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

© 2021
Alla Zakurdaeva
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

Abstract .................................................................................................................................... ii

Acknowledgements ............................................................................................................... iii

Table of contents .................................................................................................................... iv

List of tables ................................................................................................................................ vii

List of figures .......................................................................................................................... ix

List of appendices .................................................................................................................. xi

Glossary of terms .................................................................................................................. xii

1. Introduction ........................................................................................................................ 1
   1.1 Objective .................................................................................................................... 2
   1.2 Deliverables ............................................................................................................... 4
   1.3 Relevance ................................................................................................................... 5
   1.4 Contribution ............................................................................................................... 6
   1.5 Overview of method ................................................................................................... 7
   1.6 Organization of the document ..................................................................................... 8

2. Literature review .............................................................................................................. 10
   2.1 Software architecture and quality analysis .............................................................. 10
   2.2 Hotspot pattern approach to detecting architectural integrity violations associated with bugs 15
       2.2.1 Design Rule Space (DRSpace) ............................................................................. 16
       2.2.2 Hotspot pattern definitions .............................................................................. 18
       2.2.3 Existing (non-ML-based) approach to detection of hotspot patterns .............. 25
   2.3 Machine learning techniques .................................................................................... 26
   2.4 Summary of key findings from the literature ............................................................ 30

3. Research method .............................................................................................................. 32
4. Case study: Tiki open source web-application ............................................................ 34

5. Part 1: Hotspot pattern derivation through machine learning ................................. 37
   5.1 Details of the method ............................................................................................... 37
   5.2 Results and discussion .............................................................................................. 42
      5.2.1 Heuristic verification ............................................................................................ 42
      5.2.2 Interpretation of the rule sets generated by white box heuristic-based models ...... 43
      5.2.3 Analysis of the rule sets generated by white box heuristic-based models............. 48
      5.2.4 Performance comparison of different algorithms (heuristic-based models)............. 52
      5.2.5 Summary ............................................................................................................. 56
   5.3 Limitations ............................................................................................................... 57

6. Part 2: Exploration into node embeddings as machine-engineered features preserving the network structure of a DRSpace ............................................................. 61
   6.1 Details of the method ............................................................................................... 61
   6.2 Results and discussion .............................................................................................. 70
      6.2.1 Best machine learning algorithm (embedding-based models).............................. 70
      6.2.2 Best embedding-based machine learning model.................................................. 71
      6.2.3 Best combination of p, q, and CCN Threshold ...................................................... 71
      6.2.4 3D graphs based on average recall values ............................................................. 72
      6.2.5 Impact of return parameter p (local walk close to the starting node vs moderate exploration) ...................................................................................................................... 72
      6.2.6 Impact of in-out parameter q (DFS vs BFS) .......................................................... 72
      6.2.7 Impact of CCN Threshold .................................................................................... 79
      6.2.8 Execution time ..................................................................................................... 80
      6.2.9 Summary ............................................................................................................. 80
   6.3 Limitations ............................................................................................................... 81

7. Discussion .......................................................................................................................... 84
7.1 Implications for software architecture analysis and research ........................................ 84
7.2 Implications for technology innovation management and technology entrepreneurship 86
7.3 Practical applications to the Tiki project ....................................................................... 87

8. Conclusion .......................................................................................................................... 89

References ............................................................................................................................ 91

Appendices ............................................................................................................................. 99

Appendix A: Architectural relations between the source files in the library package of the Tiki project .................................................................................................................................. 99

Appendix B: Architectural data extraction ........................................................................... 100
  B.1 Step-by-step algorithm ................................................................................................. 100
  B.2 Source code ................................................................................................................ 103

Appendix C: Bug data extraction .......................................................................................... 119
  C.1 Step-by-step algorithm ................................................................................................. 119
  C.2 Source code ................................................................................................................ 120

Appendix D: Data post-processing for heuristic-based models ......................................... 124
  D.1 Step-by-step algorithm ................................................................................................. 124
  D.2 Source code ................................................................................................................ 125

Appendix E: Data post-processing for embedding-based models ..................................... 128
  E.1 Step-by-step algorithm ................................................................................................. 128
  E.2 Source code ................................................................................................................ 130

Appendix F: Performance indicators for different embedding-based models, with the best-performing model highlighted ................................................................. 134
List of tables

Table 1  Highlights of the scholarly and practitioner literature ........................................ 10
Table 2  Summary of known architectural hotspots .......................................................... 19
Table 3  Research method summary ................................................................................... 32
Table 4  Data acquisition overview ..................................................................................... 36
Table 5  Feature sets constructed ........................................................................................ 37
Table 6  Bug frequency mapping to categorical bug-proneness ....................................... 39
Table 7  Maximum feature values and maximum bug frequency value ......................... 39
Table 8  Machine learning algorithm parameters in heuristic-based models ............... 41
Table 9  Correspondence of the complete feature set to the heuristics ......................... 43
Table 10 Examples of the rules produced by the CN2 Rule Inducer based on the binary target variable, and their correspondence to the heuristics .............................................. 45
Table 11 The distribution of files in possible hotspot patterns (previously unknown) detected by the CN2 Rule Induction ..................................................................................... 52
Table 12 Performance indicators for different heuristic-based models built on complete feature set ................................................................................................................................ 54
Table 13 False negative values for heuristic-based machine learning models built on complete feature set .................................................................................................................. 55
Table 14 Performance indicators for heuristic-based Decision Tree classifiers built on complete and reduced feature sets ........................................................................................................ 55
Table 15 Execution time of different heuristic-based models built on complete feature set ............................................................................................................................................... 56
Table 16  Execution time of heuristic-based Decision Tree classifiers built on complete and reduced feature sets

Table 17  Machine learning algorithm parameters in embedding-based models

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

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

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

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

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

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

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

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

Table 26  Average execution times of different embedding-based models
List of figures

Figure 1  Research method overview ................................................................. 7
Figure 2  Visualization of architectural relations .............................................. 14
Figure 3  DRSpace exemplary layout ............................................................... 18
Figure 4  Unstable Interface exemplary layout ............................................... 20
Figure 5  Unhealthy Inheritance Hierarchy exemplary layout ......................... 21
Figure 6  Modularity Violation Group exemplary layout ................................. 22
Figure 7  Crossing exemplary layout ............................................................... 23
Figure 8  Clique exemplary layout ................................................................. 24
Figure 9  Package Cycle exemplary layout ...................................................... 25
Figure 10 An example of the CN2 Rule Induction output in Orange toolkit ....... 46
Figure 11 Detected instance of Unstable Interface. Excerpt of the Design Structure
Matrix for lib/core/Services/Utilities.php (one-hop neighbors) ......................... 46
Figure 12 Detected instance of Modularity Violation Group. Excerpt of the Design
Structure Matrix for lib/core/Feed/ForwardLink.php (one-hop neighbors) ........ 46
Figure 13 Detected instance of Crossing. Excerpt of the Design Structure Matrix for
lib/core/Tracker/Field/Abstract.php (one-hop neighbors) ............................... 47
Figure 14 An example of the Decision Tree output in Orange toolkit ............... 48
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) ....................... 51
Figure 16 False negative values for heuristic-based Decision Tree models built on the
categorical target variable ................................................................................. 56
List of appendices

Appendix A: Architectural relations between the source files in the library package of the Tiki project................................................................. 99

Appendix B: Architectural data extraction................................................................. 100
  B.1 Step-by-step algorithm........................................................................... 100
  B.2 Source code......................................................................................... 103

Appendix C: Bug data extraction................................................................. 119
  C.1 Step-by-step algorithm........................................................................... 119
  C.2 Source code......................................................................................... 120

Appendix D: Data post-processing for heuristic-based models......................... 124
  D.1 Step-by-step algorithm........................................................................... 124
  D.2 Source code......................................................................................... 125

Appendix E: Data post-processing for embedding-based models......................... 128
  E.1 Step-by-step algorithm........................................................................... 128
  E.2 Source code......................................................................................... 130

Appendix F: Performance indicators for different embedding-based models, with the best-performing model highlighted.............................................. 134
## Glossary of terms

<table>
<thead>
<tr>
<th>Glossary Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural flaw</td>
<td>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”)</td>
</tr>
<tr>
<td>Architectural integrity violation</td>
<td>An architectural flaw caused by a failure to abide by software design principles</td>
</tr>
<tr>
<td>Architectural space (software architecture)</td>
<td>A conglomerate of software files and architectural relationships (e.g., structural or logical dependencies) between them</td>
</tr>
<tr>
<td>Architecture decay</td>
<td>Degradation of software architecture caused by poorly thought-through design decisions</td>
</tr>
<tr>
<td>Area under curve (AUC)</td>
<td>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 ( \text{FPR} = \frac{FP}{FP + TN} ) at various discrimination thresholds, which depicts relative trade-offs between true positives (benefits) and false positives (costs)</td>
</tr>
<tr>
<td>Classification accuracy (CA)</td>
<td>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: ( \text{CA} = \frac{(TP + TN)}{(TP + TN + FP + FN)} )</td>
</tr>
<tr>
<td>Confusion matrix</td>
<td>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</td>
</tr>
<tr>
<td>Dependees (in this research)</td>
<td>The files which the studied file depends on</td>
</tr>
<tr>
<td>Dependers (in this research)</td>
<td>The files which depend on the studied file</td>
</tr>
<tr>
<td>Design Rule Hierarchy (DRH)</td>
<td>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</td>
</tr>
<tr>
<td>Design Rule Space (DRSpace)</td>
<td>Architectural space, formed by one or more leading files and, if applicable, modules directly or indirectly dependent on them via structural and/or evolutionary</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Design Structure Matrix (DSM)</td>
<td>A square matrix, which visualizes architectural relations between the files marking its rows and columns in the same order</td>
</tr>
<tr>
<td>Embedding-based machine learning model <em>(in this research)</em></td>
<td>A model built on node embeddings as features learned by node2vec in an automated manner</td>
</tr>
<tr>
<td>Encapsulation <em>(object-oriented programming)</em></td>
<td>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 <em>(information hiding)</em></td>
</tr>
<tr>
<td>Evolutionary/logical dependency/coupling</td>
<td>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</td>
</tr>
<tr>
<td>F1 score</td>
<td>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: $F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}$</td>
</tr>
<tr>
<td>False negatives (FN)</td>
<td>The number of positive instances which were misclassified by machine learning algorithm as negative (e.g., actual value: “bug-prone”; predicted/assigned value: “benign”)</td>
</tr>
<tr>
<td>False positives (FN)</td>
<td>The number of negative instances which were misclassified by machine learning algorithm as positive (e.g., actual value: “benign”; predicted/assigned value: “bug-prone”)</td>
</tr>
<tr>
<td>Heuristic-based machine learning model <em>(in this research)</em></td>
<td>A model built on features manually engineered on the basis of formalization of known hotspot patterns (i.e., heuristics)</td>
</tr>
<tr>
<td>Hotspot pattern</td>
<td>A rule for pinpointing architectural integrity violations, associated with software bugs and incurring high maintenance costs</td>
</tr>
<tr>
<td>Inheritance <em>(object-oriented programming)</em></td>
<td>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</td>
</tr>
<tr>
<td>Interface <em>(object-oriented programming)</em></td>
<td>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</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>without knowing its exact inner workings (see also “encapsulation”)</td>
<td></td>
</tr>
<tr>
<td>Machine learning (ML) model</td>
<td>A mathematical representation of a specific process or phenomenon, inferred by the mathematical and statistical learning algorithm from the provided (“fed”) data.</td>
</tr>
<tr>
<td>Machine learning feature</td>
<td>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).</td>
</tr>
<tr>
<td>Network representation learning (NRL)</td>
<td>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.</td>
</tr>
<tr>
<td>Node embedding</td>
<td>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.</td>
</tr>
<tr>
<td>Precision</td>
<td>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)</td>
</tr>
<tr>
<td>Recall</td>
<td>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)</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Improving the internal structure of an existing body of code without changing its external behavior.</td>
</tr>
<tr>
<td>Software bug</td>
<td>An error, flaw or fault in a computer program or system that causes it to produce incorrect results or behave in an unintended way.</td>
</tr>
<tr>
<td>Software modularity</td>
<td>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.</td>
</tr>
<tr>
<td>Software package</td>
<td>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.</td>
</tr>
<tr>
<td>Structural dependency</td>
<td>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.,</td>
</tr>
</tbody>
</table>
would not compile); is exemplified by inheritance, implementation of an interface, or a call to another element’s function

