 85) and Random Forest (*FP* = 97).

The model #6 – $p = 2$, $q = 1$ (global balanced walk), *CCNThreshold* = 1, Naïve Bayes, is the second best, achieving a little bit better precision, but proportionally worse recall (please refer to Table 18 below).

6.2.3 Best combination of p, q, and CCN Threshold

The best combination is #6 – $p = 2$, $q = 1$ (global balanced walk) with *CCNThreshold* = 1. The analysis has been based on the recall and precision values achieved by the top 3 models at each combination. Although the best machine learning model belongs to case #5 with *CCNThreshold* = 1, the next two alternatives perform slightly better at case #6, see Table 18.

Table 18 Top three embedding-based models in each of the two best-performing combinations of p, q, and CCN Threshold

ML algorithm	Precision[12]	Recall[12]	FN	FP	TP
Case #6 – $p = 2$, $q = 1$ (global balanced walk) with *CCNThreshold* = 1					
Naïve Bayes	0.733	0.832	52	94	258
Logistic Regression	0.736	0.765	73	85	237
Neural Network	0.769	0.761	74	71	236
Case #5 – $p = 1$, $q = 1$ (default values) with *CCNThreshold* = 1					
Naïve Bayes	0.722	0.845	48	101	262
Logistic Regression	0.735	0.761	74	85	236
Random Forest	0.695	0.713	89	97	221

[12] With regard to target class = bug-prone.

6.2.4 3D graphs based on average recall values

Figures 19-24 depict 3D graphs built on average recall values obtained by different sets of embedding-based models in each of 18 combinations of p, q, and CCN Threshold: across all machine learning algorithms (Tables 19-20), for the best algorithm – Naïve Bayes – only (Tables 21-22), and across three leaders in terms of performance whose average *recall* > 70%: Naïve Bayes, Logistic Regression, and Neural Network (Tables 23-24). The graphs at first are presented individually (Figures 19-21), to provide a better view of the surface, and then are repeated with a common legend across all plots (Figures 22-24).

6.2.5 Impact of return parameter p (local walk close to the starting node vs moderate exploration)

It seems there is no consistency in the impact of p on the performance. The only observation that can potentially be made is that for models based on *CCNThreshold* = 3 p had better be set as balanced; however, it does not extend to the models with *CCNThreshold* = 1, where the results are pretty mixed, so no preferred setting can be discerned.

6.2.6 Impact of in-out parameter q (DFS vs BFS)

As can be inferred from the 3D graphs and the corresponding tables with the average recall values, *q = balanced* tends to lead to a better performance than *q = BFS* or *q =DFS*, which prioritize Breadth-first Sampling or Depth-first Sampling in the network exploration strategy.

Individually (zoomed in)

Across all algorithms

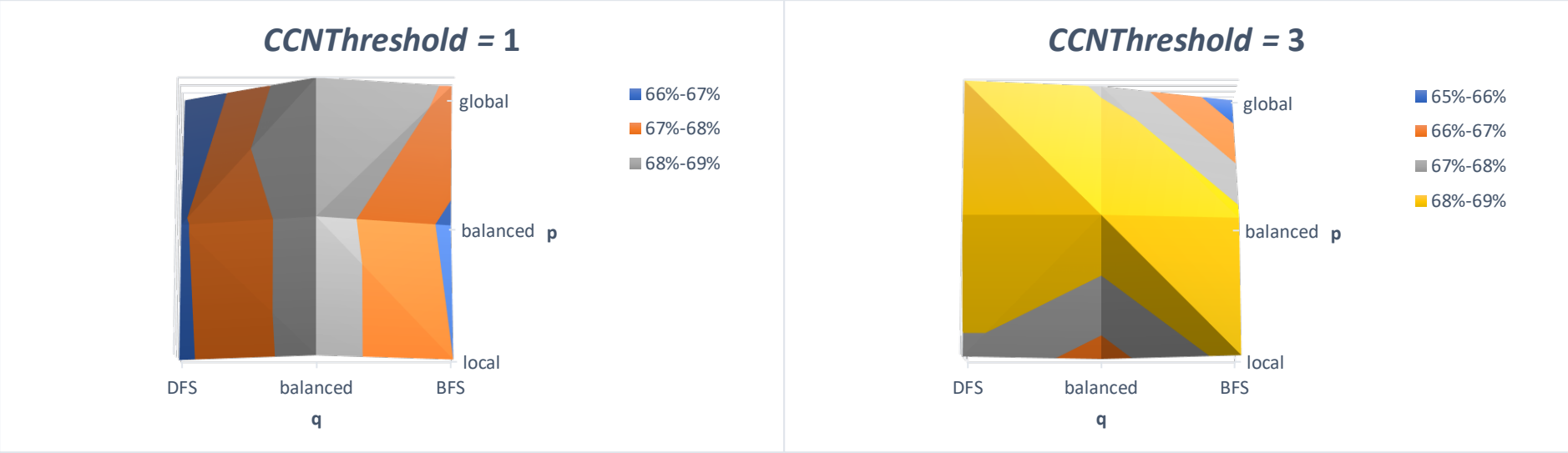


Figure 19 3D graphs built on average recall values for embedding-based models across all algorithms (individual charts)

Table 19 Average recall values for embedding-based models across all algorithms in each of 9 p/q cases with CCNThresold = 1

	q		
p	**DFS**	**balanced**	**BFS**
local	67%	69%	67%
balanced	67%	69%	67%
global	66%	69%	68%

Table 20 Average recall values for embedding-based models across all algorithms in each of 9 p/q cases with CCNThresold = 3

	q		
p	**DFS**	**balanced**	**BFS**
local	68%	67%	68%
balanced	69%	69%	68%
global	69%	68%	65%

Only Naïve Bayes

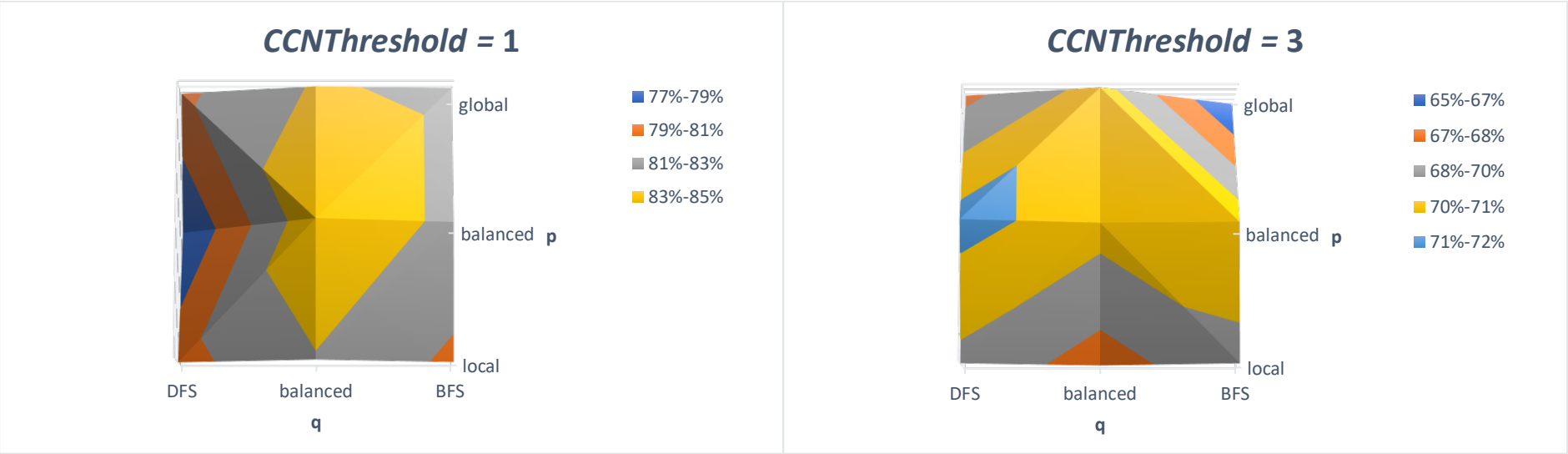


Figure 20 3D graphs built on recall values for embedding-based Naïve Bayes models (individual charts)

Table 21 Recall values for embedding-based Naïve Bayes models in each of 9 p/q cases with CCNThresold = 1

	q		
p	**DFS**	**balanced**	**BFS**
local	80%	83%	81%
balanced	77%	85%	83%
global	81%	83%	83%

Table 22 Recall values for embedding-based Naïve Bayes models in each of 9 p/q cases with CCNThresold = 3

	q		
p	**DFS**	**balanced**	**BFS**
local	69%	67%	69%
balanced	72%	70%	71%
global	68%	70%	65%

Across three leaders in terms of performance whose avg. recall > 70%: Naive Bayes, Logistic Regression, and Neural Network

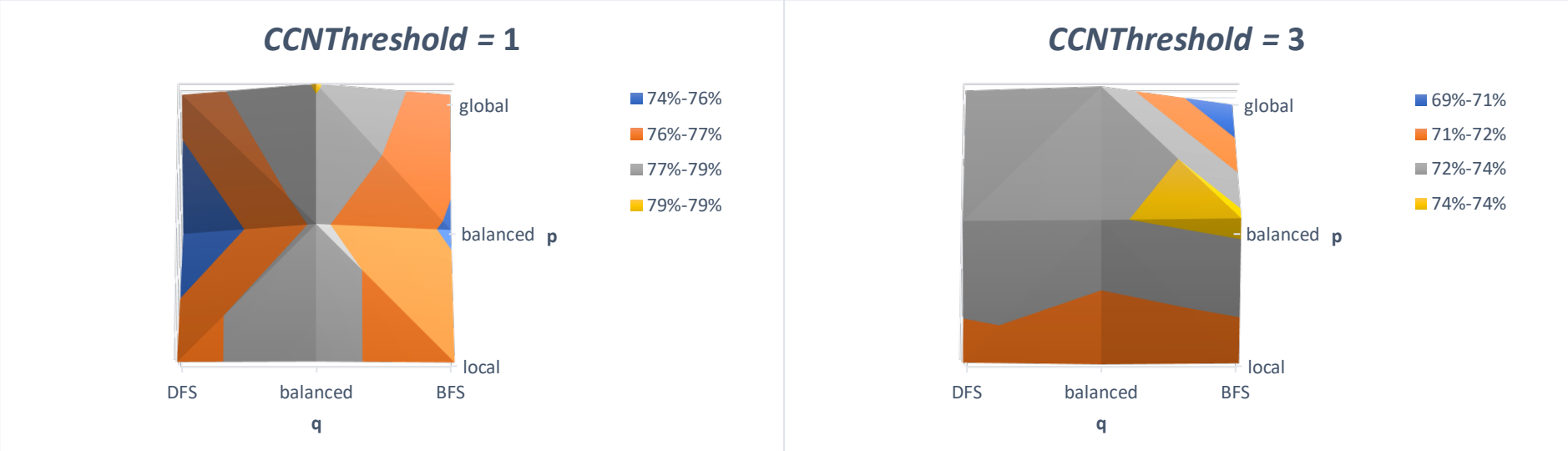


Figure 21 3D graphs built on average recall values for embedding-based Naive Bayes, Logistic Regression, and Neural Network models (individual charts)

Table 23 Average recall values for embedding-based Naive Bayes, Logistic Regression, and Neural Network models in each of 9 p/q cases with CCNThresold = 1

	q		
p	**DFS**	**balanced**	**BFS**
local	77%	77%	77%
balanced	74%	77%	75%
global	76%	79%	76%

Table 24 Average recall values for embedding-based Naive Bayes, Logistic Regression, and Neural Network models in each of 9 p/q cases with CCNThresold = 3

	q		
p	**DFS**	**balanced**	**BFS**
local	72%	71%	71%
balanced	73%	73%	74%
global	72%	73%	69%

Common legend (zoomed out)

Across all algorithms

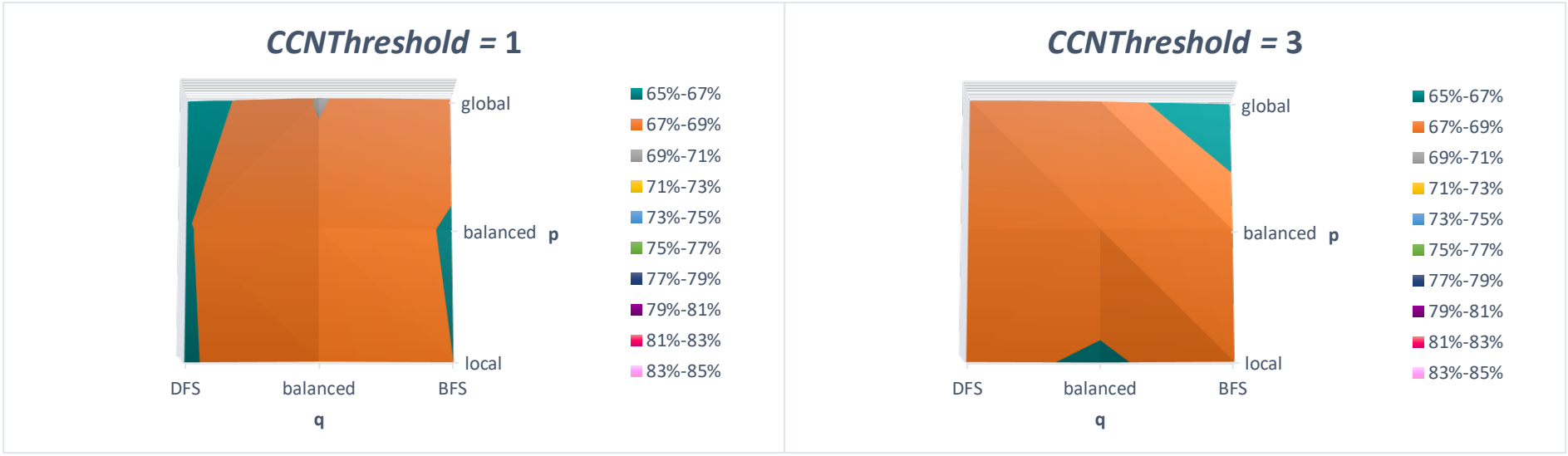


Figure 22 3D graphs built on average recall values for embedding-based models across all algorithms (common legend)

