A graph based end to end defect prediction framework.

Abir M'Baya

N° d'ordre NNT : 2022LYSE2035

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

Opérée au sein de

L'UNIVERSITÉ LUMIÈRE LYON 2

École Doctorale : ED 512 Informatique et Mathématiques

Discipline : Informatique

Soutenue publiquement le 6 juillet 2022, par :

Abir M'BAYA

A graph based end to end defect prediction framework.

Devant le jury composé de :

Hervé PANETTO, Professeur des universités, Université de Lorraine, Président

Teresa GONÇALVES, Professeure des universités, Université d'Evora, Rapporteure

Virginie GOEPP, Maîtresse de conférences HDR, INSA Strasbourg, Rapporteure

Sebti FOUFOU, Professeur des universités, Université de Bourgogne, Examinateur

Nejib MOALLA, Professeur des universités, Université Lumière Lyon 2, Directeur de thèse

Contrat de diffusion

THESE de DOCTORAT DE L'UNIVERSITÉ DE LYON

Opérée au sein de

L'UNIVERSITÉ LUMIÈRE LYON 2

École Doctorale : ED 512 Informatique et Mathématiques

Discipline : Informatique

Soutenue publiquement le 06 juillet 2022, par :

Abir M'BAYA

A GRAPH BASED END TO END DEFECT PREDICTION FRAMEWORK

Devant le jury composé de :

Mme Virginie Goepp, Maître de Conférences-HDR à l'INSA de Strasbourg, Rapporteur
Mme Teresa Gonçalves, Professeur d'Université à l'Université l'Evora (Portugal), Rapporteur
M. Sebti Foufou, Professeur des Universités à l'Université de Bourgogne, Examinateur
M. Hervé Panetto, Professeur des Universités à l'Université de Lorraine, Examinateur
M. Néjib Moalla, Professeur des Universités à l'Université Lumière Lyon 2, Directeur de thèse

Acknowledgments

I would like to express my deep gratitude towards everyone who consistently supported me during my PHD thesis period.

Firstly, I would like to offer my sincerest thanks to my supervisor Prof. Néjib MOALLA for his constant encouragement and support throughout my thesis period. I am extremely grateful for the freedom and the confidence he gave me, which boosted my self-confidence to continue research in this filed in future. His valuable advice and insights were essential for the completion of this dissertation.

I am thankful for the committee members consisting of Prof. Virginie GOEPP, Prof. Teresa GONCALVES, Prof. Sebti FOUFOU, and Prof. Hervé PANETTO who have spent their precious time for reviewing my thesis and providing constructive feedback that ameliorated my thesis.

Thanks to all professors and colleagues in our research laboratory, DISP for a friendly atmosphere, finest working conditions, and ideal research environment.

Finally, I am indebted to my dear parents for their strong belief in education and their unconditional support and immense love over the years. Their love was and remains an important drive that inspired me to complete this thesis. I would also like to thank my younger sister, my brother and my brother-in law for their undying encouragement and help. They are the persons that I can always rely on. And lastly, thanks to my beloved husband for his unconditional love, moral support, care, sacrifice and patience that motivated me to overcome any obstacles. I feel so thankful and lucky to have you in my life. Thank you for

allowing me to reach my dreams and objectives and be as challenging as I wanted. I am grateful to my lovely children for their patience. I appreciate them for making my days bright with humorous jokes that help me to forget my tiredness.

Abstract

Software defect prediction is one of the most explored research topics in software engineering. Modern software applications are often overly complicated and prone to failures. Software defect prediction (SDP) can alert on the risk of failure of a software component in the initial stages of development and help developers to appropriately schedule and prioritize their test efforts, reduce costs, and ensure software quality. Traditional statistical software defect prediction tools are always time-consuming and ineffective. We argue that machine learning algorithms with its ability in learning, classification, knowledge representation, etc. can capture useful properties of code that are difficult to extract by humans or other alternative research methods. However, the performance of machine learning tools varies depending on the quality of input data. Since the programming languages of modern applications hold increasingly complex characteristics which are difficult to understand, it is a prerequisite to provide a powerful representation of code analysis that can explore deeply the code software artifacts and capture useful information from different levels of abstraction of the programs. For these reasons, many efforts have been made to propose an efficient defect prediction tool, but the achievements do not represent yet high performance.

In this thesis, we focus on software defect prediction and propose a novel deep learning-based technique to enhance existing defect prediction approaches. To build predictive models, previous studies focused on classic machine learning algorithms and handcrafted traditional features (i.e., software metrics). The software metrics are designed manually to capture the static properties of the code. Such methods are time-consuming and inaccurate since they fail to capture the semantic meanings of programs. Recently, researchers exploited deep learning algorithms based on either tree representations of programs or precise graphs representing program execution flows. However, these models do not offer high performance and do not cover all types of bugs. They often fail to capture intra-procedural dependencies. Indeed, several bugs are related to these dependencies. Such information is important in modelling program functionality and can lead to a more accurate defect prediction.

The training procedure requires a sufficient historical data from a project to build a prediction model. Therefore, it is not practical for new projects, which have no or not enough historical

data. An alternative solution is to train a prediction model by using data from other projects. The traditional approaches are based on metrics to select appropriate projects whose characteristics are close to the new project. However, the metrics are not enough to capture meaningful information from projects and then choose the best candidates that generalize well the new project. The differences between projects in several aspects such as the architecture, developer experience, coding style, the functional, etc. makes the selection task more complicated.

In this thesis, the emphasis was placed on two main tasks: First, to bridge the gap between programs' dependencies and defect prediction features, we propose an end-to-end deep learning algorithm to learn a powerful code representation including different levels of abstractions of code such as the syntax, the semantic and the dependencies automatically from code and further train and construct defect prediction classifier by using these complex features. The experimental results indicate that our approach can significantly improve the existing defect prediction approaches. Second, we propose a novel method to choose the best candidate projects for the project that lacks historical data. We evaluate the effectiveness of our method on 10 open-source projects. Results show that selecting carefully the projects can boost the performance of existing techniques and even of our proposed defect prediction framework, which considers all the other available projects and does not involve any selection strategy of projects.

Keywords: Defect Prediction; Deep Learning; Code Property Graph; Graph Convolutional Neural Network; Abstract Syntax Tree; Control Flow Graph; Program Dependency Graph; Program Analysis.

Table of contents

List of figures

List of Tables

list of abbreviations and acronyms

ANN	Artificial Neural Networks
ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
Bi-LSTM	Bidirectional Long Short-Term Memory
BLSTM	Bi-directional Long Short-Term Memory Neural Network
BTS	Bug Tracking System
CBO	Coupling Between Object
CCDP	Change-Level Cross-Project Defect Prediction
CDG	Control Dependency Graph
CFG	Control Flow Graphs
CNN	Conventional Neural Network
CPDP	Cross-Project Defect Prediction
DBN	Deep Belief Network
DDG	Data Dependency Graph
DGCL	Disordered Graph Conventional Layer
DGCNN	Deep Graph Convolutional Neural Network
DP-CNN	Defect Prediction via Convolutional Neural Network
DIT	Depth of Inheritance Tree
DL	Deep learning
DT	Decision Tree
G-CNN	Graph Conventional Neural Network
GCNs	Graph Conventional Networks
ITS	Issue Tracking System
JIT-DP	Just-In-Time Defect Prediction
JIT-SDP	Just-In-Time Software Defect Prediction

LCOM	Lack of Cohesion in Methods
LOC	Lines Of Code
LR	Logistic Regression
LSTM	Long Short-Term Memory
MLP	Multi-Layer Perceptron
NB	Naive Bayes
NN	Neural Network
PNN	Probabilistic Neural Network
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SDAEs	Stacked Denoising Autoencoders
SE	Software Engineering
SVM	Support Vector Machine
TB-LSTM	Tree-Based Long Short-Term Neural Network
TDS	Training Data Selection
VCS	Version Control System
WPDP	Within-Project Defect Prediction

FIRST CHAPTER

1 INTRODUCTION

1.1 Research context

Modern software applications are increasingly subject to failures during releases of their upgrades due to the growth of software scale and complexity. In fact, developers should constantly make modifications to these software applications by including new functionalities or fix defects to ensure their corrective and developing maintenance. However, these modifications can complicate the software system further and give rise to the introduction of new defects. Defects arising from that particular software have a significant impact on business credibility and may lead to fatal consequences such as loss of time and additional costs, and even health problems in critical software. According to a published survey [1], [2], worldwide software management spent about $3.7 trillion in 2013 and 23 % of this cost was attributed to quality assurance and testing. Moreover, the authors of this study [3] estimated that the bugs occurred in the software application cost the US economy about $59.5 billion every year, and improving the testing procedure can save more than a third of such amount. A noticeable example that shows the importance of effective testing phase would be the $125 million NASA spacecraft which was lost in space because of a small data conversion bug [4]. Similarly, another example of a critical system is the Therac-25, a radiation therapy machine for dealing with cancer patients. Bugs introduced in the control system of Thrac-25 in 1980s resulted in increased time for producing radiation leading to several injuries and even deaths. All these statistics highlight the importance of quality assurance and software testing.

Due to the scarce resources, program budgets, and tight release schedules, the inspection of the entire code source is often challenging, and testing all units is not practical. To ensure high software quality and reduce costs, early prediction of defects is often necessary. Furthermore, detecting a software bug after the commissioning of the software system is 100 times more costly than detecting it during development [5]. Thus, localizing, and fixing

defects at an earlier stage become an urgent requirement to improve the software quality and maximize customer satisfaction. To this purpose, software defect prediction is used to predict whether a source code artifact contains defects in the initial stages of development. This helps the developers to appropriately rank software components for inspection. Based on such ranks, developers expend less effort in inspecting the software components' source code that is potentially defective.

The challenges behind the defect prediction subject have made it an interest research area since the beginning of the software era. Firstly, researchers are contributing their efforts to mainly improve the effectiveness of the predictive models (i.e. having a higher accuracy in prediction) to quickly narrow down the most likely defective parts of software codebase [6]–[8] at coarse granularity levels such as file [7], module; or package [8]–[11]. In this field, several studies perform prediction at file-level. This means that they build predictive models by analysing a training software history data in previous releases and use the developed model to predict whether files in future releases are prone to defects or not. Then, the research defect prediction studies are driven towards performing finer-grained level defect prediction which is represented by the so-called just-in-time software defect prediction (JIT-SDP) (i.e., short-term prediction at commit level, line level, etc). With this strategy, developers can have immediate feedback [12] and quickly narrow down the most likely defective commits/lines/etc.

In practice, software defect prediction cannot work well in certain cases like for legacy systems or new projects which lack data history or has no data at all. A promising and popular solution, known as cross-project defect prediction (CPDP); is proposed to deal with the shortage of training data. The underlying idea behind the cross project is to build the training model by using historical data from other projects (i.e., source projects), and then predict the defects in the prevalent project (or target project) which has insufficient data. Cross-project defect prediction has become a recent trend in software defect prediction [13]–[15], and many researchers are been investigating ways to improve its prediction performance which is always low. This is due to the differences between source and target projects in terms of domain, architecture, coding style, programming language, and developer experience. Zimmerman et al. [16] conducted 622 cross-project predictions among 12 real applications and concluded that only 21 experiments (about 3.4% predictions) could reach better performance. Therefore, several questions have been arisen in how to meet the challenges posed by CPDP and improve its effectiveness. Recommended solutions included selecting

suitable source projects, instead of randomly choosing one or a set of sources projects and proposing methods that minimize the data distribution difference between the source and target projects as well as selecting relevant features and improving the prediction model or classifier.

1.2 Traditional approaches and limitation

The defect prediction studies fall into two main directions: firstly, by applying metric-based methods which manually design software metrics to extract features (predictors) from source artifacts (i.e., files, changes, packages, etc.) and investigating many machine learning algorithms to build predictive models on the metric data and discriminate defective code from non-defective code. Secondly, by using either tree-based program representation or control flow graph-based representation and deep learning networks, thereby, applying them automatically to learn distinguishing features from either trees or graphs.

Metric-based techniques mainly focus on designing manually and arbitrarily discriminative features or a new combination of features called software metrics to measure some properties of source code. For example, Halstead metric based on numbers of operators and operands [17]; McCabe's metric estimates the complexity of a program by assessing its control flow graph [18] and; CK metrics that are based on function and inheritance counts [19]. Moreover, process metrics quantify many aspects of historical development archived in software repositories (version control and bug tracking systems) [20]–[22] such as code metric churn code [23], code entropy [24], change churn [25] and, change features [9] based on a number of lines of code added or, removed. Although several robust learning algorithms such as Naive Bayes (NB), Decision Tree (DT), Dictionary Learning [10], Support Vector Machine (SVM), and Neural Net-work (NN) are applied for software defect prediction, these predictors have not achieved high performance [11] since they are based on metrics which have several definitions and ambiguous counting and are manually and arbitrarily selected by each researcher. Also, all the above-mentioned metrics do not reveal the syntax and semantics of the code.

Thereafter, the input data provided to the classifier no longer concern traditional metrics but represent the syntax and semantic elements of the program by exploiting tree representation

of programs – The Abstract Syntax Trees (ASTs). Then, deep neural networks are applied to automatically learn to distinguish features from ASTs since their architecture can effectively capture complex non-linear features. Tree-based methods significantly outperform metrics-based software. Wang et al. [12] leverages a deep belief network (DBN) in learning semantic features from token vectors extracted from programs‟ ASTs. Li et al. [26] proposed a tree-based convolutional neural network to extract structural information of ASTs to improve defect prediction.

The AST-based methods are flawed as they do not reveal all the types of software defects in the programs, especially those induced by the execution process of programs. AV Phan et al. [27] proposed an application of a graphical data structure namely control flow graphs (CFG) to SDP. In the field of machine learning, the quality of input data directly affects the performance of classifiers. Considering this, CFG provides enhanced results relative to previous studies based on metrics and ASTs.

Although CFGs perform well, they are only able to capture the execution process within a program and do not identify the intra-procedural dependencies. In other words, they cannot capture the behaviour of the program. However, many bugs are directly related to the dependencies within the program [20], [28]. A recent study on file-level proved that syntax and semantics are not enough to cover several types of bugs, thus, suggesting the combination of semantic and structural features to improve the prediction accuracy [29]. Structural features are related to the dependency information in the programs. To conclude, both AST and CFG features do not cover all the types of defects in programs to respond to the constant evolution of software programs in terms of complexity.

A code program with different dependencies between data can have the same semantics and syntax. For example, in Figure 1, an implementation of a simple functionality in a human resources context whose purpose is to compute the salary increase percentage is illustrated. The value of the raise variable should be assigned to the display function. However, it is missing in file1.java. Thus, it will never be displayed on the users' screen. This is obviously a logical bug, which can happen in real cases just as it did at McDonald's[1]. From a technical point of view, the raise variable's value is assigned but never used, making it a dead assignment. In general, this weakness could be an indication of a significant logic error in the program or a deprecated variable that was not removed and is an indication inferior quality of

[1] https://www.mbs.news/a/2020/02/a-bug-made-orders-for-mcdonalds-in-france-practically-free-videos.html

assessment. Figure 1 depicts two Java files file1.java and file2.java, both having the same syntax and sematic. Thus, using traditional features to represent these two code snippets such as process metrics or static code metrics or AST have identical feature vectors. However, dependency information is different. Features that can discriminate such structural differences should have significant impact on the improvement of prediction accuracy. Taking the example in Figure 1, features that ensure that any variable assigned in the program has a dependency relationship, and so it is used by another instruction should be meaningful. It is therefore important to highlight the dependencies of data or of control in the program, allowing the deep learning algorithm to learn all the failures related to the dependencies. Such information may help to select expressive features for defect prediction. Specifically, it may be significant for selecting defective artifacts (files, packages, changes, etc.) and ameliorate the defect prediction process.

Deep learning has proved its efficiency in developing more accurate defect prediction models by leveraging selected expressive features generated automatically from the source code. These features are used to train and construct the defect prediction models [30]–[32]. However, the existing prediction models do not provide an optimum performance whether at file or change level. Moreover, in the case of cross project, the problem of defect prediction is generally considered as a specific case of transfer learning which aims to retrieve knowledge from the training data (i.e., one or a set of source projects) and transfer it to a target project. However, the majority of the existing cross project approaches does not mimic any strategy to select the suitable source projects for the target project, which can lead to impairment of the performance and the effectiveness of cross-project defect prediction.

File1.java

```
1.  public double SalaryIncrease(double increase) {
2.      double oldSalary = getEmployeSalary();
3.      if (salary > 0){
4.          double raise = (increase/oldSalary)*100;
5.          print("Salary increase: " + "%");
6.      }
7.  }
```

File2.java

```
1.  public double SalaryIncrease (double increase) {
2.      double oldSalary = getEmployeSalary();
3.      if (salary > 0){
4.          double raise = (increase/oldSalary)*100;
5.          print("Salary increase: " + raise + "%");
6.      }
7.  }
```

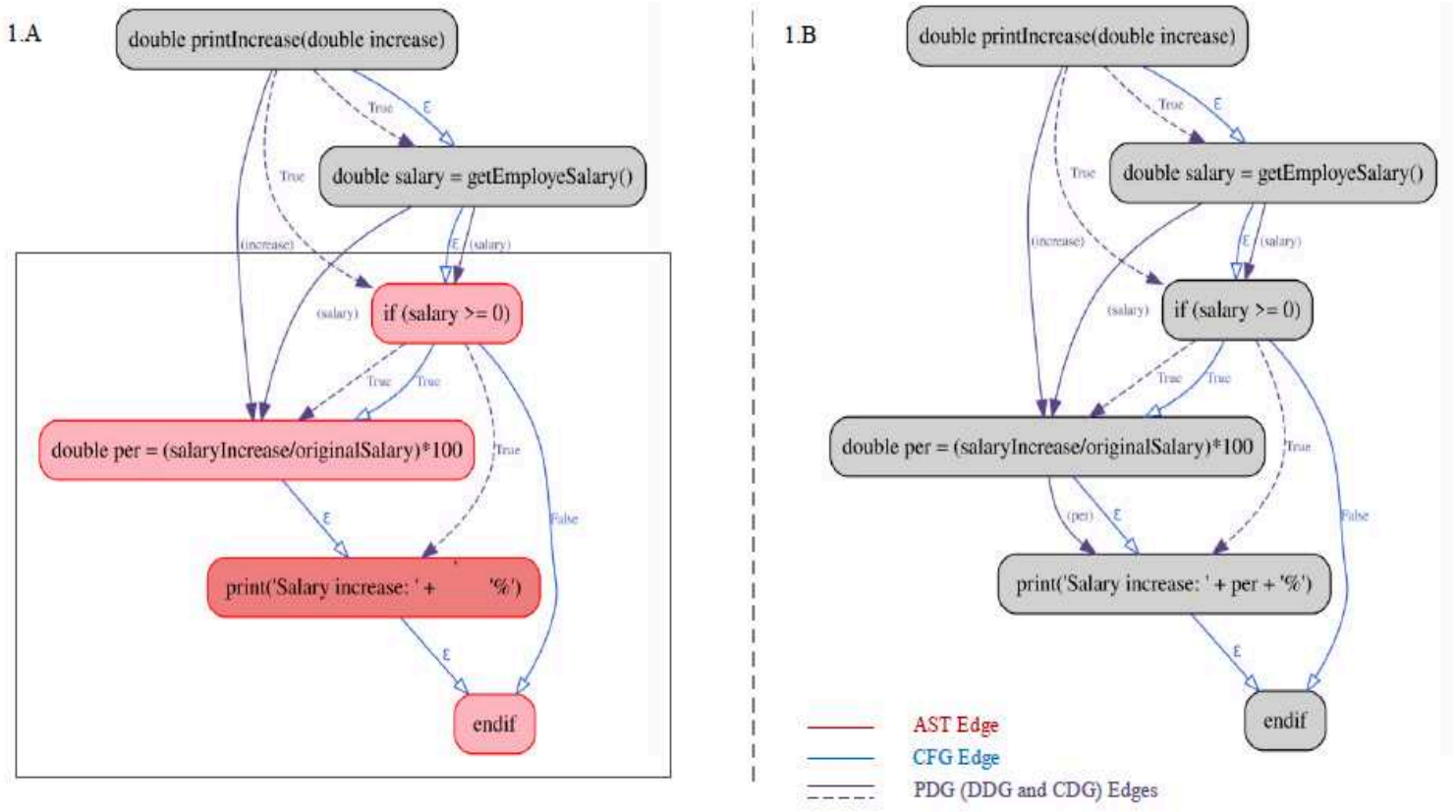

Figure 1: A motivating example. The variable increase corresponds to the difference between the new salary and the old salary. The raise percentage which is defined by the variable raise is computed in line 4 in both file1.java and file2.java.

1.3 Research objectives

With reference to the above context and the limitations of the existing defect prediction approaches, this research targets to improve the existing software reliability practices and provide a more complete prediction technique that detects diverse types of bugs more efficiently and effectively. Therefore, our research objectives can be summarized as follows:

- Propose a more reliable prediction model in terms of decision at both file-level and change-level defect prediction. In other words, providing an end-to-end defect prediction model that can determine defective artifacts (files or changes in this research) by extracting meaningful features directly from the source code. These features encode the properties of programs and are used by deep learning algorithms to train and construct defect prediction models.

- Choose the most appropriate source projects in the case of cross-project defect prediction to improve the feasibility and the performance of the prediction model.

1.4 Research questions

To achieve the above-stated objectives, answering the following fundamental research questions is required:

1. Considering the dependency information intra programs, do they improve the performance of prediction models on file-level?
2. Considering the dependency information intra methods, do they improve the performance of prediction models on change-level?
3. Is end-to-end deep learning based on graph analysis allowed to improve the quality of defect prediction models?
4. What criteria should be used to construct the dataset of external projects?
5. How qualification models and matching functions can contribute to build prediction model for cross projects?

1.5 Scientific problem

Defect prediction has gained much attention and has been considered especially important in the field of software engineering. It involves the preparation of data in which useful features are extracted directly from source code; and use machine learning algorithms, more specifically, deep learning algorithms that take features as input to train the defect prediction models.

Exploring and analysing the structures and semantic meanings of programs helps to boost the quality of the learning as well as the effectiveness and the performance of software defect prediction. Therefore, providing a suitable and relevant representation of code that aims to select the best set of features is a big challenge due to the complexity and the comprehension of rich information of programs. In addition, it is important to apply a suitable deep learning algorithm which can train automatically complex features of different types to provide an end-to-end defect prediction modelling process. The existing approaches do not formulate the

program well resulting in less exposure of several types of bugs. This may undermine the results of defect prediction model.