<table>
<thead>
<tr>
<th>Technical debt</th>
<th>Implied cost of accumulated deficiencies in the code, technical documentation and development practices, deliberately incurred due to software project’s time constrains</th>
</tr>
</thead>
<tbody>
<tr>
<td>True negatives (TN)</td>
<td>The number of instances which were correctly classified by machine learning algorithm as negative (e.g., actual value: “benign”; predicted/assigned value: “benign”)</td>
</tr>
<tr>
<td>True positives (TP)</td>
<td>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”)</td>
</tr>
<tr>
<td>Undersampling</td>
<td>A group of techniques designed to balance the class distribution in a dataset by removing samples from the majority class</td>
</tr>
<tr>
<td>White box machine learning models</td>
<td>ML models whose rationale behind the produced predictions can be output, examined, and interpreted</td>
</tr>
</tbody>
</table>
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 composition² (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](image)

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

<table>
<thead>
<tr>
<th>Stream</th>
<th>Key highlights of the stream</th>
<th>Citations to key sources</th>
</tr>
</thead>
</table>
| 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](image)

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 \(\text{cell}(rx; cy)\) 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, \(\text{cell}(r3; c1)\) is labeled with "x", which indicates that file3 structurally depends on file1. Number in a cell \(\text{cell}(rx; cy)\), \(x \neq y\), 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, \(\text{cell}(r2; c6)\) 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 \(\text{cell}(rx; cy)\), \(x \neq y\), contains both an "x" and a number, then the file \(x\) is both structurally related to file \(y\) and evolutionary coupled with it. See \(\text{cell}(r4; c3)\) 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 \(M4: (rc9-rc10)\).
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>
<thead>
<tr>
<th></th>
<th>file1</th>
<th>file2</th>
<th>file3</th>
<th>file4</th>
<th>file5</th>
<th>file6</th>
<th>file7</th>
<th>file8</th>
<th>file9</th>
<th>file10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>file2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>file3</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>file4</td>
<td></td>
<td>x,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>file5</td>
<td></td>
<td>x,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>file6</td>
<td></td>
<td></td>
<td></td>
<td>x,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>file7</td>
<td></td>
<td></td>
<td>x,5</td>
<td>x,8</td>
<td>x,6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>file8</td>
<td></td>
<td>x,6</td>
<td>x,8</td>
<td>x,2</td>
<td>x,1</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>file9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x,7</td>
<td>(9)</td>
</tr>
<tr>
<td>10</td>
<td>file10</td>
<td></td>
<td>x,2</td>
<td>x,3</td>
<td>x</td>
<td>x</td>
<td>x,2</td>
<td>x</td>
<td></td>
<td>x,5</td>
</tr>
</tbody>
</table>

Figure 3  DRSpace exemplary layout
Table 2  Summary of known architectural hotspots

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstable Interface</td>
<td>too frequent co-changing of highly influential (e.g., leading) files</td>
</tr>
<tr>
<td>Unhealthy Inheritance Hierarchy</td>
<td>mutual dependence of parent and child, or client of the hierarchy, that depends on both parent and its child</td>
</tr>
<tr>
<td>Modularity Violation Group</td>
<td>the minimal number of file sets that cover all file pairs that are evolutionary coupled but have no structural dependencies</td>
</tr>
<tr>
<td>Crossing</td>
<td>“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</td>
</tr>
<tr>
<td>Clique</td>
<td>a set of files that are closely coupled with each other through one or more structural dependency cycles</td>
</tr>
<tr>
<td>Package Cycle</td>
<td>cyclic structural dependency of different packages</td>
</tr>
</tbody>
</table>

2.2.2  Formal definitions. Let us introduce some notions for hotspots formalization [27, 28, 30]: \( \text{StructImpact}_{\text{thr}} \) is the threshold of the structural impact scope of a file, \( \text{HistoryImpact}_{\text{thr}} \) is the threshold of the number of files evolutionary coupled with a file, \( \text{cochange}_{\text{thr}} \) 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), \( \text{crossing}_{\text{thr}} \) 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: \( F = \{ f_i \mid i \in \mathbb{N} \} \), \( \text{depend}(x, y) \) – \( x \) structurally depends on \( y \), \( \text{#cochange}(x, y) \) – 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 $\text{StructImpact}_{thr}$ files that structurally depend on the file under study (which represents a highly influential file) and more than $\text{HistoryImpact}_{thr}$ of these files have been changed together with it more than $\text{cochange}_{thr}$ 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: $\exists f_1 \in F \mid \{F_{set,S}: \forall f_i \in F_{set,S} \mid SRelation(f_i, f_1) \land \{F_{set,H}: \forall f_j \in F_{set,H} \mid \#cochange(f_j, f_1) > cochange_{thr}\} \land (|F_{set}| > \text{StructImpact}_{thr}) \land (|F_{set,S} \cap F_{set,H}| > \text{HistoryImpact}_{thr})$, where $i \in \{1, 2, 3, \ldots, n\}$, $n$ is the number of files in $F_{set,S}$, $j \in \{1, 2, 3, \ldots, m\}$, $m$ 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.
**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, \( f_{\text{parent}} \), and one or more children, \( F_{\text{child}} \), there exists a child file \( f_i \) that \( f_{\text{parent}} \) depends on.

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

Mathematical formalization:

\[
\exists f_{\text{parent}}, F_{\text{child}} \in F \land \exists f_i \in F_{\text{child}} \mid \text{depend}(f_{\text{parent}}, f_i) \lor \exists f_j \in F \mid \text{depend}(f_j, f_{\text{parent}}) \land \exists f_i \in F_{\text{child}} \mid \text{depend}(f_j, f_i),
\]

where \( i, j \in [1, 2, 3, \ldots, n], f_j \notin F_{\text{child}} \) and \( f_j \neq f_{\text{parent}} [27, 30] \).

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

1) The parent file, *file1*, depends on its child, *file2*.


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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>file1</td>
<td>dp,9</td>
<td>.2</td>
<td>.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>file2</td>
<td>ex,9</td>
<td>.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>file3</td>
<td>dp,2</td>
<td></td>
<td>.4</td>
<td>.3</td>
<td>.7</td>
</tr>
<tr>
<td>4</td>
<td>file4</td>
<td>.8</td>
<td>ex,4</td>
<td>.1</td>
<td>.11</td>
<td>.4</td>
</tr>
<tr>
<td>5</td>
<td>file5</td>
<td>.5</td>
<td>ex,3</td>
<td>.11</td>
<td>.5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>file6</td>
<td>dp,7</td>
<td>dp,4</td>
<td>dp,5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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\textsubscript{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: \( MVG_1 \cup MVG_2 \cup MVG_3 \cup \ldots \cup MVG_n = P \mid f_{\text{core}} \in MVG_i, \forall f_j \in MVG_i, f_j \neq f_{\text{core}} \land \neg SRelation(f_{\text{core}}, f_j) \land \neg SRelation(f_j, f_{\text{core}}) \land (\#\text{cochange}(f_{\text{core}}, f_j) > \text{cochange}_{\text{thr}}) \), 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 \( f_{\text{core}} \), 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>file1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>file2</td>
<td>x,2</td>
<td>(2)</td>
<td>7</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>file3</td>
<td>x,3</td>
<td>(3)</td>
<td>3</td>
<td>2</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>file4</td>
<td>x,3</td>
<td>(4)</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>file5</td>
<td>x,7</td>
<td>x</td>
<td>(5)</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>file6</td>
<td>x,2</td>
<td>x</td>
<td>(6)</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>file7</td>
<td>x,4</td>
<td>3</td>
<td>x</td>
<td>(7)</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>file8</td>
<td>32</td>
<td>16</td>
<td>15</td>
<td>4</td>
<td>14</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 6  Modularity Violation Group exemplary layout
Crossing. The file structurally depends on more than crossing files evolutionary coupled with it, and more than crossing 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: \( \exists f_i, f_j, f_c \in F \mid (|SRelation(f_i, f_j)| > crossing_{thr} \land |\#cochange(f_i, f_c) > cochange_{thr}| > crossing_{thr}) \land (|SRelation(f_c, f_j)| > crossing_{thr}) \land |\#cochange(f_c, f_j) > cochange_{thr}| > crossing_{thr}) \), where \( i, j \in \{2, 3, 4, ..., n\} \), \( n \) 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).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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: \( \forall f_i \in F_{cq}, \forall f_j \in F_{cq}, i \neq j \mid \text{depend}(f_i, f_j) \land \text{depend}(f_j, f_i) \) \( \lor \) \( \exists f_1, ..., f_n \in F_{cq}, n \geq 1 \mid \text{depend}(f_i, f_1) \land \text{depend}(f_1, f_2) \land ... \land \text{depend}(f_n, f_j) \), where \( i, j \in [1, 2, 3, ..., n] \), \( F_{cq} \) 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: \textit{file1} \( \rightarrow \) \textit{file9} \( \rightarrow \) \textit{file2} \( \rightarrow \) \textit{file1}.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>file1</td>
<td>(1)</td>
<td>.5</td>
<td>.2</td>
<td>x,2</td>
<td>.2</td>
<td>.10</td>
<td>.3</td>
<td>.3</td>
<td>x,7</td>
</tr>
<tr>
<td>2</td>
<td>file2</td>
<td>x,5</td>
<td>(2)</td>
<td>.3</td>
<td>.6</td>
<td>.3</td>
<td>.6</td>
<td>.4</td>
<td>.9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>file3</td>
<td>x,2</td>
<td>(3)</td>
<td>.7</td>
<td>.2</td>
<td>x,7</td>
<td>.5</td>
<td>.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>file4</td>
<td>.2</td>
<td>x,3</td>
<td>.7</td>
<td>(4)</td>
<td>.5</td>
<td>.8</td>
<td>.2</td>
<td>x,11</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>file5</td>
<td>x,2</td>
<td>.6</td>
<td>x,2</td>
<td>x,5</td>
<td>(5)</td>
<td>.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>file6</td>
<td>.10</td>
<td>.3</td>
<td>.8</td>
<td>(6)</td>
<td>x,3</td>
<td>.4</td>
<td>.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>file7</td>
<td>.3</td>
<td>x,6</td>
<td>.7</td>
<td>x,7</td>
<td>(7)</td>
<td>.4</td>
<td>.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>file8</td>
<td>x,3</td>
<td>.4</td>
<td>.5</td>
<td>x,2</td>
<td>x,3</td>
<td>x,4</td>
<td>(8)</td>
<td>.9</td>
<td>.2</td>
</tr>
<tr>
<td>9</td>
<td>file9</td>
<td>.7</td>
<td>x,9</td>
<td>x</td>
<td>.4</td>
<td>x,9</td>
<td>(9)</td>
<td>x,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>file10</td>
<td>x,8</td>
<td>x,5</td>
<td>.11</td>
<td>x,7</td>
<td>.2</td>
<td>.2</td>
<td>.3</td>
<td>(10)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8** Clique exemplary layout

**Package Cycle.** The file \( f_i \) from the package \( P_a \) depends on some file from the package \( P_b \), and the file \( f_j \) 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: \( \exists f_1, f_i \in P_a \land \exists f_2, f_j \in P_b \mid \text{depend}(f_1, f_j) \land \text{depend}(f_2, f_i) \), where \( P_a, P_b \) are the packages of the system, \( i, j \in [1, 2, 3, ..., n] \), \( n \) 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Package1.file1</td>
<td>(1)</td>
<td>10</td>
<td>x,2</td>
</tr>
<tr>
<td>2</td>
<td>Package1.file2</td>
<td>,10</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Package2.file1</td>
<td>,2</td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>4</td>
<td>Package2.file2</td>
<td>x</td>
<td></td>
<td>,5</td>
</tr>
</tbody>
</table>