Only Naïve Bayes

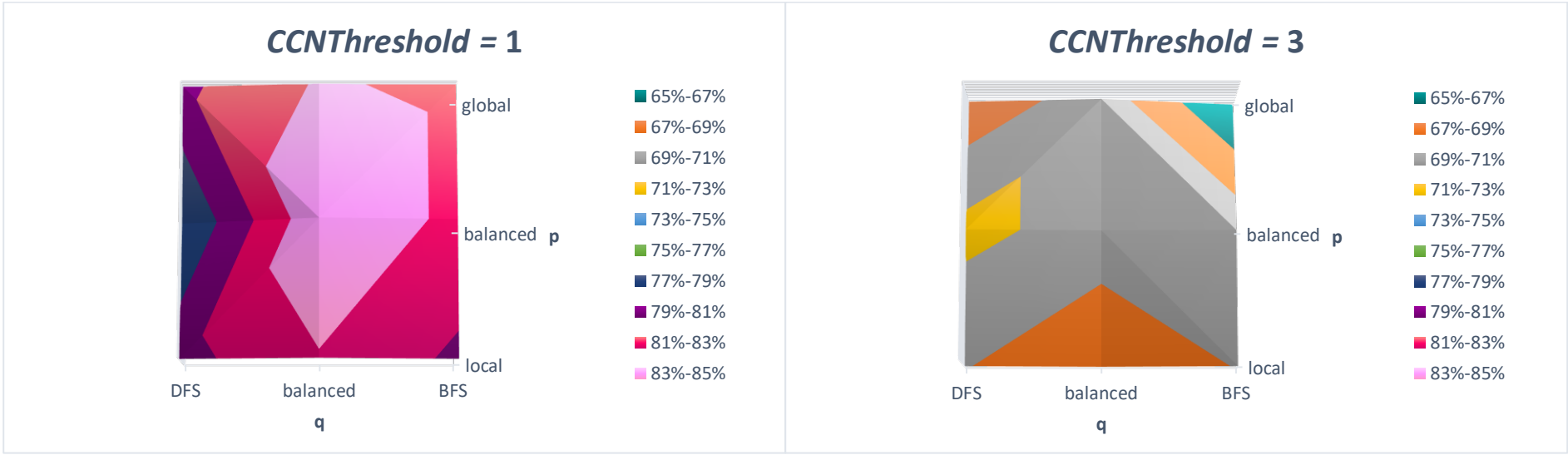


Figure 23 3D graphs built on recall values for embedding-based Naïve Bayes models (common legend)

Across three leaders in terms of performance whose avg. recall > 70%: Naive Bayes, Logistic Regression, and Neural Network

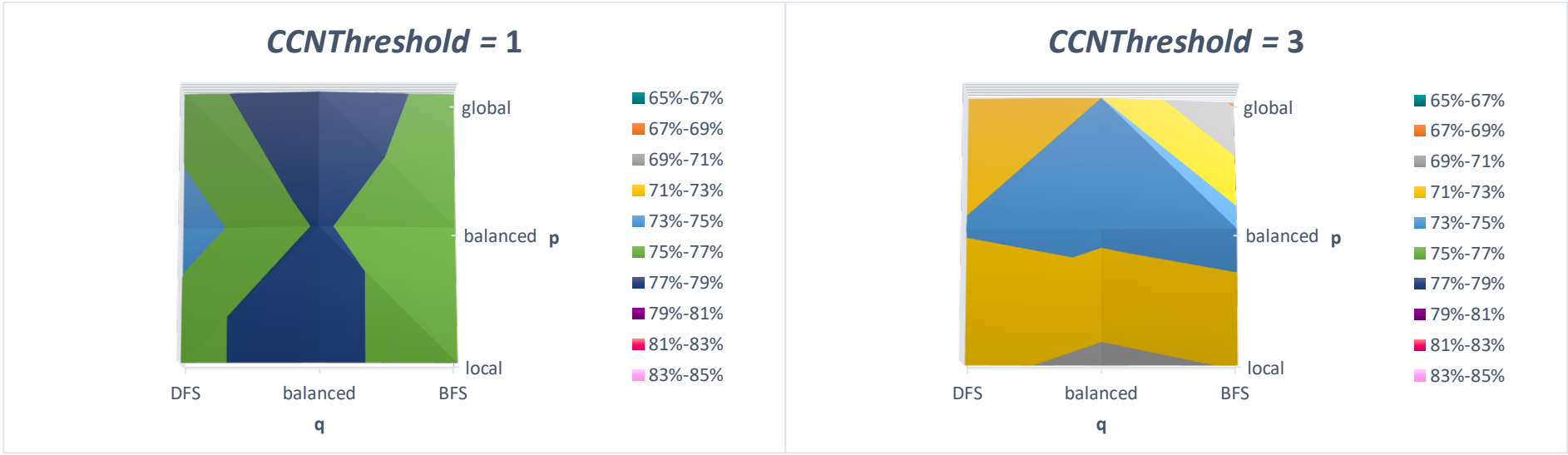


Figure 24 3D graphs built on average recall values for embedding-based Naive Bayes, Logistic Regression, and Neural Network models (common legend)

6.2.7 Impact of CCN Threshold

Table 25 shows how the recall values have changed on average across ML algorithms in the transition from the dataset based on *CCNThreshold* = 1 to the one based on *CCNThreshold* = 3. For most of machine learning algorithms the difference is insignificant. The greatest variation is observed for Naïve Bayes (12.7%), followed by AdaBoost (7.1%) and Tree (4.7%). Meanwhile, if the performance of Naïve Bayes deteriorates when not all logical dependencies are considered, the performance of AdaBoost and Tree improves with the exclusion of the supposed "noise." However, both AdaBoost and Tree only achieved recall of approximately 60-65%, which is not a big improvement over random allocation of samples into bug-prone or non-bug-prone (benign) categories which would give us a recall of approx. 50%.

Table 25 The impact of CCN Threshold on the performance (recall) of embedding-based models

Machine learning algorithm	**Average recall at *CCNThreshold* = 1**	**Average recall at *CCNThreshold* = 3**	**Change in the recall value for the dataset based on *CCNThreshold* = 3 compared to the one with *CCNThreshold* = 1**
Naïve Bayes	81.6%	68.9%	-12.7%
AdaBoost	60.0%	67.1%	7.1%
Decision Tree	60.6%	65.3%	4.7%
Logistic Regression	74.6%	71.3%	-3.3%
Random Forest	65.1%	68.4%	3.3%
CN2 Rule Induction	57.4%	59.0%	1.6%
Neural Network	73.2%	75.6%	2.4%

Overall, as can be seen from 3D graphs with a common legend, the machine learning models performed better when all evolutionary information had been inspected, including occasional co-changes between the files.

6.2.8 Execution time

The embedding-based models took more time to train than heuristic-based ones. Testing on modified input data (`emb_mlinput_th1.csv` or `emb_mlinput_th3.csv`, depending on the co-change threshold, see Appendix E.2) lasted approximately the same compared to the heuristic-based models. Table 26 displays the average execution times across algorithms. CN2 Rule Induction has again been the slowest method. Neural Network and it were the only algorithms which took more than a second to train.

Table 26 Average execution times of different embedding-based models

	CCNThreshold = 1		*CCNThreshold* = 3	
ML algorithm	**Train**	**Test**	**Train**	**Test**
Decision Tree	00s : 66ms	00s : 29ms	00s : 59ms	00s : 19ms
CN2 Rule Induction	57s : 82ms	00s : 24ms	39s : 54ms	00s : 30ms
Random Forest	00s : 30ms	00s : 24ms	00s : 29ms	00s : 15ms
Naïve Bayes	00s : 19ms	00s : 13ms	00s : 19ms	00s : 12ms
AdaBoost	00s : 36ms	00s : 22ms	00s : 28ms	00s : 25ms
Logistic Regression	00s : 30ms	00s : 25ms	00s : 20ms	00s : 17ms
Neural Network	02s : 20ms	00s : 33ms	01s : 88ms	00s : 29ms

6.2.9 Summary

As follows from our analysis, the best performance has been achieved when neither structural equivalence ($q = BFS$), nor homophily ($q = DFS$) was prioritized, but instead a balanced network exploration strategy ($q = balanced$) had been adopted (RQ3). It is similar to the concept of a DRSpace (see Figure 18): on the one hand, DRSpaces represent local neighborhoods of each node regarded as a design rule (structural equivalence), and on the other hand, files with elevated bug-proneness due to underlying design flaws tend to be architecturally connected in a small number of DRSpaces [11, 18, 24, 27, 28, 30, 42, 44] (homophily), so not only immediate neighbors should be taken into consideration. Therefore, we need a somewhat balanced random walk, which steps away from the starting node, but does not go too far. There is a chance, though, that the

superiority in performance for models with $q = balanced$ could be induced by the homophilic and structural equivalences in the Tiki dependency network [21], so more research is required to confirm our findings.

In addition, our analysis indicates that even if two files only occasionally change together, their relationship still should be considered in the task of architectural hotspot detection. Thus, the answer to the research question about the composition of the logical dependency (sub)graph – RQ4 – is negative.

6.3 Limitations

The results of the analysis of the embedding-based models, although improved on some limitations specified in Part 1, still may have been impacted by several factors.

First, apart from p and q, guiding the network exploration strategy, other node2vec parameters – number of dimensions, number of walks per source, length of walk per source, context (neighborhood) size, and number of epochs in stochastic gradient descent [21] – were fixed at default. Although determining the appropriate values of these fixed parameters may be difficult [46], fine-tuning them may seriously affect machine learning model performance, and, accordingly, the conclusions made in this research.

Second, in contrast to our previous method, in the embedding-driven analysis we automate feature extraction by casting it as a representation learning problem [21]. Doing so not only simplifies the prediction process from a practical point of view, but, as suggested by recent studies [46], may improve classification performance. Unfortunately, we are unable to directly compare the performance of the two types of models since in

Part 1 the data was not undersampled. However, it would be interesting to verify this suggestion as applied to the task of hotspot detection.

Third, despite we studied a single package of a software system and thus the number of data points was relatively small (3001 files for *CCNThreshold* = 1 and 1924 files for *CCNThreshold* = 3), the training time of embedding-based models was higher than the time of heuristic-based models, especially for some machine learning algorithms. Although the computer we used was not really powerful (2.50GHz processor with 4GB of RAM and Windows 10), node2vec algorithm may pose some challenges in terms of scalability to large systems with many files and dependencies. It appears that, once trained, a node2vec model does not require much time to predict categories; at least, the testing time is comparable to heuristic-based models. However, regularly retraining the model in full on an updated labelled data may be time-consuming, which may not be suitable to large systems. The number of dimensions d (in our case, we used a default value $d = 128$) determines the number of components in the produced vector embeddings [21] and thus the number of features in the resulting dataset for a machine learning model. Perhaps adjusting (decreasing) this number would help to achieve a good balance. The machine learning algorithm selected for the task also seems to play an important role in terms of execution time and thus should be selected carefully. Naïve Bayes, yielding the best performance in terms of accuracy, is also the fastest of the methods.

Fourth, we looked at one snapshot of the Tiki project to train and test our models (both heuristic- and embedding-based ones). It would be valuable to confirm our findings at different time slices, e.g. evaluating how well the models would perform on the actual prediction of the impact of "future" changes based on the "past" experience.

Fifth, we only considered the main branch of the Tiki project in the data collection and excluded commits labeled as merges on the grounds that they provide little history of each file, just outcomes of development activities. Therefore, some important evolutionary information could have been lost. If instead we took into account multiple branches, it would significantly expand the set of commits in the analysis and we would have a more complete view of the history of each file [27].

Finally, as has been mentioned earlier, the work presented in the current thesis is only a case study, so the drawn conclusions need to be reinforced/verified by replicating our research on a larger scale.

7. Discussion

7.1 Implications for software architecture analysis and research

In this thesis we have reported a case study of the Tiki open source project, in which we have examined the hotspot pattern approach to pinpointing software design flaws, associated with bugs, through the lens of machine learning. We have employed Decision Tree and CN2 Rule Induction classifiers for detecting patterns of architectural integrity violations and detailed the process of interpreting the generated outputs. We have related the results to the suite of hotspots, previously proposed in the literature [18, 27, 28, 30], and have confirmed that machine learning is capable of reproducing known patterns, and even of disclosing novel ones – we have been able to replicate the heuristics discoverable with the features we engineered and have also detected a clearly discernible pattern of Reverse Unstable Interface (or Unstable Coupled Client), which is based on the pair {high Dependees, high DependeesCCN} and has not been described before. By and large, our research demonstrates that machine learning can be used to discover patterns of architectural integrity violations in a semi-automatic manner and thus can serve as an auxiliary means of advancing the theory on architectural flaws or architectural anti-patterns encountered in software and associated with bugs.

We have observed that the number of times a file was changed together with other files of a system is the most precise indicator of its bug-proneness. There was no full concordance between different machine learning models with regard to ranking of identified patterns, which would let us make uniform statistical generalizations. However, overall the results converge and comply with the findings presented in the previous studies [18, 27, 28, 30]: Modularity Violation Groups generally has been the most

widespread hotspot, covering the majority of bug-prone cases in the library package of the Tiki project, Unstable Interface, although not always detectable and by and large pretty rare (even when detected, it has comprised no more than 13 bug-prone instances only), has shown the highest classification accuracy, which makes it the most severe hotspot with regard to the impact on file error- and change-proneness, so the files involved in it have a high probability of being bug-prone, while the Crossing typically has appeared as the second most severe hotspot pattern.