Therefore, the main research problem that this research addresses is how to represent the source code by considering the dependency information within programs, to permit a better exploration of the source code and improve qualitatively and quantitatively the detection of several types of bugs.

1.6 Contribution

The main contribution of this work is to design a full automated framework for defect prediction at different granularities, i.e., file-level and change-level, in two settings: within-project defect prediction and cross-project defect prediction. The proposed framework aims to explore deeply the code to discover maximum bugs to improve the quality of prediction. Consequently, the framework avoids investigating a huge amount of time and cost spent to release error-free software to the end users. We present the list of contributions hereafter:

1. Exploring deeply the code for defect prediction to detect the three defect typologies i.e. syntactical, semantic and dependency information by leveraging a proposed concept of code property graph [22] that merges properties of abstract syntax trees, control flow graphs and program dependence graphs into a single entity structure. This graphical program representation allows expressing patterns linked to defective code including the three typologies. After a systematic literature review using the keywords "code property graph" and "defect predict" on WoS, ScienceDirect and Scopus, we can assume that this research introduces for the first time the concept of code property graphs in the field of defect prediction. The experimental results prove that leveraging code property graphs is effective in developing high-performance classifiers.

2. Propose an end-to-end automated prediction model on file-level and change-level for software defect prediction to automatically learn graph-based expressive features that are fed to the deep learning algorithm “the deep graph convolutional neural network DGCNN”.

3. Demonstrating the inability of the traditional features in automatically extracting distinct types of bugs and especially those which are related to the dependencies from files and code changes.

4. An extensive evaluation under both the non-effort-aware and effort-aware scenarios; performed on Java projects demonstrating the empirical strengths of our model for defect prediction and shows that our approach achieves a significant improvement for within-project defect prediction and cross-project defect prediction.

5. Propose a project selection framework to choose meaningful projects based on structural and semantic information hidden in the code and the global knowledge of the projects, instead of using all the available projects.

6. An extensive evaluation performed on 10 large-scale Java project from Promise dataset [33] confirming the effectiveness of our framework in selecting similar source projects and demonstrates that our framework outperforms previously succeeded CPDP baselines and also our approach without making any selection of source projects beforehand.

1.7 Thesis scope

In this thesis, the proposed approaches for enhancing software reliability are analysed only on Java projects. Hence, they might not work for other programming languages, e.g., C++, script languages and assembly languages. Moreover, these suggested solutions are restricted to the examination of software bugs gathered from software histories from same projects or different projects. However, they cannot be generalized for other certain types of defects such as real-time bugs from concurrency bugs and embedded systems.

1.8 Manuscript organization

This dissertation consists of 5 chapters. After the introduction in the chapter 1, the rest of the content is organized as follows:

Chapter 2: Background

This chapter provides the background including the basic concepts of all subjects addressed in this thesis to formulate the defect prediction framework. It starts with describing the defect prediction foundations such as the defect prediction process and the main defect prediction approaches. Next, the different basic concepts of program analysis including the abstract syntax tree, the control flow graph and the program dependency graph that aim to analyse and model robustly the code are explained. Then, how these representations are transformed and merged into a single and powerful representation, code property graph, from which the complex features are extracted, is described.

Finally, an overview of deep learning and its different architectures, specifically, graph neural network that train and construct the predictive model by considering input multi-scale graph-based representation without losing information is illustrated.

Chapter 3: Related Work

This chapter is dedicated to the literature review. It introduces proposed and current approaches to the problems confronted above. The first and second area of research is related to file-level defect prediction and change-level defect prediction in two settings within-project and cross-project defect prediction, respectively. First, a brief discussion on the traditional proposed methods that aim to select useful features representing the code properties by considering recent research papers is given. Then, considerable shortcomings that these solutions suffer from are analysed and illustrated, followed by highlighting the eventual contributions that can be achieved. Different traditional machine learning algorithms as well as the deep learning architectures used by most of researchers to train and construct their prediction models are also presented. Next, we demonstrate the interest of the novel deep learning algorithms applied to graphs, including the deep graph convolutional neural network.

Finally, we present the study on the cross-project feasibility as well as the specific approaches dedicated to cross-project.

Chapter 4: An end-to-end deep learning defect prediction over code property graphs

This chapter depicts our proposed framework to automatically learn expressive features from both code files and code changes in order to determine defective files and changes in an earlier stage of the development phase before the production of the system. This framework provides a suitable and powerful representation of code by merging three basic concepts of program analysis into code property graph exploring deeply the code files and changes, and express patterns linked to diverse types of bugs. Then, the designing features are fed to the deep graph convolutional neural network to build the defect prediction model that can predict whether a file/or change is buggy or not. The experiment results proved that our approach significantly improves the existing works.

Chapter 5: A source project selection framework for cross-defect prediction

This chapter describes our novel methodology for selecting the best candidate source projects among several available projects to improve the cross-prediction performance. The selection is based on computing both high-level similarity and low-level similarity between the source projects and the target project. Finally, the chosen projects have the highest distribution difference between source projects and the target project. We evaluate our methodology on open-source projects and the results prove that our approach can boost the cross-prediction performance.

Chapter 6: Conclusion and perspectives

This chapter summarizes this research and highlights the challenges of this thesis that can be revised into future work.

SECOND CHAPTER

2 BACKGROUND

This chapter gives the background and the defect prediction foundations related to our objective of leveraging deep learning techniques based on graphs to ameliorate existing software quality practices. In the section 2.1, an overview of the software defect prediction process as well as the investigated defect prediction tasks in this thesis is provided followed by a broad overview of the architecture of the program analysis methodology adopted in the first stage of defect prediction (i.e., pre-processing phase) to extract the useful features from the source code in section 2.2. Deep learning-based techniques and more especially graph based deep learning methods such as Deep Graph Convolutional Neural Network are also covered in section 2.3.

2.1 Defect prediction process

A defect is a bug or an error in a program that causes the software product to cease operation in the manner requested by customer and developer when executing the system. The existing defects in software products are increasing over time, and are inevitable due to various reasons like poor communication between developers, incomplete requirement specification [34], [35], lack of user input [34], [36], [37], unclear and inadequate objectives and goals [34], [35].

Software defect prediction is adopted as a solution to improve the software quality and avoid the failure of the project. It alerts the developers about the presence of failures in software components (file, package, module, change, etc.) during the initial stages of the development of a software system. Thus, it helps the developers to devote extra resources and time to the non-defective software artifacts and, consequently, allowing companies to save money and resources. In general, constructing a defect prediction model needs a large amount of

historical data from a project to exploit the source code and express patterns linked to defective code. These patterns are expressed by traditional features (metrics) or by features collected from classic representation of code. The instances with the features corresponding to the software component and labels (buggy or not) are used to train machine learning classifier. Then, the trained model is applied to predict new instances as defective or not. The set of instances used to build model is referred to the training set and those are used to evaluate the built models is referred to the test set. However, for the project that has just started and do not have enough historical data or for the legacy systems in which history data are not available, building a predictive model is a challenging task. Therefore, the prediction model can be built by using history data from other projects (source projects) to predict defects to the project (target project) that lacks data. Figure 2 gives an overview of software defect prediction process and the three phases of software defect prediction modelling (i.e., pre-processing, model construction, and model validation). To build a prediction model, the first step is to extract features from data instances gathered from software archive. Depending on defect prediction granularity, the data instances can represent method instances, package instances, code change instances, source code file instances, etc. The features are useful data that serve to mine the code and its characteristics. These data can represent various levels of abstraction such as software development process, complexity, structural information, etc. Then, to constitute the training and test corpus, data instances are labelled as non-defective or defective according to whether the instance data includes bugs or not. Generally, the defect data are always noisy, and a mislabelling data detection method is needed to reduce the noise. Besides, the deep learning algorithm (DGCNN in this work) takes as input only numerical data, and the input vectors should have a same length. Thus, a mapping approach is required to map between tokens and integers. In the phase of model construction, after setting the deep learning parameters, the defect prediction models are built by using the training set to train the data. Finally, the defect prediction model is validated by using the test and performance metrics to assess the model performance.

In this thesis, we consider the following defect prediction tasks described in Table 1. Table 1 shows their abbreviations.

Table 1: Defect Prediction tasks

Defect Prediction Level	Within-project	Cross-Project
File-level	WPDP	WPCP
Change-Level	WCDP	CCDP

2.1.1 File-level defect prediction

File-level defect prediction is among the most adopted prediction techniques in the literature [10], [38]–[42]. File-level defect prediction is a traditional prediction technique that conducts long-term prediction at a coarse grained-level. Its process is typically as described in the Figure 2. The software history data represents the previous releases of the project. During the file-level defect prediction, the first step is to label each file as buggy or clean based on post-release defects accumulated from bug tracking system, and extract code properties from these files. The feature extraction techniques commonly used in the literature to train the machine learning based classifiers, can be divided into two types: one is by using traditional metrics which are designed manually, and the second is by applying automatically learned features from either arborescence-based structure or graph-based structure. According to these feature extraction methods, the level of code properties understanding differs. This means that by ensuring an effective representation of code, the better extraction of useful features of diverse types (semantic, structural, etc.) can be guaranteed. More details are provided in the next chapter *state of the art*. The prediction models are built by analysing software archives during previous releases and the developed model is used to predict whether files in the future releases are prone to defects or not. In this work, we evaluated the performance of DGCNN-based semantic and dependency features on both file-level within-project defect prediction (WPDP) and file-level cross-project defect prediction (CPDP).

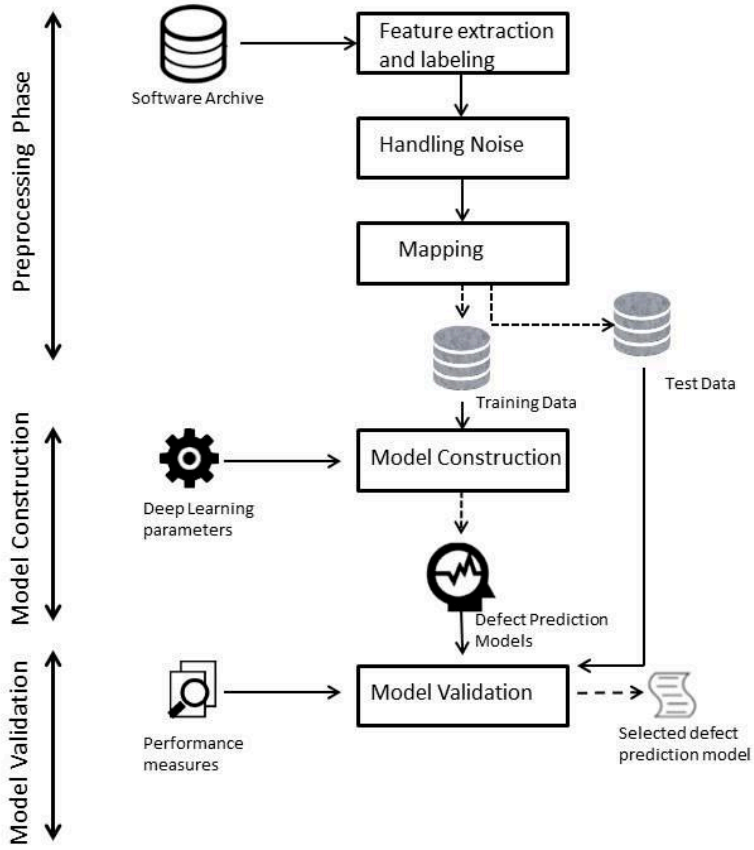

Figure 2: Defect Prediction Process

2.1.2 Change-level Defect prediction

Just-in-time defect prediction (JIT-DP) or change-level defect prediction has attracted more attention than the file-level defect prediction which conducts predictive models in an extremely late stage of the software development cycle, e.g., predicting faulty-proneness of files during releases of upgrades. The program modules are continuously modified and a thousand lines of code are made by several developers from one version to another, so it is quite difficult for developers to identify and repair potential defects in such a file [43]. JIT-DP has a more fine-granularity and the prediction can be executed once the code changes are committed [44]. JIT-SDP has several advantages compared to file-level defect prediction: 1) it is more practical as it reduces the risk of introducing new defects during the commit. 2) Developers can inspect code changes with limited effort and fix bugs when the code changes are still fresh in their minds. Same as file-level defect prediction, change-level defect prediction process contains three main steps:

- Pre-processing phase: Label each change as buggy or clean based on previous commits and extract the features that represent the code changes. Then, reduce the noisy data by applying mislabelling data detection method and finally perform a map between tokens and integers.
- Model learning: Construct the predictive model by using the deep learning algorithm that takes the meaningful generated features as input.
- Model validation: Predicting testing data to evaluate the constructed model.

Different from file-level defect data, labelling change-level data are specified for each project' s historical change and is stored in Version Control System (VCS) as buggy or clean by using the B-SZZ algorithm (i.e. the original SZZ approach) [45]. The aim of the algorithm is to make a further link of defect-fixing change to defect-introducing change. Bug-fixing change refers to the code which fixes the bug, while the bug-introducing change refers to the code that incorporates bugs. The bug introducing changes are identified by employing an *annotate/blame* technique provided by SZZ algorithm. This technique is commonly applied by several researchers [9], [46]–[49]. We label the bug-introducing changes as buggy, and others changes as clean. In this thesis, same as file-level defect prediction, we evaluate the performance of DGCNN-based semantic and dependency features on both change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP).

2.2 Data representation

Software defects are deeply hidden in programs' semantics which can cause unexpected output and profound consequences after software's commissioning. It is therefore required to exploit the source code and devise a useful representation of code that enables the developers to make mining of copious amounts of code and express patterns linked to defective code. Considering this, it cannot be expected from a system to automatically learn these patterns or meaningful features for defect prediction without ensuring a suitable representation that can be robustly extracted from code due to the rich information of programs possessed by the system.

As a solution, this section presents our methodology adopted in the data pre-processing stage for robust source code analysis, which sets as a foundation for our methodology of defect prediction. The key insight underlying this methodology is to explore deeply the programs by jointly considering account the syntax and semantic, control flow and data flow to discover the maximum of bugs and improve the quality of prediction. To this end, we combined classic ideas from compiler construction which are also known as classic concepts for analysing code robustly. Ultimately, we showed that our approach amounts to a useful tool for defect prediction by leveraging a joint representation of a program's control flow, data flow, and syntax called code property graph. In fact, it enabled to reveal several types of bugs in the source code with respect to control flow, data flow, and syntax, to respond to the constant evolution of software programs in terms of complexity.

In this section, we give an overview of the architecture of the program analysis methodology adopted in the data pre-processing stage. Then, we discuss how source code can be parsed and transformed into an intermediate graph-based program representation. Then, we show how the basic concepts of code representation can be combined into a meaningful representation to create the structural and semantic features for defect prediction.

2.2.1 A code mining system

Figure 3 depicts the code analysis architecture adopted in this work and gives the following key components of the method to produce an intermediate representation of the code.

- LL Parser. The first step to robust code analysis. LL parsers called LL (K) parsers are a top-down parsers for a subset of context-free languages. LL (K) parser is a form of recursive descent parsing which recursively parse the input to make parse tree. This parsing technique does not require any backtracking and is known as predictive parser.
- Code Property Graphs. To allow a deeply exploring of program and extracting complex patterns from code that combine syntax, control flow and data flow properties, the code property graph, a type of program representation was employed. This representation can be easily built from the output of LL (K) parser.

In the data pre-processing stage, the source code is firstly passed to the LL (k) parser to generate an intermediate comprehensive representation of the code, the code property graph. This graph including complex structural information helps to generate the best set of features for defect prediction. Then, graph based deep neural network techniques detailed in the next section are implemented to process graph data on the server side. In the following sub-sections, we provide the necessary background information for each of our methodology components in greater detail.

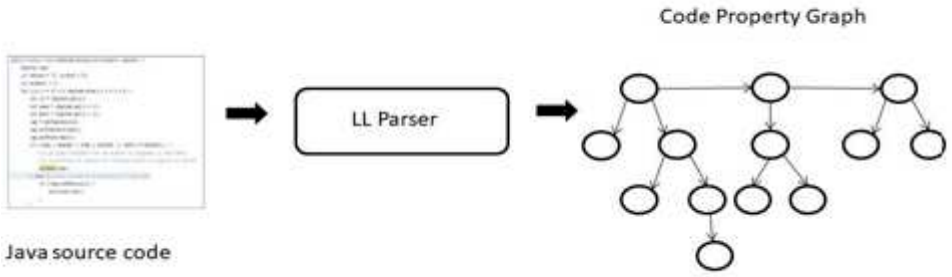

Figure 3: Overview of our architecture for robust code analysis methodology

2.2.2 LL PARSER

Automatic analysis of source code requires providing an intermediate comprehensive code representation that makes program properties explicit. Program analysis literature offer already several representations created for different purposes such as syntax analyse, control-flow and dataflow analyse, etc. All these representations are either directly or indirectly provided from a program's parse tree. By contrast, to the best of our knowledge, the well-known program analysis tools such as SOOT [50], Spoon [51], etc. carry out control-flow and data-flow analyses at an intermediate code or byte-code level rather than on an Abstract Syntax Tree (AST). However, operating these studies directly at the source code level or more precisely at the AST level can be beneficial since it makes it possible to deal with the source code. Additionally, there is no need to compile elevated level of abstractions during the translation to intermediate code. This is especially important for tools that are incorporated in interactive development environment as in our case. In fact, this type of analysis allows a faster computing time rather than on byte code by the application of interactive settings. For

example, if the user makes some actions in a program like code modifications, the model of the edited program which is typically AST will be kept in memory and will be updated in response to the modifications. However, a translation to byte code will need re-computation of information and consequently would potentially slow down performance in terms of response time [52]. To this end, it is more interesting to provide useful representations based on AST. First, we constructed the parse tree, which is later transformed into an abstract syntax tree (AST). Next, a control flow graph (CFG) was constructed from the abstract syntax tree to analyse the program's control flow. Based on the information that control flow graphs includes, we can provide control and data dependencies as expressed by control dependency graph (CDG) and data dependency graph (DDG), respectively. In the following segment, we present how syntax, control flow and program dependencies are determined by these representations and how they are generated from the output of the parser. Figure 4 presents an overview of the representations we can generate based on the LL parser output and underlines their dependencies.

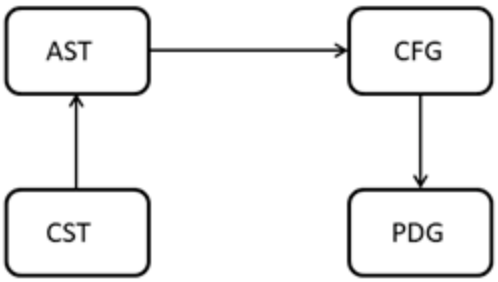

Figure 4: Dependencies between program representations

The first step in most software analysis projects is to parse the source code base. The main purpose of the parsers is to break the source code into components that can be converted into a target language. Parsers take a chain of tokens as input and construct a data structure such as AST which represents the input and includes all the information from the target program.

Generally, there are two kinds of parsers: top-down strategy and bottom-up strategy. A top-down strategy begins from the parse tree's root node and working down to the bottom leaves, following the rules of the formal grammar while the bottom-up parsing operates vice versa, beginning from the leaves and working up to the root. Most standard parsers use context-free grammar to describe languages. They are composed of a set of grammar rules. Production

rules are composed of a set of rewrite rules specifying symbol substitutions to convert nonterminal symbols into a set of either terminal or non-terminal symbols. A final representation of the input is generated when the rules are used recursively [53]. A terminal symbol is a standalone language construct, while a non-terminal symbol represents a syntactical phrase consisting of one or more terminal symbols and can include other allowable phrase structures.

The most popular subclasses of grammars are LL (k) for top-down and LR (k) for bottom-up parsers [54]. For code analysis, we opted for LL parsers family in this work. The first "L" states that the input is read from left to right, the second "L" indicates that the parser generates the leftmost derivation for its AST; and "k" is the number of look-ahead symbols applied at each parsing step to make decisions by comparing the symbols that begin at each alternative. LL (k) is limited to a fixed number of tokens of look-ahead to examine the entire remaining input rather than the LL (*) which can make decisions by providing deterministic strategy and using regular expressions, represented as deterministic finite automata (DFA). LL (*) parsers are a sort of recursive-descent parsers, designed from a set of recursive procedures where each implemented procedure corresponds to a production of the grammar. LL (*) strategy applies predictive parsing; meaning that it uses look-ahead that allows it to never backtrack and consequently it is able to run in linear time [54]. The main problem of LL (*) is that it performs grammar analysis statically that sometimes fails to find regular expressions with which it can distinguish between alternatives productions. To this end, an extension of LL (*) called Adaptive LL (*) or ALL (*) is proposed to address the problem of LL (*). It can, therefore, perform grammar analysis dynamically at runtime, before the generated parser executes. The idea behind ALL (*) prediction mechanism is to launch sub-parsers to determine which path leads to a valid parse. It has therefore access to all remaining input sequences to make decisions in sequence recognition while the others perform grammar in a static way and must consider all possible input sequences (infinitely long). The ALL (*) algorithm is the foundation of the ANTLR 4 parser generator tool (ANTLR 3 is based upon LL (*)). In this work, we selected the ANTLR 4 to generate the useful representations from code.

2.2.2.1 ANTLR 4

ANTLR (Another Tool for Language Recognition) parser is a powerful and flexible parser generator that accepts any context-free grammars. It is provided as a Java library to process, read, execute or convert structured texts or binary files. ANTLR 4 generates a recursive-descent parser that uses an ALL (*) production prediction function. Currently, it generates parsers in Java or C#. It was released in January 2013 and has about 5000 downloads/month. Thus, ALL (*) is widely used by academic and industrial users. As we explained before, the main idea of ANTLR parser is to read an input grammar and convert it into a program which can recognize a text and process it according to the rules of the defined grammar. ANTLR has two distinct stages: lexical analysis and parsing targeting the regular language it recognizes. Acknowledging a phrase refers to determine its various elements and distinguish it from other phrases. To do so, the lexer creates tokens (vocabulary symbols) by breaking up the input streams into tokens on which the parser feeds off and tries to recognize the sentence's structure.

In our context, the lexer' role is to understand the syntax of Java language while the parser is dedicated to checking the semantic and understanding the semantics of Java language by providing syntax trees which represent the sentences of the context-free. We take the following simple Java statement as an example: Length = 50; in the phase of lexical analysis, ANTLR analyses the input of characters and then collect them into tokens with tokens = {"Length", "=", "50", and ";"}. In the syntax analysis, ANTLR assures that the token sticks to the rules of the grammar and recognizes that it is an assignment statement in this example. Then, it builds the parse tree which saves how the parser acknowledged the input sentence' structure and components. The following diagram represented in Figure 5 illustrates the main steps of a language recognizer.