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 Understand³. 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.
(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>
<thead>
<tr>
<th>Step</th>
<th>Description of activity undertaken</th>
<th>Outcome of the activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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</td>
<td>Lists of heuristics, features, suitable machine learning techniques</td>
</tr>
<tr>
<td>2</td>
<td>Computing of architectural and bug metrics based on records in the selected project’s version control system</td>
<td>Algorithm for metrics extraction, e.g. raw input creation for machine learning models (Appendices B and C)</td>
</tr>
<tr>
<td>3</td>
<td>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</td>
<td>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)</td>
</tr>
<tr>
<td>4</td>
<td>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</td>
<td>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)</td>
</tr>
<tr>
<td>5</td>
<td>Results summarization</td>
<td>Report of the analysis results, suggestions for future investigation (Sections 5.2.5, 5.3, 6.2.9, 6.3, Chapter 7)</td>
</tr>
</tbody>
</table>

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 7\(^{th}\), 2002, to March 3\(^{rd}\), 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 3\(^{rd}\), 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

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

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, Dependents – the number of files which depend on a file, Dependees – the number of files a file depends on, DependentsCCN – 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 Dependences (that, in essence, is a sum of Dependents 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>
<thead>
<tr>
<th>Feature</th>
<th>Role</th>
<th>Complete feature set</th>
<th>Reduced feature set</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Name</td>
<td>meta</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CCN</td>
<td>feature</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Dependents</td>
<td>feature</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Dependees</td>
<td>feature</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Dependences</td>
<td>feature</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>DependentsCCN</td>
<td>feature</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>DependeesCCN</td>
<td>feature</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>BugProneness</td>
<td>target</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
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, 

$$BugProneness = 0 \text{ if the file had never been a part of a bug-fixing commit}$$

($$BugFrequency = 0$$) and 

$$BugProneness = 1 \text{ if had been at least once}$$ ($$BugFrequency > 0$$). 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 ($$BugFrequency = 1,9$$) have been mapped to category $$BugProneness = Low$$, 10 to 19 bug-fixing commits ($$BugFrequency = 10,19$$) – to category $$BugProneness = Medium$$, 20 and more bug-fixing commits ($$BugFrequency \geq 20$$) – 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 $$BugFrequency \geq 20$$, and 6.3% of files with $$BugFrequency = 10,19$$ (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:
\[ g_{\text{norm}} = \frac{g - \min(g)}{\max(g) - \min(g)}, \] 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:

\[ g_{\text{norm}} = \frac{g}{\max(g)}. \]

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

<table>
<thead>
<tr>
<th>Bug frequency</th>
<th>Normalized bug frequency</th>
<th>Bug-proneness</th>
<th>Number of files</th>
<th>Percentage of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>None</td>
<td>1012</td>
<td>33.7%</td>
</tr>
<tr>
<td>1-9</td>
<td>0.0019-0.0168</td>
<td>Low</td>
<td>1679</td>
<td>56.0%</td>
</tr>
<tr>
<td>10-19</td>
<td>0.0187-0.0355</td>
<td>Medium</td>
<td>190</td>
<td>6.3%</td>
</tr>
<tr>
<td>20+</td>
<td>0.0374-1</td>
<td>High</td>
<td>120</td>
<td>4.0%</td>
</tr>
</tbody>
</table>

**Table 7**  Maximum feature values and maximum bug frequency value

<table>
<thead>
<tr>
<th>( g )</th>
<th>( \max(g) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCN</td>
<td>1291</td>
</tr>
<tr>
<td>Dependents</td>
<td>645</td>
</tr>
<tr>
<td>Dependees</td>
<td>97</td>
</tr>
<tr>
<td>Dependencies</td>
<td>663</td>
</tr>
<tr>
<td>DependentsCCN</td>
<td>653</td>
</tr>
<tr>
<td>DependeesCCN</td>
<td>149</td>
</tr>
<tr>
<td>BugFrequency</td>
<td>535</td>
</tr>
</tbody>
</table>

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\textsuperscript{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.

\textsuperscript{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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Target variable setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binary</td>
</tr>
<tr>
<td></td>
<td>Categorical</td>
</tr>
<tr>
<td><strong>Decision Tree</strong></td>
<td></td>
</tr>
<tr>
<td>Induce binary tree</td>
<td>yes</td>
</tr>
<tr>
<td>Min. number of instances in leaves</td>
<td>2</td>
</tr>
<tr>
<td>Do not split subsets smaller than</td>
<td>5</td>
</tr>
<tr>
<td>Limit the maximal tree depth to</td>
<td>100</td>
</tr>
<tr>
<td>Stop when majority reaches</td>
<td>100%</td>
</tr>
<tr>
<td><strong>CN2 Rule Induction</strong></td>
<td></td>
</tr>
<tr>
<td>Rule ordering</td>
<td>Unordered</td>
</tr>
<tr>
<td>Covering algorithm</td>
<td>Exclusive</td>
</tr>
<tr>
<td>Evaluation measure</td>
<td>Laplace accuracy</td>
</tr>
<tr>
<td>Beam width</td>
<td>5</td>
</tr>
<tr>
<td>Minimum rule coverage</td>
<td>1</td>
</tr>
<tr>
<td>Maximum rule length</td>
<td>5</td>
</tr>
<tr>
<td>Statistical significance (default $\alpha$)</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Relative significance (parent $\alpha$)</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td><strong>Random Forest</strong></td>
<td></td>
</tr>
<tr>
<td>Number of trees</td>
<td>10</td>
</tr>
<tr>
<td>Number of attributes considered at each split</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Replicable training</td>
<td>yes</td>
</tr>
<tr>
<td>Limit depth of individual trees</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Do not split subsets smaller than</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td><strong>AdaBoost</strong></td>
<td></td>
</tr>
<tr>
<td>Base estimator</td>
<td>Tree</td>
</tr>
<tr>
<td>Number of estimators</td>
<td>50</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.90</td>
</tr>
<tr>
<td>Fixed seed for random generator</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Classification algorithm</td>
<td>SAMME.R</td>
</tr>
<tr>
<td>Regression loss function</td>
<td>Linear</td>
</tr>
</tbody>
</table>
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 \{\text{high Dependees}, \text{high DependeesCCN}\}, or \{\text{high Dependees}, \text{high DependeesCCN}\}, 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 \{\text{high Dependees}, \text{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 \{\text{high Dependees}, \text{high DependeesCCN}\}). The inverse relationship is based on the similar reasoning; however, the symmetry will only work if DependeesCCN or DependeesCCN 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

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Machine learning model features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstable Interface</td>
<td>high Dependents, high DependentsCCN</td>
</tr>
<tr>
<td>Unhealthy Inheritance Hierarchy</td>
<td>N/A (structural dependencies have been aggregated for each file separately)</td>
</tr>
<tr>
<td>Modularity Violation Group</td>
<td>low Dependents, low Dependees, high CCN</td>
</tr>
<tr>
<td>Crossing</td>
<td>{high Dependents + high DependeesCCN}, or {high Dependees + high DependentsCCN}, or the combination of these</td>
</tr>
<tr>
<td>Clique</td>
<td>N/A (structural dependencies have been aggregated for each file separately)</td>
</tr>
<tr>
<td>Package Cycle</td>
<td>N/A (single package has been examined)</td>
</tr>
</tbody>
</table>

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) $0.0333 \leq \text{CCN, Depend} \leq 0.0016, \text{Dependee} \leq 0.0103, \text{DependeesCCN} \leq 0.0134$ (Quality 0.982; 107 bug-prone cases).

2) $0.0279 \leq \text{CCN, Depend} \leq 0.0016, \text{Dependee} \leq 0.0103, \text{DependeesCCN} \leq 0.0134$ (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, Dependee defines 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

<table>
<thead>
<tr>
<th>Generated rule</th>
<th>Interpretation</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0139 ≤ CCN, 0.0109 ≤ Dependers, 0.0031 ≤ DependentsCCN ≤ 0.0092 (Quality 0.800; 3 bug-prone cases)</td>
<td>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.</td>
<td>Unstable Interface</td>
</tr>
<tr>
<td>CCN ≤ 0.0178, 0.0233 ≤ Dependents, 0.0168 ≤ DependentsCCN ≤ 0.0184 (Quality 0.667; 1 bug-prone case)*</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>0.0101 ≤ CCN ≤ 0.0263, Dependents ≤ 0.0016, Dependees ≤ 0.0103, DependentsCCN ≤ 0.0015 (Quality 0.789; 518 bug-prone cases)</td>
<td>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.</td>
<td>Modularity Violation Group</td>
</tr>
<tr>
<td>0.0279 ≤ CCN, Dependents ≤ 0.0016, Dependees ≤ 0.0103, DependentsCCN ≤ 0.0134 (Quality 0.948; 169 bug-prone cases)**</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>0.0116 ≤ CCN ≤ 0.0434, Dependents ≤ 0.0124, 0.0092 ≤ DependentsCCN, 0.0309 ≤ Dependees (Quality 0.867; 12 bug-prone cases)</td>
<td>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.</td>
<td>Crossing</td>
</tr>
<tr>
<td>0.0199 ≤ DependentsCCN, 0.0206 ≤ Dependees (Quality 0.833; 4 bug-prone cases)***</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

*  Representative file: lib/core/Services/Utilities.php
** Representative file: lib/core/Feed/ForwardLink.php
***  Representative file: lib/core/Tracker/Field/Abstract.php
Figure 10  An example of the CN2 Rule Induction output in Orange toolkit

<table>
<thead>
<tr>
<th>IF conditions</th>
<th>THEN class</th>
<th>Distribution</th>
<th>Probabilities [%]</th>
<th>Quality</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNn: 0.07285536018390243 AND CCNn: 0.07937231506525378 AND CCNn: 0.0721648456330824 AND</td>
<td>DisposableBugPromineness=1</td>
<td>[0, 74]</td>
<td>1 : 99</td>
<td>0.987</td>
<td>3</td>
</tr>
<tr>
<td>CCNn: 0.029412020313426820 AND CCNn: 0.038202014021394268 AND CCNn: 0.0771106687405113 AND</td>
<td>DisposableBugPromineness=1</td>
<td>[0, 40]</td>
<td>2 : 98</td>
<td>0.975</td>
<td>3</td>
</tr>
<tr>
<td>Cls: 0.013422818791946308 AND CCNn: 0.033307513535338342 AND</td>
<td>DisposableBugPromineness=1</td>
<td>[1, 107]</td>
<td>2 : 98</td>
<td>0.982</td>
<td>5</td>
</tr>
<tr>
<td>Cls: 0.010059871340464756 AND CCNn: 0.0134058730515464 AND</td>
<td>DisposableBugPromineness=1</td>
<td>[0, 12]</td>
<td>7 : 93</td>
<td>0.929</td>
<td>2</td>
</tr>
<tr>
<td>Cls: 0.020132628187919462 AND CCNn: 0.0285358180546243 AND</td>
<td>DisposableBugPromineness=1</td>
<td>[1, 40]</td>
<td>5 : 95</td>
<td>0.953</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Services/Utilities.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>lib/core/Feed/ForwardLink.php</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
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.
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 ($Dependers_{CCN} = 0, Dependees_{CCN} = 0$).

- 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 ($\approx 88\%$ for Decision Tree and $\approx 83\%$ 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 Dependents, high DependentsCCN\}, 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 (\textit{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.