Comparing the performance indicators for different machine learning algorithms, we have determined that CN2 Rule Induction is the most effective for distinguishing between benign files and those which ever participated in bug-fixing commits, regardless of how many times. Meanwhile, when it comes to detecting files with severe problems (i.e. those with the medium or high level of bug-proneness), either Random Forest or Decision Tree should become the classifier of choice. Meanwhile, there has been no significant difference revealed between machine learning models based on feature sets with different granularity, so from the practical point of view the consolidated information on historical and structural relations (consolidated features) may equally well be used for pinpointing bug-prone files.

The analysis of the embedding-based models has shown that in order to identify design flaws associated with bugs we need to examine more than just immediate architectural relationships of a file, but to inspect a whole neighborhood of the files (nodes) structurally or evolutionary connected to it, at the same time not going too deep into the overall network (essentially, what in a DRSpace is achieved through clustering). We should adopt a balanced network exploration strategy which would prioritize neither

Breadth-first Sampling, nor Depth-first Sampling. Although we have established that one needs to consider a mixture of local and global dependencies, such that a sampled neighborhood conveys both structural and homophilic properties of the studied node, it is yet unclear what exactly this sampled neighborhood should look like. node2vec can be set to output random walks instead of embeddings[13], which may allow us to gain a more in-depth understanding of the neighborhood composition.

Lastly, our results suggest that, contrary to what is normally being done in the studies on software architecture analysis (not only those related to the hotspot pattern approach [27, 28, 30, 42], but in general [5, 16, 36]), even "weak" logical dependencies should be taken into account when composing a graph of architectural relations. Ignoring occasional co-changing between the files may lead to a loss of important information.

7.2 Implications for technology innovation management and technology entrepreneurship

The presented study is of value to various stakeholder groups. From a practical point of view, the results of this research can be incorporated into a tool, which would help software vendors or software development team managers to identify more hotspot patterns than before, ensuring better product quality and reducing maintenance costs [18, 22]. This work also opens up promising opportunities for technology entrepreneurs, who, for instance, could create plugins for software development environments or build quality control tools, which would assess the architectural integrity of the application once a new change is committed to the project's repository, providing real-time feedback and a

[13] See either Scala implementation of node2vec with Spark available at https://github.com/aditya-grover/node2vec (`node2vec_spark` directory) or a high performance implementation at https://github.com/snap-stanford/snap/tree/master/examples/node2vec.

timely warning if a violation has been detected. Such plugins could likewise benefit open source software project management if the said development environment is open source.

The results of architectural integrity analysis can be published by a team of open source developers to signal quality to product development managers selecting software libraries. A proprietary company could use them in a similar fashion to attract clients interested in purchasing specific software.

Finally, by pinpointing design flaws, software developers would be able to target issues before they propagate to other parts of the system, allowing for more efficient refactoring [11, 18, 19, 27], which overall would save time for making functional updates [9]. The hotspot pattern analysis would also provide information on how to fix the situation and correct underlying design flaws which have led to the problem being flagged [18]. These are only a few potential implications.

7.3 Practical applications to the Tiki project

Although the discussion with the Tiki developers has not occurred during the course of our research, we can speculate on practical applications of our work to the Tiki project. First and foremost, we pinpointed multiple files, whose elevated bug-proneness may be attributed to violations of software design principles. We believe that a list of these files would be useful for the team. Secondly, we have developed an algorithm, honed for the Tiki project, which we could hand over to the developers so they are able to evaluate the architectural integrity of their system themselves at any point of time in the future, either by manually following the process described in Appendices B to E, or, as discussed in the previous section, by creating a special tool for that. Thirdly, one of the points of interest for the Tiki team was the visualization of the architecture after a change,

which can also be accomplished now by running scripts from Appendix B and exporting the output file to the visualization software, e.g. Gephi[14] open source platform. We hope that the conversation with the Tiki development team will take place and we will receive direct feedback from the practitioners who directly work with the system that we have studied.

[14] https://gephi.org/.

8. Conclusion

To sum up, machine learning appears to have promising implications for scholarly research, especially in terms of pattern detection. The decision-making of some machine learning models is explicit, and the rules generated by them to guide the decisions can easily be derived, interpreted and related to the formalization of known rules. Thus, as shown by our work on discovering patterns of architectural integrity violations associated with bugs, they can complement the existing studies. In particular, they provide an opportunity to reveal human-readable patterns that have been previously unknown. We hypothesize that machine learning can also help identify new features (product or project characteristics) that have never been considered relevant or of high impact. We hope that the present thesis stirs further exploration into this approach, specifically as it applies to pinpointing software design flaws which may be the root causes of elevated bug- and change-proneness[15] of system elements (classes, files, or packages). Uncovering unforeseen aspects related to the localization of defects caused by architectural integrity violations would help us deal with the technical debt more efficiently – the longer a software defect persists, the most difficult it is to eradicate and the more technical debt recasts a technical concept as an economic one [9].

[15] Although it can be used for a whole range of different problems, not necessarily related to software at all.

***

Post Scriptum. Although beyond the scope of our research, we wanted to make an observation about a promising avenue node embeddings open up for future investigation and, especially, practice. In the study of financial news recommendation task [33], scholars quantified the strength of the correlation between the vertices in a network by calculating the cosine similarity between their representations and proposed a method for incremental updating of the embeddings. If we create a target subgraph consisting of two categories, "bug-prone" and "non-bug-prone," we can bring the node classification problem down to the link prediction problem and, thus, can potentially adapt the proposed method to the task of (architectural) bug detection, so we don't need to re-generate embeddings and re-train machine learning model every time there is a change – the developed system would be capable of incrementally updating (replacing) embeddings in the run-time at the moment of the commit. This should enable to achieve a good balance between time efficiency and "classification" accuracy even for large software projects with many dependencies.

References

1. Nemitari Ajienka and Andrea Capiluppi. 2017. Understanding the interplay between the logical and structural coupling of software classes. *The Journal of Systems and Software* 134 (December 2017), 120-137. DOI: http://doi.org/10.1016/j.jss.2017.08.042.

2. Carliss Y. Baldwin and Kim B. Clark. 2000. *Design Rules, Vol. 1: The Power of Modularity* (1st. ed.). MIT Press, Cambridge, MA.

3. Carliss Y. Baldwin, Alan MacCormack, and John Rusnak. 2014. Hidden structure: Using network methods to map system architecture. *Research Policy* 43, 8 (October 2014), 1381-1397. DOI: https://doi.org/10.1016/j.respol.2014.05.004.

4. Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. 1997. If your version control system could talk. In *Proceedings of the ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 5 pages.

5. Fabian Beck and Stephan Diehl. 2012. On the impact of software evolution on software clustering. *Empirical Software Engineering* 18, 5 (October 2013), 970-1004. DOI: https://doi.org/10.1007/s10664-012-9225-9.

6. Robert Benkoczi, Daya Gaur, Shahadat Hossain, and Muhammad A. Khan. 2018. A Design Structure Matrix approach for measuring co-change-modularity of software products. In *Proceedings of the ACM/IEEE 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, 331-335. DOI: https://doi.org/10.1145/3196398.3196409.

7. Dirk Beyer and Ahmed E. Hassan. 2006. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on*

Reverse Engineering (WCRE '06). IEEE Computer Society, Washington, DC, 199-210. DOI: https://doi.org/10.1109/WCRE.2006.14.

8. Dirk Beyer and Andreas Noack. 2005. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*. IEEE Computer Society, Washington, DC, 259-268. DOI: https://doi.org/10.1109/WPC.2005.12.

9. Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, New York, NY, 47-51. DOI: https://doi.org/10.1145/1882362.1882373.

10. Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. 2013. Leveraging design rules to improve software architecture recovery. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA '13)*. ACM, New York, NY, 133-142. DOI: https://doi.org/ 10.1145/2465478.2465480.

11. Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. 2019. Design Rule Spaces: A new model for representing and analyzing software architecture. *IEEE Transactions on Software Engineering* 45, 7 (July 2019), 657-682. DOI: https://doi.org/10.1109/TSE.2018.2797899.

12. Clarence Chio and David Freeman. 2018. *Machine Learning and Security: Protecting Systems with Data and Algorithms* (1 st ed.). O'Reilly Media, Inc., Sebastopol, CA.

13. Peter Clark and Robin Boswell. 1991. Rule induction with CN2: Some recent improvements. In *Machine Learning — EWSL-91. Proceedings of the 5th European Working Session on Learning*, Yves Kodratoff (Ed.). Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), Vol. 482. Springer, Berlin, Heidelberg, 151-163. DOI: https://doi.org/10.1007/BFb0017011.

14. Di Cui, Ting Liu, Yuanfang Cai, Qinghua Zheng, Qiong Feng, Wuxia Jin, Jiaqi Guo, and Yu Qu. 2019. Investigating the impact of multiple dependency structures on software defects. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 584-595. DOI: https://doi.org/10.1109/ICSE.2019.00069.

15. Marco D'Ambros, Michele Lanza, and Romain Robbes. 2009. On the relationship between change coupling and software defects. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE '09)*. IEEE Computer Society, Washington, DC, 135-144. DOI: https://doi.org/10.1109/WCRE.2009.19.

16. Marcos C. de Oliveira, Rodrigo Bonifacio, Guilherme N. Ramos, and Marcio Ribeiro. 2016. Unveiling and reasoning about co-change dependencies. In *Proceedings of the 15th International Conference on Modularity (MODULARITY '16)*. ACM, New York, NY, 25-36. DOI: https://doi.org/10.1145/2889443.2889450.

17. Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. 2018. Learning structural node embeddings via diffusion wavelets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. ACM, New York, NY, 1320-1329. DOI: https://doi.org/10.1145/3219819.3220025.

18. Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and Lu Xiao. 2016. Towards an architecture-centric approach to security analysis. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA '16)*. IEEE Computer Society, Washington, DC, 221-230. DOI: https://doi.org/10.1109/WICSA.2016.41.

19. Markus M. Geipel and Frank Schweitzer. 2012. The link between dependency and cochange: Empirical evidence. *IEEE Transactions on Software Engineering* 38, 6 (November-December 2012), 1432-1444. DOI: https://doi.org/10.1109/TSE.2011.91.

20. Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research* (1st. ed.). Aldine Publishing, Chicago, IL.

21. Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, 855-864. DOI: http://doi.org/10.1145/2939672.2939754.

22. Diego Guemes-Pena, Carlos Lopez-Nozal, Raul Marticorena-Sanchez, and Jesus Maudes-Raedo. 2018. Emerging topics in mining software repositories: Machine learning in software repositories and datasets. *Progress in Artificial Intelligence* 7 (2018), 237-247. DOI: https://doi.org/10.1007/s13748-018-0147-7.

23. Jen Tsu Hsu. 2018. *Investigating the Causes of Software Technical Debt at the Architectural Level*. Master's thesis. University of British Columbia, Vancouver, BC.

24. Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. 2015. A case study in locating the architectural roots of technical debt. In *Proceedings of the 37th International Conference*

on Software Engineering (ICSE '15). IEEE Computer Society, Washington, DC, 179-188. DOI: https://doi.org/10.1109/ICSE.2015.146.

25. Octavio Loyola-González. 2019. Black-box vs. white-box: Understanding their advantages and weaknesses from a practical point of view. *IEEE Access* 7 (October 2019), 154096-154113. DOI: http://doi.org/10.1109/ACCESS.2019.2949286.

26. Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidović, and Robert Kroeger. 2018. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering* 44, 2 (February 2018), 159-181. DOI: https://doi.org/10.1109/TSE.2017.2671865.

27. Ran Mo. 2018. *Automatically Measuring Software Architecture and Identifying Architecture Problems*. Doctoral Dissertation. Drexel University, Philadelphia, PA.

28. Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA '15)*. IEEE Computer Society, Washington, DC, 51-60. DOI: https://doi.org/10.1109/WICSA.2015.12.

29. Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2016. Decoupling level: A new metric for architectural maintenance complexity. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, 499-510. DOI: http://doi.org/10.1145/2884781.2884825.

30. Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2021. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE*

Transactions on Software Engineering 47, 5 (May 2021), 1008-1028. DOI: http://doi.org/10.1109/TSE.2019.2910856.

31. Steve Mutuvi. 2019. Introduction to Machine Learning Model Evaluation. (April 2019). Retrieved September 9, 2020 from https://heartbeat.fritz.ai/introduction-to-machine-learning-model-evaluation-fa859e1b2d7f.

32. Phu Chien Nguyen, Kouzou Ohara, Akira Mogi, Hiroshi Motoda, and Takashi Washio. 2006. Constructing decision trees for graph-structured data by chunkingless graph-based induction. In *Proceedings of the 10th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining (PAKDD '06)*. Springer-Verlag Berlin, Heidelberg, 390-399. DOI: https://doi.org/10.1007/11731139_45.

33. Jiangtao Ren, Jiawei Long, and Zhikang Xu. 2019. Financial news recommendation based on graph embeddings. *Decision Support Systems* 125 (October 2019), 113115. DOI: https://doi.org/10.1016/j.dss.2019.113115.

34. Frank Rosenblatt. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* (1st. ed.). Spartan Books, Washington, DC.

35. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (October 1986), 533–536. DOI: https://doi.org/10.1038/323533a0.