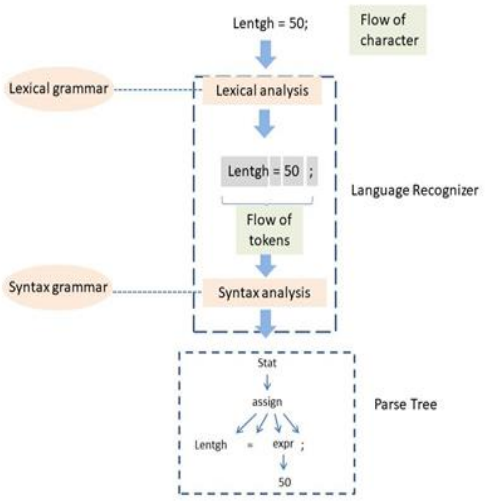

Figure 5: The data flow of the language recognizer

To perform the parsing, there is a need to design grammar which includes the syntactic rules of the language. An example of a simplified abstract grammar is depicted in the Figure 6. It represents an object-oriented form and illustrates the abstract classes *Stmt* and *Expr,* and subclasses for both statements and expressions such as *IfStmt* and *MethodCall*. The grammar uses a typical syntax with angle brackets to represent tokens, square brackets for optional children, and kleen star for list children. Children are either named with given names such as *LValue* and *RValue* that are the left and right children of the type of name *AssignExpr*, or with their types, such as *Block* child of a *MethodDecl*. Certain constructs are plotted as expressions in the grammar and may operate both as expressions and statements. For example, assignments and method calls fall in this category. The complete grammar for Java is disposable on the JastAdd web site [55].

2.2.3 Exposing Program Syntax

Based only on structural and syntax program, prediction of software defects is often possible. To this end, syntax trees or parse trees are meaningful tool to extract syntactical properties of the code. These trees form the output of ANTLR parser, and all other classic representations considered in this work are generated based on them. We now shortly explain how the parser creates the parse trees and then outline their conversion into abstract syntax tress, a normalized and simplified syntax tree for static analysis.

```
abstract Stmt;
Block      : Stmt ::= Stmt*;
IfStmt     : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt  : Stmt ::= Expr Stmt;
ExprStmt   : Stmt ::= Expr;
VarDecl    : Stmt ::= <Type:String> <Name:String> [Init:Expr];
ReturnStmt : Stmt ::= [Expr];
EmptyStmt  : Stmt;

abstract Expr;
AssignExpr : Expr ::= LValue:Expr RValue:Expr;
VarAccess  : Expr ::= <Name:String>;
MethodCall : Expr ::= <Name:String> Arg:Expr*;
```

Figure 6: A simplified Java abstract grammar

We regard the code sample depicted in Figure 7 to illustrate the three basic representations in addition to the code property graph to expose the weakness and strengths of each representation. Particularly, the example depicts a function called printTaxableAmount, which reads inputs amount by calling the function getAmount (line3). This variable is then verified if it is greater than a 0 (line 4) before being applied in an arithmetic operation (line 5) and going to the function print (line 6).

```
1.  public void printTaxableAmount() {
2.
3.          int amount = getAmount();
4.          if (amount > 0){
5.              double VATAmount = 0.20 * amont;
6.              print(VATAmount);
7.          }
8.  }
```

Figure 7: Example of code sample

2.2.3.1 Parse trees

Parse trees or concrete syntax trees are ordered, rooted trees that represents the syntactic structure of string according to some context-free grammar. It is always created as the next phase following the lexical analysis and can be easily illustrated when parsing source code

according to the defined grammar rules of the language in question. When performing grammar productions to acknowledge the input, a node is created for each meted terminal or non-terminal. We obtained the desired tree structure by connecting each node to its parent production. The root of the parse tree represents the general symbol of the grammar such as the start symbol. The interior nodes refer to the nonterminal symbols such as method call while the leaves refer to the terminals of the grammar which emerges as constants, and keywords such as *for*, *if*, *8*, etc.

Parse tree is the only representation that can be firstly generated from the text and thus is considered as the basis for the creation of the other classic representations presented in this section. Moreover, concrete syntax tree is considered as a concrete representation of the input as it saves all the information of the input, in another words, it is a grammatical copy of the code, token by token, in tree format. It takes every little piece of sentence and translates it to a data structure; even the punctuations and whitespace like the end of line are represented in the parse tree by a punctuation symbol and empty box, respectively. However, parse tree structure is not an extremely useful representation to work with as it contains all the information of the text even those that are not important to analyze code and extract distinguishing patterns. For that purpose, we transformed the parse tree structure into more useful representations of program syntax, the abstract syntax tree (AST).

2.2.3.2 Abstract Syntax Trees (ASTs)

The abstract syntax trees neglect useless information in program which has no significant semantic meanings throughout the program, against the parse tree. Indeed, ASTs do not consider the punctuation symbols such as parentheses or braces. Moreover, ASTs do not differentiate between two variables which are declared either in a declaration list or by using two consecutive declarations. Thus, contrary to the parse trees, ASTs are conceived to be the same for both declarations. Finally, the AST always discards the inner nodes with a single non-terminal child node which makes it a compacted version of a concrete syntax tree. Thus, it is a tree representation that records the structure of the input and is insensitive to the grammar that produces it.

According to the state-of-the-art [12], there are three types of AST nodes extracted as tokens: 1) Nodes of class instance creations and method invocations that are saved as their class names or method names, 2) declaration nodes (method declaration, type declarations, and 3) control-flow nodes such as *if* statements, *while* statements, *catch* clauses, etc.

ASTs include the essential elements of code and exclude the intrinsic ones such as comments, braces and inherent type declaration which may weaken the significance of other nodes as they are always method-specific or class-specific and they do not influence the semantics of the whole project, etc. In addition, AST is an ordered tree where a construct occurring in the code is represented. For example, the top elements of the file such as class declarations or import are represented by the children of the root which in turn depicts the whole file. Then, each class declaration node has its children to represent the fields or the included methods. We assigned to each tree node its AST type (e.g., *DeclarationStmt, ForStmt, WileStmt, etc*.) or its AST name (e.g., *variable name, class name, method name, etc*.).

As an example, Figure 8 illustrates an AST for the function PrintTaxableAmount. The details presented in the parse tree like the brackets are no longer included in the AST.

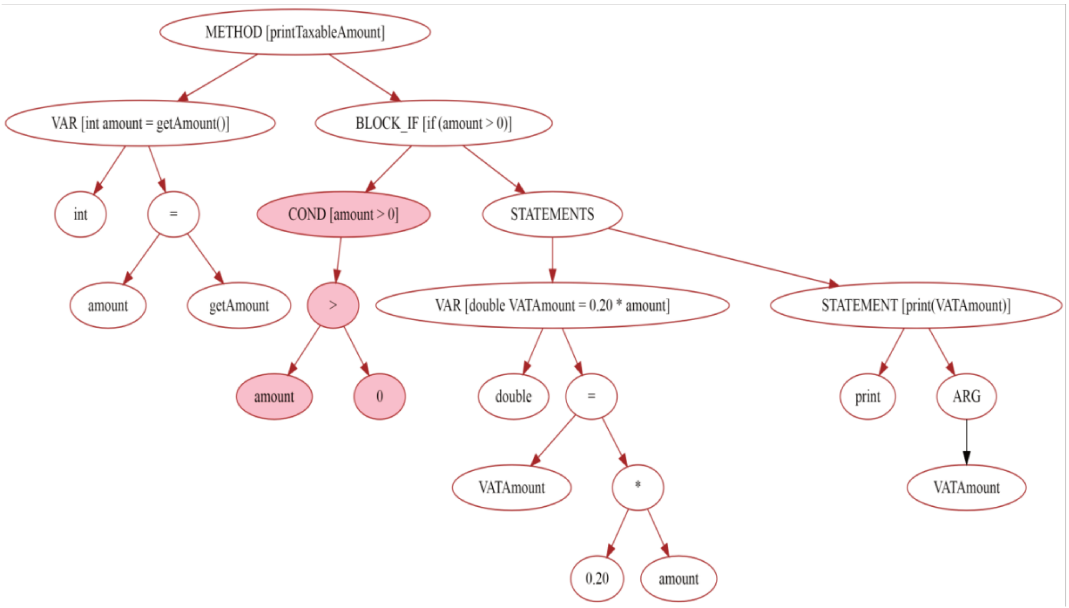

Figure 8: The abstract syntax tree corresponding to the listed code sample.

The AST is created as the final result of the syntax analysis phase and can be directly generated from parse trees. The parser may or may not always construct a concrete syntax tree, or parse tree. When it might need to construct a parse tree in between, the AST is realized by recursively walking the parse tree and determining translations of elementary parse trees into their corresponding abstract syntax trees.

Therefore, depending on how the compiler was designed, the parser may directly go straight onto generating an AST, or syntax tree [56]. However, AST will be always generated as the output of the parser, and no matter how many passes it might need to take in order to do so.

ASTs form the basis of many code representations as they serve as the first intermediate representations provided by parsers. Despite the syntax and the structural information that they explore from the source code, they do not show the control flow or the data dependencies of programs. Consequently, in the field of SDP, they may not reveal many types of defects in programs, as defect characteristics are deeply hidden in programs' semantics.

2.2.4 Exposing Control Flow

The control flow graph (CFG) [57] exposes its control flow: i.e., all the statements which can be executed following the conditions which must be traversed through a program where the abstract syntax tree is not well suited to study statements interaction and establish the execution order of the statements , a major requirement to model defect patterns in programs.

A control flow graph is a directed graph used in program analysis for determining properties and behaviour of program without executing it. The CFG nodes represent a basic block that explicit a linear segment of statements (both control and non-control statements) and conditions and the directed edges indicate the transfer of control from one instruction to another. The statements have one entry point (the first instruction carried out) and one exit point (the last instruction carried out) while the conditions are the instructions that require to be encountered for a particular path of execution. Unlike ASTs, the edges in CFGs are not ordered but rather need to be labelled and, more precisely, the statement node has one outgoing edge labelled Ɛ while the condition node has two outgoing edges labelled true and false.

Figure 9 shows an example of the control flow graph constructed from the code sample printTaxableAmount. The control flow graph begins with a start node identified by START and ends with an exit node designated with EXIT. Moreover, each statement is represented by a node. The illustrated example involves four non-control statements, the declaration of amount and VATamount, the call of print, and the invocation of the method printTaxableAmount. There also exists one control statement provided by if (amount > 0). The non-control statement is linked to just one other node via the edge labelled as Ɛ. The control statements have two labelled outgoing edges. The labels take the values true or false to specify under which condition the next block will be executed.

Control flow graphs can be generated directly from ASTs. To do this, it is required to give information about all keywords the language provides to permit developers to modify control flow, e.g., the keywords while, for, if, break, etc. Having this information, ASTs can be translated into control flow graphs by performing the following two step procedures:

- **Structured control flow.** In this step, the first version of control flow is provided by operating control flow statements such as for, if or while. This can be made by determining for each control flow statement how the abstract syntax tree is translated into a control flow graph and then the defined rules are recursively applied to all statements in the abstract syntax tree.
- **Unstructured control flow.** In this step, the control flow graph is rectified by introducing unstructured control flow defined by jump statements. Operating jump statements is easy after producing the first version of control flow graph as all the required information such as all loops, the targets of break and continue, and the labels which refer to go to statements are known. In fact, the complete control flow graph is provided by simply introducing other control flow edges from jump statements to their targets, and thus constructs the final control flow graph.

Control flow graph has been widely used for various problems including malware analysis [58], [59] and, software plagiarism [60], [61]. Moreover, it is considered as a standard code representation in reverse engineering to help in program understanding. In the field of defect prediction, method-based-CFG may be beneficial for distinguishing patterns. The following paper [27] proposes to leverage CFG for detecting faulty source code written in C language

and proves, by performing experiments, that CFG-based-method outperforms significantly the AST-based-method because of showing execution sequences of programs. However, CFG fails to provide dependency information in those programs, despite the fact that many bugs are directly related to the data flow information and the relationship between executed instructions. Many researchers established a clear link between dependencies and the appearance of several bugs [20], [28].

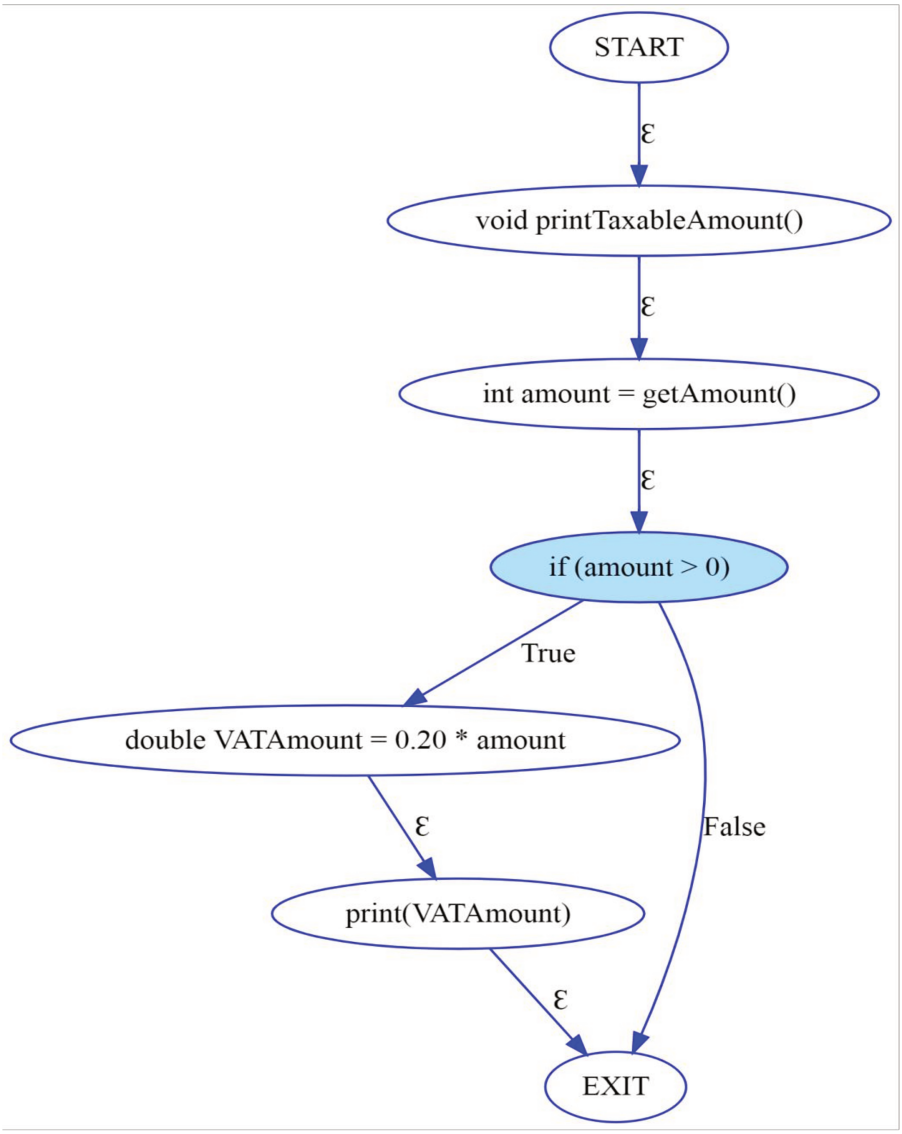

Figure 9: The control flow graph corresponding to the listed code sample

2.2.5 Exposing dependency information

The existence of control flow graphs is a pre-condition for the creation of program dependence graphs which play a crucial role in our method to fix and predict defect patterns in the program.

2.2.5.1 Program Dependence Graphs (PDGs)

A program dependence graph proposed by Ferrante [62], is a graphical-representation of a single method in a computer program that demonstrates program semantics and facilitates program comprehension. Indeed, it shows data and control dependence between instructions in a program in that method. The nodes correspond to the nodes in CFG of the program (declarations, conditional statements, function calls, etc.) and the edges represent both the data dependency and control dependence corresponding to the influence of one variable on another variable and the influence of statements on the values of variables respectively. Therefore, PDG is a combination of a Data Dependency Graph (DDG) and a Control Dependency Graph (CDG).

Two types of dependencies are expressed from control flow: data and control dependencies. The data dependence exhibits the correlation between instructions with respect to the usage and production of data. As shown in the example below, if it exists data dependence between two statements, a variable in one statement may has an incorrect value if we reverse the two statements.

```
d = 4;
c = d * 3;
```

The first statement declares the variable ‘d’ which is applied by the second statement, so the second statement depends on the first one. This dependence is called a direct dependence. Reversing the two statements can generate an error.

The second type of dependency is control dependence. It is used to determine statements that may be carried out before a given statement is carried out. For example:

```
if (x > 3)
y = 10;
```

The execution of the second instruction always provides the same results; however, it depends on the first instruction. This type of dependence is called control or indirect dependence.

As an example, Figure 10 depicts the program dependence graph corresponding to the code sample. The program dependency graph holds a node for each program statement like the control flow graph; however, the right sequence of statement execution can no longer be obtained from it. Rather, we see the dependence between statements under the outgoing edges from the statements identifying variables to the statements using these variables. As an example, we see in the Figure 10, the variable amount is defined by the first statement, and it is applied in the statement of definition of VATamount as well as the predicate. This predicate is itself linked to the print-function and the statement defining the variable VATamount by a control-dependence edge, showing that both the statements are only performed if the predicate executes to true.

PDG provides indication on the connectivity inside each method of the software. Without such interactions, software will not be able to perform its required tasks in those methods. Such connectivity can be eventually a major contributor to the appearance of bugs and to the difficulty to maintain such software. Therefore, analysing the dependences between instructions inside each component may be helpful for distinguishing faulty patterns from non-faulty ones.

Both of PDG' edges can be determined based on the control flow graph, and, in the case of control dependencies, the post dominator [57].

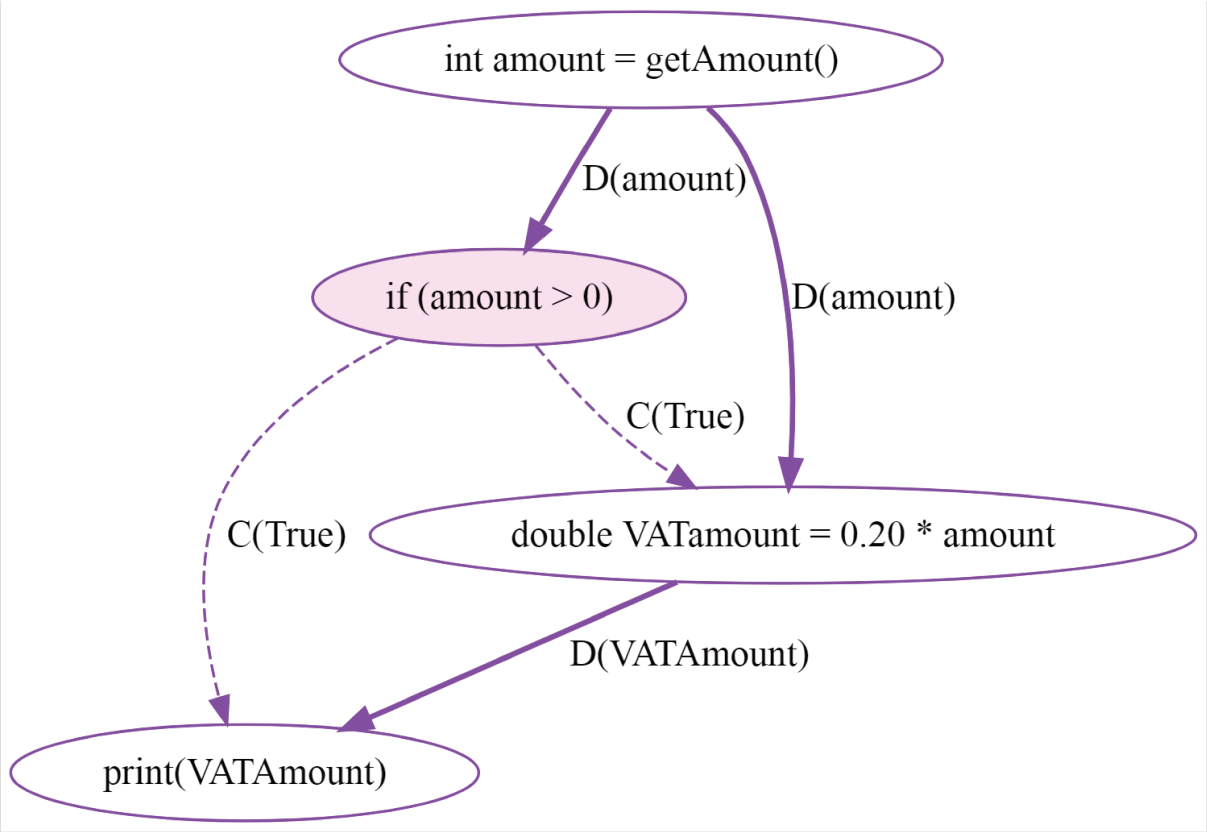

Figure 10: The program dependency graph corresponding to the listed code sample

Calculation of control dependencies- Dominators and post-dominators determine which basic block must be executed prior to, or after, a block n. For a given node v, we say that it is post-dominated by a node w if every directed path from v to stop (a unique exit node of CFG) contains w. Intuitively, the post-dominance frontier of a basic block n represents the set of all blocks that are immediate predecessors to blocks and are not themselves post-dominated by n. The formal definition [62] of control dependence is, taking two nodes A and where B in CFG, B is control dependent on A if there exists a direct path from A to B with any X in P post-dominated by B; and A is not post-dominated by B. Another rule to express the control dependence is that for a block A, which post dominates a block B, which in turn is control dependent on statement C, and A does not post-dominate C, then A is control dependent on C. Therefore, the computation of control dependencies depends on the availability of the post dominator tree which refers to the dominator tree of the reverse control flow graph.

Calculation of data dependencies- Intra-procedural data dependences is defined in terms of def-use chains [63]. A def-use chain determines all possible uses of a variable, for each definition of that variable. By this way, all the information about the variables that concern their definitions is propagated to all of their uses. A definition of a variable means that the statement affects a value to the variable, while a variable used refers to any statement which accesses the variable's value. Thus, in the Data Dependence Graph, statements represent nodes and def-use chains refer to the edges. So, creating a DDG edge is simply a way of striking the uses for each definition of a variable.