<table>
<thead>
<tr>
<th></th>
<th>lib/core/Search/Expr/Or.php</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lib/core/Search/Expr/Or.php</td>
<td></td>
<td>(1)</td>
<td>.5</td>
<td>.5</td>
<td>.4</td>
<td>.2</td>
<td>.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>lib/core/Search/Expr/Not.php</td>
<td>.5</td>
<td>(2)</td>
<td>.5</td>
<td>.4</td>
<td>.2</td>
<td>.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>lib/core/Search/Expr/MoreLikeThis.php</td>
<td>(3)</td>
<td>.2</td>
<td>.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>lib/core/Search/Expr/Distance.php</td>
<td>(4)</td>
<td>.1</td>
<td>.1</td>
<td>.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>lib/core/Search/Expr/And.php</td>
<td>.5</td>
<td>.5</td>
<td>(5)</td>
<td>.4</td>
<td>.2</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>lib/core/Search/Expr/Token.php</td>
<td>.4</td>
<td>.4</td>
<td>(6)</td>
<td>.1</td>
<td>.2</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>lib/core/Search/Elastic/TypeFactory.php</td>
<td>.1</td>
<td>.1</td>
<td>(7)</td>
<td>.10</td>
<td>.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>lib/core/Search/Expr/Range.php</td>
<td>.2</td>
<td>.2</td>
<td>.2</td>
<td>x,2</td>
<td>(8)</td>
<td>.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>lib/core/Search/Elastic/Index.php</td>
<td>.2</td>
<td>.1</td>
<td>.1</td>
<td>x,10</td>
<td>(9)</td>
<td>x,12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>lib/core/Search/Elastic/QueryBuilder.php</td>
<td>x,1</td>
<td>x,1</td>
<td>x,3</td>
<td>x,2</td>
<td>x,1</td>
<td>x,2</td>
<td>x,2</td>
<td>x,2</td>
<td>x,12</td>
<td>(10)</td>
</tr>
</tbody>
</table>

**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 Dependents, 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

<table>
<thead>
<tr>
<th>Pattern number</th>
<th>Target variable</th>
<th>Bug-prone files</th>
<th>Files total</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Binary</td>
<td>17</td>
<td>18</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>Categorical</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>Binary</td>
<td>23</td>
<td>27</td>
<td>85%</td>
</tr>
<tr>
<td></td>
<td>Categorical</td>
<td>4</td>
<td>4</td>
<td>100%</td>
</tr>
</tbody>
</table>

#### 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 ($\text{BugProneness} = 1$ for

---

7 For the categorical setting, files with $\text{BugProneness} = \text{Medium or High}$ have been regarded as “bug-prone,” and files with $\text{BugProneness} = \text{None or Low}$ 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-
prone ness 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-prone ness 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**

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

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

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

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

<table>
<thead>
<tr>
<th>Feature set</th>
<th>AUC</th>
<th>CA</th>
<th>F1</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Binary option (Target class = 1)</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>0.692</td>
<td>0.695</td>
<td>0.784</td>
<td>0.738</td>
<td>0.836</td>
</tr>
<tr>
<td>Reduced</td>
<td><strong>0.711</strong></td>
<td>0.688</td>
<td>0.779</td>
<td>0.733</td>
<td>0.832</td>
</tr>
<tr>
<td><em>Categorical option (Target class = Medium)</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>0.670</td>
<td><strong>0.920</strong></td>
<td>0.143</td>
<td>0.225</td>
<td>0.105</td>
</tr>
<tr>
<td>Reduced</td>
<td><strong>0.689</strong></td>
<td>0.920</td>
<td>0.131</td>
<td>0.212</td>
<td>0.095</td>
</tr>
<tr>
<td><em>Categorical option (Target class = High)</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>0.745</td>
<td>0.961</td>
<td>0.410</td>
<td>0.512</td>
<td>0.342</td>
</tr>
<tr>
<td>Reduced</td>
<td><strong>0.757</strong></td>
<td>0.960</td>
<td>0.388</td>
<td>0.500</td>
<td>0.317</td>
</tr>
</tbody>
</table>
Table 15  Execution time of different heuristic-based models built on complete feature set

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>Binary option</th>
<th>Categorical option</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>00s : 41ms</td>
<td>00s : 30ms</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>08s : 63ms</td>
<td>00s : 40ms</td>
</tr>
<tr>
<td>Random Forest</td>
<td>00s : 18ms</td>
<td>00s : 11ms</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>00s : 16ms</td>
<td>00s : 10ms</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>00s : 20ms</td>
<td>00s : 18ms</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Feature set</th>
<th>Binary option</th>
<th>Categorical option</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Complete</td>
<td>00s : 41ms</td>
<td>00s : 30ms</td>
</tr>
<tr>
<td>Reduced</td>
<td>00s : 21ms</td>
<td>00s : 16ms</td>
</tr>
</tbody>
</table>

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 Dependents, 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]. 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 $cochangethr = 0$.

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\(^1\) per se, some studies show that treating structural dependencies as undirected, instead of directed, relationships does not qualitatively impact the network recovery [19].

\(^1\) 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 \( \text{DefaultWeightForDependency} = \max(CCN)/2 \) (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

---

\(^{11a}\) When information on evolutionary couplings is merged with the structural dependency data, the value decreases accordingly, because of the file pairs with \( \text{Status} = 1 \) (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.

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.

![Diagram of node2vec random walk transition probability bias](image)

**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 ≥ 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 > cochangethr). 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 ≥ CCNThreshold). Therefore, our CCNThreshold = 3 corresponds to cochangethr = 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 ≤ 9 are considered benign, and files with BugFrequency ≥ 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decision Tree</strong></td>
<td></td>
</tr>
<tr>
<td>Induce binary tree</td>
<td>yes</td>
</tr>
<tr>
<td>Min. number of instances in leaves</td>
<td>2</td>
</tr>
<tr>
<td>Do not split subsets smaller than</td>
<td>5</td>
</tr>
<tr>
<td>Limit the maximal tree depth to</td>
<td>100</td>
</tr>
<tr>
<td>Stop when majority reaches</td>
<td>95%</td>
</tr>
<tr>
<td><strong>CN2 Rule Induction</strong></td>
<td></td>
</tr>
<tr>
<td>Rule ordering</td>
<td>Ordered</td>
</tr>
<tr>
<td>Covering algorithm</td>
<td>Exclusive</td>
</tr>
<tr>
<td>Evaluation measure</td>
<td>Entropy</td>
</tr>
<tr>
<td>Beam width</td>
<td>5</td>
</tr>
<tr>
<td>Minimum rule coverage</td>
<td>1</td>
</tr>
<tr>
<td>Maximum rule length</td>
<td>5</td>
</tr>
<tr>
<td>Statistical significance (default $\alpha$)</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Relative significance (parent $\alpha$)</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td><strong>Random Forest</strong></td>
<td></td>
</tr>
<tr>
<td>Number of trees</td>
<td>10</td>
</tr>
<tr>
<td>Number of attributes considered at each split</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Replicable training</td>
<td>no</td>
</tr>
<tr>
<td>Limit depth of individual trees</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Do not split subsets smaller than</td>
<td>5</td>
</tr>
<tr>
<td><strong>AdaBoost</strong></td>
<td></td>
</tr>
<tr>
<td>Base estimator</td>
<td>Tree</td>
</tr>
<tr>
<td>Number of estimators</td>
<td>50</td>
</tr>
<tr>
<td>Learning rate</td>
<td>1.0</td>
</tr>
<tr>
<td>Fixed seed for random generator</td>
<td>unchosen/unchecked</td>
</tr>
<tr>
<td>Classification algorithm</td>
<td>SAMME.R</td>
</tr>
<tr>
<td>Regression loss function</td>
<td>Linear</td>
</tr>
<tr>
<td><strong>Logistic Regression</strong></td>
<td></td>
</tr>
<tr>
<td>Regularization type</td>
<td>Ridge (L2)</td>
</tr>
<tr>
<td>Strength</td>
<td>$C = 1$</td>
</tr>
<tr>
<td><strong>Neural Network</strong></td>
<td></td>
</tr>
<tr>
<td>Neurons in hidden layers</td>
<td>100,</td>
</tr>
<tr>
<td>Activation</td>
<td>ReLu</td>
</tr>
<tr>
<td>Solver</td>
<td>Adam</td>
</tr>
<tr>
<td>Regularization</td>
<td>$\alpha = 0.0001$</td>
</tr>
<tr>
<td>Maximal number of iterations</td>
<td>200</td>
</tr>
<tr>
<td>Replicable training</td>
<td>yes</td>
</tr>
</tbody>
</table>
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 \( recall \approx 82\% \), whereas the top models for \( CCNThreshold = 3 \) (all belonging to Neural Network) only attain \( recall \approx 76\% \). 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

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>Precision(^{12})</th>
<th>Recall(^{12})</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case #6 (- p = 2, q = 1) (global balanced walk) with (CCNThreshold = 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>0.733</td>
<td>0.832</td>
<td>52</td>
<td>94</td>
<td>258</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.736</td>
<td>0.765</td>
<td>73</td>
<td>85</td>
<td>237</td>
</tr>
<tr>
<td>Neural Network</td>
<td>0.769</td>
<td>0.761</td>
<td>74</td>
<td>71</td>
<td>236</td>
</tr>
<tr>
<td>Case #5 (- p = 1, q = 1) (default values) with (CCNThreshold = 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>0.722</td>
<td>0.845</td>
<td>48</td>
<td>101</td>
<td>262</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.735</td>
<td>0.761</td>
<td>74</td>
<td>85</td>
<td>236</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.695</td>
<td>0.713</td>
<td>89</td>
<td>97</td>
<td>221</td>
</tr>
</tbody>
</table>

\(^{12}\) 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 $CCN_{Threshold} = 3$ p had better be set as balanced; however, it does not extend to the models with $CCN_{Threshold} = 1$, 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>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>67%</td>
<td>69%</td>
<td>67%</td>
</tr>
<tr>
<td>balanced</td>
<td>67%</td>
<td>69%</td>
<td>67%</td>
</tr>
<tr>
<td>global</td>
<td>66%</td>
<td>69%</td>
<td>68%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>68%</td>
<td>67%</td>
<td>68%</td>
</tr>
<tr>
<td>balanced</td>
<td>69%</td>
<td>69%</td>
<td>68%</td>
</tr>
<tr>
<td>global</td>
<td>69%</td>
<td>68%</td>
<td>65%</td>
</tr>
</tbody>
</table>

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

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

**CCNThreshold = 1**

![3D graph](image)

**CCNThreshold = 3**

![3D graph](image)

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 CCNThreshold = 1

<table>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>80%</td>
<td>83%</td>
<td>81%</td>
</tr>
<tr>
<td>balanced</td>
<td>77%</td>
<td>85%</td>
<td>83%</td>
</tr>
<tr>
<td>global</td>
<td>81%</td>
<td>83%</td>
<td>83%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>69%</td>
<td>67%</td>
<td>69%</td>
</tr>
<tr>
<td>balanced</td>
<td>72%</td>
<td>70%</td>
<td>71%</td>
</tr>
<tr>
<td>global</td>
<td>68%</td>
<td>70%</td>
<td>65%</td>
</tr>
</tbody>
</table>
Across three leaders in terms of performance whose avg. recall > 70%: Naive Bayes, Logistic Regression, and Neural Network

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

<table>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>77%</td>
<td>77%</td>
<td>77%</td>
</tr>
<tr>
<td>balanced</td>
<td>74%</td>
<td>77%</td>
<td>75%</td>
</tr>
<tr>
<td>global</td>
<td>76%</td>
<td>79%</td>
<td>76%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p</th>
<th>DFS</th>
<th>balanced</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>72%</td>
<td>71%</td>
<td>71%</td>
</tr>
<tr>
<td>balanced</td>
<td>73%</td>
<td>73%</td>
<td>74%</td>
</tr>
<tr>
<td>global</td>
<td>72%</td>
<td>73%</td>
<td>69%</td>
</tr>
</tbody>
</table>
Figure 22  3D graphs built on average recall values for embedding-based models across all algorithms (common legend)
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 $CCN_{Threshold} = 1$ to the one based on $CCN_{Threshold} = 3$. 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 $CCN_{Threshold} = 1$ | Average recall at $CCN_{Threshold} = 3$ | Change in the recall value for the dataset based on $CCN_{Threshold} = 3$ compared to the one with $CCN_{Threshold} = 1$ |
|----------------------------|----------------------------------------|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------
| 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

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCNThreshold = 1</th>
<th>CCNThreshold = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>00s : 66ms</td>
<td>00s : 29ms</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>57s : 82ms</td>
<td>00s : 24ms</td>
</tr>
<tr>
<td>Random Forest</td>
<td>00s : 30ms</td>
<td>00s : 24ms</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>00s : 19ms</td>
<td>00s : 13ms</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>00s : 36ms</td>
<td>00s : 22ms</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>00s : 30ms</td>
<td>00s : 25ms</td>
</tr>
<tr>
<td>Neural Network</td>
<td>02s : 20ms</td>
<td>00s : 33ms</td>
</tr>
</tbody>
</table>

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 $q = \text{balanced}$ 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\(^\text{13}\), 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/adityagrover/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\textsuperscript{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.