36. Luciana L. Silva, Marco T. Valente, and Marcelo de A. Maia. 2015. Co-change clusters: Extraction and application on assessing software modularity. In *Transactions on Aspect-Oriented Software Development XII*, Shigeru Chiba, Eric Tanter, Eric Ernst, and Robert Hirschfeld (Eds.). Lecture Notes in Computer Science, Vol. 8989. Springer, Berlin, Heidelberg, 96–131. DOI: https://doi.org/10.1007/978-3-662-46734-3_3.

37. Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, 5 pages. DOI: https://doi.org/10.1145/1082983.1083147.

38. Maximilian Steff and Barbara Russo. 2012. Co-evolution of logical couplings and commits for defect estimation. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*. IEEE Press, Piscataway, NJ, 213-216. DOI: https://doi.org/10.1109/MSR.2012.6224283.

39. Jørgen Tellnes. 2013. *Dependencies: No Software is an Island*. Master's thesis. University of Bergen, Bergen, the Netherlands.

40. Adam Tornhill. 2015. *Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs* (1st. ed.). The Pragmatic Programmers, LLC.

41. Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. 2009. Design Rule Hierarchies and parallelism in software development tasks. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, 197-208. DOI: https://doi.org/ 10.1109/ASE.2009.53.

42. Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Design Rule Spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, 967-977. DOI: https://doi.org/ 10.1145/2568225.2568241.

43. Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, 763-766. DOI: https://doi.org/10.1145/2635868.2661677.

44. Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. 2016. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, 488-498. DOI: http://doi.org/10.1145/2884781.2884822.

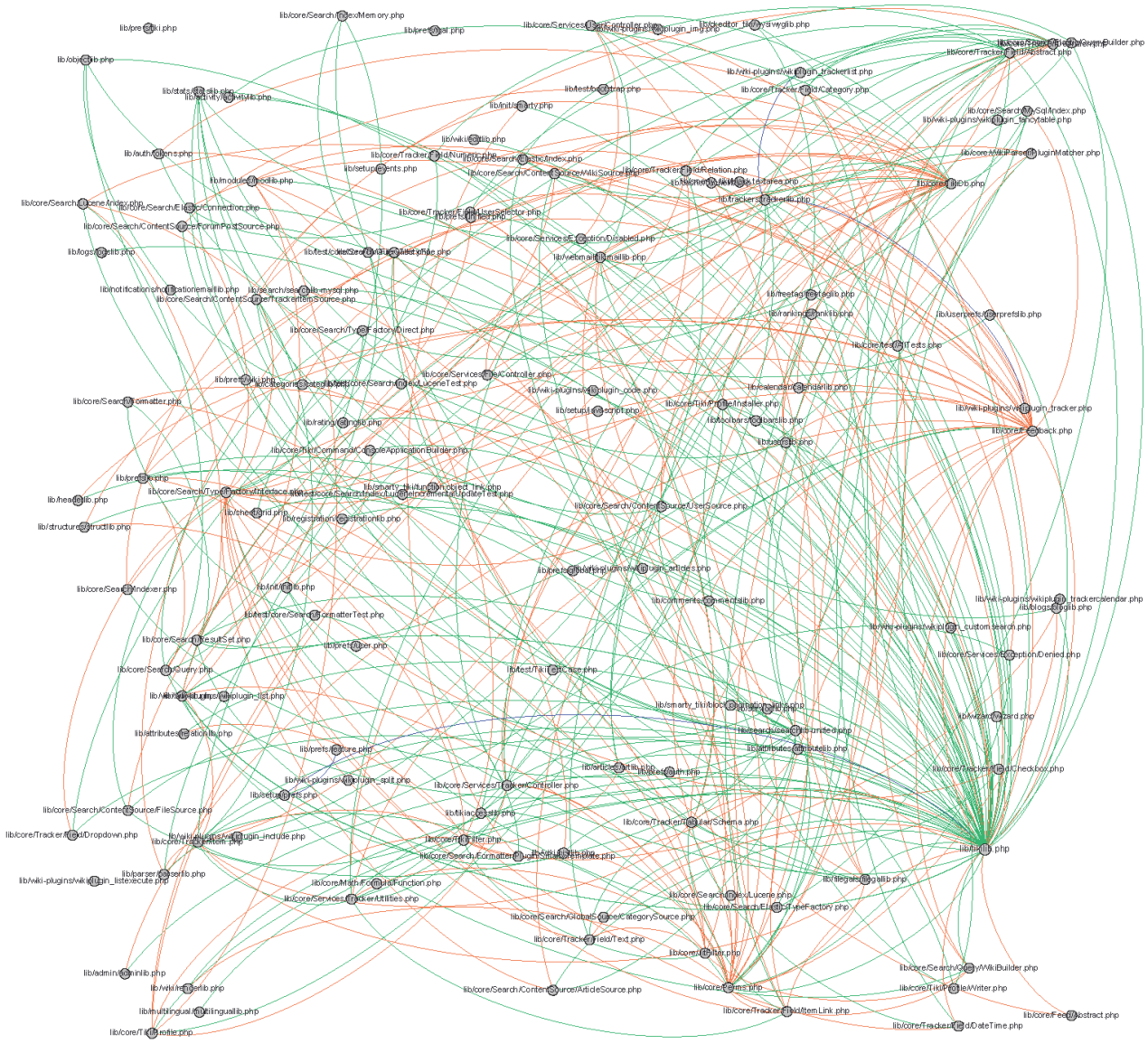
45. Alla Zakurdaeva, Michael Weiss, and Steven Muegge. 2020. Detecting architectural integrity violation patterns using machine learning. In *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20)*. ACM, New York, NY, 1480-1487. DOI: http://doi.org/10.1145/3341105.3374008.

46. Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. 2020. Network representation learning: A survey. *IEEE Transactions on Big Data* 6, 1 (March 2020), 3-28. DOI: https://doi.org/10.1109/TBDATA.2018.2850013.

47. Thomas Zimmermann, Peter Weiβgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (June 2005), 429-445. DOI: https://doi.org/10.1109/TSE.2005.72.

Appendices

Appendix A: Architectural relations between the source files in the library package of the Tiki project



Only structural dependency;
Only evolutionary coupling;
Both.

Appendix B: Architectural data extraction

B.1 Step-by-step algorithm

1) Clone Tiki Git repository into some directory. For the sake of example, let's assume that the root directory of the cloned repository on our machine is `C:/Tiki-rep/tiki`. Also for the sake of example, let's assume that all scripts from Appendix B.2 (and subsequently, Appendices C.2, D.2, and E.2) are located in the HOME (~) folder.

2) In Git Bash, go to the project directory and set the `diff.renameLimit`:

```
cd C:/Tiki-rep/tiki
git config diff.renameLimit 999999
```

3) Return to the HOME folder and run `git_commits_extraction.sh` to retrieve commits from the Tiki Git:

```
Note: If you copy files to another folder, please, open git_commits_extraction.sh in Notepad and
change ~/dataforprocessing.log (lines 109, 111) to path-to-your-folder/dataforprocessing.log.
```

```
cd ~
./git_commits_extraction.sh
```

Note: If you want to recreate our data, it doesn't matter what you respond to the question about whether to include commits with 1 file only. We had used the output file `dataforprocessing.log` solely for the next step (co-change count), although one may use it as an input to the bug metric extraction script as well (see Appendix C).

4) Employ `cochangecount.py` to derive co-changes for the necessary period of time (e.g., in our research - from October 7th, 2002, the outset of the project, to March 3rd, 2019, 03:09:25 UTC) and only between the code files located in the library (`lib` directory):

```
python cochangecount.py
```

Save the output as `cochanges19.csv`.

5) Recover structural dependencies through `dephpend`:

> Note: You may need to use Git to revert the repository to the moment corresponding to the end of the co-change extraction period.

```
dephpend text C:/Tiki-rep/tiki/lib > ~/tiki-lib.txt
```

This produces a file `tiki-lib.txt` with dependencies indicated by a `-->` symbol.

6) Convert .txt output of `dephpend` to .csv with comma as a delimiter via `txtdephpend2csv.pl`:

```
./txtdephpend2csv.pl tiki-lib
```

7) Since `dephpend` extracts structural dependencies between PHP classes, and `cochangecount.py` outputs co-changes between files, in order to merge the derived information, we need to map source code classes to files via `class.py`:

```
python class.py C:/Tiki-rep/tiki
```

The script generates `mapping.csv` with the class names and the corresponding .php file paths.

8) Run `visualization.py`, which merges the data from 3 .csv files by mapping classes from `tiki-lib.csv` to files through `mapping.csv` and combining them with `cochanges19.csv`:

`python visualization.py`

The output is provided in `merged.csv` in the following format:

```
Source,Target,CCN,Status,Weight
, where

Source and Target are two files, and Source structurally depends on Target /for Status = 1 or 3/
if the files are only evolutionary coupled /Status = 2/, the direction doesn't matter;

CCN is an abbreviation for "co-change number," i.e. a number of co-changes;

Status = 1 - denotes a structural dependency
Status = 2 - denotes an evolutionary coupling
Status = 3 - denotes both a structural dependency and an evolutionary coupling;

Weight = DefaultWeightForDependency + CCN /for Status = 1 or 3/
or
Weight = CCN /for Status = 2/,
where DefaultWeightForDependency = max(CCN)/2.
```

While `merged.csv` is ready to be imported into any preferred visualization tool (we had used Gephi to produce Figure 2, full version of which is presented in Appendix A), there are two more output files. The first one, `notfound.txt`, contains classes that haven't been mapped to files. In case `visualization.py` omits too many class/file pairs, it may be used as an input to `search.py` which looks for possible class locations (saved then in `locations.csv`) and is an enhancement tool for manual inspection and prospective improvement of the `class.py` script:

```
python search.py C:/Tiki-rep/tiki
```

The second output file, `visualization.log`, contains the full log describing the mapping and clean-up process (e.g., apart from classes copied to `notfound.txt`, it also provides a list of classes which haven't been matched with files because their name is not unique and the attempt to determine the correct file from the list of candidates based on a file path was unsuccessful).

B.2 Source code

git_commits_extraction.sh

```
1   #!/bin/sh
2   #
3   #prompt:
4   #use --since=<date> and --after=<date> options of git log to additionally narrow search if needed
5   #
6   #SETTINGS
7   #input variables
8   MaxNFilesInCommit=20    #commits with more than this number of files will be left out as automated
minor changes
9   #computational variables
10  Iter=0
11  NCommitsCopied=0
12  #move to the directory with the cloned Tiki Git repository
13  read -p "Provide the local path to the cloned Tiki repository: " GitPath
14  cd "$GitPath"
15  #ask how far in the history the user wants to go (all history or X number of latest records; how many?)
16  while True; do
17      read -p "Do you want to extract all history (\"1\") or a finite number of the most recent commits
(\"2\")?" onetwo
18      case $onetwo in
```

```
19          [1]* ) ExtractFiniteNumber=False; NCommitsToExtract=0; break;;
20          [2]* ) ExtractFiniteNumber=True;
21              while True; do
22                  read -p "How many?" NCommitsToExtract
23                  if [[ $NCommitsToExtract =~ ^[0-9]+$ ]]; then
24                    break
25                  else
26                    echo "Please enter a positive integer."
27                  fi
28              done
29              break;;
30          * ) echo "Please answer 1 or 2.";;
31      esac
32  done
33  #Cai et al. mention that there is no need to extract commits with one file only, because there are no
co-changes in them
34  #however, it may be useful for some other metrics (e.g., bug frequency). we will make it optional
35  while True; do
36      read -p "Include commits with 1 file only (y/n)?" yn
37      case $yn in
38          [Yy]* ) DontIncludeOneFileCommits=False; break;;
39          [Nn]* ) DontIncludeOneFileCommits=True; break;;
40          * ) echo "Please answer yes or no.";;
41      esac
42  done
43  #DATA EXTRACTION
44  #copy major statistics on commits, files and changes into a separate file
45  #we will sift through Git records one by one
46  CommitHash=$(git log -1 --skip="$Iter" --pretty=format:%H)
47  while [[ "$CommitHash" != "" ]]
48  do
49      if $ExtractFiniteNumber && [ $NCommitsCopied -eq $NCommitsToExtract ]; then
50          echo "Have reached the limit on the number of commits: $NCommitsCopied/$NCommitsToExtract."
51          break
52      fi
53      #we skip unrelated commits with [BRANCH], [DOC], [LANG], [MRG], [OOPS], [TEST], [TRA] labels
54      IncludeCommit=0
55      labelsarray=() #all labels in a particular commit
```

```
56      CommitTitle=$(git log -1 --skip="$Iter" --pretty=format:%s)
57      if [[ "$CommitTitle" =~ '[' ]]; then                    #if there is no label - we include it
58          #let's first derive a list of labels from the commit title
59          IFS='[' read -r -a difflabels <<< "$CommitTitle"
60          for label in "${difflabels[@]}"
61          do
62              #skip empty elements of $difflabels
63              #(if the line starts with '[', after separating it by '[' into array (line 59) the first
element will be empty)
64              #if there is no closing bracket, then we consider it as a commit with no label and include
it in the file
65              if [[ X"" != X"$label" ]] && [[ $label =~ ']' ]]; then
66                  label=${label%]*}
67                  #ensure that the label contains at least one letter
68                  if [[ "$label"  =~ ^[a-zA-Z0-9\ /]+$ ]] && [[ ! "$label"  =~ ^[0-9]+$ ]]; then
69                      #make the label UPPERCASE
70                      label=${label^^}
71                      #cleanup specific to Tiki Git labels:
72                      #1- match different labels meaning the same thing
73                      if [[ $label == MERGE* ]]; then
74                          label="MRG"
75                      fi
76                      #2- exclude meaningless words (which are not labels) put by committers in [ ]
brackets
77                      if [[ $label != *AND* ]] && [[ $label != *FOO* ]] && [[ $label != *FP* ]] && [[
$label != *FROM* ]] && [[ $label != *REVISION* ]] && [[ $label != *THANKS* ]]; then
78                          labelsarray+=( $label )
79                      fi
80                  fi
81              fi
82          done
83          #if there is at least one label which is not in {[BRANCH], [DOC], [LANG], [MRG], [OOPS],
[TEST], [TRA]}, then we include the commit
84          #however, the algorithm is not ideal because we don't exclude noise completely in here - as
mentioned in line 76, sometimes developers put general information in square brackets (as opposed to
labels). these comments may include words beyond the ones specified above ("and", "foo", "fp", "from",
"revision", "thanks")
85          if [ ${#labelsarray[@]} -ne 0 ]; then
```

```
86              for commitlabel in "${labelsarray[@]}"
87              do
88                  if [[ $commitlabel != BRANCH ]] && [[ $commitlabel != DOC ]] && [[ $commitlabel != LANG
]] && [[ $commitlabel != MRG ]] && [[ $commitlabel != OOPS ]] && [[ $commitlabel != TEST ]] && [[
$commitlabel != TRA ]]; then
89                      IncludeCommit=1
90                      break
91                  fi
92              done
93              if [ $IncludeCommit -eq 0 ]; then
94                  echo "SKIPPED. Commit: $CommitHash $CommitTitle. Does not contain necessary labels.
Commit is skipped."
95                  Iter=$((Iter+1))
96                  CommitHash=$(git log -1 --skip="$Iter" --pretty=format:%H)
97                  continue
98              fi
99          fi
100     fi
101     #if chosen so, don't copy commits with just 1 file inside
102     NFilesInCommit=$(git log -1 --skip="$Iter" --shortstat | tail -1 | awk '{print $1;}')
103     if  $DontIncludeOneFileCommits && [ $NFilesInCommit -eq 1 ]; then
104         echo "SKIPPED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit. Commit is
skipped."
105     else
106         #we do not copy commits with more than MaxNFilesInCommit files inside, as we assume they
contain automated minor changes
107         if [ $NFilesInCommit -le $MaxNFilesInCommit ]; then
108             if [ $NCommitsCopied -eq 0 ]; then
109                 git log -1 --skip=$Iter --numstat > ~/dataforprocessing.log
110             else
111                 git log -1 --skip=$Iter --numstat >> ~/dataforprocessing.log
112             fi
113             NCommitsCopied=$((NCommitsCopied+1))
114             echo "COPIED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit."
115         else
116             echo "SKIPPED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit. Commit is
skipped."
117         fi
```

```
118     fi
119     Iter=$((Iter+1))
120     CommitHash=$(git log -1 --skip="$Iter" --pretty=format:%H)
121 done
```