2.2.6 Code property graph

To recognize and extract patterns linked to diverse types of defective code (i.e., syntactic defective code, semantic defective code, defective code related to the dependencies); it is required to exploit deeply the source code and ensure a powerful representation of code that can be robustly extracted from code. Such representation should be able to automatically learn these patterns or meaningful features for defect prediction. The existing studies adopt either AST or CFG to represent the code and extract useful defective patterns which will be fed to the deep learning algorithm. However, these representations allow either to extract the syntactical properties of code that are derived from AST or control flow from CFG; and none of them can represent the dependencies within the program. The three basic concepts of program analysis including AST, CFG and PDG introduced in the previous section are complementary to provide rich information of programs. Each of them stores and provides certain properties of the software; nevertheless, a single representation alone is not able to detect all types of errors and predict quality of the developed software with the least amount of possible human efforts. In order to take advantage of the benefits of these representations, we merged these three representations into a common entity structure called code property graph (CPG), first introduced by Yamagushi et al. [22]. Such a structure combines the strengths of each representation; and consequently, it allows patterns to be analysed based on the combination of syntax, control, and data flow. After a systematic literature review using these keywords ("code property graph" and "defect prediction") on WoS, ScienceDirect and Scopus, we can assume that this work introduces first the concept of code property graphs in the field of defect prediction. The code property graph use the concept of property graph [64]

which forms a basic representation of structured data in many graph databases, as for example Neo4J, ArangoDB, and OrientDB. The key insight underlying the code property graph is to reveal several types of bugs in the source code with respect to syntax, semantic, control flow and data flow to respond to the constant evolution of software programs which are increasingly prone to failures. Consequently, the concept of CPG serves as a basis for our framework that allows discovering the maximum of bugs and improving the quality of prediction. Furthermore, the concept of code property graphs has proven its success in other works in the field of vulnerability detection [22], [65]–[70].

In this section we define the property graph as an abstract data type, including the basic operations to construct the code property graph. In addition, we detail how the classic program representations described in the previous section can be modelled as instances of property graphs; and merging them by using the same contextual properties for the construction of the code property graph.

Formally, a property graph is defined as follows.

Definition 1. A property graph [64] is a directed, edge-labelled, attributed graph G= (N, E, δ, β) where N is a set of nodes and E ⊂ (N × N) represents a set of edges which are labelled by an edge labelling function γ assigning a label to each edge from Σ (i.e., γ: E → Σ) and properties can be assigned to both nodes and edges from keys to values by using the function β where (β : (N × E) × K→ V); K is a set of property keys and V is related to the property values.

A key k ϵ K is affected to each node, where only vertices A, C and D have property values

These properties can be used for linking a graph with other graphs.

2.2.6.1 Model the Abstract Syntax Tree as Property Graph

AST forms the basis of many code representations as it provides detailed information about the software code. We, thus, start the building of the joint data structure by modelling AST as

property graph GA= (NA, EA, δA, βA) where NA corresponds to the tree nodes and EA represent the tree edges which are labelled as AST edges by applying the labelling function δA. As explained previously, we assigned to each AST node a key property by using βA to model the AST as property graph. We can thus define property keys for several types of AST nodes such as the string property code and name which are corresponding to the code snippet the node represents (e.g., statement, expression, operand, operator, etc.) and the name of represented object (e.g., method name), respectively. Moreover, we define some properties that indicate where the code can be found like the keys order and line-number which represent the order structure of the tree and the line where the code can be found. As a result, the property keys are defined as: KA = {code, name, order, line number, etc.} while the set of property values VA is given by all the operator and operands, statements and expressions, method name and the natural numbers.

2.2.6.2 Model the Control Flow Graph as Property Graph

CFG nodes represent blocks of instructions that correspond to the statements and expressions in AST. Hence, we express the CFG as property graph to prepare its incorporation into a joint data structure GC= (NC, EC, δC, βC), where δC reflects the edge labelling function that assigns a label to each edge in the CFG property graph from the set ΣC = {true, false, ϵ} while βC defines the properties assigned to each CFG node which only takes these property values VA = {Stmt, expression} for the key code.

2.2.6.3 Model the Program Dependency Graph as Property Graph

The PDG represents data and control dependencies among statements and expressions. Therefore, PDG has the same nodes as CFG, but it does not represent the same edges. For this purpose, we defined PDG property graph as follows: GP= (NP, EP, δP, βP), where we simply identify a new set of edges EP and properties compared to CFG. We have therefore a new edge labelling function δP which assigns the values of data and control dependencies from the

set ΣP = {C, D}. Moreover, we added the properties symbol and evaluation to indicate the corresponding symbol to each data dependency and the state true or false evaluation of the expression to each control dependency.

2.2.6.4 Merging the representations AST, CFG and PDG

As the last step, we merged the three representations into a unique data structure called Code property graph which maps all the code elements into various levels of abstraction, including AST, CFG and PDG. This joint data structure provides a much deeper understanding of code source and how the various components interact with each other. This understanding allows a more effective analysis of the code source for the extraction of errors. This is especially effective for improving the performance of the prediction model and identifying complex bugs of distinct types and especially those which are related to the dependencies.

As each statement and expression is represented by a node in each of the three graphs and the AST is the only one of the three representations, which introduces additional nodes, statements and expressions are therefore served as transition points from one representation to another. We can thus incorporate CFG and PDG into AST through the statements and expressions. As explained above, each node is assigned by a property key and its corresponding set of property values such as the key code and its property values (for-statement, while-statement, if-statement, etc.) and the key-property line and the corresponding property values (line-number, etc.) that indicates where the code can be found. For example, to link the AST and CFG we get the property of each node of CFG and we search in AST the nodes that have the same property value as well as the same line number of codes. Then, we add the edges of CFG in AST between the two nodes (source node and target node). Figure 12 illustrates the corresponding CPG to the code sample in Figure 7. The property values of the node corresponding to the statement {if (amount > 0)} are IF-Statement and 3. In the AST, we add the edges (in-coming edges and out-going edges) of CFG and PDG. Same process to merge the AST and PDG to construct the code property graph. Figure 11 shows the process to

merge the blocks of AST1, CFG2, and PDG3 to construct the node *merge node* in code property graph. Formally, a property graph is defined as follows.

Definition 2. A code property graph is a property graph G= (N, E, δ, β) constructed from the merging of the three representations AST, CFG and PDG of source code where:

N = NA ,

E = EA ⊂ EC ⊂ EP,

δ = δA ⊂ δC ⊂ δP and

β = βA ⊂ βC ⊂ βP

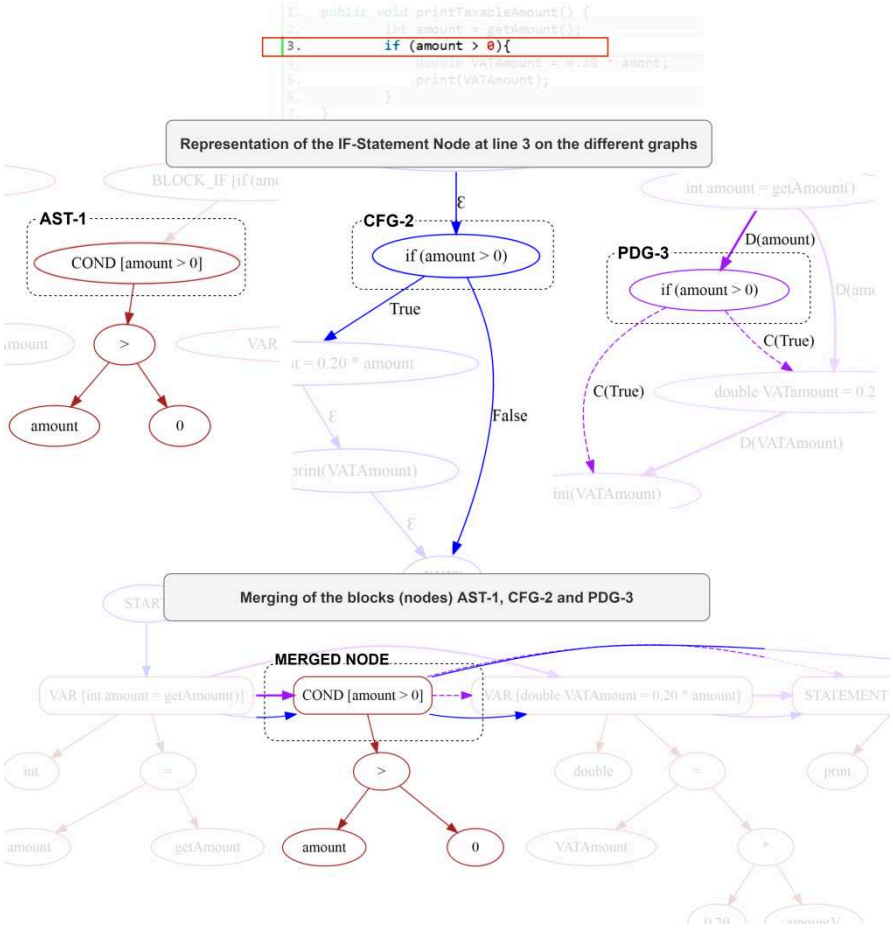

Figure 11: The merging process to construct the merging node from AST 1, CFG 2, and PDG 3

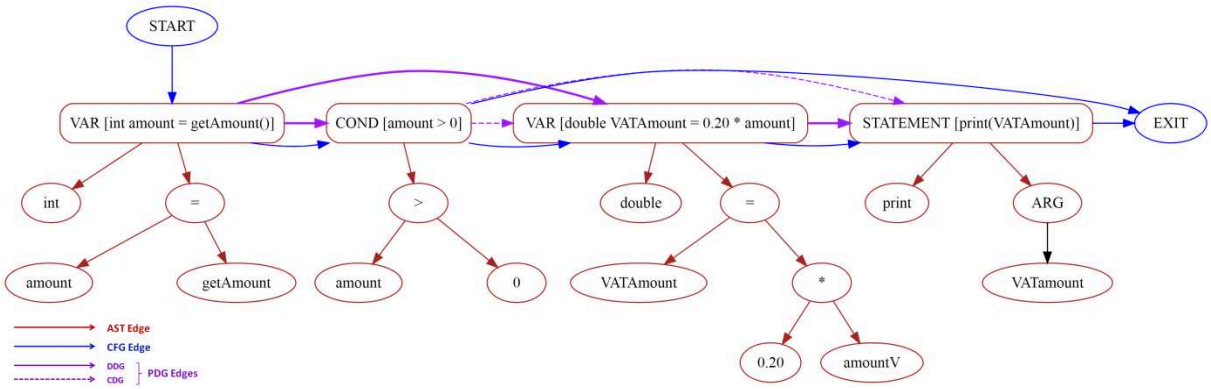

Figure 12: The code property graph corresponding to the listed code sample

2.3 Deep Learning

Deep learning (DL) is increasingly investigated by academic researchers and industrial practitioners in recent years into terms of software engineering (SE) tasks due to its remarkable success in resolving a broad range of problems from software documentation, software language modelling, testing, predicting defects in software, etc. According to the surveys [71], [72], since 2015 and especially in the years that followed, the number of papers that include DL in SE have increased significantly, from 11 papers in 2017 to 28 papers in 2018 and 35 in 2019. This proves the immense importance given by researchers to apply the deep learning techniques in software engineering. The usage of DL focuses mainly on three problems: documentation, testing, and defect prediction. The bibliography analysis conducted in the survey [72] indicate that 8.6% or about 9 papers among 81 papers cover deep learning for defect prediction problem. We give more details of these works in the next chapter Literature. The application of DL for DF was motivated primarily by the weakness of traditional machine learning algorithms. Indeed, DL can model high-level data representation based on multiple layers of Neural Networks (NN) while machine learning is based on structured data. To better understand this, we aim to predict whether the software instance such as file, code change, etc. introduces bugs or not in this thesis. To this purpose, we need to extract meaningful features from history data that represent fault-prone instances from not fault-prone, and then train and build the prediction model. We applied several feature extraction methods to extract relevant features, and this is where the difference between

machine learning and deep learning can be perceived. The features are handcrafted in machine learning while they are recognized and extracted automatically based on Neural Networks with DL. Thus, the use of DL automates the defect prediction process.

The most commonly used classes of NN in SE are recurrent neural network, convolutional neural network, and multilayer perceptron. In the following, we give an overview of each of these types of NN:

Multi-Layer Perceptron (MLP): It is a type of feed-forward artificial neural network. MLPs are extensively used to resolve numerous problems including classification, identification, control systems, pattern recognition, etc. Moreover, MLP results can be used as a baseline for comparison to assess the performance of various algorithms on a specific problem. The MLP involves one or three layers; the input layer gets the data, the hidden layers deliver abstraction levels, and the output layer makes the prediction. For training, MLP applies the supervised learning technique which called back-propagation.

Conventional Neural network (CNN): It has been applied to many fields such as speech recognition [73], [74], image classification [75], [76], and natural language processing [77], [78], and has shown great success in the practice as it handle large-scale and high dimensional inputs. The CNN learn local structure and feature of data by using three types of layers including convolutional layer, pooling layer, and fully connected layer. The convolutional layer replaces the multilayer perceptron in which hidden layers are fully connected.

CNNs have two key factors which are sparse connectivity and shared weights. Sparse connectivity means that CNN performs a local correlation between neurons by connecting every neuron in one layer to only a restricted number of neurons in another layer. In CNN, sparse connectivity is checked by a kernel size and pooling size. For example, when a kernel size is equal to 4, the nodes in the hidden layer m only connects to four nodes in layer m-1 instead of needing to connect to all the adjacent neurons like traditional artificial neural networks (ANNs). The connection between the two hidden adjacent layers refers to a local filter which captures a local pattern. To compute the output of the next layer, a nonlinear transformation is performed after having multiplied the output of the previous layer with each local filter and adds a bias. The function is described as follows:

$$h_i^m = ReLU(W^{m-1} * V^{m-1})_i + b^{m-1}$$

Where h_i^m refers to the i[th] hidden unit in layer m, W^{m-1} *and* b^{m-1} present to the weights and bias of the local filer, respectively. Rectified linear unit (ReLU) denotes the nonlinear function.

Shared weights mean that the local filter shares the same weight and bias. This allows capturing features, whatever their positions. Besides, the weight sharing enables to improve the learning efficiency by reducing model capacity.

Another important concept in CNN is the pooling layer. It is always applied after the convolutional layer and aims to partition the output vector into several non-overlapping sub regions, and then, derives a single output from them. Thus, the pooling layer acts to reduce the spatial size by applying an operation including max, average, or L2-norm. The most common operation of pooling is the maximum pooling: the value recovery is the maximum of each sub-region. Pooling operation enables to decrease the number of intermediate representations and make the defect prediction process more robust.

The effectiveness of CNN relies on other parameters such as filter length, pooling size, convolutional and pooling layer numbers. To have a successful CNN, these parameters should be well tuned for a good setting.

Recurrent Neural Network (RNN): It is a type of NN designed for sequence prediction problems. This means that the RNNs are intended to receive historical sequence inputs to predict the next output values. Thus, contrary to the back-propagation learning, the hidden layer is updated with both the current inputs and the preceding received inputs. This concept enables to the network to memorize long-term dependencies. This concept confers the advantage of memory the network. The Long Short-Term Memory (LSTM) is among the most used RNNs in various different applications [79], [80].

2.3.1 Graph Convolutional Network

Various complex systems across different areas such as social networks, knowledge graphs, and protein-protein interaction networks, etc., require a powerful representation to model the objects and their relationships. A graph is a rich representational data structure that describes complex systems in the real world. Therefore, the objects are represented by nodes while the relationship between the objects is represented by edges. The existing traditional deep

learning approaches cannot be applied directly to graphs and need to be transformed into regular data forms (i.e., same fixed data). For example, although the Convolutional Neural Networks are capable to extract multi-scale features, they cannot operate on irregular data structures defined in a non-Euclidean space. Thus, the graphs should be transformed to be processed by CNNs. However, the structural information of the graphs can be lost, and redundant information can be involved during the transformation process. The key factors (i.e., local connectivity, shared weights, and the use of multi-layer) that distinguish the CNN algorithms motivate to generalize them to graphs for several reasons: 1) most of graphs are locally connected, 2) the shard weights limit the computational cost, and the multi-layer structure allows to process hierarchical patterns. To this purpose, there has been a great interest to adapt convolutional neural networks to the graph domain to properly carry out feature learning on graphs directly [81]–[85]. A novel architecture is proposed, called graph convolutional neural network (G-CNN) that extends the CNN by adding a pre-processing layer called the disordered graph convolutional layer (DGCL). G-CCN has proved its effectiveness to extract useful features for graph classification [86].

Various approaches are proposed to generalize CNNs to graph-structured data [84], [86]–[91], and can be categorized into two strategies: spatial-based approaches and spectral-based approaches. Spectral methods typically transform the graph into the spectral domain by using the eigenvectors of the Laplacian matrix as the convolution operator. Most of the spectral-based methods are restricted to same-sized graph structures and are always applied for vertex classification.

Regarding the spatial-based methods, they can be employed to real-word graph classification problems as they are not limited to fixed-sized graph structures. These methods generalize the graph convolution operation by using the neighbourhood information from the graph data space. However, they still need to further transform the multi-scale features learned from graph convolution layers into same-sized representations which can be managed by the standard CNN. To reach this target, the learned local-level vertex features are aggregated from the graph convolution operation as global-level graph features by applying a SumPooling layer.

To overcome the above limitations of the existing spatial-based Graph Convolutional Network, a novel spatially-based Deep Graph Convolutional Neural Network (DGCNN) is developed by Zhang et al [92] to store more vertex information. They proposed a new SortPooling layer whose aim is transform the learned vertex features from the spatial graph

convolution layers into a same-size local-level vertex structure, by sequentially storing a number of vertices with prior orders. Then, the standard CNN model followed by a SoftMax layer can be directly performed on the obtained fixed-sized graph structure.

2.3.2 Deep Graph Convolutional Neural Network (DGCNN)

Deep Graph Convolutional Neural Network (DGCNN) is a new architecture of convolutional neural network that takes graphs of arbitrary structure as inputs. This new proposed architecture tackles two main challenges: 1) how to gather robustly relevant features describing rich information involved in graphs and 2) how to sequentially read these graphs in a consistent order. Graph convolution layers aim to extract the local substructure from nodes and define a consistent node ordering. The extraction of this information is inspired by the Weisfeiler-Lehman sub-tree kernel approach (WL) [93]. Then, to address the second challenge, a Sortpooling layer is used to sort the node features under a predefined order and unifies input sizes. Thus, a fixed and ordering representation is achieved and then, standard convolutional and dense layers can be introduced to read ordered graph representations and make the prediction. In this thesis, we apply DGCNN as feature extractor from code property graphs to train a model which predicts whether the new instance (file or change) is buggy or not. As described in the previous section, the vertex labels of the code property graphs contain rich complex information of programs. For example, in CPG, each vertex is an instruction that may include instruction name, and many operands. Additionally, each instruction can be seen by other perspectives including instruction types and functions, besides its contents. Thus, a powerful deep learning architecture such as DGCNN is required to directly process these graphs. A complete architectural view of DGCNN is presented in Figure 13. In the remaining section, we explain in detail the three consecutive stages to be performed by the DGCNN: 1) the convolutional layers, 2. the Sort-pooling layer of DGCNN, and 3) the traditional convolution and dense layers reading the sorted graph depictions.

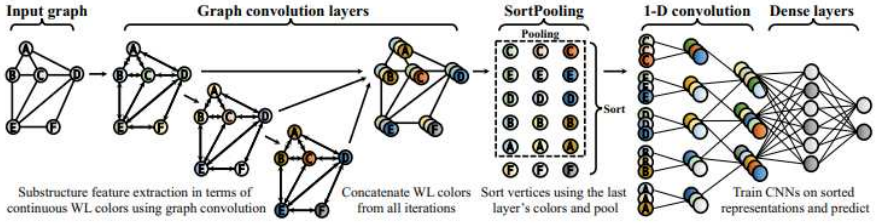

Figure 13: The overall structure of DGCNN **[92]**

2.3.2.1 Convolutional layers

As indicated above, there is a theoretical relationship between the DGCNN model and the WL algorithm [93]. The procedure aims to extract features from various parts of graphs. Its operation is determined as follows: the key idea of WL is to concatenate the colour of a vertex with the colours of its neighbours, then to store the concatenated labels lexicographically to attribute to each vertex a new colour (i.e., a new label). This operation is reiterated until the maximum iteration h is reached. The vertices having the same labels or converge to the same colour are considered as non-distinguished and share the same structural role in the graph. WL is applied to verify the isomorphism of graphs, so the graphs which have the same set of WL colours at any iteration, are considered as similar. Furthermore, the similarity between two graphs is identified by the computation of the kernel function in WL method [93]. However, DGCNN approach differs from of WL in the uses of only a soft version of the approach. Indeed, the kernel function is not computed and the use of colours is not the same: the DGCNN approach concatenates the generated colours in the form of a tensor Z_t (with $t = 1, .., h$) horizontally in a tensor $Z_{1:h}$ (h being the number of iteration / convolution performed) [92].

2.3.2.2 SortPooling

The main purpose of the SortPooling layer is to sort the feature descriptors, each representing a vertex, in a consistent order before feeding them into the standard 1-D convolutional and dense layers. The issue here is the sorting the vertices. To classify images, the pixels are stored according to their spatial order. However, in text classification, the alphabetical order is adopted to arrange the words. For graphs, the vertices are sorted following their structural roles within graph as determined by the WL algorithm. The SortPooling layer serves as a bridge between the convolutional layers and standard neural network by back propagating

loss gradients and consequently achieves an end-to-end learning. More precisely, the layer of SortPooling takes as input, the output of the convolutional layers which is the tensor Zt (Z1...h) of size n * ∑ct, where each line represents the feature descriptor, and each column refers to the feature channel. To order the vertices of the graph, the SortPooling layer sort the Zt by starting by the last channel Zh in descending order. Then, the output of the layer is a tensor k * ∑ct, where k represents an integer to be defined.

2.3.2.3 The traditional convolutional layer

a 1-D convolution (Conv1D) layers is added. Then the output of sort pooling is reshaped, Z^{sp} of size k $\sum_1^h(c_t)$ with every row signifying a vertex and each column depicting a feature channel, to a vector of size k $\sum_1^h(c_t)$ x 1. After this the vector is fed into an entirely connected 1 layer perceptron for classification of graph.

THIRD CHAPTER

3 Related Work

There exists a plethora of research whose objective is to improve software quality. In recent years, increasing studies have been conducted to build accurate predictive models. Several trends have been emerged from software defect prediction studies and have evolved over time. Firstly, most of researchers focus on proposing an efficient and precise classifier at file level for academic and industrial application. As mentioned in the previous chapter, this means that they build predictive models by analysing software history data in previous releases and use the developed model to predict whether files in future release are prone to defects or not. However, this kind of prediction has the limitation in terms that this traditional prediction model could be more interesting before the product release for the purpose of the quality assurance. Indeed, the prediction would be more effective and helpful if we can predict bugs whenever the source code is changed. To remedy this limitation, a recent defect prediction research is represented by the so-called just-in-time software defect prediction (JIT-SDP) (i.e. short-term prediction at commit level) [94]–[96]. With this strategy, developers can have immediate feedback [95] and quickly narrow down the most likely defective commits once code changes are committed.