\textsuperscript{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\textsuperscript{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].

\textsuperscript{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


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


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.


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:

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

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.

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):

```bash
cpython cochangecount.py
```

Save the output as `cochanges19.csv`.

5) Recover structural dependencies through `dephpend`:

```bash
dehpend 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`:

```bash
./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`:

```bash
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

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

```
#!/bin/sh
#
#prompt:
#use --since=<date> and --after=<date> options of git log to additionally narrow search if needed
#
#SETTINGS
#input variables
MaxNFilesInCommit=20  #commits with more than this number of files will be left out as automated minor changes
#computational variables
Iter=0
NCommitsCopied=0
#move to the directory with the cloned Tiki Git repository
read -p "Provide the local path to the cloned Tiki repository: " GitPath
cd "$GitPath"
#ask how far in the history the user wants to go (all history or X number of latest records; how many?)
while True; do
  read -p "Do you want to extract all history ("1") or a finite number of the most recent commits ("2")?" onetwo
  case $onetwo in
```
[1]* ) ExtractFiniteNumber=False; NCommitsToExtract=0; break;;
[2]* ) ExtractFiniteNumber=True;
while True; do
    read -p "How many?" NCommitsToExtract
    if [[ $NCommitsToExtract =~ ^[0-9]+$ ]]; then
        break
    else
        echo "Please enter a positive integer."
    fi
    done
    break;;
    esac
done
#Cai et al. mention that there is no need to extract commits with one file only, because there are no co-changes in them
#however, it may be useful for some other metrics (e.g., bug frequency). we will make it optional
while True; do
    read -p "Include commits with 1 file only (y/n)?" yn
    case $yn in
        [Yy]* ) DontIncludeOneFileCommits=False; break;;
        [Nn]* ) DontIncludeOneFileCommits=True; break;;
        * ) echo "Please answer yes or no.";;
    esac
done
#DATA EXTRACTION
#copy major statistics on commits, files and changes into a separate file
#we will sift through Git records one by one
CommitHash=$(git log -1 --skip="$Iter" --pretty=format:%H)
while [[ "$CommitHash" != "" ]]
do
    if $ExtractFiniteNumber && [ $NCommitsCopied -eq $NCommitsToExtract ]; then
        echo "Have reached the limit on the number of commits: $NCommitsCopied/$NCommitsToExtract."
        break
    fi
    #we skip unrelated commits with [BRANCH], [DOC], [LANG], [MRG], [OOPS], [TEST], [TRA] labels
    IncludeCommit=0
    labelsarray=() #all labels in a particular commit
CommitTitle=$(git log -1 --skip="$Iter" --pretty=format:%s)

if [[ "$CommitTitle" =~ ' [' ]]; then
  # let's first derive a list of labels from the commit title
  IFS='[' read -r -a difflabels <<< "$CommitTitle"
  for label in "${difflabels[@]}"
  do
    # skip empty elements of $difflabels
    # (if the line starts with ' [', after separating it by ' [ into array (line 59) the first
    # element will be empty)
    # if there is no closing bracket, then we consider it as a commit with no label and include
    # it in the file
    if [[ X"" != X"$label" ]] && [[ $label =~ ' ' ]]; then
      label="$[label]"
      # ensure that the label contains at least one letter
      if [[ "$label" =~ ^[a-zA-Z0-9\ ]+$ ]] && [[ ! "$label" =~ ^[0-9]+$ ]]; then
        # make the label UPPERCASE
        label="${label^^}"
        # cleanup specific to Tiki Git labels:
        # 1- match different labels meaning the same thing
        if [[ $label == MERGE* ]]; then
          label="MRG"
          fi
        # 2- exclude meaningless words (which are not labels) put by committers in [ ]
        brackets
        if [[ $label != *AND* ]] && [[ $label != *FOO* ]] && [[ $label != *FP* ]] && [[ $label != *FROM* ]] && [[ $label != *REVISION* ]] && [[ $label != *THANKS* ]]; then
          labelsarray+=( $label )
          fi
        fi
      done
  fi
  # if there is at least one label which is not in {{BRANCH}, [DOC], [LANG], [MRG], [OOPS],
  [TEST], [TRA]}, then we include the commit
  # 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")
  if [ $[#labelsarray[@]] -ne 0 ]; then
for commitlabel in "${labelsarray[@]}"
    do
        if [[ $commitlabel != BRANCH ]] && [[ $commitlabel != DOC ]] && [[ $commitlabel != LANG ]] && [[ $commitlabel != MRG ]] && [[ $commitlabel != OOPS ]] && [[ $commitlabel != TEST ]] && [[ $commitlabel != TRA ]]; then
            IncludeCommit=1
            break
        fi
    done
if [ $IncludeCommit -eq 0 ]; then
    echo "SKIPPED. Commit: $CommitHash $CommitTitle. Does not contain necessary labels. Commit is skipped."
    Iter=$((Iter+1))
    CommitHash=$(git log -1 --skip="$Iter" --pretty=format:%H)
    continue
fi
fi
#if chosen so, don't copy commits with just 1 file inside
NFilesInCommit=$(git log -1 --skip="$Iter" --shortstat | tail -1 | awk '{print $1;}')
if $DontIncludeOneFileCommits && [ $NFilesInCommit -eq 1 ]; then
    echo "SKIPPED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit. Commit is skipped."
else
    #we do not copy commits with more than MaxNFilesInCommit files inside, as we assume they contain automated minor changes
    if [ $NFilesInCommit -le $MaxNFilesInCommit ]; then
        if [ $NCommitsCopied -eq 0 ]; then
            git log -1 --skip=$Iter --numstat > ~/dataforprocessing.log
        else
            git log -1 --skip=$Iter --numstat >> ~/dataforprocessing.log
        fi
        NCommitsCopied=$((NCommitsCopied+1))
        echo "COPIED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit."
    else
        echo "SKIPPED. Commit: $CommitHash. Number of files in commit: $NFilesInCommit. Commit is skipped."
    fi
cochangecount.py

#script doesn't distinguish moved or renamed files, such as tiki/doc/{tiki09.doc => tiki.doc}, from the rest
import re
import pandas as pd
import datetime

#auxiliary functions

def getadate(quest_num, question):
    isvalid=False
    while not isvalid:
        answer = input("Timestamp " + str(quest_num) + ": " + question)
        if (quest_num == 1) and (answer == "0"):
            d = answer
            isvalid=True
        elif (quest_num == 2) and (answer == "99"):
            d = answer
            isvalid=True
        else:
            try:
                d = datetime.datetime.strptime(answer, "%Y/%m/%d")
                isvalid=True
            except:
                print("Error, check your input!\n")
    return d

def getadir():
    answer = input("Subdirectory name: ")
    if not answer.endswith('/ '):
        answer = answer + "/"
    return answer
def yes_or_no(question):
    answer = input(question).lower().strip()
    print(" ")
    while not(answer == "y" or answer == "yes" or answer == "n" or answer == "no"):
        print("Input y/n or yes/no")
        answer = input(question).lower().strip()
        print(" ")
    if answer[0] == "y":
        return True
    else:
        return False

def is_code_file(filename):
    if (filename[-4:] == ".php") or (filename[-4:] == ".tpl") or (filename[-4:] == ".sql"):
        return True
    else:
        return False

#main part
since = "0"
till = "99"
subdirectory = ""
copycommit = True

#script settings/user input
if yes_or_no("Do you want to specify time interval for co-change count (otherwise the script will process the entire repository)? "): 
    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"): ")
    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"): ")
if yes_or_no("Output co-changes between files located in a specific subdirectory only? "): 
    subdirectory = getadir()
onlycodefiles = yes_or_no("Output co-changes between code files only (.php, .tpl, .sql)? ")

#pattern representing the beginning of a new commit
newcommit = re.compile(r'^commit (.+)')

#pattern representing the line in a commit log, which contains information about the date
dateline = re.compile(r'^Date:  (\.+)$')
#pattern representing the line in a commit log, which contains information about files: 2 columns for numbers of added and deleted lines + 1 column for file name
filename = re.compile(r'^\d+\s+\d+\s+(\.+)$')
#we will aggregate all pairs of files within each separate commit and combine them in a single dataframe (one dataframe for all commits)
filesincommit=[]  #list of files in one particular commit
rows_list = []  #rows of the future dataframe
with open('dataforprocessing.log', encoding="utf8") as extractedcommits:
    for line in extractedcommits:
        if newcommit.search(line):
            if filesincommit!=[]:
                for filei in filesincommit:
                    for filej in filesincommit:
                        if filei != filej:
                            if onlycodefiles:
                                #if user chose to output only co-changes between the code files,
                                #check whether both files have proper extensions
                                if (is_code_file(filei) and is_code_file(filej)):
                                    rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number': 1})
                            else:
                                rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number': 1})
            filesincommit=[]
        elif ((since != "0") or (till != "99")) and dateline.search(line):
            commitdate = datetime.datetime.strptime(line[8:-7], "%a %b %d %H:%M:%S %Y")
            if (since == "0") and (till != "99"):  #since the beginning of the log till some timestamp
                copycommit = (commitdate - datetime.timedelta(seconds=1) < till)  #python doesn't support <=, so worked a way around
                if (since != "0") and (till == "99"):  #since some timestamp till the end of the log
                    #the log is processed from the most recent record to the most old
                    #copy while commitdate => since, but break as soon as the interval is passed:
                    if (commitdate < since):
                        break;
                    else:
#commit should be copied if since <= commitdate <= till,
#but as soon as the interval has passed, break the loop:
if (commitdate < since):
    break;
copycommit = (commitdate - datetime.timedelta(seconds=1) < till)
elif copycommit and fileline.search(line):
    #derive file name from the commit log
    if subdirectory != "":
        #if the user chose to process only a certain subdirectory,
        #check the beginning of the filename
        filename = (re.sub(fileline, r'\3', line)).rstrip('
')
        if filename.startswith(subdirectory):
            filesincommit.append(filename)
else:
    filesincommit.append((re.sub(fileline, r'\3', line)).rstrip('
'))
#process the last commit (for the case since="0", because there is no commit after, see lines 70-71)
if filesincommit != []:
    for filei in filesincommit:
        for filej in filesincommit:
            if filei != filej:
                if onlycodefiles:
                    if (is_code_file(filei) and is_code_file(filej)):
                        rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number': 1})
                else:
                    rows_list.append({'Primary File':filei, 'Secondary File':filej, 'Cc Number': 1})
cochanges = pd.DataFrame(rows_list)
#dataframe has a format:
#Primary File    Secondary File    Cc Number (i.e., co-change number)
#file1           file2             1
#file1           file2             1
#file1           file3             1
#file1           file4             1
#we just have to group the dataframe by 1st and 2nd columns
if cochanges.empty:
    print('No files are found based on input parameters.')  #to prevent falling in error when grouping
    an empty dataframe
else:
cochanges = cochanges.groupby(['Primary File', 'Secondary File'])['Cc Number'].sum()

# copy dataframe to the separate file
with open("cochanges.csv", 'w') as outfile:
    cochanges.to_csv(outfile, header=False, line_terminator='
')

print("See cochanges.csv for further details")

txtdephpend2csv.pl

#!/usr/bin/perl
#
#txtdephpend2csv.pl
open(INFILE, $ARGV[0] . " txt");
open(OUTFILE, " > " . $ARGV[0] . " csv");

# provide a choice of what to use as a delimiter - "," or " ; "
while (1) {
    print "What to use as a delimiter: ", ’ or ’; ’?";
    chomp(my $input = <STDIN>);
    if ($input eq "," or $input eq ";") {
        while (<INFILE>) {
            if (/(.+ \->[\(\)+])/) {
                print OUTFILE "$1$input$2
            }
        }
        last;
    } else {
        print "Please, type ", ’ or ’; ’. \n"
    }
}

close(INFILE);
close(OUTFILE);

class.py

# usage: class.py <directory>
#output: all pairs of class names and corresponding file paths