cochangecount.py

```
1   #script doesn't distinguish moved or renamed files, such as tiki/doc/{tiki09.doc => tiki.doc}, from the
rest
2   import re
3   import pandas as pd
4   import datetime
5   #auxiliary functions
6   def getadate(quest_num, question):
7       isvalid=False
8       while not isvalid:
9           answer = input("Timestamp " + str(quest_num) + ": " + question)
10          if (quest_num == 1) and (answer == "0"):
11              d = answer
12              isvalid=True
13          elif (quest_num == 2) and (answer == "99"):
14              d = answer
15              isvalid=True
16          else:
17              try: #strptime() throws an exception if the input doesn't match the pattern
18                  d = datetime.datetime.strptime(answer, "%Y/%m/%d")
19                  isvalid=True
20              except:
21                  print("Error, check your input!\n")
22      return d
23
24  def getadir():
25      answer = input("Subdirectory name: ")
26      if not answer.endswith('/'):
27          answer = answer + "/"
28      return answer
29
```

```
30  def yes_or_no(question):
31      answer = input(question).lower().strip()
32      print(" ")
33      while not(answer == "y" or answer == "yes" or answer == "n" or answer == "no"):
34          print("Input y/n or yes/no")
35          answer = input(question).lower().strip()
36          print(" ")
37      if answer[0] == "y":
38          return True
39      else:
40          return False
41
42  def is_code_file(filename):
43      if (filename[-4:] == ".php") or (filename[-4:] == ".tpl") or (filename[-4:] == ".sql"):
44          return True
45      else:
46          return False
47
48  #main part
49  since = "0"
50  till = "99"
51  subdirectory = ""
52  copycommit = True
53  #script settings/user input
54  if yes_or_no("Do you want to specify time interval for co-change count (otherwise the script will
process the entire repository)? "):
55      since = getadate(1, "Since [date]. Type the date in format 'yyyy/mm/dd', or type 0 to indicate the
earliest commit found (e.g., if the input file contains all commit history,  \"0\" will mean \"since the
beginning of the project\"): ")
56      till = getadate(2, "Till [date]. Type the date in format 'yyyy/mm/dd', or type 99 to indicate the
latest commit found (e.g., if the input file contains all commit history,  \"99\" will mean \"till the end
of the project\"): ")
57  if yes_or_no("Output co-changes between files located in a specific subdirectory only?  "):
58      subdirectory = getadir()
59  onlycodefiles = yes_or_no("Output co-changes between code files only (.php, .tpl, .sql)? ")
60  #pattern representing the beginning of a new commit
61  newcommit = re.compile(r'^commit (.+)')
62  #pattern representing the line in a commit log, which contains information about the date
```

```
63  dateline = re.compile(r'^Date:   (.+)')
64  #pattern representing the line in a commit log, which contains information about files: 2 columns for
numbers of added and deleted lines + 1 column for file name
65  fileline = re.compile(r'^(\d+)\s+(\d+)\s+(.+)')
66  #we will aggregate all pairs of files within each separate commit and combine them in a single
dataframe (one dataframe for all commits)
67  filesincommit=[]    #list of files in one particular commit
68  rows_list = []      #rows of the future dataframe
69  with open('dataforprocessing.log', encoding="utf8") as extractedcommits:
70      for line in extractedcommits:
71          if newcommit.search(line):
72              if filesincommit!=[]:
73                  for filei in filesincommit:
74                      for filej in filesincommit:
75                          if filei != filej:
76                              if onlycodefiles:
77                                  #if user chose to output only co-changes between the code files,
78                                  #check whether both files have proper extensions
79                                  if (is_code_file(filei) and is_code_file(filej)):
80                                      rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc
Number': 1})
81                              else:
82                                  rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc
Number': 1})
83              filesincommit=[]
84          #if the user has specified a time interval, then only copy commits within the timespan
85          elif ((since != "0") or (till != "99")) and dateline.search(line):
86              commitdate = datetime.datetime.strptime(line[8:-7], "%a %b %d %H:%M:%S %Y")
87              if (since == "0") and (till != "99"): #since the beginning of the log till some timestamp
88                  #commit should be copied if commitdate <= till:
89                  copycommit = (commitdate - datetime.timedelta(seconds=1) < till) #python doesn't
support <=, so worked a way around
90              elif (since != "0") and (till == "99"): #since some timestamp till the end of the log
91                  #the log is processed from the most recent record to the most old
92                  #copy while commitdate => since, but break as soon as the interval is passed:
93                  if (commitdate < since):
94                      break;
95              else:
```

```
96                  #commit should be copied if since <= commitdate <= till,
97                  #but as soon as the interval has passed, break the loop:
98                  if (commitdate < since):
99                      break;
100                 copycommit = (commitdate - datetime.timedelta(seconds=1) < till)
101         elif copycommit and fileline.search(line):
102             #derive file name from the commit log
103             if subdirectory != "":
104                 #if the user chose to process only a certain subdirectory,
105                 #check the beginning of the filename
106                 filesname = (re.sub(fileline, r'\3', line)).rstrip('\n')
107                 if filesname.startswith(subdirectory):
108                     filesincommit.append(filesname)
109             else:
110                 filesincommit.append((re.sub(fileline, r'\3', line)).rstrip('\n'))
111 #process the last commit (for the case since="0", because there is no commit after, see lines 70-71)
112 if filesincommit!=[]:
113     for filei in filesincommit:
114         for filej in filesincommit:
115             if filei != filej:
116                 if onlycodefiles:
117                     if (is_code_file(filei) and is_code_file(filej)):
118                         rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number':
1})
119                 else:
120                     rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number': 1})
121 cochanges = pd.DataFrame(rows_list)
122 #dataframe has a format:
123 #Primary File    Secondary File    Cc Number (i.e., co-change number)
124 #file1           file2             1
125 #file1           file2             1
126 #file1           file3             1
127 #file1           file4             1
128 #we just have to group the dataframe by 1st and 2nd columns
129 if cochanges.empty:
130     print('No files are found based on input parameters.') #to prevent falling in error when grouping
an empty dataframe
131 else:
```

```
132     cochanges = cochanges.groupby(['Primary File','Secondary File'])['Cc Number'].sum()
133     #copy dataframe to the separate file
134     with open("cochanges.csv",'w') as outfile:
135         cochanges.to_csv(outfile, header=False, line_terminator='\n')
136     print("See cochanges.csv for further details")
```

txtdephpend2csv.pl

```
1   #!/usr/bin/perl
2   #
3   #txtdephpend2csv.pl
4
5   open(INFILE, $ARGV[0] . ".txt");
6   open(OUTFILE, ">" . $ARGV[0] . ".csv");
7
8   #provide a choice of what to use as a delimiter - "," or ";"
9   while (1) {
10      print "What to use as a delimiter: ',' or ';'?";
11      chomp(my $input = <STDIN>);
12      if ($input eq "," or $input eq ";") {
13          while (<INFILE>) {
14              if (/(.+) --> (.+)/) {
15                  print OUTFILE "$1$input$2\n";
16              }
17          }
18          last;
19      } else {
20          print "Please, type ',' or ';'.\n";
21      }
22  }
23
24  close(INFILE);
25  close(OUTFILE);
```

class.py

```
1   #usage: class.py <directory>
```

```
2   #output: all pairs of class names and corresponding file paths
3   import sys
4   import os
5   import re
6   import pandas as pd
7
8   if len(sys.argv) < 2:
9       print("You must set an argument - a path to the project directory!")
10      quit()
11  rootdir = sys.argv[1]
12  if not rootdir.endswith('/'):
13      rootdir = rootdir + "/"
14  rootdirlength = len(rootdir)
15  #pattern representing a class
16  isclass = re.compile(r'^class\s+(\w+)')
17  #pattern representing an abstract class
18  isabstractclass = re.compile(r'^abstract\s+class\s+(\w+)')
19  #pattern representing an interface
20  isinterface = re.compile(r'^interface\s+(\w+)')
21  rows_list = []      #rows of the future dataframe
22  for root, dirs, files in os.walk(rootdir):
23      for file in files:
24          if file.endswith(".php"):
25              fullpath = os.path.join(root, file).replace("\\","/")
26              with open(fullpath, encoding="utf8") as sourcefile:
27                  for line in sourcefile:
28                      if isclass.search(line):
29                          classname = (re.sub(isclass, r'\1', line)).rstrip().split(' ')[0]
30                          rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
31                      elif isabstractclass.search(line):
32                          classname = (re.sub(isabstractclass, r'\1', line)).rstrip().split(' ')[0]
33                          rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
34                      elif isinterface.search(line):
35                          classname = (re.sub(isinterface, r'\1', line)).rstrip().split(' ')[0]
36                          rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
37  #create a dataframe with classes to files mapping
38  mapping = pd.DataFrame(rows_list)
39  #sort the dataframe by columns Class and File
```

```
40  #create a temporary column Class.Upper for sorting to be case-insensitive
41  mapping["Class.Upper"] = mapping["Class"].str.upper()
42  mapping.sort_values(by=['Class.Upper', 'File'], inplace=True)
43  del mapping["Class.Upper"]
44  #copy dataframe to the separate file
45  with open("mapping.csv",'w') as outfile:
46      mapping.to_csv(outfile, header=False, index=False, line_terminator='\n')
47  print("See mapping.csv for further details.")
```