Another limitation has been existed for defect prediction. It is too difficult to build prediction models for some projects typically new projects or legacy ones which have not enough historical data available to train models. This limitation is considered among the most difficult problems in defect prediction studies [16]. To resolve this issue, prior works have proposed various cross-project defect prediction models, i.e. models trained using historical data from other projects [41], [97]–[99]. In the cross defect prediction study, researchers have conducted another interesting topic which concerns the study on cross-prediction feasibility [16], [100]. In this thesis, we will address three actively studied topics in recent years: 1) software defect prediction at file level, 2) software defect prediction at change level or just-in-time defect prediction, and 3) the feasibility of cross-prediction. To conduct these three topics,

researchers have investigated various pre-processing approaches and have focused on machine learning models to train historical data. Machine learning algorithms have been widely used in the last years to improve the accuracy of the prediction models [7], [24], [25], [38], [101]–[103]. As explained in the previous chapter, pre-processing is a step in the defect prediction process where we extract meaningful features from instances generated from software archives. The traditional methods for program analysis proposed in the pre-processing step can be classified into three categories: metric-based methods, tree-based methods and graph-based methods.

In this chapter, we give an overview of the different existing research of leveraging deep learning models to ameliorate software reliability. The reminder of this chapter is as follows: Section 3.1 represents different axes covered in this research. In section 3.2, we present different traditional pre-processing methods used in the literature to extract features from the code. Section 3.3 reviews various defect prediction models based on machine learning and especially deep learning in software engineering. Section 3.4 presents various approaches for cross-project defect prediction. Finally, based on this literature, we identify and discuss the current challenges faced by researchers.

3.1 Overview of various research axes

Table 2 lists the representative research axes in software defect prediction. Proposing pre-processing techniques is especially important research branch in defect prediction studies. These techniques including feature extraction [104], normalization [41], [103], and handling noise [105], [106] and can improve the performance of the prediction. Many studies have focused firstly on proposing distinct categories of metrics to develop prediction models. Software metrics are quantifiable or countable measurements that can be applied to characterize properties of a software product and predict the quality of software [107]. Generally, the widely adopted metrics are code metrics and process metrics [108]. Code metrics provide a snapshot of software, whereas process metrics take the software changes over time. Then, these techniques are evolved over time and the researchers exploit the source code by representing the program by abstract syntax trees or control flow graphs to mine large amount of code' syntax and semantics.

As mentioned above, most defect prediction studies are conducted based on the most meaningful subfield of machine learning, deep learning. It has been widely used in last years in many traditional software engineering such as software testing [109], [110], defect prediction [30]–[32], and documentation [111], [112]. It has proved its efficiency in developing more accurate defect prediction models by leveraging selected expressive features automatically generated from the source code and then these features are used to train and construct the defect prediction models.

Table 2: Representative research axes in software defect prediction

Granularity	Type	Categories	Methods
File/Change	Within/Cross	Pre-processing techniques (Feature-extraction, normalization and noise handling)	Metrics: process metrics, code metrics, network metrics. AST based methods CFG based methods
		Algorithms/ models	Classic machine learning Deep learning (DBN, CNN, etc.)
		Pre-processing (Transfer learning)	NN filter, TCA+, etc.
	Cross	Feasibility	Decision Tree

Defect prediction models tried to locate bugs at different granularities. Most of them investigate traditional file-level defect prediction and made meaningful contributions by proposing new representations of code or exploiting different machine learning. Subsequently, researchers have focused on finer granularity such as change-level that can give more accurate and earlier feedback to developers.

The existing research studies described above are verified under within prediction setting, i.e., prediction models are trained and tested in the same project. However, for the new project, locating bugs becomes a tedious task for researchers due to the lack of historical data. To address this issue, various approaches are proposed such as Nearest Neighbour (NN) [16], TCA+ [41], etc.

According to many studies [16], [100], cross-prediction is very hard to achieve. Determining the feasibility of the cross project can play a crucial role for cross-project defect prediction. This research branch was not deeply investigated despite its importance. Only, a handful of studies are conducted to deal with this problem such as decision trees, etc.

3.2 Traditional pre-processing techniques

In this section, we highlight some current traditional pre-processing techniques in the area of software defect prediction. These techniques play a significant role to build the prediction model and improve its performance.

The following subsections present the different adopted approaches to select features from the source code.

3.2.1 Software Metrics

Most software metrics can be classified into two kinds: code metrics and process metrics: Code metrics are directly collected from the source code and process metrics are gathered from historical data recorded in software repositories such as issue tracking system and version control [113].

3.2.1.1 Code metrics (or product metrics)

Code metrics are calculated directly from the source code. The presumption here is that the more complex the code, the more they are bug-prone as they are more hard to understand and to change [7]. Several metrics are proposed by the researchers to measure the code complexity.

Halstead computes the software complexity of a module by proposing size metrics based on the number of operators and operands [17]. These metrics have been widely used in defect prediction literature [99], [103], [114].

McCabe designs a software module by a directed flow graph where each program statement is represented by a node and the flow of control between two statements is represented by an arc. Different form of Halstead metrics measures quantity and volume of code. McCabe introduced the cyclomatic metric to analyze the complexity of source code structure by

considering the complexity of the control paths i.e., it computes the number of nodes, arcs, and connected components in control flow graph of program.

A set of metrics have been proposed for object-oriented languages to build a prediction model. The most popular metrics for object-oriented programs are CK metrics proposed by Chidamber and Kemerer [19]. Its underlying assumption is to quantify the complexity and the size of distinct aspects of object-oriented program at class level. This metric is composed of six metrics and was applied by many studies to build prediction models [7], [20], [38], [101], [105], [106], [115]–[119].

Another widely used metric in the literature to assess the size of a software system is lines of the code (LOC) [7], [38], [103], [114], [120], [121]. Several types of lines of code are proposed to count several types of lines such as loc-comments, loc-code-and-comments, etc.

3.2.1.2 Process metrics

Process metrics have been employed by several researchers to enhance the performance of defect prediction. Software defects often change in time as the software evolves. Indeed, many researchers considered that the use of process metrics such as code changes may be interesting on the evolution version [122]. Version control systems (VCS) such as GIT, CVS, or SVN, store detailed information about the change: the files that have been changed, the names of developers, the manual log message, etc.

Many researchers have focused on these evolution metrics. Nagappan and Ball [25] applied eight representative code churn metrics to measure the amount the quantity of code change, and the experimental results show so that these metrics are considered as good predictors to predict the defect density of bug-proneness. Moser et al. [123] proposed different history metrics such as the number of revisions, ages of files, and past fixes to predict defects and concluded that they are more efficient predictors than code metrics when performing experiments on the Eclipse project. Hassan [24] used entropy metrics to predict new changes and compared them with two change metrics (the number of previous faults and previous revisions on six open-source projects). The experimental results showed that the complexity metrics outperformed the two changed features. However, this experimentation revealed the weakness of the evaluation as the comparison was between complexity metrics and those by only two change metrics. Moreover, Rahman et al. [108] investigated a number of evolution

metrics such as code change, committer/developer information, etc., and indicated that process metrics outperformed code metrics because of stagnation of code metrics. Also, Madeyski et al. [124] extracted process metrics from software change history, such as the number of revisions, modified lines, and defects in the previous versions, and concluded that the process metrics could improve the prediction performance than product metrics. Graves et al [125] retrieved process metrics from software change history and concluded that they performed better than static code metrics. Kamei et al. [95] selected 14 change metrics of different categories such as size, history, experience, etc. and developed logistic regression models to predict commits as buggy or not. Later on, they extended their work and evaluated the feasibility of their proposed method in a cross-project context [115]. Qiao et al. [126] proposed two new process metrics that change the degree of classes and the defect rates of historical packages for software defect prediction in object-oriented programs. The authors made comparisons between their proposed process metrics and code metrics as long with other process metrics to show the effectiveness of their approach. Kim et al. used text-based metrics accumulated from change in logs, file names, and the identifiers in deleted and added source code; then applied support vector machine SVM to predict whether a change contains bugs or not [96]. Some researchers such as Stanic et al. [127] used combinations of process metrics and code metrics, and their results showed that the combination of these metrics could predict more defective files. Shivaji et al. [104] investigated combinations of churn metrics, object oriented metrics, textual features and static code metrics while Bird et al. [128] employed combinations of developer contribution network metrics.

3.2.1.3 Other metrics

Researchers have proposed another kind of measures based on network measure [20], [129]. Zimmermann et al. [20] generated dependency graphs and conducted network analysis measures such as betweenness, closeness, and degree of centrality on that graph. Their results demonstrated that network measures are better in predicting more bug-prone binaries than process and code metrics. Qu et al. [130] proposed network embedding technique called node2defect to learn structural features into low-dimensional vector space. The node2defect used traditional software engineering traditional metrics such as lack of cohesion in methods (LCOM), coupling between object classes (CBO), and depth of inheritance tree (DIT) to learn structural features into vectors. Then, it concatenated the learned vectors with other traditional

metrics to predict bugs more accurately. Meenely et al. [129] used a set of developer metrics extracted from developer social network that represents the structure of the collaboration between developers. This study indicates that there is a correlation between software failures and developer network metrics.

3.2.1.4 Discussion

At present, researchers have designed various traditional metrics extract code properties and describe the characteristics of software evolution. Table 3 summarizes the representative metrics by category. The code metrics such as CK, size, Hastead, McCabe and OO metrics are mainly used as predictors for file-level defect prediction. Due to the stagnation of these metrics, process metrics are proposed for just-in-time defect prediction to describe the evolution characteristics of software. Most of these metrics appeared in 2000s when software repositories such as issue tracking systems and version control became popular. The subject of metrics is still an open debate today. Despite the proposition of several metrics in the field of defect prediction, no research can prove that there is one set of metrics that outperforms all the others. Furthermore, even the combination of these metrics does not provide optimal performance in identifying defects as they rely on the performance in each single feature involved in the combination. To conclude, there is still no consensus about the best set of metrics (predictors) for software defects.

Table 3: Representative metrics by category

Category	Metrics
File-level defect prediction	CK, size, Hastead, McCabe OO metrics
Change-level (just-in-time) defect prediction	Code metric churn (), change () Change entropy ()

3.2.2 Software defect prediction methods based on trees and graphs

Recently, several approaches have proposed more advanced methods based on trees and graphs to analyze the source code. Furthermore, they have applied deep learning techniques (i.e., CNN, DBN) to generate automatically meaningful features. It is no longer about the traditional manually designing features (methods based on metrics) that are fed into classifiers to identify code defects but, instead, it is more about the features that are captured automatically from the source code and that can have richer representations of programs and conduct a more precise prediction. Among those approaches are those that convert the code source into AST and others that convert the code into CFG. The abstract syntax tree includes semantic and structural information specifying the hierarchical relationship among different components in the source code [131]; while the control flow graph represents all the paths that can be crossed during the program execution. In previous studies of software defect prediction, Wang et al [12] applied a deep learning model (DBN) to automatically learn features over AST. For file level defect prediction, they first parsed the source code into AST and then created token vectors from the AST node. Finally, they built defect prediction based on the token vectors. According to the evaluation, their approach outperformed traditional metric-based methods. Li et al [26] introduced an approach for defect prediction performing deep learning (i.e. CNN) for structural and semantic feature extraction. Similarly, to Wang' approach, they recorded AST nodes to build the token vectors that will be fed to the deep learning algorithm. However, instead of recording the names of AST nodes, they extracted values to build the token vectors. The framework known as Defect Prediction via Convolutional Neural Network (DP-CNN) outperforms existing defect prediction methods such as DBN and defect prediction metrics. Shi et al. [132] proposed a framework called Multi-perspectives tree embedding MPT-embedding. They represented the code as AST from multiple perspectives and used the convolutional neural network CNN to construct the defect prediction model. For a better exploitation of the tree structure AST and better capture of many level of syntactic and semantic information in source code, Dam et al. [133] proposed a tree-based long short-term neural network (TB-LSTM) that can acquire vector representation of the whole AST. Fan et al [134] extracted relevant features from the AST and considered that they contain significant syntaxes and semantics information. As a result, they applied bidirectional long short-term memory (Bi-LSTM) with attention mechanism to capture

defective programming patterns. To build the token vectors, they recorded plain text in the source code for method invocations nodes and extracted the node types for control flow nodes. All the nodes of declarations are simply recorded as node names. Chen et al. [135] leveraged a deep learning cross-project defect prediction method, called “DeepCPDP”. This method represents the source code by a simplified AST to extract token vectors and apply the Bi-directional Long Short-Term Memory neural network (BLSTM) to build the classifier. Nguyen et al. [136] examined changes at the AST level.

Unlike other researchers who have represented the code in the form of AST, Anh et al. [27] adopted the graph based representation to extract meaningful features. Indeed, they represented the code as CPG and used the deep learning network DGCNN to learn semantic features. The results of the experiment showed that the CFG based methods outperformed tree-based methods as they captured the execution process of programs. To sum up, the studies based on trees and graphs described above allow considering only account the syntax and semantic information and the program execution processes. Therefore, they can cover more typologies of bugs, typically syntactic and semantic bugs or those related to the execution processes.

3.2.2.1 Discussion

A significant number of researchers have made contributions, mainly in data-pre-processing phase in which they explore the source code to make mining of large amount of code and extract meaningful features to use machine learning techniques that take the expressive features as input to identify code defects and build predictive models [39]. In the beginning, they applied traditional metrics. Then, they focused on using tree representations of programs or precise graphs representing program execution flows. The contribution of the AST based code representation was important compared to the traditional metrics as it provides significant semantic and syntactic information. Therefore, AST based methods can capture more bugs and especially those related to syntactic and semantic defects which traditional metrics often fail to detect. Although CFG based methods has given better results compared to techniques based on AST and metrics, they only capture the execution process within the program and do not provide information related to the relationships between components intra program modules. In other words, they are not able to capture the behaviour of the program.

To sum up, existing models do not offer high performance and do not cover all types of bugs. They often fail to capture intra-procedural dependencies. Indeed, several bugs are directly related to these dependencies. Such information is important in modelling program functionality and can result to a more effective and accurate defect prediction. To thoroughly explore program structure and semantics and detect defective programming patterns, it is required to devise a more complete prediction technique which can extract several types of bugs. This should help to mitigate the fragility of current learning methods based on either trees or graphs to extract meaningful features from the source code and improve the quality of prediction.

To this effect, it is of paramount importance to conceive an efficient source code representation able to make mining of large amounts of code and provide simultaneously different project characteristics belonging to structural, semantic, and intra-dependencies aspects. Indeed, we need to merge all these aspects in our code analysis representation as each aspect has its own benefits and highlights specific bugs. Moreover, none of them can fully replace the others. Table 4 presents a comparison between the target solution and the existing feature extraction methods. As it is shown in the table, the target method should highlight all the various aspects of programs including the syntax and the semantic information, the execution process and the intra-procedural dependencies. Contrary to metric-based methods, they do not deal with any aspect expect the static code properties and some software evolution characteristics. Furthermore, the designed metrics may not be highly correlated with class labels or redundancy. These all can affect the effectiveness and efficiency of the prediction model. As for other methods, they are not complete either because they also lack certain aspects such as the syntactic aspects for CFG based methods and the execution process for AST based methods.

As reported in the previous chapter, the code property graph representation meets all these criteria. Indeed, taking in consideration this code representation should logically significantly improve the prediction quality. We confirm this hypothesis in the following chapter.

Table 4: A comparison between the target method and the existing feature extraction methods

	Traditional metrics	AST based methods	CFG based methods	The target solution
Syntactic information	x	√	x	√
Semantic information	x	√	√	√
Execution process	x	X	√	√
Intra-procedural dependencies	x	X	x	√

3.3 Software Defect prediction models

3.3.1 Traditional Machine learning algorithms

Machine learning models have been extensively used in the literature to predict the fault-proneness of software systems since they have achieved enormous success in solving real-world problems of software engineering. Therefore, a variety of traditional machine learning algorithms such as Decision Trees, Logistic Regression, Naïve Bayes, Random Forest, etc. are published to develop software fault prediction models. The ultimate objective of machine learning techniques in defect prediction is to find relevant defective pattern extracted directly from code and improve the model accuracy. Depending on what to predict, we categorize machine learning models into two types, classification, and regression. Classification models determine defect-proneness while regression models predict the number of defects. In this study, we highlight on classification problems and try to classify which modules are prone defects in a software system. To do this, it is required to have a labelled dataset with meaningful features to identify defective from non-defective modules. To create this dataset, feature extraction methods are applied to extract useful features, and then train the model. Traditional learning algorithms are usually applied on lots of information provided by metrics-based methods to extract software bugs information. Erturk et al. [137] used McCabe software metrics with the algorithms SVM and ANFIS (new adaptive model proposed) to predict defects. In the paper of Naidu et al. [138], the defect was categorized into five parameters such as Program length, Volume, Time, Difficulty, Estimator , and Effort. They applied ID3 classification algorithm, to classify defects. Aleem et al. [139] used around

fifteen data sets from PROMISE data repository and applied a collection of machine learning algorithms such as (Random Forest, , Support Vector Machine (SVM), Naive Bayes, Ensemble Classifier (Bagging and Boosting), etc.) to the selected datasets. They suggested that SVM and bagging had high performances and accuracy by measuring the performance of each method. Ghouti et al., [140] proposed a software prediction model based on Probabilistic Neural Network (PNN) and SVM and used Promise datasets for evaluation. Their results showed that predictive performance of PNN outperforms SVM for any size of datasets. Guo et al., [141] applied ensemble approach (Random Forest) to predict defective software components. They compared their approach' performance against other existing machine leaning approaches involved in NASA datasets. Azeem et al. [142] discussed the utilization of several machine learning techniques such as regression, classification, clustering, and association in software bug prediction but did not supplied a comparative analysis of these approaches. Okutan and Yildiz [143] used PROMISE data repository and suggested that most powerful metrics for software are response for lines of code, class, and lack of coding quality. Wang et al. [144] compared only ensemble classifiers for software defect prediction.

Traditional learning approaches are proven to be useful to assess software quality and predict predictive software components. However, they fail to extract and exploit meaningful features automatically from the code. In other words, the features are handcrafted (i.e., manually encoded) with traditional machine learning approaches as they rely on metrics. To remedy this issue, Deep learning techniques are proposed recently to select automatically the features through neural networks [145]–[147].

3.3.2 Deep Learning in software engineering

Deep Learning is a subfield of Machine Learning that depends on many layers of Neural Networks (NN) to represent high-level representations [148]. Currently, there are several types of NNs, such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Deep Reinforcement Learning, Auto-Encoders, and Generative Adversarial Networks. Deep learning techniques have been drawn increasing attention in software engineering due to their powerful feature learning capability, and have been successfully applied and revealed remarkable improvement in several fields like documentation [111], [112], [149], testing [109], [110], [150], malware classification [151], [152], software traceability [153], defect prediction [12], [30]–[32], etc. Researchers are investigating the

application of DL to address traditional problems in Software engineering [153]–[163]. According to a study [72], the number of papers concerning deep learning and software engineering constantly increases over years. The work [162] used recurrent neural networks (RNN) to explore source code and showed its effectiveness. They later extended the RNN models published in [164] to detect code clones. The authors in this paper [165] proposed a special RNN Encoder-Decoder model to remedy the API call sequence recovery problem based on the natural language queries of the user. Besides, the researchers applied RNN Encoder-Decoder for identifying common faults in C programs in this study [166].

Deep learning techniques have been widely applied also in defect prediction [72]. In this field, researchers have explored the performance of different deep learning algorithms for learning semantic features [26], [167]–[170]. For example, Wang et al. [12] applied deep belief networks (DBN) on token vectors retrieved from programs' ASTs to learn semantic features automatically for file-level defect prediction. Their findings proved that the deep features-based technique outperforms traditional software metrics on average 14.2% in F-score. A longer version of their work is proposed in this paper [32]. Among the improvements, they applied DBN to automatically learn semantic features for just-in-time defect prediction and extended their experiments by including four open-source commercial projects. Dam et al. [171] suggested that LSTM is a more efficient predictive model for source code. They claimed that features based on metrics do not extract the multiple levels of semantics in the code. Then, they extend their work in [133] and relied on the usage of a tree-structured LSTM network to better match with AST representation of the code. The experiment results confirmed the effectiveness of the proposed method on file-level for both within and cross-project. The performed evaluations included real projects provided by an industrial partner (Samsung). Wen et al [170] leveraged RNN to learn features for change-level defect prediction task. The experimental results indicated that the RNN-based features showed better results compared to traditional change-level defect prediction features. Li et al. [26] proposed an approach for defect prediction called DP-CNN in which the authors employ CNN to extract patterns involving syntactic and semantic information from programs and combine traditional features with CNN-learned-features to improve the file-level prediction accuracy. Their evaluations showed that DP-CNN can outperform DBN-based defect prediction model. Hoang et al. [30] described an end-to-end deep learning framework, called DeepJIT, which automatically captures features from commit messages and code changes. DeepJIT outperforms the state-of-the-art benchmark by around 10% in terms of F1. Tong et al. [31]

claimed that Stacked Denoising Autoencoders (SDAEs) was never used in defect prediction. They proposed a novel two-phase software defect prediction based on SDAEs and two-stage ensemble learning to obtain more representative and robust features. The experimental results showed that their novel approach is significantly effective compared to the traditional software metrics. Zhou et al. [172] suggested a new deep forest model to predict defects by transforming random forest classifiers into a layer-by-layer structure. Mou et al. [160] investigated the applicability of deep learning to model complex structure of programs and demonstrated that deep learning can extract different levels of programs' structural information. Finally, Yang et al. [49] leveraged DBN from a set of traditional change metrics such as code deleted, modified directories, metrics related to developers 'experience, modified directories, files before/after changes, etc. to predict a commit as a buggy or not. This work is proposed to overcome the weakness of Logistic Regression (LR) which cannot merge features to generate new one.