```python
import sys
import os
import re
import pandas as pd

if len(sys.argv) < 2:
    print("You must set an argument - a path to the project directory!")
    quit()

rootdir = sys.argv[1]
if not rootdir.endswith('/ '):
    rootdir = rootdir + "/ "
rootdirlength = len(rootdir)

#pattern representing a class
isclass = re.compile(r'^class \s+(\w+)')

#pattern representing an abstract class
isabstractclass = re.compile(r'^abstract \s+ class \s+(\w+)')

#pattern representing an interface
isinterface = re.compile(r'^interface \s+(\w+)')

rows_list = []
#rows of the future dataframe
for root, dirs, files in os.walk(rootdir):
    for file in files:
        if file.endswith('.php'):
            fullpath = os.path.join(root, file).replace("\\","/")
            with open(fullpath, encoding="utf8") as sourcefile:
                for line in sourcefile:
                    if isclass.search(line):
                        classname = (re.sub(isclass, r'\1', line)).rstrip().split(' ')[0]
                        rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
                    elif isabstractclass.search(line):
                        classname = (re.sub(isabstractclass, r'\1', line)).rstrip().split(' ')[0]
                        rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})
                    elif isinterface.search(line):
                        classname = (re.sub(isinterface, r'\1', line)).rstrip().split(' ')[0]
                        rows_list.append({'Class':classname, 'File':fullpath[rootdirlength:]})

#create a dataframe with classes to files mapping
mapping = pd.DataFrame(rows_list)

#sort the dataframe by columns Class and File
```

#create a temporary column Class.Upper for sorting to be case-insensitive
mapping["Class.Upper"] = mapping["Class"].str.upper()
mapping.sort_values(by=["Class.Upper", 'File'], inplace=True)
del mapping["Class.Upper"]

#copy dataframe to the separate file
with open("mapping.csv",'w') as outfile:
    mapping.to_csv(outfile, header=False, index=False, line_terminator='\n')

print("See mapping.csv for further details.")
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("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)
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
dependfiles['CCN'] = 0
#add column Status to dataframes
#Status = 1 - denotes a structural dependency
#Status = 2 - denotes an evolutionary coupling
#Status = 3 - denotes both a structural dependency and an evolutionary coupling
dependfiles['Status'] = 1
cochanges['Status'] = 2
mergeddf = dependfiles.append(cochanges, ignore_index=True, sort=False).groupby(['File1','File2'])[['CCN', 'Status']].sum().reset_index()
#as cochangecount.py creates reciprocal pairs of files for each processed commit
#(i.e., for example, for commit with just two files "aal" and "bb2", it will create two rows in a .csv file:
#specifying that "aal" has been changed with "bb2" one time, as well as "bb2" - with "aal" one time),
#when grouping mergeddf, only one of these reciprocal pairs will receive Status=3 (depending on which file depends on which):
#therefore, we have to drop the duplicate with Status=2
#P.S. if grouping the dataframe has led to:
#we should keep both rows with Status=3, since they correspond to one of the hotspot patterns
statusthree = mergeddf.loc[mergeddf['Status'] == 3].copy()
for trow in statusthree.itertuples():
    rowfound = mergeddf.loc[(mergeddf['File1'] == getattr(trow, 'File2')) & (mergeddf['File2'] == getattr(trow, 'File1')) & (mergeddf['Status'] == 2)]
    if rowfound.empty:
        rowfound = mergeddf.loc[(mergeddf['File1'] == getattr(trow, 'File2')) & (mergeddf['File2'] == getattr(trow, 'File1')) & (mergeddf['Status'] == 3)]
    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.
print("Error! Can't find a reciprocal pair for files ", getattr(trow, 'File1'), ", ", 
getattr(trow, 'File2'), ", ", which have status = 3.")

# we had an idea to equate CCN to 0 in one of the rows for the case described above in P.S.,
# assuming that leaving everything as it was would cause doubled CCN for the edge.
# however, it turned out that this is not how Gephi works.
# so we opted for keeping the both rows with the same CCN and the same weight (the same weight is 
especially important)
# the following code is commented out and kept here just in case (other visualization tools may act 
differently)

    if getattr(trow, 'CCN') != 0:
        mergeddf.at[rowfound.index, 'CCN'] = 0
        statusthree.at[rowfound.index, 'CCN'] = 0
    else:
        mergeddf.drop(rowfound.index, inplace=True)

# we also have to delete reciprocal rows, corresponding to Status = 2, as we still have situations like
# File1 | File2 | CCN | Status
#  ccl  | dd2  | 1   | 2
#  dd2  | ccl  | 1   | 2
statustwo = mergeddf.loc[mergeddf['Status'] == 2].copy()
fullindex = statustwo.index
statustwo['Status2_check'] = statustwo.apply(lambda row: ''.join(sorted([row['File1'], row['File2']]))), axis=1
statustwo.drop_duplicates(subset=['Status2_check'], inplace=True)
mergeddf.drop(list(set(fullindex)-set(statustwo.index)), inplace=True)

# calculate weight for structural and logical dependencies
print("Max CCN is ", mergeddf['CCN'].max(), ".")
depweight = int(mergeddf['CCN'].max()/2) # default weight for dependency subgraph

# although we had used the weight function specified above to quantify the impact of a structural 
dependency,
# other functions may be developed to try to balance the importance of the two types of architectural 
relationships
print("The default weight for structural dependency subgraph is ", depweight, ".")
mergeddf['Weight'] = mergeddf.apply(lambda row: row['CCN'] if row['Status'] == 2 else row['CCN'] + depweight, axis=1)
mergeddf.rename(columns={'File1': 'Source', 'File2': 'Target'}, inplace=True)
with open("merged.csv", 'w') as outfile:
    mergeddf.to_csv(outfile, index=False, line_terminator='\n')
print("See merged.csv for data visualization.")

with open("visualization.log", 'w') as logfile:
    print("Couldn't find a match for ", len(notprocessed), " classes: ", ".join(sorted(notprocessed)), ".

dependencies.to_string(logfile)

print("Dependencies dataframe after mapping classes to files (no post-processing, contains all
duplicates and NaN values):

dependencies.to_string(logfile)

print("See visualization.log for the full script output.")

search.py

#usage: search.py <directory>
#clean-up tool: searches for classes, which haven't been matched to files by class.py, in the project
directory
#output: a list of possible locations for manual inspection
import sys
import os
import pandas as pd

if len(sys.argv) < 2:
    print("You must set an argument - a path to the project directory!")
    quit()

rootdir = sys.argv[1]
if not rootdir.endswith('/':
    rootdir = rootdir + '/'

rootdirlength = len(rootdir)
with open('notfound.txt', 'r') as inputfile:
    text = inputfile.read()

class_list = text.split(',')

rows_list = []
    #rows of the future dataframe
for root, dirs, files in os.walk(rootdir):
    for file in files:
        if file.endswith('.php') or file.endswith('.sql') or file.endswith('.tpl'):
            fullpath = os.path.join(root, file).replace('\', '/')
            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

```perl
#!/usr/bin/perl
#
#bugfrequency.pl

my %commits;
my %authors;
my %emails;
my %dates;
my %labels;
my %files;
my %buggyfiles;
my %buggyfilesclean;
my %frequency;

my $n = 99999;  #set $n to a small number for testing
my $commitsnum = 0;
my $commit = "";
my $count = 0;
open(INFILE, $ARGV[0] . ".log");
while (<INFILE>) {
  if (/^commit (.+)/) {
    last unless ($n-- > 0);
    $commit = $1;
    push @commits, $commit;
    $commitsnum++;
  } elsif (/^Author: (.+?) <(.+)>/) {
    #author names can include spaces
    my $author = $1;
    my $email = $2;
    $authors{$commit} = $author;
    $emails{$commit} = $email;
  } elsif (/^Date: (.+)/) {
    my $date = $1;
    $dates{$commit} = $date;
```
$count = 2;

elsif ($count == 2) {
  $count--;
  $labels{$commit} = "";
}

elsif ($count == 1) {
  chomp($_);
  $label = trim($_);
  if ($label =~ /^[I\w\v]+$/i) {
    if ($labels{$commit}) {
      $labels{$commit} .= "\n";
    }
    $labels{$commit} .= $label;
  } elseif ($_ == "") {
    $count--;
  }
}

elif (/^\d+\s+\d+\s+\.+$/) {
  my $file = $3;
  if ($files{$commit}) {
    $files{$commit} .= "\n";
  } else {
    $files{$commit} = ""
  }
  $files{$commit} .= $file;
}

} }

close(INFILE);

open(my $log, '>', $full_log) or die "Could not open file '$full_log' \!";

open(my $out, '>', $output) or die "Could not open file '$output' \!";

print $log "commitsnum commits total\n";

print $log "The list of bug-fixing commits:\n";

my @BuggyCommits = grep { uc($labels$_) =~ /\[(FIX|SEC|BUG)\]/ } @commits;

my $bugcommitsnum = 0;

foreach $i (@BuggyCommits) {
  print $log "$i\n";
  $bugcommitsnum++;
  my @BuggyCommitFiles = split("\n", $files{$i});
foreach $j (@BuggyCommitFiles) {
    # adding escape characters for '{' and '}' to fix perl syntax error
    $jclean = $j;
    $jclean =~ s/\}/\}/g;
    $jclean =~ s/\{/\}/g;
    unless (grep( /^$jclean$/, @buggyfiles)) {
        push @buggyfiles, $j;
        $frequency{$j} = 0;
    }
    $frequency{$j}++;
}

print $log "***OVERALL SUMMARY***
";
print $log "$bugcommitsnum bug-fixing commits total\n";
print $log "The list of unique files in bug-fixing commits with frequencies of occurrence:\n";
my $stattotalnum = 0;
my $statfrequency;
my $package = "";
my $packages;
my $findslash = 0;
foreach $buggyfile (sort @buggyfiles) {
    print $out "$buggyfile, $frequency{$buggyfile}\n";
    print $log "\t$buggyfile, $frequency{$buggyfile}\n";
    $stattotalnum++;
    $findslash = index($buggyfile, '/');
    if ($findslash != -1) {
        $package = substr($buggyfile, 0, $findslash);
    } else {
        $package = "root directory"
    }
    unless (grep( /^$package$/, @packages)) {
        push @packages, $package;
        $statfrequency{$package} = 0;
    }
    $statfrequency{$package}++;
}
print $log "Bug fix distribution among project packages:\n";
my $percentage = 0;

foreach $k (@packages) {
    $percentage = sprintf("%.2f", $statfrequency{$k}*100/$stattotalnum);
    print $log "$k:
        number of unique files = $statfrequency{$k}
        percentage = $percentage%
";
}

close $out;

sub trim {
    my $text = shift;
    $text =~ s/\s+/\s+/g;
    $text =~ s/\s+$/\s+$/g;
    return $text;
}
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:

<table>
<thead>
<tr>
<th>- file</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalized CCN</td>
</tr>
<tr>
<td>normalized dependencies on a file (Dependers - # of files which depend on a file)</td>
</tr>
<tr>
<td>normalized dependencies of a file (Dependees - # of files a file depends on)</td>
</tr>
<tr>
<td>normalized total number of dependencies (Dependers + Dependees)</td>
</tr>
<tr>
<td>normalized number of co-changes with dependers</td>
</tr>
<tr>
<td>normalized number of co-changes with dependees</td>
</tr>
<tr>
<td>normalized bug frequency</td>
</tr>
</tbody>
</table>

Keeping Dependents, 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\(^\text{16}\), 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**

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

\(^{16}\) In our research, added variable \textit{BugProneness} = 0 if \textit{BugFrequency} == 0 else 1 with \textit{values} = 0, 1 for the binary option, and \textit{BugProneness} = 0 if \textit{BugFrequency} == 0 else 1 if \textit{BugFrequency} <= 0.016822429906542057 else 2 if \textit{BugFrequency} <= 0.03551401869158879 else 3, \textit{values} = None, Low, Medium, High for the categorical bug-proneness setting.
based on our data, \( \min(x) = 0 \) for all 6 columns.

therefore, the formula comes down to:

\[
\text{normalized}_x_i = \frac{x_i}{\max(x)}
\]

\( \text{statusone} = \text{merged}.\text{loc}[(\text{merged}[\text{Status}] == 1)].\text{copy()} \)

\( \text{statustwo} = \text{merged}.\text{loc}[(\text{merged}[\text{Status}] == 2)].\text{copy()} \)

\( \text{statusthree} = \text{merged}.\text{loc}[(\text{merged}[\text{Status}] == 3)].\text{copy()} \)