visualization.py

```
1   import pandas as pd
2   #import .csv files produced by cochangecount.py, txtdephpend2csv.pl, and class.py
3   cochanges = pd.read_csv('cochanges19.csv', names=['File1', 'File2', 'CCN']) #CCN - co-change number
4   dependencies = pd.read_csv('tiki-lib.csv', names=['Class1', 'Class2'])
5   mapping = pd.read_csv('mapping.csv', names=['Class', 'File'])
6   #firstly, we need to bring all data to the same format (to files)
7   notprocessed=[]         #list of classes which will not have been mapped to files
8   output1 = ""
9   output2 = ""
10  dependencies['Classname1'] = dependencies.apply(lambda row: row['Class1'][row['Class1'].rfind("\\") + 1
: ], axis=1)
11  dependencies['Classname2'] = dependencies.apply(lambda row: row['Class2'][row['Class2'].rfind("\\") + 1
: ], axis=1)
12  mapping['Class'] = mapping.Class.str.upper()
13  for rows in dependencies.itertuples():
14      for i in 1, 2:
15          classname = getattr(rows, "Classname" + str(i)).upper()
16          returned = mapping.loc[mapping['Class'] == classname]
17          if returned.empty:
18              notprocessed.append(getattr(rows, "Class" + str(i)))    #we will output the full name of a
class
19          elif len(returned.index) > 1:                               #elif classname is not unique
20              #try to find similarities between the full class name and the file path
21              fullclassname = getattr(rows, "Class" + str(i)).replace("\\","/").upper()
22              #settings specific to the Tiki project
23              if fullclassname[0:5] == "TIKI/":
```

```
                fullclassname = fullclassname[5:]
            if fullclassname[0:6] == "TESTS/":
                fullclassname = fullclassname[6:]
            if fullclassname[0:7] != "OPENPGP":
                fullclassname = fullclassname.replace("_", "/")
            for rrow in returned.itertuples():
                check = 0
                if fullclassname in getattr(rrow, "File").upper():
                    dependencies.at[rows.Index,'File' + str(i)] = returned.at[rrow.Index,'File']
                    check = 1
                    break;
            if check == 0:
                output1 = output1 + "can't find a file for a not-unique class " + fullclassname + "\n"
        else:
            dependencies.at[rows.Index,'File' + str(i)] = returned.at[list(returned.index)[0],'File']
notprocessed = list(set(notprocessed))
#copy nonmatched classes to notfound.txt in case we want to use search.py to look for their possible
locations
with open("notfound.txt",'w') as notfound:
    print(", ".join(sorted(notprocessed)), file=notfound)
print("See notfound.txt for classes that haven't been matched to files.")
#prepare everything for merging
dependfiles = dependencies[['File1', 'File2']].copy()
#as files can contain several classes, there may be cases in the dataframe when:
#File1        File2
#aaa.php      aaa.php    , i.e., a file "depends" on itself
#foremost, we drop such rows
dependfiles.drop(dependfiles.loc[dependfiles['File1'] == dependfiles['File2']].index, inplace=True)
#then we drop duplicates and rows with NaN values
original = len(dependfiles)
dependfiles.drop_duplicates(inplace=True)
dropped = len(dependfiles)
output2 = output2 + "During post-processing, " + str(original - dropped) + " duplicated structurally
dependent file pairs were dropped, "
dependfiles.dropna(inplace=True)
output2 = output2 + str(dropped - len(dependfiles)) + " rows containing NaN values were dropped.\n"
#now, when classes are fully mapped to files and the structural dependency dataframe is free from
duplicates and NaN values, start merging
```

```
59  dependfiles['CCN'] = 0
60  #add column Status to dataframes
61  #Status = 1 - denotes a structural dependency
62  #Status = 2 - denotes an evolutionary coupling
63  #Status = 3 - denotes both a structural dependency and an evolutionary coupling
64  dependfiles['Status'] = 1
65  cochanges['Status'] = 2
66  mergeddf = dependfiles.append(cochanges, ignore_index=True,
sort=False).groupby(['File1','File2'])['CCN', 'Status'].sum().reset_index()
67  #as cochangecount.py creates reciprocal pairs of files for each processed commit
68  #(i.e., for example, for commit with just two files "aa1" and "bb2", it will create two rows in a .csv
file:
69  #File1      File2      CCN
70  #aa1        bb2        1
71  #bb2        aa1        1
72  #specifying that "aa1" has been changed with "bb2" one time, as well as "bb2" - with "aa1" one time),
73  #when grouping mergeddf, only one of these reciprocal pairs will receive Status=3 (depending on which
file depends on which):
74  #File1      File2      CCN           Status
75  #aa1        bb2        1             3
76  #bb2        aa1        1             2
77  #therefore, we have to drop the duplicate with Status=2
78  #P.S. if grouping the dataframe has led to:
79  #File1      File2      CCN           Status
80  #aa1        bb2        1             3
81  #bb2        aa1        1             3
82  #we should keep both rows with Status=3, since they correspond to one of the hotspot patterns
83  statusthree = mergeddf.loc[mergeddf['Status'] == 3].copy()
84  for trow in statusthree.itertuples():
85      rowfound = mergeddf.loc[(mergeddf['File1'] == getattr(trow, 'File2')) & (mergeddf['File2'] ==
getattr(trow, 'File1')) & (mergeddf['Status'] == 2)]
86      if rowfound.empty:
87          rowfound = mergeddf.loc[(mergeddf['File1'] == getattr(trow, 'File2')) & (mergeddf['File2'] ==
getattr(trow, 'File1')) & (mergeddf['Status'] == 3)]
88          if rowfound.empty: #check just in case: if there is no reciprocal row with Status 2, there
should be a reciprocal row with Status 3. if it's not the case, this may indicate a severe issue with the
data or the code
```

```
89              print("Error! Can't find a reciprocal pair for files ", getattr(trow, 'File1'), " and ",
getattr(trow, 'File2'), ", which have status = 3.")
90  #we had an idea to equate CCN to 0 in one of the rows for the case described above in P.S.,
91  #assuming that leaving everything as it was would cause doubled CCN for the edge.
92  #however, it turned out that this is not how Gephi works.
93  #so we opted for keeping the both rows with the same CCN and the same weight (the same weight is
especially important)
94  #the following code is commented out and kept here just in case (other visualization tools may act
differently)
95  #         elif getattr(trow, 'CCN') != 0:
96  #             mergeddf.at[rowfound.index,'CCN'] = 0
97  #             statusthree.at[rowfound.index,'CCN'] = 0
98      else:
99          mergeddf.drop(rowfound.index, inplace=True)
100 #we also have to delete reciprocal rows, corresponding to Status = 2, as we still have situations like
101 #File1      File2      CCN            Status
102 #cc1        dd2        1              2
103 #dd2        cc1        1              2
104 statustwo = mergeddf.loc[mergeddf['Status'] == 2].copy()
105 fullindex = statustwo.index
106 statustwo['Status2_check'] = statustwo.apply(lambda row: ''.join(sorted([row['File1'], row['File2']])),
axis=1)
107 statustwo.drop_duplicates(subset=['Status2_check'], inplace=True)
108 mergeddf.drop(list(set(fullindex)-set(statustwo.index)), inplace=True)
109 #calculate weight for structural and logical dependencies
110 print("Max CCN is ", mergeddf['CCN'].max(), ".")
111 depweight = int(mergeddf['CCN'].max()/2) #default weight for dependency subgraph
112 #although we had used the weight function specified above to quantify the impact of a structural
dependency,
113 #other functions may be developed to try to balance the importance of the two types of architectural
relationships
114 print("The default weight for structural dependency subgraph is ", depweight, ".")
115 mergeddf['Weight'] = mergeddf.apply(lambda row: row['CCN'] if row['Status'] == 2 else row['CCN'] +
depweight, axis=1)
116 #provide output
117 mergeddf.rename(columns={'File1':'Source', 'File2':'Target'}, inplace=True)
118 with open("merged.csv",'w') as outfile:
119     mergeddf.to_csv(outfile, index=False, line_terminator='\n')
```

```
120 print("See merged.csv for data visualization.")
121 with open("visualization.log",'w') as logfile:
122     print(output1, file=logfile)
123     print("\n\nCouldn't find a match for ", len(notprocessed), " classes: ", ",
".join(sorted(notprocessed)), ".\n", file=logfile)
124     print("Dependencies dataframe after mapping classes to files (no post-processing, contains all
duplicates and NaN values):\n", file=logfile)
125     dependencies.to_string(logfile)
126     print(output2, file=logfile)
127 print("See visualization.log for the full script output.")
```

search.py

```
1   #usage: search.py <directory>
2   #clean-up tool: searches for classes, which haven't been matched to files by class.py, in the project
directory
3   #output: a list of possible locations for manual inspection
4   import sys
5   import os
6   import pandas as pd
7
8   if len(sys.argv) < 2:
9       print("You must set an argument - a path to the project directory!")
10      quit()
11  rootdir = sys.argv[1]
12  if not rootdir.endswith('/'):
13      rootdir = rootdir + "/"
14  rootdirlength = len(rootdir)
15  with open('notfound.txt', 'r') as inputfile:
16      text=inputfile.read()
17  class_list = text.split(', ')
18  rows_list = []      #rows of the future dataframe
19  for root, dirs, files in os.walk(rootdir):
20      for file in files:
21          if file.endswith(".php") or file.endswith(".sql") or file.endswith(".tpl"):
22              fullpath = os.path.join(root, file).replace("\\","/")
23              with open(fullpath, encoding="utf8") as sourcefile:
```

```
                for line in sourcefile:
                    for classname in class_list:
                        if classname in line:
                            rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
#create a dataframe with classes to files mapping
locations = pd.DataFrame(rows_list)
#sort the dataframe by columns Class and File
#create a temporary column Class.Upper for sorting to be case-insensitive
locations["Class.Upper"] = locations["Class"].str.upper()
locations.sort_values(by=['Class.Upper', 'File'], inplace=True)
del locations["Class.Upper"]
locations.drop_duplicates(inplace=True)
#copy dataframe to the separate file
with open("locations.csv",'w') as outfile:
    locations.to_csv(outfile, header=False, index=False, line_terminator='\n')
print("See locations.csv for further details.")
```

Appendix C: Bug data extraction

C.1 Step-by-step algorithm

1) Extract the commit log and have it ready.

> Note: In general, you can use `dataforprocessing.log` produced by `git_commits_extraction.sh` from Appendix B. However, an additional step is required if you want to recreate our data. As can be noted from Table 4, the commit extraction script filters out 5,000 commits. To ensure we don't miss important information (e.g., there may happen to be a big-fixing commit with more than 20 files) we had retrieved a separate, complete log by means of Git Bash:
>
> ```
> cd C:/Tiki-rep/tiki
> git log --numstat > ~/tikidata.log
> ```

2) Execute `bugfrequency.pl`. If the extracted commit log has the full name `file-name.log`, the script should be run as `bugfrequency.pl file-name` (in case both the script and the input file lie in the same directory):

```
cd ~
./bugfrequency.pl tikidata
```

The tool outputs two files: `bugs.log` (containing the full statistics on bug-fixing commits) and `bugs.csv` (containing the subset of the data presented in `bugs.log`, specifically the list of files involved in bug-fixing commits with the associated frequencies of occurrence). Bug-fixing commits are determined as those labeled in Tiki Git with [FIX], [BUG], or [SEC] (the latter denotes security bug fixes). If you want to retrieve statistics on security bug fixing commits only, please modify the regular expression in line `66` of the `bugfrequency.pl` from `/\[(FIX|SEC|BUG)\]/` to `/\[SEC\]/`.

C.2 Source code

bugfrequency.pl

```
1   #!/usr/bin/perl
2   #
3   #bugfrequency.pl
4
5   my %commits;
6   my %authors;
7   my %emails;
8   my %dates;
9   my %labels;
10  my %files;
11  my %buggyfiles;
12  my %buggyfilesclean;
13  my %frequency;
14
15  my $n = 99999;                                  #set $n to a small number for testing
16  my $commitsnum = 0;
17  my $commit = "";
18  my $count = 0;
19  open(INFILE, $ARGV[0] . ".log");
20  while (<INFILE>) {
21      if (/^commit (.+)/) {
22          last unless ($n-- > 0);
23          $commit = $1;
24          push @commits, $commit;
25          $commitsnum++;
26      } elsif (/^Author: (.+?) <(.+)>/) {         #author names can include spaces
27          my $author = $1;
28          my $email = $2;
29          $authors{$commit} = $author;
30          $emails{$commit} = $email;
31      } elsif (/^Date:   (.+)/) {
32          my $date = $1;
33          $dates{$commit} = $date;
```

```
34          $count = 2;
35      } elsif ($count == 2) {
36          $count--;
37          $labels{$commit} = "";
38      } elsif ($count == 1) {
39          chomp($_);
40          $label = trim($_);
41          if ($label =~ /^\[[\w\/]+?\]/) {
42              if ($labels{$commit}) {
43                  $labels{$commit} .= "\n";
44              }
45              $labels{$commit} .= $label;
46          } elsif ($_ == "") {
47              $count--;
48          }
49      } elsif (/^(\d+)\s+(\d+)\s+(.+)/) {
50          my $file = $3;
51          if ($files{$commit}) {
52              $files{$commit} .= "\n";
53          } else {
54              $files{$commit} = "";
55          }
56          $files{$commit} .= $file;
57      }
58  }
59  close(INFILE);
60  my $full_log = 'bugs.log';
61  open(my $log, '>', $full_log) or die "Could not open file '$full_log' \!";
62  my $output = 'bugs.csv';
63  open(my $out, '>', $output) or die "Could not open file '$output' \!";
64  print $log "$commitsnum commits total\n";
65  print $log "The list of bug-fixing commits:\n";
66  my @BuggyCommits = grep { uc($labels{$_}) =~ /\[(FIX|SEC|BUG)\]/ } @commits;
67  my $bugcommitsnum = 0;
68  foreach $i (@BuggyCommits) {
69      print $log " $i\n";
70      $bugcommitsnum++;
71      my @BuggyCommitFiles = split("\n", $files{$i});
```

```
72      foreach $j (@BuggyCommitFiles) {
73          #adding escape characters for '{' and '}' to fix perl syntax error
74          $jclean = $j;
75          $jclean =~ s/\}/\\\}/g;
76          $jclean =~ s/\{/\\\{/g;
77          unless (grep( /^$jclean$/, @buggyfilesclean )) {
78              push @buggyfilesclean, $jclean;
79              push @buggyfiles, $j;
80              $frequency{$j} = 0;
81          }
82          $frequency{$j}++;
83      }
84  }
85  print $log "***OVERALL SUMMARY***\n";
86  print $log "$bugcommitsnum bug-fixing commits total\n";
87  print $log "The list of unique files in bug-fixing commits with frequencies of occurrence:\n";
88  my $stattotalnum = 0;
89  my %statfrequency;
90  my $package = "";
91  my %packages;
92  my $findslash = 0;
93  foreach $buggyfile (sort @buggyfiles) {
94      print $out "$buggyfile, $frequency{$buggyfile}\n";
95      print $log "\t$buggyfile, $frequency{$buggyfile}\n";
96      $stattotalnum++;
97      $findslash = index($buggyfile, '/');
98      if ($findslash != -1) {
99          $package = substr($buggyfile, 0, $findslash);
100     } else {
101         $package = "root directory";
102     }
103     unless (grep( /^$package$/, @packages )) {
104         push @packages, $package;
105         $statfrequency{$package} = 0;
106     }
107     $statfrequency{$package}++;
108 }
109 print $log "Bug fix distribution among project packages:\n";
```

```
110 my $percentage = 0;
111 foreach $k (@packages) {
112     $percentage = sprintf("%.2f", $statfrequency{$k}*100/$stattotalnum);
113     print $log "$k:\n\t\t\tnumber of unique files = $statfrequency{$k}\t\tpercentage = $percentage%\n";
114 }
115 close $out;
116 close $log;
117 print "See bugs.csv for the list of files involved in bug-fixing commits with the associated
frequencies. The full statistics on bug-fixing commits (containing the same list plus some additional
information) is presented in bugs.log.";
118
119 sub trim {
120     my $text = shift;
121     $text =~ s/^\s+//g;
122     $text =~ s/\s+$//g;
123     return $text;
124 }
```

Appendix D: Data post-processing for heuristic-based models

D.1 Step-by-step algorithm

1) Complete steps from Appendices B and C.
2) Post-process architectural data extracted to `merged.csv` (see Appendix B) and bug data extracted to `bugs.csv` (see Appendix C) via `heuristics.py` to prepare an input dataset for heuristic-based machine learning models, i.e. models built on features manually engineered on the basis of formalization of known hotspot patterns:

   ```
   python heuristics.py
   ```

   The output will be provided in `mlinput.csv` in the following format:

   ```
   - file
   - normalized CCN
   - normalized dependencies on a file (Dependers - # of files which depend on a file)
   - normalized dependencies of a file (Dependees - # of files a file depends on)
   - normalized total number of dependencies (Dependers + Dependees)
   - normalized number of co-changes with dependers
   - normalized number of co-changes with dependees
   - normalized bug frequency
   ```

   Keeping Dependers, Dependees and total Dependencies all in the same place enables us to use the same input file for both complete and reduced feature sets (simply by deleting redundant columns, e.g. in *Select Columns* widget in Orange). Similar logic is behind exporting bug frequency instead of binary or categorical bug-proneness – so we can use the same input file

regardless of what bug-proneness setting we opt for. We can just leverage *Feature Constructor* widget to quickly bring the target to the required form, depending on the requirements[16], and then remove BugFrequency from the data table in the *Select Columns* widget to ensure we only have one target.