3.3.3 Graph convolutional neural network

The above-mentioned deep learning algorithms have proved their effectiveness to automatically explore complex structures of features. However, none of them can tackle graph data. They require transforming the graphs into a fixed data structure. Standard convolutional neural networks (CNNs) are considered among available deep learning techniques. They are widely used in different fields such as object recognition, image classification and semantic segmentation. However, they can only learn fixed-sized local structure of data, so they cannot manage graph-based data, whose neighbourhoods are uneven. As the deep learning networks have proved a great success in machine learning, their application have been generalized for processing the data representation in graph domain [82]–[85], [173]. The novel deep learning networks applied on graphs are called Graph Convolutional Networks (GCNs). According to this study [86], GCNs are important and worth investigating. They have emerged as a powerful novel architecture to learn highly relevant features on graph data directly. In the literature, numerous methods have been proposed to apply CNN to graph-structured data to exploit rich information involved in graphs. These methods can be classified in two main categories: a) spectral and b) spatial strategies. The spectral-based approaches use the eigenvectors of the Laplacian matrix as the convolution operation to transform the graph into a spectral domain. Bruna et al [87] have performed a spectral filter by multiplying the graph

by a series of filter coefficients to apply a graph convolution network. Unfortunately, most of spectral-based methods cannot be applied on graph-structured data with same number of vertices. On the other hand, special-based methods are not limited to same-sized graph structures. The main aim of these methods is to find an operation on the neighbouring vertices and convert the convolution of the graph data into spatial structure [174]–[176]. Duvenaud et al [175] performed a spatially-based GCN by identifying a spatial graph convolutional operation on 1-layer neighbouring vertices. Atwood and Towsley [174] have carried out a spatially-based GCN by applying convolution operation on different layers of neighbouring vertices rooted at a vertex. Spatially based methods still require a further transformation of the learned local-level features from graph convolutional layers into fixed-sized representations to perform the standard classifiers. They therefore sum up the learned local features as global-level graph features by using the Sumpooling layer. However, spatially based methods are low performing on graph classification since it is hard to learn multi-scale and rich local information from global features. To overcome these limitations of current spatially based GCN methods, Zhang et al. [92] proposed a novel spatially based Deep Graph Convolutional Neural Network (DGCNN) to retain more vertex information through its new proposed SortPooling layer. The purpose of the SortPooling layer is to convert the extracted irregular vertex (i.e., unordered vertices) from the spatial graph convolutional layers into a fixed-sized local-level vertex by consecutively keeping a number of vertices with prior orders. DGCNN model reaches better performance compared to the existing state-of-the-art GCN models as it can extract complex graph information locating in local-level vertices.

In the field of defect prediction, Phan et al [27] have applied DGCNN directly on control flow graph (CFG) to learn semantic features. In CFG, each vertex is an instruction which may contain many operands and instruction name. To directly treat such graph characteristics and avoid losing significant information, the directed graph convolutional neural network is designed. Due to the enormous success of the graph convolutional networks and especially DGCNN in exploiting complex information resident in graph structure, we adopted the application of DGCNN to deeply explore the semantic meanings as well as the dependencies and therefore formulate an end-to-end deep learning approach. As explained above, we expect to leverage a joint representation of a program' syntax, control flow and data flow called code property graph which makes mining of complex and enormous amounts of code. The vertices of CPG represent statements, predicates, operands, and operators, etc. while the edges of CPG involve the edges of AST, CFG, and PDG. This common entity structure maps various code

elements that lead to a better understanding of code structure and how the various components interact with each other. Moreover, it allows extracting diverse levels of bugs. This structure will be fed directly to the deep learning algorithm DGCNN to build predictive model. The code property graph is a complex data structure that can be easily exploited by DGCNN to learn diverse types of bugs. Contrary to the above standard deep learning models which require transforming data structure into vectors, a lot of information can be lost.

3.4 Specific approaches for cross defect prediction

3.4.1 Cross prediction feasibility

Developing an accurate and effective prediction model is considered challenging for projects where there is not enough data. To overcome this issue, an increasing number of papers have proposed the cross-project defect prediction (CPDP) in which machine learning models use a bunch of training data from multiple projects [41], [177], [178]. In most cases, the data distribution between various projects cannot meet similar distribution hypothesis. Consequently, a CPDP model cannot easily extract generalizable properties of defective software components in one or set of source projects, so that the defect prediction on target project is always ineffective and unstable.

Many researchers have conducted many empirical studies to explore the feasibility of CPDP. They claimed that it is necessary to select relevant source projects, features or instances whose characteristics are remarkably close to the target project to improve the predictive model accuracy. Indeed, if the source-project data are selected carefully, the CPDP can work even better than WPCP [100], [179]. Alternatively, large irrelevant data can lead to low efficiency. For this, cross-prediction topic is considered as a challenging task as, there are only few cross-prediction combinations that work [16]. Zimmerman et al. [16] carried out 622 cross-projects experiments on real large-scale projects, and found only 21 experiments (3%) could achieve satisfactory performance due to the data distribution difference between the source and target projects. He et al [100] also conducted a study on cross-project application with the aim to select the most suitable cross-project models among all existing models to predict defects on target projects. He found that only 0.3 % to 4.7% of cases can reach

satisfactory performance depending on different classifiers. These studies confirmed the importance to tackle the cross-prediction feasibility.

3.4.2 Transfer learning approaches

To address the limitation of data distribution in CPDP models and improve the cross-prediction feasibility, researchers should strive to create a bridge from source projects to target projects to reduce the heterogeneity between them. To do this, most of researchers rely on using transfer learning techniques which is considered among the interesting areas of machine learning [180]. The ultimate objective of transfer learning techniques is to transfer knowledge from a domain with sufficient training data to another domain with insufficient training data to construct a learning model. In this field, researchers have proposed increasingly studies to improve the performance of CPDP. A systematic literature review [14] provides a detailed study on the representative techniques of transfer learning for cross-defect prediction. These proposed solutions aim to improve the CPDP performance by relying on different strategies. These strategies are shown as follows. Some approaches focused on metric value transformation. For example, Cruz et al. [181] applied metric value transformation by using power transformation to the metric values.

Other approaches are based on the selection of relevant instances from the source project that are similar to the target project. For example, Turhan et al. [179] converted the metric data into value ‘logarithm and then used the nearest-neighbour filter (NN Filter) to select the similar instances to the target. In the data pre-processing phase, he focused on feature selection, instead of feature mapping by applying the NN filter. The basic idea of NN filter is to select the source instances which have the similar data characteristics to assess data to avoid having irrelevant samples. Peters et al [182] applied Peters filters to select source instances. Ma et al [98] investigated the homogenous set of cross project metrics by applying a data gravitation mechanism to adjust the weights of training instances. They leveraged the Transfer Naive Bayes (TNB) to build a classifier on the adjusted cross data. The weights of the training instances are computed according to the similarity between the source instances and the target instance. To calculate the similarity between them, data gravitation mechanism is used. In other words, they stored the min and max values of each feature attributes of the target data, and the resemblance is estimated by the number of feature attributes of a source instance whose feature attributes’ values are between min and max values of the

corresponding target feature attribute. Then, the weights were computed according to the formulation of data gravitation [183]. The more weight of a source data means the more similarity to target data. Finally, the TNB is employed with new probabilities by using these weights either filtering features or instances of the source project that are irrelevant for the target project. Amasaki et al. [184] explored the influence of data simplification by eliminating irrelevant and redundant information. Chen et al [185] adjusted the data distribution between source projects and target project by employing the data gravitation method [183]. The mechanism of this method is to reshape (re-weighting) the instances of cross data (i.e., data of source projects) and make them close to the within data (i.e., data of the target project). Then, they used the transfer boosting learning called TrAdaboost [186] to refine the weights of source projects and build an effective defects classifier with a restricted amount of labelled data in the target project. Yuan et al [187] proposed an approach called ALTRA which aim to reduce the great data distribution difference between source projects and target project by using firstly burak filters to select the most similar labelled instances from source projects, then applying active learning to highlight significant unlabelled instances from the target project, and finally applying TrAdaboost to identify weights of labelled instances of both source and target project and constructing the prediction models via the weights. Some methods are based on feature selection or feature mapping. Yu et al. [188] used correlation-based feature selection to select features which are very close to those of the target project. Nam et al [41] applied the transfer component analysis (TCA) technique to CPDP. Instead of selecting appropriate source projects, TCA makes the distribution of the features between a given source project and the target project similar by mapping the features data from both projects into a common latent space. They further extended their approach to TCA+ with a data pre-processing phase in which they managed the data distribution divergence between the projects by adding decision rules.

Some other approaches focus on the selection of relevant source projects. Herbold et al. [189] proposed a training data selection (TDS) technique to select suitable source projects by using NN filter and measuring the Euclidian distance between the data distributions of the source and target projects. He suggested that TDS plays a significant role in CPDP. The results showed that TDS approach can significantly improve the prediction performance by a percentage of 18%. Liu et al. [190] proposed a method called TPTL which achieves to enhance the CPDP performance by selecting two source projects that are considered as the best candidates for the target project as they share similar characteristics, and then, applying

the transfer learning algorithm to construct a two-phase transfer learning model. Krishna et al. [191] applied the concept of bellwether to select the best candidate among several source projects and construct with it the transfer learner.

As described above, a lot of related works have been proposed on WPDP, and each of them is based on either feature selection or instance selection or project selection. In this research, we mainly focus on project selection.

3.5 Discussion

A significant number of contributions have been proposed to tackle the CPDP challenge by applying different strategies including the selection of source projects, instances, or features to alleviate the great distribution data between source project candidates and the target project. TCA + is considered as the state-of-the-art transfer-learning method for CPDP [41]. However, TCA +'performance is unstable and depends on the selected source projects. It means that the source projects candidates should be carefully considered, instead of randomly selecting one or many source projects. In this way, the performance of CPDP methods could be substantially improved. Additionally, the ability to identify similar projects from which the learning can be performed is of high importance even for the other proposed methods.

Nevertheless, all the proposed approaches, including TCA + are based on metrics to choose the source instances, projects or features whose data characteristics are remarkably close to those of target project. It was proved in the subsection 3.2.2, that the metrics do not provide high performance to extract meaningful code properties and patterns linked to defective code. Therefore, several metric-based methods fail to detect numerous types of bugs. Drawing on this observation, we assumed that applying approaches based on metrics to CPDP setting to select the most adequate source projects and extract useful patterns from them may alter the selection of projects and consequently, degrade the prediction quality considerably. The problem of detecting similar open-source projects is considered as an obviously difficult problem, since it implies the need to consider several parameters to detect similarities among open-source projects. Therefore, it is essential to establish more solid criteria, such as structural and semantic information and dependencies hidden in the source code, the organizational aspect of the project, etc. for a best selection of suitable source projects. We can categorize all these project aspects into two main types of software similarity computation, i.e., high-level similarity and low-level similarity (more details in chapter 5).

From the above CPDP state-of-the-art analysis, we noticed that none of the CPDP methods consider these aspects while choosing the appropriate project. They performed a filtering based on the metric calculation.

To resolve this issue, a flexible framework is required for calculating similarities among projects by using all these aspects to identify project characteristics. This helps to improve the cross-project feasibility and provide a more strong and effective training for the project that has insufficient historical data. Moreover, it helps to detect several types of defects linked to many levels, i.e., structural bugs, semantic bugs, etc. for target projects for target projects.

3.6 Synthesis

In this chapter, we analysed existing research work in software defect prediction extracted from the state-of-the-art analysis. We classified these works into three main research areas including 1) software defect prediction on file-level, 2) software defect prediction on change level, and 3) cross-project defect prediction methods.

We first analysed different approaches applied to represent the source code and extract useful features from it. These approaches are categorized into traditional metric-based approaches and traditional approaches based on trees or graphs. Then, we studied the benefits and drawbacks of each category. The methods based on metrics are easy and simple to compute but they cannot extract meaningful properties of code such as semantic and structural information, despite the importance of these aspects for modelling program functionalities. Moreover, these methods mainly focus on the manual and arbitrary selection of metrics which alter the prediction process. Based on this observation, other traditional methods are proposed to represent the source code. These methods are based either on trees or graphs. The trees based methods fail to extract the execution process of code while the graph based methods provided by CFG fail to extract the intra procedural information. Both of them have its advantages and none of them can replace the other.

From this analysis, we concluded the need to consider simultaneously structural and semantic information in addition to intra-dependencies to deeply mining the code. Secondly, we discussed different learning algorithms used in constructing a prediction model. There are traditional learning algorithms and deep learning algorithms. Contrary to traditional learning models, deep learning methods can extract automatically meaningful features from the code. We also addressed a specific field of deep learning, i.e., graph based convolutional neural

network, which can tackle complex graph data. Based on above data, we noted that a powerful and effective deep learning algorithm is required to leverage automatically complex graph features from the code to construct a prediction model of high quality. Finally, we discussed the different transfer learning methods proposed for cross-project defect prediction. We also showed the importance of selecting adequate training projects that are like the target project, which has inadequate historical data, instead of selecting arbitrary source projects. On these terms, we highlighted the need to consider multiple criteria including structural and semantic information as well as dependencies hidden in the code and consider various levels of project description such as the organizational level, project's usage, etc. to select the suitable source projects.

From the above analysis, we propose the following research direction:

- Propose a more reliable prediction model in terms of decision.
- Enhance the structure and the semantic meanings of the code to detect the maximum types of bugs and improve the performance of the prediction models.
- Choose the optimum source project in the case of cross-project to improve the feasibility and the performance of the prediction model

FOURTH CHAPTER

4 An end-to-end deep learning defect prediction over code property graphs

This chapter presents our proposed end-to-end deep learning framework for developing reliable software by detecting potential bugs. This framework extracts semantic features as well as the dependencies between software components; and then used a deep graph convolutional neural network DGCNN to learn and build defect prediction models on these semantic and dependency features. This framework is applied at file and change granularities and is conducted in two settings: within-project defect prediction and cross-project defect prediction.

4.1 Motivation

Software defect prediction techniques have been proposed to ensure corrective and evolutive maintenance of modern software applications while minimizing software development costs [192]. Any modification in such software applications may give rise to the introduction of new failures. Thus, developers should regularly check that the software application does not involve new defects. Unfortunately, the inspection of the entire code is often challenging and testing all units is not practical due to the limited resources and the tight schedules. To this end, localizing and fixing bugs at earlier stage become an urgent requirement to improve the software quality and make the software-free with least cost.

Software defect prediction models use software history data to learn and build the software models which can predict whether new instances of code, e.g., file or change in this work, include defects or not.

All the above mentioned methods fall into two main direction: one is applying metric based methods which design manually software metrics to extract features (predictors) from history instances (files from previous releases or previous commits) and traditional machine learning are investigated to build predictive models on the metric data and discriminate defective instance from non-defective instance; while the second is using either programs‟ tree representations or control flow graphs to extract relevant semantic features and deep learning networks are further applied to automatically learn distinguishing semantic features from either ASTs or CFGs.

Metric-based techniques mainly focus on designing manually and arbitrarily discriminative features or a new combination of features called software metrics to measure some properties of source code. For example, Halstead metric based on numbers of operators and operands [17]; McCabe's metric estimates the complexity of a program by assessing its control flow graph [18]; CK metrics based on function and inheritance counts [19]; code change features [9] based on a number of lines of code added, removed, etc. Although several robust learning algorithms have been applied for software defect prediction, involving Naive Bayes (NB), Decision Tree (DT), Dictionary Learning [10], Support Vector Machine (SVM), and Neural Net-work (NN), the predictors have not achieved so high performance [11] since they are based on metrics which have several definitions and ambiguous counting and are manually and arbitrarily selected by each researcher.

Thereafter, the input data provided to the classifier no longer concern traditional metrics but represent the syntax and semantic elements of the program by exploiting tree representation of programs – The Abstract Syntax Trees (ASTs). Then, deep neural networks are applied to automatically learn to distinguish features from ASTs since their architecture can effectively capture complex non-linear features. Tree-based methods significantly outperform software metrics-based. Wang et al. [12] leverages a deep belief network (DBN) in learning semantic features from token vectors extracted from programs' ASTs. This paper [26] proposed a tree-based convolutional neural network to extract structural information of ASTs to improve defect prediction.

The AST-based methods are also not perfect because they do not reveal all the types of software defects in the programs, especially those induced by the execution process of programs. Phan et al. [27] proposed an application of a graphical data structure namely control flow graphs (CFG) to SDP. In the field of ma-chine learning, the quality of input data directly affects the performance of classifiers. Regarding this, CFG provides enhanced results relative to previous studies based on metrics and ASTs.

Although the good performance of CFGs, they are only able to capture the execution process within a program and do not identify the intra-procedural dependencies, in other words they cannot capture the behaviour of the program. However, many bugs are directly related to the dependencies within the program [20], [28]. Therefore, both AST and CFG features do not cover all the types of defects in programs and especially those are related dependencies to respond to the constant evolution of software programs in terms of complexity.

To bridge the gap between program's intra-procedural dependencies and defect prediction, this thesis applies the powerful representation code property graph that merges classic concepts of program analysis including abstract syntax trees, control flow graphs, and program dependencies graphs into a common entity structure to represent the properties of the programs, and then performs DGCNN on code property graphs to automatically learn defect features. The key insight underlying this approach is to perform a suitable representation of code that can explore deeply the code by jointly taking into account the syntax, semantic, and control flows well as the intra-procedural dependencies of code to discover the maximum of bugs and improve the quality of prediction. The code property graph has proven successful in the field of vulnerability detection as it enables to efficiently mine large amount of code properties [22]. We also need to examine the code characteristics as comprehensively as possible, not to perform vulnerability analysis, but to feed the deep learning algorithm to improve prediction model accuracy. Therefore, we perform the CPG to carry out a more precise analysis of code and consequently predict different types of bugs. Furthermore, the application of the DGCNN enables to explore large-scale graphs and operate rich and complex information of vertices and edged like code property graphs.

We examine our DGCNN-based approach that generates features including semantic and dependency information on both file-level defect prediction task (i.e., predict whether files in

a current version is buggy or clean) and change-level defect prediction task (i.e., predict whether a current code commit is buggy or clean). Most of existing defect prediction approaches are carried out on these two levels [9], [41], [48], [99], [193]–[197]. Investigating these two tasks enables us to better qualify our approach compared to other existing defect prediction features and methods in the literature. For file-level defect prediction, we stand for the source files from history data by using code property graph to extract features including semantic and dependency information, while for change-level defect prediction, we represent the code change by using code property sub-graphs extracted from code property graphs. Besides, we evaluate our approach in two settings: within-project defect prediction [9], [47], [48], [197] and cross-project defect prediction [41], [99], [194], [196].

4.2 Background

4.2.1 Bug fixing change

After reporting a bug on an Issue Tracking System (ITS, e.g., JIRA and Bugzilla), changes are made to fix this bug. During bug-fixing changes, many lines are changed, removed or added. These lines are called bug-fix lines. Several existing heuristics are used to identify bug-fix changes [198]–[200]. For example, If the change log includes the bug identifier as recorded in its corresponding ITS, then such a change is considered as bug-fixing.

4.2.2 4.2.2. Bug-introducing changes

Bug-introducing changes refer to code changes that possibly lead to a bug fix change in the future system. The bug-introducing change holds a set of lines that are added, modified or deleted. These lines are called bug-introducing lines.

4.2.3 The SZZ Algorithm

SZZ is a widely used algorithm in software engineering community to detect changes that are likely to introduce defects. The original approach (B.SZZ) was introduced by Śliwerski et al. [201]. SZZ aims to make a further link of bug-fix changes to bug-introducing changes based on historical data from issue tracking systems (ITS) and versioning. The SZZ algorithm consists of two subsequent parts:

In the first part, the SZZ approach retrieves all the bug identifiers from the bug report stored in ITS and then checks whether the change log includes the bug identifier [202]. The most recent change of the commit involving the bug identifier is considered as buggy, otherwise it is clean. For the projects which do not have a properly maintained ITS, SZZ considers that the changes whose commit messages involve the keyword “fix” as bug-fixing changes. Then, for each of the identified bug-fixing commits, SZZ extracts the modified lines in the source code.

In the second part, SZZ identifies the bug-introducing changes. For this, SZZ uses the *diff* command provided by the control version system CVS (e.g., git) to determine the lines that have been modified (to fix defects) between the bug-fixing commit version and its previous version. Then, SZZ algorithm employs the *git blame/annotate* functionality to trace back the change history and recover the change that introduced the bugs. i.e., the bug-introducing change. The improvement version of SZZ and provided by Kim et al [46] does not consider the non-semantic lines involving comment lines, blank, and format modifications, to avoid the mislabelling of the changes.

A representative example is shown in Figure 14: An example of a change committed in a file. that stands for three code snippets. The first commit version, *15cf5s*, is the commit that introduces bugs; the bug is introduced in line 15 where the *if* statement represents an incorrect condition. The second change, *6scf27d*, inserts code to the *print()* function in line 18,19, and 20 after the bug was identified. The third change, *27cdf37*, modified the two lines 15 (the buggy line) and 19 to fix bugs.

Figure 14 shows how the SZZ algorithm works to introduce the bug-introducing change. Firstly, SZZ identifies the bug-fixing commit *27cdf37* after looking the bug ID #134 within the log for a commit with the commit message. The bug ID corresponds to the notified bug #134 in the issue tracking system. Then, by using the *diff* and *annotate/blame* functionality, SZZ identifies the bug-introducing changes. In this example, lines 15 and 19 are the lines that have been changed to fix the bugs so there is a doubt that a bug was introduced in these two lines. The two lines have been included in two different commits. However, line 19 was inserted in a commit after the bug was reported, so the bug-introducing commit is the one that changed line 15. Therefore, SZZ deducts that *15cf5s* refers to the bug-introducing change.