```python
for rows in mlinput.iter tuples():
    filename = getattr(rows, "File")
    #2. CCN (overall number, normalization follows)
    cnn = merged.loc[(merged[\'Source\'] == filename) | (merged[\'Target\'] == filename)][\'CCN\'].sum()
    #however, we need to remember that there are reciprocal rows (doubled file pairs) for Status=3/3 case (see visualization.py*):
    #Source   Target   CCN   Status   Weight
    #aa1      bb2      1     3     33
    #bb2      aa1      1     3     33
    #*when creating merged.csv we've already dropped reciprocal rows for Status=2 and for Status=3/2
    #therefore, we may need to subtract some number from the overall CCN:
    statusthreefile = statusthree.loc[(statusthree[\'Source\'] == filename)]
    for trow in statusthreefile.iter tuples():
        rowfound = statusthree.loc[(statusthree[\'Source\'] == getattr(trow, \'Target\')) & (statusthree[\'Target\'] == get attr(trow, \'Source\'))]
        if not rowfound.empty:
            cnn = cnn - rowfound[\'CCN\'].sum()
    mlinput.at[rows.Index, \'CCN\'] = cnn
    #3. number of dependers (overall number, normalization follows)
    mlinput.at[rows.Index, \'Dependers\'] = len(statusone.loc[(statusone[\'Target\'] == filename)]) + len(statusthree.loc[(statusthree[\'Target\'] == filename)])
    #4. number of dependees (overall number, normalization follows)
    mlinput.at[rows.Index, \'Dependees\'] = len(statusone.loc[(statusone[\'Source\'] == filename)]) + len(statusthree.loc[(statusthree[\'Source\'] == filename)])
    #5. total number of dependencies (overall number, normalization follows)
    mlinput.at[rows.Index, \'Dependencies\'] = mlinput.at[rows.Index, \'Dependers\'] + mlinput.at[rows.Index, \'Dependees\']
    #6. CCN with dependers (overall number, normalization follows)
    #CCN in statusone should always be 0 because of the logic behind Status=1
    #so we only need to focus on statusthree
    mlinput.at[rows.Index, \'DependersCCN\'] = statusthree.loc[(statusthree[\'Target\'] == filename)][\'CCN\'].sum()
```
#7. CCN with dependees (overall number, normalization follows)
mlinput.at[rows.Index,'DependeesCCN'] = statusthree.loc[(statusthree['Source'] == filename)]['CCN'].sum()

#8. bug frequency (overall number, normalization follows)
mlinput.at[rows.Index,'BugFrequency'] = bugs.loc[(bugs['File'] == filename)]['BugsNum'].sum()

# normalize columns 2-8
mlinput['CCN'] = mlinput['CCN']/mlinput['CCN'].max()
mlinput['Dependers'] = mlinput['Dependers']/mlinput['Dependers'].max()
mlinput['Dependees'] = mlinput['Dependees']/mlinput['Dependees'].max()
mlinput['Dependencies'] = mlinput['Dependencies']/mlinput['Dependencies'].max()
mlinput['DependersCCN'] = mlinput['DependersCCN']/mlinput['DependersCCN'].max()
mlinput['DependeesCCN'] = mlinput['DependeesCCN']/mlinput['DependeesCCN'].max()
mlinput['BugFrequency'] = mlinput['BugFrequency']/mlinput['BugFrequency'].max()

# output the dataframe
with open("mlinput.csv","w") as outfile:
    mlinput.to_csv(outfile, index=False, line_terminator='
')
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:

```bash
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 learnt 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):

```bash
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

dnode2vec_input.py

```python
# prepare the input file for node2vec (list of edges forming undirected unweighted graph)
import pandas as pd
import csv

def yes_or_no(question):
    answer = input(question).lower().strip()
    print(" ")
    while not(answer == "y" or answer == "yes" or answer == "n" or answer == "no"):
        print("Input y/n or yes/no")
        answer = input(question).lower().strip()
        print(" ")
    if answer[0] == "y":
        return True
    else:
        return False