D.2 Source code

heuristics.py

```
import pandas as pd
#output dataset format:
#  - file
#  - normalized CCN
#  - normalized dependencies on a file (Dependers - # of files which depend on a file)
#  - normalized dependencies of a file (Dependees - # of files a file depends on)
#  - normalized total number of dependencies (Dependers + Dependees)
#  - normalized number of co-changes with dependers
#  - normalized number of co-changes with dependees
#  - normalized bug frequency
merged = pd.read_csv('merged.csv')
#merged dataframe format:
#Source     Target     CCN        Status        Weight
bugs = pd.read_csv('bugs.csv', names=['File', 'BugsNum'])
#1. let's get a list of unique files
mlinput = pd.DataFrame(pd.unique(merged[['Source', 'Target']].values.ravel('K')), columns = ['File'])
#the formula for column normalization:
#if x is a column consisting of xi values, i=1,..,N, then
#normalized_xi = (xi-min(x))/(max(x)-min(x)).
```

[16] In our research, added variable *BugProneness* = 0 *if BugFrequency* == 0 *else* 1 with *values* = 0, 1 for the binary option, and *BugProneness* = 0 *if BugFrequency* == 0 *else* 1 *if BugFrequency* <= 0.016822429906542057 *else* 2 *if BugFrequency* <= 0.03551401869158879 *else* 3, *values* = *None*, *Low*, *Medium*, *High* for the categorical bug-proneness setting.

```
20  #based on our data, min(x)=0 for all 6 columns.
21  #therefore, the formula comes down to:
22  #normalized_xi = xi/max(x)
23  statusone = merged.loc[merged['Status'] == 1].copy()
24  statustwo = merged.loc[merged['Status'] == 2].copy()
25  statusthree = merged.loc[merged['Status'] == 3].copy()
26  for rows in mlinput.itertuples():
27      filename = getattr(rows, "File")
28      #2. CCN (overall number, normalization follows)
29      cnn = merged.loc[(merged['Source'] == filename) | (merged['Target'] == filename)]['CCN'].sum()
30      #however, we need to remember that there are reciprocal rows (doubled file pairs) for Status=3/3
case (see vizualization.py*):
31      #Source     Target     CCN        Status        Weight
32      #aa1        bb2        1          3             33
33      #bb2        aa1        1          3             33
34      #*when creating merged.csv we've already dropped reciprocal rows for Status=2 and for Status=3/2
35      #therefore, we may need to substract some number from the overall CCN:
36      statusthreefile = statusthree.loc[(statusthree['Source'] == filename)]
37      for trow in statusthreefile.itertuples():
38          rowfound = statusthree.loc[(statusthree['Source'] == getattr(trow, 'Target')) &
(statusthree['Target'] == getattr(trow, 'Source'))]
39          if not rowfound.empty:
40              cnn = cnn - rowfound['CCN'].sum()
41      mlinput.at[rows.Index,'CCN'] = cnn
42      #3. number of dependers (overall number, normalization follows)
43      mlinput.at[rows.Index,'Dependers'] = len(statusone.loc[(statusone['Target'] == filename)]) +
len(statusthree.loc[(statusthree['Target'] == filename)])
44      #4. number of dependees (overall number, normalization follows)
45      mlinput.at[rows.Index,'Dependees'] = len(statusone.loc[(statusone['Source'] == filename)]) +
len(statusthree.loc[(statusthree['Source'] == filename)])
46      #5. total number of dependencies (overall number, normalization follows)
47      mlinput.at[rows.Index,'Dependencies'] = mlinput.at[rows.Index,'Dependers'] +
mlinput.at[rows.Index,'Dependees']
48      #6. CCN with dependers (overall number, normalization follows)
49      #CCN in statusone should always be 0 because of the logic behind Status=1
50      #so we only need to focus on statusthree
51      mlinput.at[rows.Index,'DependersCCN'] = statusthree.loc[(statusthree['Target'] ==
filename)]['CCN'].sum()
```

```
52      #7. CCN with dependees (overall number, normalization follows)
53      mlinput.at[rows.Index,'DependeesCCN'] = statusthree.loc[(statusthree['Source'] ==
filename)]['CCN'].sum()
54      #8. bug frequency (overall number, normalization follows)
55      mlinput.at[rows.Index,'BugFrequency'] = bugs.loc[(bugs['File'] == filename)]['BugsNum'].sum()
56  #normalize columns 2-8
57  mlinput['CCN'] = mlinput['CCN']/mlinput['CCN'].max()
58  mlinput['Dependers'] = mlinput['Dependers']/mlinput['Dependers'].max()
59  mlinput['Dependees'] = mlinput['Dependees']/mlinput['Dependees'].max()
60  mlinput['Dependencies'] = mlinput['Dependencies']/mlinput['Dependencies'].max()
61  mlinput['DependersCCN'] = mlinput['DependersCCN']/mlinput['DependersCCN'].max()
62  mlinput['DependeesCCN'] = mlinput['DependeesCCN']/mlinput['DependeesCCN'].max()
63  mlinput['BugFrequency'] = mlinput['BugFrequency']/mlinput['BugFrequency'].max()
64  #output the dataframe
65  with open("mlinput.csv",'w') as outfile:
66      mlinput.to_csv(outfile, index=False, line_terminator='\n')
67  print("See mlinput.csv for feature output.")
```

Appendix E: Data post-processing for embedding-based models

E.1 Step-by-step algorithm

1) Complete steps from Appendices B and C.
2) Prepare an input to node2vec algorithm. Based on architectural relationships recovered in `merged.csv` (see Appendix B), create an undirected unweighted graph which forms a union of structural and logical dependencies between the files in the library package by executing `node2vec_input.py` for the necessary CCN Threshold(s), e.g., *CCNThreshold* = 1 and *CCNThreshold* = 3 in our research:

   ```
   python node2vec_input.py
   ```

   The script generates two files: `node2vec_input_[CCNThreshold].edgelist` which contains a list of edges in the union of two dependency subgraphs and is to be passed into node2vec, and `file-node_mapping_[CCNThreshold].csv` which specifies the correspondence of file names to node IDs (since an edge list for node2vec should be composed of two columns with integer values). The names of output files are auto-suffixed with the CCN Threshold set by the user (e.g., "th1" for *CCNThreshold* = 1 and "th3" for *CCNThreshold* = 3).
3) Represent each vertex of the resulting graph as a real-valued vector embedding preserving this node's network neighborhood via node2vec. You may need to get node2vec ready first. We used the reference implementation available at

https://github.com/aditya-grover/node2vec. For the sake of example, let's assume that the tool repository was cloned to the `C:/` drive (`C:/node2vec` is the root directory). Run `node2vec` with different p and q on all input files (in our case, for all CCN Thresholds), keeping other parameters as default:

```
cd C:/node2vec
python src/main.py --input ~/node2vec_input_th1.edgelist --output
~/node2vec_output_th1_p0.25_q0.5.emb --p 0.25 --q 0.5
python src/main.py --input ~/node2vec_input_th3.edgelist --output
~/node2vec_output_th3_p0.25_q0.5.emb --p 0.25 --q 0.5
```

In the file(s) produced by node2vec, the first row specifies the number of nodes the algorithm has yielded the representations for and the number of dimensions (i.e., the number of components) in the leant vectors. Then the output lists each node ID and the components of a corresponding to it embedding.

4) Generate an input to embedding-based machine learning models by executing `node2vec_output.py`, which maps the features engineered in an automated manner by node2vec back to file names (in accordance with `file-node_mapping_[CCNThreshold].csv`) and merges the result with the bug data extracted to `bugs.csv` (see Appendix C):

```
cd ~
python node2vec_output.py
```

The tool asks to provide a path to the file with embeddings, which makes it pretty flexible and convenient to use in case one needs to build different machine learning models on multiple network representations. It appends a suffix from initial file with

embeddings to the name of its output file to enable easy distinction between different outputs. The produced files are to be used as inputs to machine learning algorithms.

> Note: You may need to undersample the data before passing it into the machine learning algorithm. In Orange, for example, it can be accomplished via three widgets: first, separate out the majority class in the *Select Rows* widget, then sample the fixed number of instances from the isolated majority class in *Data Sampler* (sample without replacement, so all derived data points are unique), and in the end join the undersampled instances and the minority class in *Concatenate* (the minority class corresponds to the Unmatched Data output of the *Select Rows* widget).

E.2 Source code

node2vec_input.py

```
1   # prepare the input file for node2vec (list of edges forming undirected unweighted graph)
2   import pandas as pd
3   import csv
4   def yes_or_no(question):
5       answer = input(question).lower().strip()
6       print(" ")
7       while not(answer == "y" or answer == "yes" or answer == "n" or answer == "no"):
8           print("Input y/n or yes/no")
9           answer = input(question).lower().strip()
10          print(" ")
11      if answer[0] == "y":
12          return True
13      else:
14          return False
15  def get_threshold(message):
16      while True:
17          try:
18              userInput = int(input(message))
19          except ValueError:
20              print("Input has to be an integer. Try again.")
21              continue
```

```
        else:
            return userInput
        break
#import .csv file produced by visualization.py
mergeddf = pd.read_csv('merged.csv')
CCNthreshold = 1
if yes_or_no("Do you want to set a co-change threshold (i.e. the minimum number of co-changes two files should have to be considered evolutionary coupled and be included in the script output)?  "):
    CCNthreshold = get_threshold("CCN Threshold: ")
#visualization.py script leaves reciprocal edges (when files are mutually dependent on one another)
#for Status = 1 and Status = 3/3, since these cases correspond to hotspot patterns
#here we will use a simplified approach for merging, when we first get rid of all file pairs belonging to
#Status = 2 and whose number of co-changes is below the CCN threshold, and then drop all the duplicates,
#including 'reciprocal' ones, as in the following example:
#File1 File2
#  a     b
#  a     b - to be dropped
#  b     a - to be dropped too (since our network will be undirected)
mergeddf.drop(mergeddf.loc[(mergeddf['Status'] == 2) & (mergeddf['CCN'] < CCNthreshold)].index, inplace=True)
undirectedgraph = mergeddf[['Source', 'Target']].copy()
undirectedgraph['check'] = undirectedgraph.apply(lambda row: ''.join(sorted([row['Source'], row['Target']])), axis=1)
undirectedgraph.drop_duplicates(subset=['check'], inplace=True)
undirectedgraph.drop(columns=['check'], inplace=True)
#just in case let's check for self-dependency (we shouldn't have such rows already after visualization.py, so it may point out the problem)
if not undirectedgraph.loc[undirectedgraph['Source'] == undirectedgraph['Target']].empty:
    print("Houston, we have a problem! Self-dependency.\n")
#now we need to assign a nodeID to each file since nodes passed to node2vec have to be integers
uniquefiles = pd.DataFrame(pd.unique(undirectedgraph[['Source', 'Target']].values.ravel('K')), columns = ['File']).sort_values(by='File').reset_index(drop=True)
filetoID = dict(zip(uniquefiles.File,uniquefiles.index+1))
#replace file names with nodeIDs in the original dataframe
undirectedgraph.replace(filetoID, inplace=True)
#provide output
```

```
53  with open('file-node_mapping_th'+str(CCNthreshold)+'.csv', 'w', newline="") as csv_file:
54      writer = csv.writer(csv_file)
55      for key, value in filetoID.items():
56          writer.writerow([key, value])
57  with open('node2vec_input_th'+str(CCNthreshold)+'.edgelist','w') as outfile:
58      undirectedgraph.to_csv(outfile, sep=' ', header=False, index=False, line_terminator='\n')
59  print("Script has finished its work. See node2vec_input_th"+str(CCNthreshold)+".edgelist and file-
node_mapping_th"+str(CCNthreshold)+".csv for details.")
```

node2vec_output.py

```
1   #prepare the input file for machine learning models based on node2vec-derived embeddings
2   import pandas as pd
3   from tkinter import filedialog
4   from tkinter.filedialog import askopenfilename
5   root = filedialog.Tk()
6   root.withdraw()
7   filename = askopenfilename(initialdir = "/", title = "Select file output by node2vec", filetypes =
[('emb','*.emb')])
8   if filename == "":  #to process an exception arising if a user doesn't choose a file
9       quit()
10  suffix = filename[filename.rfind("/")+16:-4]
11  embeddings = pd.read_csv(filename, sep=" ", skiprows=1, header=None, index_col=0, names = ['Emb' +
str(n) for n in range(1, 129)]) #index of this dataframe corresponds to nodeID
12  #let's first create our metadata by replacing nodeIDs with file names from the appropriate mapping file
13  #note: mapping file has to be located in the same directory as the current script
14  if "_th1" in filename:
15      filenames = pd.read_csv('file-node_mapping_th1.csv', index_col=1, names = ['File'])
16  elif "_th3" in filename:
17      filenames = pd.read_csv('file-node_mapping_th3.csv', index_col=1, names = ['File'])
18  else:
19      print("Cannot identify which table with file-to-nodeID mapping to use.\nPlease be aware that the
script can only work with files output for CCN threshold = 1 or 3.\nIf you are using another CCN Threshold,
please modify lines 14-17 or add the needed file below them in a similar fashion.")
20      quit()
21  fileemb = filenames.join(embeddings)
22  #now let's add the target
```

```
23  bugs = pd.read_csv('bugs.csv', index_col=0, names=['BugsFreq'])
24  MLinput = fileemb.join(bugs, on='File')
25  MLinput["BugsFreq"] = MLinput["BugsFreq"].fillna(0).astype(int)
26  #provide output
27  with open("emb_mlinput"+suffix+".csv",'w') as outfile:
28      MLinput.to_csv(outfile, index=False, line_terminator='\n')
29  print("See emb_mlinput"+suffix+".csv for feature output.")
```