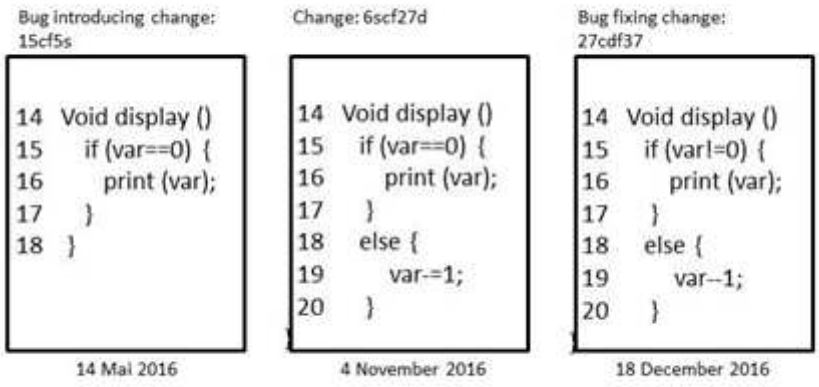

Figure 14: An example of a change committed in a file**.**

4.3 Approach

In this part, we will establish our proposed defect prediction approach depending on the code property graph, providing granular detail and a thorough understanding of data flows. The basic goal is to recognize if a source file or source code change has any defects or not by developing semantic features including dependency information from the source code or changed code snippets directly. These features are mobilized by using DGCNN to make the defect prediction better. The overall workflow of our proposed framework for generating semantic features based on code property graphs for both file-level defect prediction (inputs are source files) and change-level defect prediction (inputs and code changes) are depicted in Figure 15 and Figure 16 respectively. For file-level defect prediction, the source code is analysed and converted for the training as well as the testing source files into the Abstract Syntax Trees (ASTs), Program Dependency Graphs (PDGs), and Control Flow Graphs (CFGs). After this, these three representations are combined to form a common entity structure known as the Code property Graph (CPG). Hence, our approach takes CPG node tokens from the source code of both training and test source files as the input and generates features including semantic and dependency information. Then, the generated features are applied to build the prediction model. Note for change-level defect prediction, the input data that will be fed to the DGCNN algorithm are code property sub-graphs. Since the context and syntax information of changed code snippets is often incomplete, building AST, CFG, and

PDG for these changes directly from code is challenging. Therefore, the learning is carried out with sub-graphs of code property graphs that represent the code change, to consider the structural and semantic information characterizing the potential buggy changes and clean ones (details are in Section 4.3.2). DGCNN requires input data in the form of integer graphs. To this purpose, we build a map among integers and complex tokens and convert the token graphs (i.e., the nodes and edges contain labels) into numerical graphs (i.e., the nodes and edges contain numerical values) by applying the word-embedding. By manipulating these input graphs in the learning phase, the semantic, as well as dependency details about source code/or change code are automatically developed by the DGCNN. Then, the defect prediction models are generated via the training set, depending on the given features and their performance is analysed on the test set in the evaluation phase. Finally in the prediction phase, the high-quality model indicates the probability for every code file/or code change, if the file/or commit is defective or not.

The framework is mainly composed of five steps: 1) labelling and data extraction, 2) parsing source code (source files for file-level defect prediction and change code snippets for change-level defect prediction) into CPG to extract features, 3) encoding the token graphs into numerical graphs, 4) Using the DGCNN to develop defect features, and construct the classifier to identify if the software component (code files or code changes) are defective or clean. We outline the details of each step in the overall framework in the following sub-sections.

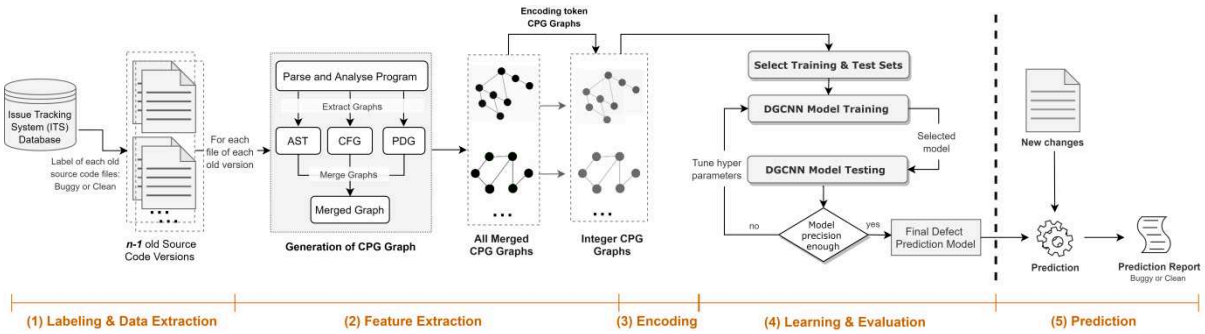

Figure 15: The overall file-level defect prediction

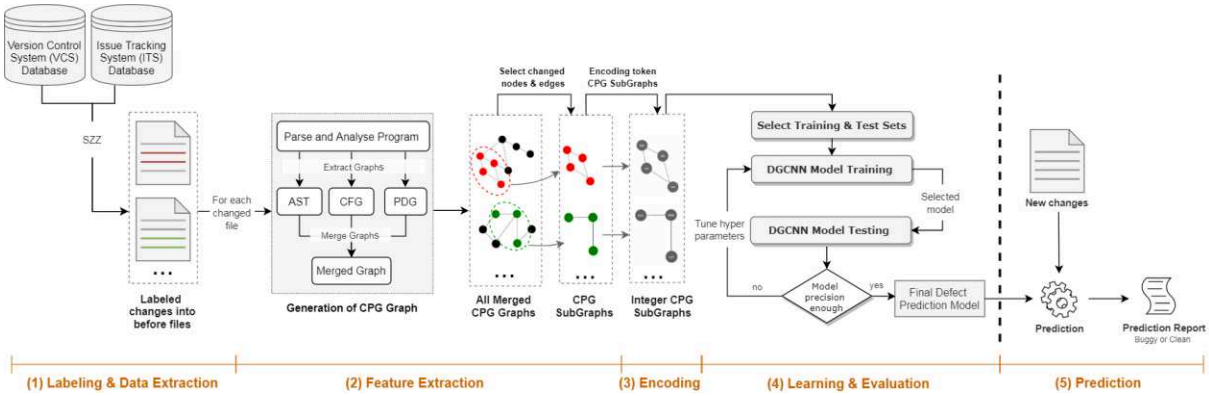

Figure 16: The overall just-in-time defect prediction

4.3.1 Labeling and data extraction

In this step, we give a label to each file/ or commit as buggy or clean. For file-level defect prediction, the labelling process is based on post-releases defects from a Bug Tracking System (BTS) by linking bug reports to its bug-fix changes. Files associated with these bug-fixing changes are considered as buggy. Otherwise, the files are labelled as clean.

Different from file-level defect prediction, labelling change-level defect prediction needs to further link bug-fixing changes to bug-introducing changes. Therefore, we could identify the bug-introducing changes from bug-fixing changes by exploiting the annotation/blame technique provided by the versioning control system SZZ algorithm. These techniques are widely applied by several existing studies [9], [46]–[49]. We firstly identify bug-fixing changes by inspecting the log message of each commit and issue tracking system. However, for the projects which have not a well maintained BTS, we followed the same labelling process of existing studies [32] and considered the commit messages that have the keyword "fix" as bug-fixing changes. Then, lines which are deleted or modified to fix a bug are considered as buggy lines, and the most recent change that introduced those faulty lines (i.e., most recent change after reporting a bug) is considered as a bug-introducing change. Same as other works, we label buggy-introducing changes as buggy and the others as clean.

4.3.2 Parsing source code

4.3.2.1 Parsing source code for files

This step involves the parsing of our Java source code into the ASTs, CFGs as well as the PDGs by employing a LL parser. The LL parser is a descending parser using the context-free grammar. Therefore, a powerful and flexible parser generation tool is used, known as ANTLR. This tool is most used in academics for reading, processing, executing, or translating the organized texts and the binary documents. It offers the lexer and the parser that targets the normal language known by it. For our context, the lexer is found to be interesting in understanding the Java language‟ syntax by generating tokens representing the whole sentence, whereas the parser plays a role in understanding the Java language‟ semantic by generating the syntax trees representing the context-free sentences.

After using the ANTLR tool, every file source is converted into the AST to capture the syntactic details from the source code. In the program, every node exhibits a construct that occurs in it. In chapter of background, it is well explained that we have the root representing the entire source file and its children as a top priority of each file i.e., the imports, class statements, etc. Every tree node has its own AST type for labelling, such as Block, while, if, for, declaration, as well as AST name i.e., the class and method name.

The CFGs can be developed by operating the ASTs i.e., the designed control nodes such as while, if, try or for, are considered to develop a primary CFG. Then the unstructured control nodes, including the break, continue, or go-to, are considered to complete its build-up.

After this, PDG is developed from the CFG. The PDG contains similar nodes as that of CFG however they are interacted by 2 ways, i.e., the control flow and the data dependence. So, it can be said that the PDG is a merge of CDG and DDG. The CDG is calculated from the CFG by using a control dependence assessment, calculating the most powerful post dominator for the various circumstances within the CFG. The computation of DDG is performed by taking the data flow analysis, usually by reaching the definitions, by which the defuse pairs can be developed, which are comprised of the edges in DDG.

Finally, all the information about the three representations is stored into a unique representation CPG through the AST nodes of declaration and expressions as explained in the section data representation in the background chapter. Hence, we get every file code

represented as the CPG, “the graph of graphs”. Usually, its nodes mainly match the AST nodes however the edges represent the AST edges and both CFG and PDG edges. The nodes and edges of CPG are taken as complex tokens having preserved structural, dependency as well as contextual in-formation and utilize these graphs as the DGCNN inputs.

4.3.2.2 Parsing source code for changes

For change-level defect prediction, the objective is to represent the code change by a suitable representation as code property graph and extract meaningful features that will be fed to the deep learning algorithm for learning the typology of bugs that occurred in previous commits. Since the syntax information of change data is often incomplete, building AST, CFG, and PDG for these changes directly from code is challenging. Therefore, the learning is carried out with sub-graphs of code property graphs that represent the code change, to take into account the structural and semantic information characterizing only the potential buggy changes and clean ones. To do this, we follow the same parsing process as for file-level and represent each file that introduced changes as a code property graph. Then, we extract the code property sub-graph from the code property graph, which represents only the code changes, instead the complete code of the file that introduced changes. To do this, we select only the nodes which are made from changed lines and all their direct neighbours as well as all the corresponding edges. Figure 17 represents the code property graph to the corresponding sample code in file1.java. The sample code is the same example given in chapter 1. It is about an implementation of a simple functionality in a human resources context whose purpose is to compute the salary increase percentage. As explained in chapter 1, this example makes a dead assignment, and this weakness could be an indication of a significant logic error in the program or an indication of poor quality. In Figure 17, the nodes of the sub-graph are coloured in red. The nodes in dark red represent the code change while the nodes in light red represent the direct neighbours. The code property sub-graphs are constructed by following the steps below: 1) we identify firstly all the lines that have been changed. For each file that represents the last version of the files before introducing changes, we annotate all the modified or deleted lines corresponding to their changed lines by adding a comment with a specific format: « //[Unique-Identifier]_T » with T = {M (modified), D (deleted)}. Figure 18 depicts a sample of code change that introduces a bug. As we can see in

file2.java in Figure 18, line 5 was modified by adding the variable raise to fix the logic bug described above. Thus, we annotate the line 5 in Figure 18, by adding the specific comment and the variable T takes the value M to indicate that this line has been modified. 2) In the second step, we need to store whether the nodes representing the CPG of each file is making from a changed line or not. Therefore, we assign the type T affected to the changed line to its corresponding node in CPG. Taking the example of the code sample in Figure 17, the node corresponding to the print () function is assigned by the character M which is given as an annotation in the corresponding line 5. 3) Finally, we select only the nodes having the type M or D and all their direct neighbours as well as all the corresponding edges to extract the code property sub-graph that represents only the code change of the file.

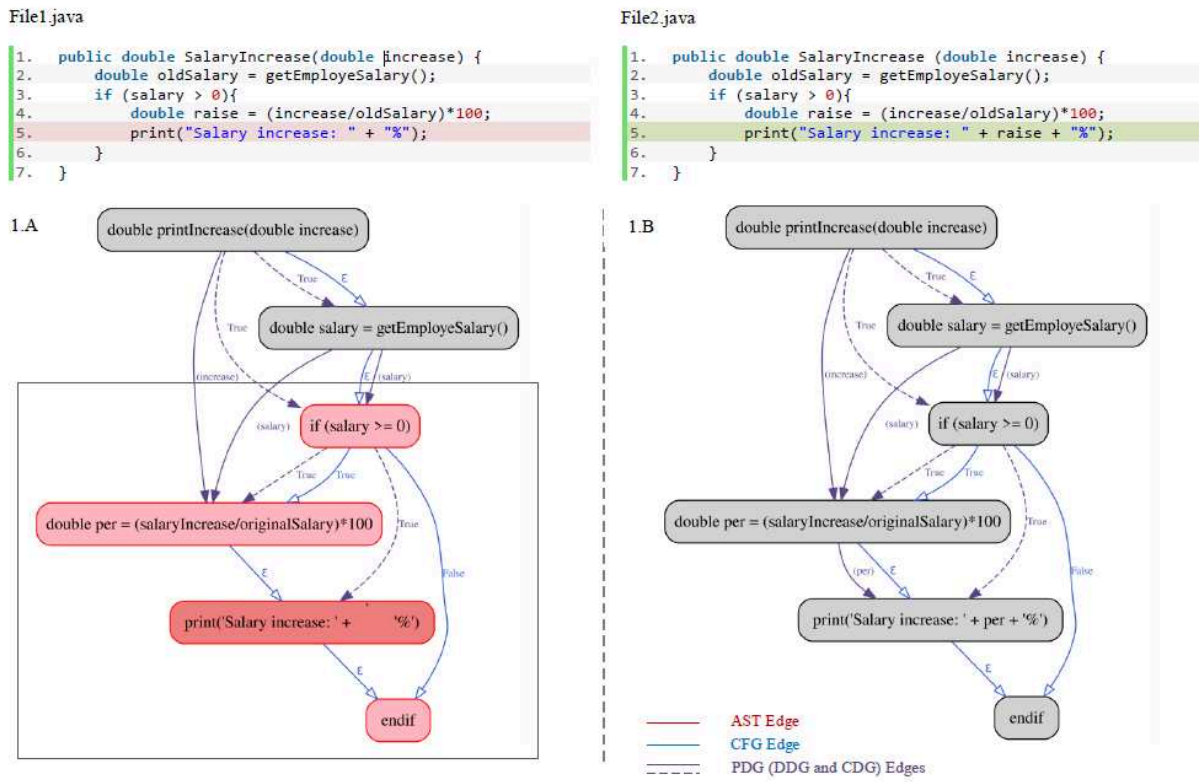

Figure 17: A motivating example. The variable increase corresponds to the difference between the new salary and the old salary. The raise percentage which is defined by the variable raise is computed in line 4 in both file1.java and file2.java.

```
1.  public double SalaryIncrease(double increase) {
2.     double oldSalary = getEmployeSalary();
3.     if (salary > 0){
4.        double raise = (increase/oldSalary)*100;
5.        print("Salary increase: " + "%");  //[8762aD6b9&l]_M
6.       }
```

Figure 18: The identification of change introducing bugs. The unique identifier is to separate the original comment from the specific one in the file introducing bugs, and the characters M and D represent the modified line and deleted line respectively.

4.3.2.3 Encoding token graphs

The numerical graphs are required by the DGCNN as input therefore the generated token graphs are not forwarded to the DGCNN directly. A token vector is used to represent the vertex label in CPG. Because it's not just a simple token as it comprises of intricate information, representing in instruction that may include several parts such as the name of instruction or different operands. Moreover, every instruction can be observed in the form of types of functions of instruction.

To use the DGCNN for generating complex features, it is essential to construct a map among the integers and the token, followed by the encoding of token graphs in the integer graphs by using a well-known method, known as Word2Vec [21]. Every token is linked with an exclusive integer identifier. By this, similar tokens are kept by one identifier and the various tokens are kept under different method names and class names. Furthermore, the input vectors are required by the DGCNN to maintain an equal length. As there is a possibility of different lengths of our converted integer vectors, we add 0 to every integer vector to equalize all the lengths and to maintain a consistent length as the longest vector. There is no impact of adding 0 because the range of encoding takes starts from 1 to the total types of tokens.

Furthermore, the infrequent tokens are usually not considered because they are developed for files/changes, not for every file/change. Therefore, only those tokens are encoded that are present in three or more than three numbers, while 0 is assigned for others.

Note that in this work, we perform the same token mapping process for both file-level and change-level defect prediction. However, the token vectors that represent the nodes and edges of token graphs for file-level are those from code property graphs while for change-level are those which represent the code property sub-graphs.

4.3.2.4 Employing Deep Graph Convolutional Neural Networks DGCNN

In this step, we employ the DGCNN to generate automatically the features and construct the predictive model by taking as input the code property graph representing each file or the code property sub-graph representing the commit.

4.3.2.5 Building Classifiers and Performing Defect Prediction

The process mentioned above, permits to generate the semantic features automatically such as the intra-procedures dependencies for every file/change in training data as well as the test data. The classifier can be built and trained by using their features as well as their labels i.e., defective or clean, and then the test data is used to analyse this classifiers' performance.

Note that k-fold cross-validation is extensively used validation method by researchers. The process of cross-validation is as follows: 1) Divide the dataset into 10 folders randomly; 2) use 9 partitions as training set and one partition as test set; 3) repeat the process by changing the test set until all data have a predicted label; 4) performing the evaluation by comparing the predicted labels and the real labels of the data.

However, according to these papers [48], [204], the k-fold cross-validation has two issues in practice, and especially, for change-level defect prediction. Changes C1-C7 are committed following a certain order in time, where C1 is firstly committed and C7 is the most recently committed. Dots represent the buggy changes, and circles represent clean changes. The narrows link each bug-fixing changes to its corresponding bug-introducing changes (buggy changes). For example, C7 fixes the bug in C6, therefore C6 is buggy.

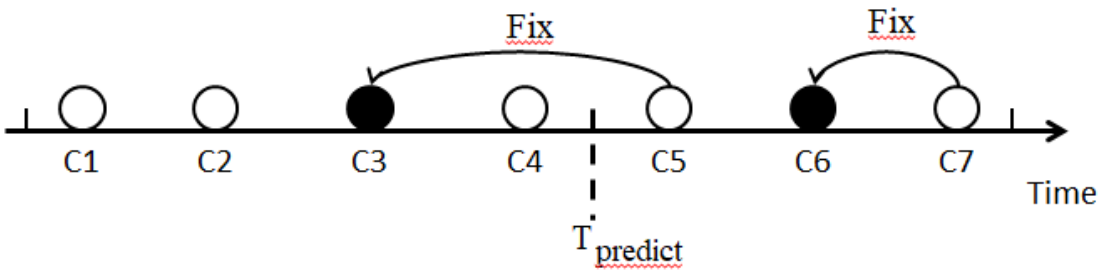

Figure 19: Demonstrating the issues arising from the use of cross-validation method to change-level defect prediction

The first problem of the method is that it may use future data to make the prediction. For example, in some iterations, the k-fold cross validation can use C2-C7 to predict whether C1 is buggy or not, which cannot represent a practical case as the prediction is made just after the C1 is committed, C2-C7 do not exist yet.

Then, cross-validation can introduce biased dataset. For example, C3 will be labelled buggy change following the Figure 19. However, in practice, when we will predict whether C4 is buggy or not at time $t_{predict}$, we will have only the information of that moment $t_{predict}$. Therefore, C3 should be labelled clean as C5 is not available yet at that time. So, it is not relevant to consider C3 as buggy when we predict C4 at time $t_{predict}$ because we won't know that C3 is buggy at time $t_{predict}$.

To conclude, the k-fold cross validation may make the evaluation inaccurate as it regularly incorporates nontrivial bias for assessing defect prediction models. Besides, it could also make the evaluation incorrect for change-level defect prediction as the changes respect a certain order in time and the k-fold cross validation method can use data from future knowledge which must not be recognized at the time of prediction.

To conclude, we do not apply the k-fold cross validation in our work to avoid the validation problem described above. For file-level defect prediction, we evaluate our proposed DGCNN-based features and other existing traditional features by building prediction models with data

from different historical releases, while for change-level defect prediction, we gather training and test data according to the time order as explained later in section 4.4.2.2 to construct and assess the prediction models.

4.4 Experiments and results

In this section, we evaluate the effectiveness of our proposed semantic and structural features based on graphs for both file-level and change-level defect prediction and compare it with the state-of-the-art-methods. The experimental environment for the generating features is an Intel(R) Core (TM) i5-M540 CPU @2.53 GHz with 8.0 GB RAM laptop running Windows 8.1 (64Bits). And the experimental environment for the deep learning execution is an Intel® Xeon® E3-1220v5 1CPU (4C/4T) with @3GHz 32GB RAM and GeForce GT 710 1GB GPU running Linux CentOS 7 (64 Bits). Initially, we present the different scenarios that our experiment takes. Then, we detail the used standard datasets and the experiment setup. After this the baseline techniques are presented and the evaluation criteria for used performance are described. Then, we evaluate the impact of the tuned parameters of DGCNN on the performance of our proposed technique for file-level defect prediction. Finally, research questions (RQ) are proposed and answered.

4.4.1 Research scenarios

Our evaluation experiments are performed under four different prediction scenarios to compare and explore the effectiveness of our proposed approach with traditional existing methods. The following Table lists the investigating different scenarios which are presented in the form of four research questions. All the investigated questions have the following format.

RQi ($1 \leq i \leq 4$): Do DGCNN-based semantic and dependency features extracted from code property graph outperform traditional semantic features at the <level> <setting> under both the non-effort aware and effort-aware evaluation scenarios.

For example, RQ1 is investigated to evaluate the performance of the proposed approach DGCNN-based features extracted from CPG at the file-level within-project under both the non-effort aware and effort-aware evaluation scenarios.

		Setting	
		Within-project	Cross-project
Level	File-level	RQ1	RQ2
	Change-level	RQ3	RQ4

4.4.2 Dataset

In this section, we list the different datasets used for assessing the method's performance 'applying DGCNN on code property graphs' for file-level and change-level defect prediction. Specifically, for file-level defect prediction, the datasets are achieved from the publicly accessible data from PROMISE repository which are widely used by researchers to assess file-level defect prediction models [10], [40], [41], [194], [196]. For change-level defect prediction, we have selected only four java projects from six existing widely used datasets for evaluating change-level defect prediction tasks [9], [48], [197] as our proposed approach is limited to Java language.

The major justification for using these widely used datasets for the assessment of defect prediction models for both file-level and change-level defect prediction is that permit us a direct comparison between our approach and the existing models of defect prediction on the same datasets; and thus, a process-safe evaluation is obtained.

4.4.2.1 Dataset for file-level defect prediction

We select all the open-source Java projects from the publicly accessible data PROMISE collected by Jureczko an Madeyski [33] for the evaluation of our experiments for file-level defect prediction. These evaluation projects released different versions from the repository. A huge domain of applications is covered in these projects, including XML parser, enterprise integration, text search engine library, and text editor. Dataset also consists of the version numbers, files class name, and well as their defective labels. The project' different versions

can be extracted from GitHub if it is available or from their official websites and can be applied in our framework. For each project, we need the number versions that constitute the source code archive from which we extract token graphs from the code property graphs of the archive data to feed our DGCNN-based generation features. In total, we gathered 10 Java projects. The Table 5 indicates each detail of this project, such as the project's description, number of files, versions, and an average defective rate for each file. The average number of files range from 122 to 815. The buggy rate has at least 9.4% and as highest as 62.9% value. The major justification for using these datasets is that they have been widely used in previous file-level deep learning-based defect prediction studies [205], [206]. Therefore, it permits a direct comparison between our approach and the existing models of defect prediction on the same datasets and thus a process-safe evaluation is obtained.