def get_threshold(message):
    while True:
        try:
            userInput = int(input(message))
        except ValueError:
            print("Input has to be an integer. Try again.")
        continue
```

130
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
# b  a - to be dropped
# a  b - 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
with open('file-node_mapping_th' + str(CCNthreshold) + '.csv', 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    for key, value in filetoID.items():
        writer.writerow([key, value])
with open('node2vec_input_th' + str(CCNthreshold) + '.edgelist', 'w') as outfile:
    undirectedgraph.to_csv(outfile, sep=' ', header=False, index=False, line_terminator='
')
print("Script has finished its work. See node2vec_input_th" + str(CCNthreshold) + ".edgelist and file-
node_mapping_th" + str(CCNthreshold) + ".csv for details.")

# prepare the input file for machine learning models based on node2vec-derived embeddings
import pandas as pd
from tkinter import filedialog
from tkinter.filedialog import askopenfilename
root = filedialog.Tk()
root.withdraw()
filename = askopenfilename(initialdir = "/", title = "Select file output by node2vec", filetypes = [("emb","*.emb")])
if filename == ": # to process an exception arising if a user doesn't choose a file
    quit()
suffix = filename[filename.rfind("/")+16:-4]
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
# let's first create our metadata by replacing nodeIDs with file names from the appropriate mapping file
# note: mapping file has to be located in the same directory as the current script
if "_th1" in filename:
    filenames = pd.read_csv('file-node_mapping_th1.csv', index_col=1, names = ['File'])
elif "_th3" in filename:
    filenames = pd.read_csv('file-node_mapping_th3.csv', index_col=1, names = ['File'])
else:
    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.")
quit()
fileemb = filenames.join(embeddings)
# now let's add the target
bugs = pd.read_csv('bugs.csv', index_col=0, names=['BugsFreq'])
MLinput = fileemb.join(bugs, on='File')
MLinput['BugsFreq'] = MLinput['BugsFreq'].fillna(0).astype(int)

#provide output
with open("emb_mlinput"+suffix+.csv",'w') as outfile:
    MLinput.to_csv(outfile, index=False, line_terminator='\n')
print("See emb_mlinput"+suffix+.csv for feature output.")
Appendix F: Performance indicators for different embedding-based models, with the best-performing model highlighted

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC(^{17})</th>
<th>CA(^{17})</th>
<th>F1(^{17})</th>
<th>Precision(^{17})</th>
<th>Recall(^{17})</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.652</td>
<td>0.648</td>
<td>0.639</td>
<td>0.656</td>
<td>0.623</td>
<td>117</td>
<td>101</td>
<td>193</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.729</td>
<td>0.655</td>
<td>0.640</td>
<td>0.669</td>
<td>0.613</td>
<td>120</td>
<td>94</td>
<td>190</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.596</td>
<td>0.550</td>
<td>0.551</td>
<td>0.550</td>
<td>0.552</td>
<td>139</td>
<td>140</td>
<td>171</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.603</td>
<td>0.603</td>
<td>0.595</td>
<td>0.607</td>
<td>0.584</td>
<td>129</td>
<td>117</td>
<td>181</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.804</td>
<td>0.742</td>
<td>0.757</td>
<td>0.716</td>
<td>0.803</td>
<td>61</td>
<td>99</td>
<td>249</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.830</td>
<td>0.753</td>
<td>0.751</td>
<td>0.757</td>
<td>0.745</td>
<td>79</td>
<td>74</td>
<td>231</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.842</td>
<td>0.765</td>
<td>0.763</td>
<td>0.768</td>
<td>0.758</td>
<td>75</td>
<td>71</td>
<td>235</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.623</td>
<td>0.644</td>
<td>0.644</td>
<td>0.644</td>
<td>0.644</td>
<td>99</td>
<td>99</td>
<td>199</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.795</td>
<td>0.728</td>
<td>0.718</td>
<td>0.747</td>
<td>0.691</td>
<td>86</td>
<td>65</td>
<td>192</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.616</td>
<td>0.585</td>
<td>0.593</td>
<td>0.581</td>
<td>0.604</td>
<td>110</td>
<td>121</td>
<td>168</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.651</td>
<td>0.651</td>
<td>0.656</td>
<td>0.647</td>
<td>0.665</td>
<td>93</td>
<td>101</td>
<td>185</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.781</td>
<td>0.732</td>
<td>0.720</td>
<td>0.753</td>
<td>0.691</td>
<td>86</td>
<td>63</td>
<td>192</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.798</td>
<td>0.739</td>
<td>0.731</td>
<td>0.755</td>
<td>0.709</td>
<td>81</td>
<td>64</td>
<td>197</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.843</td>
<td>0.759</td>
<td>0.755</td>
<td>0.767</td>
<td>0.745</td>
<td>71</td>
<td>63</td>
<td>207</td>
</tr>
</tbody>
</table>

Case #2 - p=1, q=0.5 (balanced DFS)

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC(^{17})</th>
<th>CA(^{17})</th>
<th>F1(^{17})</th>
<th>Precision(^{17})</th>
<th>Recall(^{17})</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.600</td>
<td>0.606</td>
<td>0.595</td>
<td>0.613</td>
<td>0.577</td>
<td>131</td>
<td>113</td>
<td>179</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.728</td>
<td>0.674</td>
<td>0.663</td>
<td>0.686</td>
<td>0.642</td>
<td>111</td>
<td>91</td>
<td>199</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.597</td>
<td>0.573</td>
<td>0.579</td>
<td>0.571</td>
<td>0.587</td>
<td>128</td>
<td>137</td>
<td>182</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.624</td>
<td>0.624</td>
<td>0.635</td>
<td>0.617</td>
<td>0.655</td>
<td>107</td>
<td>126</td>
<td>203</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.792</td>
<td>0.719</td>
<td>0.733</td>
<td>0.699</td>
<td>0.771</td>
<td>71</td>
<td>103</td>
<td>239</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.817</td>
<td>0.735</td>
<td>0.733</td>
<td>0.740</td>
<td>0.726</td>
<td>85</td>
<td>79</td>
<td>225</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.829</td>
<td>0.735</td>
<td>0.732</td>
<td>0.742</td>
<td>0.723</td>
<td>86</td>
<td>78</td>
<td>224</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.650</td>
<td>0.664</td>
<td>0.665</td>
<td>0.662</td>
<td>0.669</td>
<td>92</td>
<td>95</td>
<td>186</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.818</td>
<td>0.728</td>
<td>0.722</td>
<td>0.740</td>
<td>0.705</td>
<td>82</td>
<td>69</td>
<td>156</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.660</td>
<td>0.606</td>
<td>0.607</td>
<td>0.606</td>
<td>0.608</td>
<td>109</td>
<td>110</td>
<td>169</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.655</td>
<td>0.655</td>
<td>0.653</td>
<td>0.656</td>
<td>0.651</td>
<td>97</td>
<td>95</td>
<td>181</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.794</td>
<td>0.745</td>
<td>0.737</td>
<td>0.760</td>
<td>0.716</td>
<td>79</td>
<td>63</td>
<td>199</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.769</td>
<td>0.737</td>
<td>0.733</td>
<td>0.746</td>
<td>0.719</td>
<td>78</td>
<td>68</td>
<td>200</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.839</td>
<td>0.764</td>
<td>0.763</td>
<td>0.767</td>
<td>0.759</td>
<td>67</td>
<td>64</td>
<td>211</td>
</tr>
</tbody>
</table>

Case #3 - p=2, q=0.5 (global DFS)

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC(^{17})</th>
<th>CA(^{17})</th>
<th>F1(^{17})</th>
<th>Precision(^{17})</th>
<th>Recall(^{17})</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.562</td>
<td>0.569</td>
<td>0.563</td>
<td>0.571</td>
<td>0.555</td>
<td>138</td>
<td>129</td>
<td>172</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.738</td>
<td>0.663</td>
<td>0.655</td>
<td>0.671</td>
<td>0.639</td>
<td>112</td>
<td>97</td>
<td>198</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.612</td>
<td>0.569</td>
<td>0.573</td>
<td>0.568</td>
<td>0.577</td>
<td>131</td>
<td>136</td>
<td>179</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.616</td>
<td>0.616</td>
<td>0.592</td>
<td>0.631</td>
<td>0.558</td>
<td>137</td>
<td>101</td>
<td>238</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.784</td>
<td>0.737</td>
<td>0.754</td>
<td>0.708</td>
<td>0.806</td>
<td>60</td>
<td>103</td>
<td>250</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.813</td>
<td>0.732</td>
<td>0.736</td>
<td>0.726</td>
<td>0.745</td>
<td>79</td>
<td>87</td>
<td>231</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.836</td>
<td>0.748</td>
<td>0.745</td>
<td>0.755</td>
<td>0.735</td>
<td>82</td>
<td>74</td>
<td>156</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.670</td>
<td>0.674</td>
<td>0.675</td>
<td>0.674</td>
<td>0.676</td>
<td>90</td>
<td>91</td>
<td>188</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.785</td>
<td>0.703</td>
<td>0.692</td>
<td>0.720</td>
<td>0.665</td>
<td>93</td>
<td>72</td>
<td>185</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.677</td>
<td>0.622</td>
<td>0.622</td>
<td>0.622</td>
<td>0.622</td>
<td>105</td>
<td>105</td>
<td>185</td>
</tr>
</tbody>
</table>

\(^{17}\) With regard to target class = bug-prone.
<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC</th>
<th>CA</th>
<th>F1</th>
<th>Precision</th>
<th>Recall</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.687</td>
<td>0.687</td>
<td>0.691</td>
<td>0.682</td>
<td>0.701</td>
<td>83</td>
<td>91</td>
<td>195</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.785</td>
<td>0.728</td>
<td>0.713</td>
<td>0.755</td>
<td>0.676</td>
<td>90</td>
<td>61</td>
<td>188</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.803</td>
<td>0.745</td>
<td>0.742</td>
<td>0.750</td>
<td>0.734</td>
<td>74</td>
<td>68</td>
<td>204</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.855</td>
<td>0.781</td>
<td>0.774</td>
<td>0.798</td>
<td>0.752</td>
<td>69</td>
<td>53</td>
<td>209</td>
</tr>
</tbody>
</table>

**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)**
<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC^17</th>
<th>CA^17</th>
<th>F1^17</th>
<th>Precision^17</th>
<th>Recall^17</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.621</td>
<td>0.611</td>
<td>0.595</td>
<td>0.621</td>
<td>0.571</td>
<td>133</td>
<td>108</td>
<td>177</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.764</td>
<td>0.687</td>
<td>0.671</td>
<td>0.707</td>
<td>0.639</td>
<td>112</td>
<td>82</td>
<td>198</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.620</td>
<td>0.584</td>
<td>0.590</td>
<td>0.581</td>
<td>0.600</td>
<td>124</td>
<td>134</td>
<td>186</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.618</td>
<td>0.618</td>
<td>0.604</td>
<td>0.626</td>
<td>0.584</td>
<td>129</td>
<td>108</td>
<td>181</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.787</td>
<td>0.732</td>
<td>0.751</td>
<td>0.702</td>
<td>0.806</td>
<td>60</td>
<td>106</td>
<td>250</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.810</td>
<td>0.750</td>
<td>0.754</td>
<td>0.743</td>
<td>0.765</td>
<td>73</td>
<td>82</td>
<td>237</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.817</td>
<td>0.747</td>
<td>0.741</td>
<td>0.758</td>
<td>0.726</td>
<td>85</td>
<td>72</td>
<td>225</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.637</td>
<td>0.667</td>
<td>0.663</td>
<td>0.672</td>
<td>0.655</td>
<td>96</td>
<td>89</td>
<td>182</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.809</td>
<td>0.734</td>
<td>0.718</td>
<td>0.764</td>
<td>0.676</td>
<td>90</td>
<td>58</td>
<td>188</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.670</td>
<td>0.631</td>
<td>0.631</td>
<td>0.632</td>
<td>0.629</td>
<td>103</td>
<td>102</td>
<td>175</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.691</td>
<td>0.691</td>
<td>0.693</td>
<td>0.688</td>
<td>0.698</td>
<td>84</td>
<td>88</td>
<td>194</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.777</td>
<td>0.714</td>
<td>0.707</td>
<td>0.725</td>
<td>0.691</td>
<td>86</td>
<td>73</td>
<td>192</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.782</td>
<td>0.710</td>
<td>0.708</td>
<td>0.714</td>
<td>0.701</td>
<td>83</td>
<td>78</td>
<td>195</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.837</td>
<td>0.763</td>
<td>0.757</td>
<td>0.774</td>
<td>0.741</td>
<td>72</td>
<td>60</td>
<td>206</td>
</tr>
</tbody>
</table>

Case #8 - p=1, q=2 (balanced BFS)

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC^17</th>
<th>CA^17</th>
<th>F1^17</th>
<th>Precision^17</th>
<th>Recall^17</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.613</td>
<td>0.611</td>
<td>0.609</td>
<td>0.612</td>
<td>0.606</td>
<td>122</td>
<td>119</td>
<td>188</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.743</td>
<td>0.694</td>
<td>0.679</td>
<td>0.713</td>
<td>0.648</td>
<td>109</td>
<td>81</td>
<td>201</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.581</td>
<td>0.558</td>
<td>0.549</td>
<td>0.560</td>
<td>0.539</td>
<td>143</td>
<td>131</td>
<td>167</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.634</td>
<td>0.634</td>
<td>0.630</td>
<td>0.637</td>
<td>0.623</td>
<td>117</td>
<td>110</td>
<td>193</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.814</td>
<td>0.750</td>
<td>0.768</td>
<td>0.717</td>
<td>0.826</td>
<td>54</td>
<td>101</td>
<td>256</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.810</td>
<td>0.716</td>
<td>0.720</td>
<td>0.711</td>
<td>0.729</td>
<td>84</td>
<td>92</td>
<td>226</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.804</td>
<td>0.713</td>
<td>0.710</td>
<td>0.717</td>
<td>0.703</td>
<td>92</td>
<td>86</td>
<td>218</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.667</td>
<td>0.667</td>
<td>0.657</td>
<td>0.678</td>
<td>0.637</td>
<td>101</td>
<td>84</td>
<td>177</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.792</td>
<td>0.730</td>
<td>0.713</td>
<td>0.762</td>
<td>0.669</td>
<td>92</td>
<td>58</td>
<td>186</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.587</td>
<td>0.583</td>
<td>0.581</td>
<td>0.583</td>
<td>0.579</td>
<td>117</td>
<td>115</td>
<td>161</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.655</td>
<td>0.655</td>
<td>0.664</td>
<td>0.646</td>
<td>0.683</td>
<td>88</td>
<td>104</td>
<td>190</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.788</td>
<td>0.743</td>
<td>0.733</td>
<td>0.763</td>
<td>0.705</td>
<td>82</td>
<td>61</td>
<td>196</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.796</td>
<td>0.727</td>
<td>0.724</td>
<td>0.732</td>
<td>0.716</td>
<td>79</td>
<td>73</td>
<td>199</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.867</td>
<td>0.802</td>
<td>0.801</td>
<td>0.807</td>
<td>0.795</td>
<td>57</td>
<td>53</td>
<td>221</td>
</tr>
</tbody>
</table>

Case #9 - p=4, q=2 (global BFS)

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>CCN Threshold</th>
<th>AUC^17</th>
<th>CA^17</th>
<th>F1^17</th>
<th>Precision^17</th>
<th>Recall^17</th>
<th>FN</th>
<th>FP</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>1</td>
<td>0.671</td>
<td>0.660</td>
<td>0.645</td>
<td>0.674</td>
<td>0.619</td>
<td>118</td>
<td>93</td>
<td>192</td>
</tr>
<tr>
<td>Random Forest</td>
<td>1</td>
<td>0.735</td>
<td>0.677</td>
<td>0.669</td>
<td>0.687</td>
<td>0.652</td>
<td>108</td>
<td>92</td>
<td>202</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>1</td>
<td>0.627</td>
<td>0.585</td>
<td>0.597</td>
<td>0.581</td>
<td>0.613</td>
<td>120</td>
<td>137</td>
<td>190</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>1</td>
<td>0.618</td>
<td>0.618</td>
<td>0.604</td>
<td>0.626</td>
<td>0.584</td>
<td>129</td>
<td>108</td>
<td>181</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>1</td>
<td>0.818</td>
<td>0.745</td>
<td>0.764</td>
<td>0.711</td>
<td>0.826</td>
<td>54</td>
<td>104</td>
<td>256</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>1</td>
<td>0.819</td>
<td>0.734</td>
<td>0.733</td>
<td>0.735</td>
<td>0.732</td>
<td>83</td>
<td>82</td>
<td>227</td>
</tr>
<tr>
<td>Neural Network</td>
<td>1</td>
<td>0.828</td>
<td>0.747</td>
<td>0.742</td>
<td>0.756</td>
<td>0.729</td>
<td>84</td>
<td>73</td>
<td>226</td>
</tr>
<tr>
<td>Tree</td>
<td>3</td>
<td>0.642</td>
<td>0.655</td>
<td>0.658</td>
<td>0.651</td>
<td>0.665</td>
<td>93</td>
<td>99</td>
<td>185</td>
</tr>
<tr>
<td>Random Forest</td>
<td>3</td>
<td>0.790</td>
<td>0.705</td>
<td>0.693</td>
<td>0.723</td>
<td>0.665</td>
<td>93</td>
<td>71</td>
<td>185</td>
</tr>
<tr>
<td>CN2 Rule Induction</td>
<td>3</td>
<td>0.649</td>
<td>0.606</td>
<td>0.584</td>
<td>0.618</td>
<td>0.554</td>
<td>124</td>
<td>95</td>
<td>154</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>3</td>
<td>0.648</td>
<td>0.647</td>
<td>0.642</td>
<td>0.652</td>
<td>0.633</td>
<td>102</td>
<td>94</td>
<td>176</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>3</td>
<td>0.775</td>
<td>0.718</td>
<td>0.696</td>
<td>0.753</td>
<td>0.647</td>
<td>98</td>
<td>59</td>
<td>180</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>3</td>
<td>0.786</td>
<td>0.728</td>
<td>0.721</td>
<td>0.741</td>
<td>0.701</td>
<td>83</td>
<td>68</td>
<td>195</td>
</tr>
<tr>
<td>Neural Network</td>
<td>3</td>
<td>0.834</td>
<td>0.750</td>
<td>0.741</td>
<td>0.768</td>
<td>0.716</td>
<td>79</td>
<td>60</td>
<td>199</td>
</tr>
</tbody>
</table>