Appendix F: Performance indicators for different embedding-based models, with the best-performing model highlighted

Case #1 - p=0.25, q=0.5 (local DFS)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.652	0.648	0.639	0.656	0.623	117	101	193
Random Forest	1	0.729	0.655	0.640	0.669	0.613	120	94	190
CN2 Rule Induction	1	0.596	0.550	0.551	0.550	0.552	139	140	171
AdaBoost	1	0.603	0.603	0.595	0.607	0.584	129	117	181
Naïve Bayes	1	0.804	0.742	0.757	0.716	0.803	61	99	249
Logistic Regression	1	0.830	0.753	0.751	0.757	0.745	79	74	231
Neural Network	1	0.842	0.765	0.763	0.768	0.758	75	71	235
Tree	3	0.623	0.644	0.644	0.644	0.644	99	99	179
Random Forest	3	0.795	0.728	0.718	0.747	0.691	86	65	192
CN2 Rule Induction	3	0.616	0.585	0.593	0.581	0.604	110	121	168
AdaBoost	3	0.651	0.651	0.656	0.647	0.665	93	101	185
Naïve Bayes	3	0.781	0.732	0.720	0.753	0.691	86	63	192
Logistic Regression	3	0.798	0.739	0.731	0.755	0.709	81	64	197
Neural Network	3	0.843	0.759	0.755	0.767	0.745	71	63	207
Case #2 - p=1, q=0.5 (balanced DFS)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.600	0.606	0.595	0.613	0.577	131	113	179
Random Forest	1	0.728	0.674	0.663	0.686	0.642	111	91	199
CN2 Rule Induction	1	0.597	0.573	0.579	0.571	0.587	128	137	182
AdaBoost	1	0.624	0.624	0.635	0.617	0.655	107	126	203
Naïve Bayes	1	0.792	0.719	0.733	0.699	0.771	71	103	239
Logistic Regression	1	0.817	0.735	0.733	0.740	0.726	85	79	225
Neural Network	1	0.829	0.735	0.732	0.742	0.723	86	78	224
Tree	3	0.650	0.664	0.665	0.662	0.669	92	95	186
Random Forest	3	0.818	0.728	0.722	0.740	0.705	82	69	196
CN2 Rule Induction	3	0.660	0.606	0.607	0.606	0.608	109	110	169
AdaBoost	3	0.655	0.655	0.653	0.656	0.651	97	95	181
Naïve Bayes	3	0.794	0.745	0.737	0.760	0.716	79	63	199
Logistic Regression	3	0.769	0.737	0.733	0.746	0.719	78	68	200
Neural Network	3	0.839	0.764	0.763	0.767	0.759	67	64	211
Case #3 - p=2, q=0.5 (global DFS)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.562	0.569	0.563	0.571	0.555	138	129	172
Random Forest	1	0.738	0.663	0.655	0.671	0.639	112	97	198
CN2 Rule Induction	1	0.612	0.569	0.573	0.568	0.577	131	136	179
AdaBoost	1	0.616	0.616	0.592	0.631	0.558	137	101	173
Naïve Bayes	1	0.784	0.737	0.754	0.708	0.806	60	103	250
Logistic Regression	1	0.813	0.732	0.736	0.726	0.745	79	87	231
Neural Network	1	0.836	0.748	0.745	0.755	0.735	82	74	228
Tree	3	0.670	0.674	0.675	0.674	0.676	90	91	188
Random Forest	3	0.785	0.703	0.692	0.720	0.665	93	72	185
CN2 Rule Induction	3	0.677	0.622	0.622	0.622	0.622	105	105	173

[17] With regard to target class = bug-prone.

AdaBoost	3	0.687	0.687	0.691	0.682	0.701	83	91	195
Naïve Bayes	3	0.785	0.728	0.713	0.755	0.676	90	61	188
Logistic Regression	3	0.803	0.745	0.742	0.750	0.734	74	68	204
Neural Network	3	0.855	0.781	0.774	0.798	0.752	69	53	209
Case #4 - p=0.5, q=1 (local balanced walk)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.633	0.640	0.636	0.644	0.629	115	108	195
Random Forest	1	0.740	0.677	0.673	0.682	0.665	104	96	206
CN2 Rule Induction	1	0.589	0.561	0.564	0.561	0.568	134	138	176
AdaBoost	1	0.639	0.639	0.632	0.644	0.619	118	106	192
Naïve Bayes	1	0.808	0.742	0.763	0.706	0.829	53	107	257
Logistic Regression	1	0.816	0.744	0.745	0.741	0.748	78	81	232
Neural Network	1	0.834	0.748	0.746	0.753	0.739	81	75	229
Tree	3	0.655	0.662	0.652	0.672	0.633	102	86	176
Random Forest	3	0.791	0.719	0.708	0.738	0.680	89	67	189
CN2 Rule Induction	3	0.605	0.565	0.569	0.563	0.576	118	124	160
AdaBoost	3	0.651	0.651	0.654	0.649	0.658	95	99	183
Naïve Bayes	3	0.781	0.719	0.706	0.742	0.673	91	65	187
Logistic Regression	3	0.783	0.701	0.695	0.711	0.680	89	77	189
Neural Network	3	0.851	0.766	0.765	0.768	0.763	66	64	212
Case #5 - p=1, q=1 (default values)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.627	0.644	0.644	0.643	0.645	110	111	200
Random Forest	1	0.758	0.700	0.704	0.695	0.713	89	97	221
CN2 Rule Induction	1	0.571	0.532	0.528	0.533	0.523	148	142	162
AdaBoost	1	0.619	0.619	0.611	0.625	0.597	125	111	185
Naïve Bayes	1	0.816	0.760	0.779	0.722	0.845	48	101	262
Logistic Regression	1	0.825	0.744	0.748	0.735	0.761	74	85	236
Neural Network	1	0.828	0.737	0.730	0.751	0.710	90	73	220
Tree	3	0.661	0.673	0.664	0.682	0.647	98	84	180
Random Forest	3	0.816	0.743	0.734	0.761	0.709	81	62	197
CN2 Rule Induction	3	0.676	0.619	0.612	0.623	0.601	111	101	167
AdaBoost	3	0.705	0.705	0.695	0.719	0.673	91	73	187
Naïve Bayes	3	0.796	0.750	0.737	0.777	0.701	83	56	195
Logistic Regression	3	0.804	0.732	0.732	0.731	0.734	74	75	204
Neural Network	3	0.855	0.790	0.785	0.804	0.766	65	52	213
Case #6 - p=2, q=1 (global balanced walk)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.633	0.644	0.637	0.649	0.626	116	105	194
Random Forest	1	0.747	0.666	0.661	0.671	0.652	108	99	202
CN2 Rule Induction	1	0.650	0.613	0.609	0.615	0.603	123	117	187
AdaBoost	1	0.624	0.624	0.615	0.631	0.600	124	109	186
Naïve Bayes	1	0.803	0.765	0.779	0.733	0.832	52	94	258
Logistic Regression	1	0.831	0.745	0.750	0.736	0.765	73	85	237
Neural Network	1	0.834	0.766	0.765	0.769	0.761	74	71	236
Tree	3	0.632	0.660	0.657	0.663	0.651	97	92	181
Random Forest	3	0.779	0.719	0.713	0.729	0.698	84	72	194
CN2 Rule Induction	3	0.591	0.561	0.551	0.564	0.540	128	116	150
AdaBoost	3	0.676	0.676	0.675	0.678	0.673	91	89	187
Naïve Bayes	3	0.782	0.734	0.725	0.750	0.701	83	65	195
Logistic Regression	3	0.797	0.728	0.728	0.729	0.727	76	75	202
Neural Network	3	0.855	0.777	0.774	0.785	0.763	66	58	212
Case #7 - p=0.5, q=2 (local BFS)									

ML algorithm	CCN Threshold	AUC[17]	CA[17]	F1[17]	Precision[17]	Recall[17]	FN	FP	TP
Tree	1	0.621	0.611	0.595	0.621	0.571	133	108	177
Random Forest	1	0.764	0.687	0.671	0.707	0.639	112	82	198
CN2 Rule Induction	1	0.620	0.584	0.590	0.581	0.600	124	134	186
AdaBoost	1	0.618	0.618	0.604	0.626	0.584	129	108	181
Naïve Bayes	1	0.787	0.732	0.751	0.702	0.806	60	106	250
Logistic Regression	1	0.810	0.750	0.754	0.743	0.765	73	82	237
Neural Network	1	0.817	0.747	0.741	0.758	0.726	85	72	225
Tree	3	0.637	0.667	0.663	0.672	0.655	96	89	182
Random Forest	3	0.809	0.734	0.718	0.764	0.676	90	58	188
CN2 Rule Induction	3	0.670	0.631	0.631	0.632	0.629	103	102	175
AdaBoost	3	0.691	0.691	0.693	0.688	0.698	84	88	194
Naïve Bayes	3	0.777	0.714	0.707	0.725	0.691	86	73	192
Logistic Regression	3	0.782	0.710	0.708	0.714	0.701	83	78	195
Neural Network	3	0.837	0.763	0.757	0.774	0.741	72	60	206
Case #8 - p=1, q=2 (balanced BFS)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.613	0.611	0.609	0.612	0.606	122	119	188
Random Forest	1	0.743	0.694	0.679	0.713	0.648	109	81	201
CN2 Rule Induction	1	0.581	0.558	0.549	0.560	0.539	143	131	167
AdaBoost	1	0.634	0.634	0.630	0.637	0.623	117	110	193
Naïve Bayes	1	0.814	0.750	0.768	0.717	0.826	54	101	256
Logistic Regression	1	0.810	0.716	0.720	0.711	0.729	84	92	226
Neural Network	1	0.804	0.713	0.710	0.717	0.703	92	86	218
Tree	3	0.667	0.667	0.657	0.678	0.637	101	84	177
Random Forest	3	0.792	0.730	0.713	0.762	0.669	92	58	186
CN2 Rule Induction	3	0.587	0.583	0.581	0.583	0.579	117	115	161
AdaBoost	3	0.655	0.655	0.664	0.646	0.683	88	104	190
Naïve Bayes	3	0.788	0.743	0.733	0.763	0.705	82	61	196
Logistic Regression	3	0.796	0.727	0.724	0.732	0.716	79	73	199
Neural Network	3	0.867	0.802	0.801	0.807	0.795	57	53	221
Case #9 - p=4, q=2 (global BFS)									
ML algorithm	**CCN Threshold**	**AUC[17]**	**CA[17]**	**F1[17]**	**Precision[17]**	**Recall[17]**	**FN**	**FP**	**TP**
Tree	1	0.671	0.660	0.645	0.674	0.619	118	93	192
Random Forest	1	0.735	0.677	0.669	0.687	0.652	108	92	202
CN2 Rule Induction	1	0.627	0.585	0.597	0.581	0.613	120	137	190
AdaBoost	1	0.618	0.618	0.604	0.626	0.584	129	108	181
Naïve Bayes	1	0.818	0.745	0.764	0.711	0.826	54	104	256
Logistic Regression	1	0.819	0.734	0.733	0.735	0.732	83	82	227
Neural Network	1	0.828	0.747	0.742	0.756	0.729	84	73	226
Tree	3	0.642	0.655	0.658	0.651	0.665	93	99	185
Random Forest	3	0.790	0.705	0.693	0.723	0.665	93	71	185
CN2 Rule Induction	3	0.649	0.606	0.584	0.618	0.554	124	95	154
AdaBoost	3	0.648	0.647	0.642	0.652	0.633	102	94	176
Naïve Bayes	3	0.775	0.718	0.696	0.753	0.647	98	59	180
Logistic Regression	3	0.786	0.728	0.721	0.741	0.701	83	68	195
Neural Network	3	0.834	0.750	0.741	0.768	0.716	79	60	199