The second dataset used for file-level defect prediction task includes some GitHub projects[2]. These projects are named large-size datasets. Table 6 shows details of each project in GitHub repository in terms of description, version, average number of files and average defective rate. The versions of GitHub projects are represented by the day when their corresponding defect data is gathered. Finally, other Java projects are applied in our experiments. The projects DrJava, Genoviz, Jmri, Jmol, and Jppf are collected from a dataset introduced by Shippey et al. in their proposed approach ESEM in the paper [207]. Table 7 defines each project.

Table 5: Details of the evaluated projects for file level defect prediction from Promise Repository

Project	*Description*	*Versions*	*Average # Source Files*	*Average Buggy Rate (%)*
LUCENE	Java based build tool	2.0, 2.2, 2.4	488	13.4
LOG4J	Logging library for Java	1.0, 1.1	122	29.1
IVY	Dependency management library	1.4, 2.0	296.5	9.4
JEDIT	Text editor designed for programmers	3.2, 4.0, 4.1	297	27.4
CAMEL	Enterprise integration framework	1.2, 1.4, 1.6	815	22.5
SYNAPSE	Data transport adapters	1.0, 1.1, 1.2	211.7	25.5
ANT	Java based build tool	1.5, 1.6, 1.7	463.7	21.0
XERCES	XML parser	1.2, 1.3	446.5	15.7
XALAN	A library for transforming XML files	2.4, 2.5	763	32.6
POI	Java library to access Microsoft format files	1.5, 2.5, 3.0	354.7	62.9

[2] http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/

Table 6: Details of the evaluated projects from GitHub repository

Project	*Description*	*Versions*	*Average Source Files*	*Average Buggy Rate (%)*
BROADLEAFCOMMERCE	Entreprise eCommerce Framework	2013-09→2014-10	2114	8.19
ELASTICSEARCH	Distributed RESTfu search engine	2013-08→2014-02	4664	19.15
HAZELCAST	Highly scalable data distribution platform	2013-11→2014-11	1710	10.00
NETTY	Event-driven asynchronous network application framework	2013-02→2014-02	1136	18.57
ORIENTDB	Multi model database system	2013-06→2014-06	1635	9.68

Table 7: Details of the evaluated projects from shippey **[207]**

Project	*Website*	*Versions*	*Average Source Files*	*Average Buggy Rate (%)*
DrJava	http://drjava.org/	2008→2009	814	20.4
Genoviz	https://sourceforge.net/projects/genoviz/	6.0→6.1	704	32.4
Jmri	http://jmri.sourceforge.net/	2.4→2.6	2241	23.3
Jmol	http://jmol.sourceforge.net/	6.0→7.0	291	43.7
JPPF	http://jppf.org/	5.0→5.1	1621	15.4

4.4.2.2 Dataset for change-level defect prediction

We selected four Java open-source: Jackrabbit, Lucene, Jdt (from Eclipse), and Eclipse platform among six widely used projects for evaluating change-level defect prediction tasks. These projects have enough change histories to construct and evaluate change-level predictive models and are frequently used in the literature. We rely on the SZZ algorithm (described in the background section) to label the bug-fixing changes of these projects.

Table 8 shows the details about these evaluated projects in terms of LOC and the number of changes. The LOC of the files and their corresponding number of changes include only Java code source. We focus only on classifying source code changes in the change-level defect prediction setting. Our DGCNN-based features generation is not applicable on other

languages. Thus, we cannot select the two remaining evaluated projects to perform our experiments as their source codes are written in C language.

Table 8: Selected Java open-source Projects for change-level defect prediction. LOC is the number of lines of code. First Date is the date of the first commit of a project. Last Date is the date of the last commit of a project. Changes are the number of changes.

Projet	LOC	First Date	Last Date	Changes	Average Buggy rate (%)
JDT	1.5M	2001/06/05	2012/07/24	73K	20.5
Lucene	828K	2010/03/17	2013/01/16	76K	23.6
Jackrabbit	589K	2004/09/13	2013/01/14	61K	37.4
Platform		2001/20	2007/12	64K	25

The data used for change-level defect prediction are often imbalanced, i.e. the training dataset contains fewer buggy instances than clean instances [9], [95], [208], [209]. As it is shown in the Table 8, the average ratio does not exceed 37.4 percent. The imbalanced data can introduce noise/bias and lead to a poor prediction performance [48]. The imbalance data issue is an open question that should be investigated by researchers and remains our future work. However, in this thesis, we only follow the same settings as Tan and Wang' paper (i.e. we apply the online change classification process instead of time sensitive change classification) [48]. The online change classification process allows us to overcome the issues of the k-fold cross-validation method described above and to have a fair comparison in our evaluation experiments.

To classify a change that is committed at time $t_{predict}$ i.e., the change that represents the test set, time sensitive change classification uses the changes committed before that time as training set to build models. However, this method has three limits. Firstly, the training set can introduce noise data, meaning that many changes can be mislabelled. For example, in the Figure 19, C3 is labelled as clean at the time prediction $t_{predict}$ while it is buggy in practice. The labelling of C3 depends on the information that we have at time $t_{predict}$, and currently C5 that fix the bugs in C3 does not exist yet. Typically, bugs take years to be discovered and fixed [210]–[212]. Therefore, many buggy changes in the training set, and especially those are committed close to the time prediction will be mislabelled as they would not have been discovered and fixed yet. Secondly, the prediction performance of time sensitive change

classification relies on the training set, meaning that the training set picked up from a time period may do not represent the changes from other period of time. Finally, the changes considered in the test set may be different from those of the training set in terms of development characteristics such as programming style, developer experience, etc. especially when they are committed over a long duration. So, this can make difficult to construct more accurate prediction models as the training set may be too old compared to the test set.

To address the three limitations, Tan et al [48] proposed a new approach called *online change classification.* This approach leaves a gap between the training set and the test set which allows a more balanced training set and let to have more time for buggy changes to be found and fixed. In this way, as it shown in Figure 20, each project has several runs. Each run is composed of a training set, test set and a gap between the two sets. The gap is based on the setting values present in the Tan et al. 'paper [48]. The gap' values are between 0.2 year and 1.0 year. By using this method, the prediction is applied on multiple test sets; to avoid being dependent on a particular test set. Besides, the training set is constantly updated with new data when starting a new prediction. For each run, the data following the training set is added to the training set in the previous run. So, the training set and the test are more susceptible to have similar characteristics for building more accurate prediction models. The result is therefore the average performance of these runs.

Figure 20 depicts an execution of two runs adopted in the paper [48]. The second run combines the data included in the training set in the first run and the data from time T2 to T3 to construct the training set for the second run (i.e., the training set in the second run is constituted of changes from T1 to T3). We followed the Tan et al. 'paper in the time settings. Indeed, the duration of the update-time (i.e., T2 to T3) is the same as the duration of each test set. The gap is from T3 to T5 in the second run. The test set in the second run is composed of the most recent changes from T5 to T6. The recent time prediction is T6; therefore, the labelling of the changes that constitute the new training set depends on the information available at time T6.

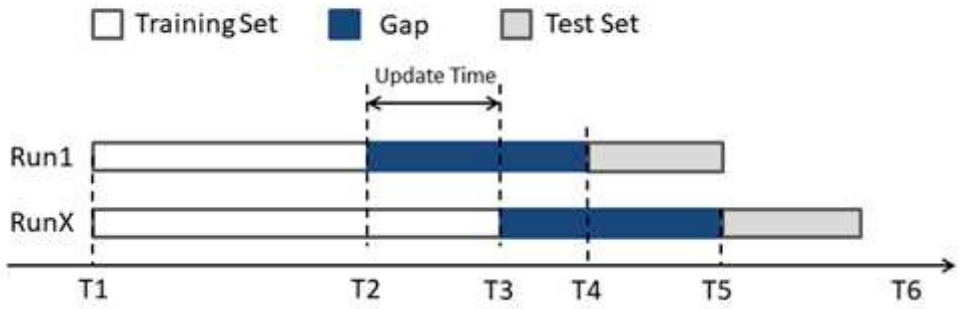

Figure 20: Example of runs

4.4.2.3 Baseline methods

Baselines for evaluating file-level defect prediction

To assess the performance of semantic and dependency features for file-level defect prediction, we camper the semantic and dependency features with traditional semantic features based on AST. It is interested to set a comparison with approaches based on CFG such as the approach proposed in the paper [27] but a comparison cannot be made for this as it targets to design a method for identifying the defective source code implemented in C language. Regarding the features based on software metrics, we do not compare the extracted features with these because they have been compared already in previous studies [12], [26], [68]. In these studies, thorough deep learning used to learn the defect predicting features automatically outperforms software metrics. Finally, we just established a comparison of our method with the given baseline methods:

- **DBN** [12]]: The state-of-art technique depending on the AST nodes that applies the DBN i.e. the Deep Belief Network on source code to capture the semantic features to predict the defects.
- **DP-CNN** [26]: a structure that develops the semantic and structural features automatically by using source code and combines the traditional metrics for a precise software defective prediction.
- **TCA+:** is proposed by Nam et al [41]. It is considered one of the state-of-the-art methods in cross-project defect prediction. The core objective of this method is to reduce the data distribution between the target project and the available candidate source projects.

- **Node2defect:** is proposed by Qu et al. [130]. It uses network embedding technique to automatically learn structural features and the machine learning Random Forest to predict defects.
- **MPT-embedding:** it is a deep learning method which represents the code by AST from multiple perspectives and apply the CNN to construct the defect prediction model [132].
- **Seml:** It is a framework that combines word embedding and deep learning LSTM for predicting defects [213].

Baselines for evaluating change-level defect prediction

The performance of semantic and structural features based on graphs proposed for change-level defect prediction; is evaluated by comparing it with the given baseline methods:

- **DBN-based features** [179]: This method tokenize code changes by considering different combinations among the three different types of tokens (added, deleted, and context) and then perform the DBN algorithm.
- **CBS+** [214]: a simple supervised predictive model that leverages the idea of both the supervised model (EALR) [10] and the unsupervised model (LT).

4.4.2.4 Performance evaluation criteria

To analyze the precision of the predictive models, we used the non-effort-aware and the effort-aware analysing metrics.

Metrics for Non-effort-aware Evaluation

In the case of non-effort-aware, three performance metrics were used that are commonly adopted by previous studies to analyze the models of defect prediction [10], [39], [41], [48], [103], [205], [215]. These measures include recall, precision, and F1 score, and are described as follow:

Precision depicts the ratio of accurately predicted defected files to all the files predicted as defective. It can be calculated as:

$$Precision = \frac{\text{true positive}}{\text{true positive+false positive}} \quad (1)$$

Recall presents the ratio of accurately predicted defected files to all of the true defected files. It can be calculated as:

$$Recall = \frac{\text{true positive}}{\text{true positive+false negative}} \quad (2)$$

F-measure calculates the weighted harmonic average of the precision and recall. It can be computed as

$$F1\ score = \frac{2*\text{Precision}*\text{Recall}}{\text{Precision+Recall}} \quad (3)$$

In these equations the true positive number exhibits the total amount of defected files (or changes), which are defected truly, whereas the false positive exhibit the amount of predicted defected files (or changes) that was clean. The false negative represents the amount of predictive non-defected files (or changes) while they were defective in real. The two measures recall, and precision may be incomplete to compare defect prediction models. For example, if all the files (or changes) are predicted as defective then a high recall will be obtained as 1 with low precision; or if the predicted instances have higher confidence values as defective instances than a higher precision score is obtained with low recall. Thus, to overcome these issues, a trade-off can be seen among the metrics precision and recall by computing the F1 measure. F1 score is considered as a complex measure of the precision and recall, to measure the performance of defect prediction. Hence the greater F1 measure the better will be the prediction performance.

Metrics for Effort-aware Evaluation

Under the effort-aware scenario, we use the PofB20 metric [9] for identifying the accurate percentage of defects observed by monitoring the first 20% lines of code. Hence, the monitored lines of code and total discovered defects are collected until the inspection of 20% lines code is completed for every instance of test data. In the end, the total percentage of identified defects is referred to as the PofB20 score. Usually, a higher level of this score is a

good sign, indicating the approach performance to be analysed. Hence the developer can detect the bugs by just inspecting the few numbers of source codes.

4.4.2.5 Parameter Settings for Training a DGCNN

Like several deep learning algorithms [173], [216], [217], DGCNN requires well-tuned parameters for an effective training, i.e. 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the number of iterations (i.e. epochs). In this section, we evaluate the impact of these parameters on the performance of the proposed defect prediction model.

Setting parameters for file-level defect prediction

To tune these parameters, we conduct experiments by varying the values of the three parameters on five different projects: camel (1.2, 1.4), ant (1.5, 1.6), jEdit (4.0, 4.1), poi (1.5, 2.5), and lucene (2.0, 2.2). For each project, we train and build the DGCNN defect prediction model, with respect to the specific values of the parameters, by using the older version of the project as training set and the newer version as test set. Finally, we compute the average F1 score of the five projects for file-level defect prediction to study the impact of the three parameters on the performance of the proposed approach.

Setting the number of hidden layers and the number of nodes in each hidden layer.

We tune the number of hidden layers and the number of nodes in each hidden layer together as they interact with each other. In the experiment, we set the number of hidden layers to 7 discrete values that include 2, 3, 5 10, 20, 50, and 100. For the number of nodes in each hidden layer, we set 7 discrete values which are 16, 32, 64, 128, 256, and 512. The number of iterations (or number of epochs) takes the value 50 and remains constant during the evaluation of these two parameters.

Figure 21 shows the average F1 score when tuning the two parameters together for file-level defect prediction. By fixing the number of nodes in each hidden layer and by varying the number of hidden layers, all the average F1scores form a convex curve. Most curves reach a

peak at the point where the number of hidden layers takes the value 3. When the number of hidden layers remains fixed while varying the number of nodes in each layer, the best values of F1 scores are reached when the number of nodes in each layer is 32 (the best values are represented by the top line in Figure 21). Therefore, we select the number of hidden layers as 3 and the number of nodes in each layer as 32.

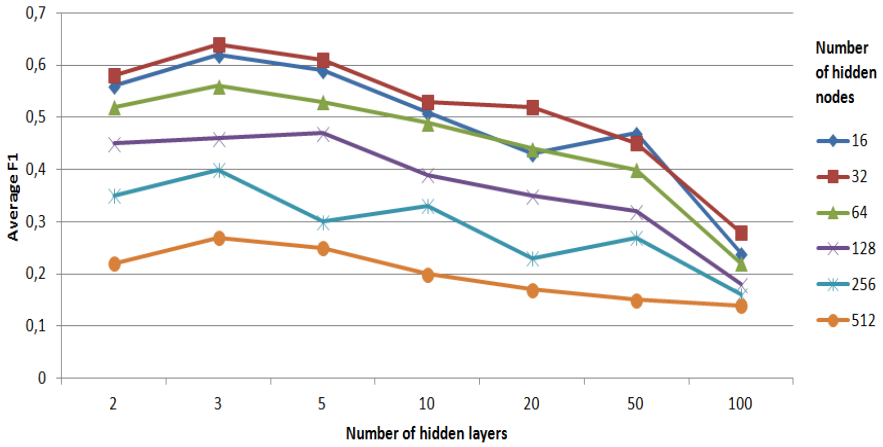

Figure 21 : File-defect prediction performance with different number of hidden layers and number of nodes in each hidden layer

Setting the number of epochs.

The number of epochs is considered as an important parameter for constructing an effective DGCNN. This parameter is related to the number of rounds of optimization that are applied during training to adjust the weights and reduce the error rate. Generally, the more the number of epochs is increased, the more the error rate decreases. However, too many epochs may cause an over-fitting and a slow training time. Therefore, it is necessary to make a compromise between the number of epochs and the execution time cost. We select the same projects to carry out the experiments to tune the number of epochs for file-level defect prediction. We set the number of epochs to 10 discrete values including 1, 10, 20, 50, 100, 200, 300, 500, 1000, and 5000. We evaluate this parameter by using the error rate. Figure 22 shows that when we increase the number of epochs, the error rate decreases slowly while the time cost raises exponentially. In this study, we set the number of epochs to 200 when the error rate takes the value 0.16 and the time cost is equal to 40.4 seconds

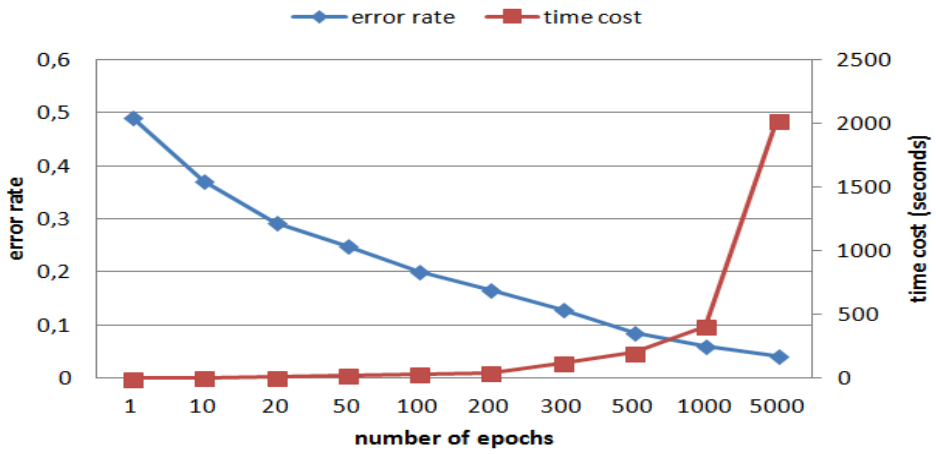

Figure 22: Average error rates and time cost when tuning the number of epochs

Setting parameters for change-level defect prediction

Note that for change-level defect prediction, we perform the same parameter tuning process and then we select the best parameter values as the file-level defect prediction. The selected parameter values are 200 for the number of epochs, 32 for the number of nodes in each layer and 3 hidden layers.

4.4.2.6 Experiment setup for file-level Within-Project defect Prediction

We performed numerous experiments to analyze our approach's performance in file-level within-project defect prediction and comparing it with already existing methods. We have developed models for defect prediction according to the standard process of defect prediction mentioned in the approach part. To analyze the precision of developed defect prediction models, the test data is utilized.

As per the state-of-the-art [32], [133], we use the following files; having 2 successive versions for each project described in Table 5, the oldest one is used as a training set, and the newest one as a test. The number of epochs is another significant feature for developing the defect predicting classifier. We set the epoch numbers to 200 just to have a trade-off among the cost of time and the iteration numbers. As explained in the previous sub-section, we acquire a suitable performance during the prediction classifiers learning stage. In the end, every

experiment was repeatedly conducted for 10 iterations and the means of results were given. In the end, every experiment was repeatedly conducted for 10 iterations and the means of results were given.

4.4.2.7 Experiment setup for file-level Cross-Project defect Prediction

Developing precise prediction models for the new projects is a challenging task because the new projects don't possess sufficient data for training. To sort out this issue, the approach of cross- project defect prediction is used by the literature.

The objective of cross-project methods is to train the predictive models by utilizing the data of existing projects, known as source projects. After this, the trained models are used for predicting the defects in new projects, known as target projects. Yet it's challenging to provide a precise cross-project defect prediction because the source and the target projects don't possess similar characteristics and the features possess various distributions [40].

One just wonders if it's possible to use our proposed method, based on the CPG in cross-project defect prediction to identify the patterns of defects that suggest if the trained features from a project can be used to predict a new project.

To our belief, the proposed semantic features can be used in cross-project however it is better to confirm its efficiency. So, the performance is evaluated for these features such as syntax, data, and control flow of cross-project defect prediction, by using the training data from source project, leading to develop model for defect prediction on target project.

We selected the DBN_CP [32] and TCA+ [41] as baselines and compared them with our method from the perspective of the cross-project. DBN_CP is a technique proposed by Wang et al. [32] to measure the performance of the semantic features in cross-project defect prediction; and TCA+ is considered one of the state of the art methods in cross-project defect prediction [41]. The generation process of the test pairs is described as follows: we select each version from one project as a target project and all the other versions from the other projects as a source project. For example, we take ANT 1.5 as a target project. We select all the versions from other projects to predict ANT 1.5 except the versions from the project ANT. In total, 606 pairs are composed.

4.4.2.8 Experiment setup for Change-level Within-Project defect Prediction

For each project listed in Table 8, we used the training data to construct the predictive model based on semantic and structural features and apply it to the test data to analyze the accuracy of the built model. As explained in the section dataset for change-level defect prediction, the change-level data are always imbalanced which led to a poor prediction performance. To overcome this issue, we used the online change classification process. In this way, we generated the semantic and dependency features and constructed the defect prediction model by using the training set, for each run of a project listed in Table 8. Then, we evaluated the performance of the model on the test data in this run. The global performance is measured by the average performance of all the runs.

4.4.2.9 Experiment setup for Change-level Cross-Project defect Prediction

To develop just-in-time defect prediction models (or change-level defect prediction) for the projects which are in their initial development, and they have not enough training data, the state-of-the-art proposes change-level cross projects. The objective of cross-project methods is to train the predictive models by utilizing the data of existing projects, known as source projects. After this, the trained models are used for predicting the defects in new projects, known as target projects.

To this end, we proposed the approach code property graph-based features for change-level cross-project defect prediction. Specifically, for each project, we select the test dataset in all runs as a target project and we use the training dataset in all runs from the other projects as a source project to constitute the test pairs in the change-level cross-project. An example of a test pair can be as follows: A test dataset from Run 1 of Project A and the training dataset from both run 1 and run 2 of project B, etc. In total, 1120 test pairs are constituted.

To evaluate our proposed approach CPG based features, we selected the DBN_CCP [32] and CBS+ [214] as baselines and compared them with our method from the perspective of the cross-project.

4.4.3 Results and analysis

In this section, we present the results of our experiments. Particularly the evaluation of efficiency of the CPG-based features proposed for file-level defect prediction and change-level defect prediction under different evaluation scenarios; by setting a comparison of the proposed approach with the state-of-the-art method and by answering the following given research question (RQ).

4.4.3.1 RQ1: Do code property graph-based features learned from DGCNN outperform traditional features for file-level within-project defect prediction?

Non-effort-aware evaluation

A file-level within-project defect prediction models based on DGCNN is built to answer to this question and then the purpose of six sets of features was compared, i.e., the semantic features containing the data and control flow, which are learned automatically by DGCNN, DBN, DP-CNN, Seml, node2defect, and MPT embedding. The two baselines feature DBN, and DP-CNN are based on the AST, and the baseline Mt embedding is based on AST from multiple perspectives to extract structural and semantic features. The baseline Seml extracts the features from the code based on word embedding from AST while the baseline node2defect used network measures to extract code properties. The main reason to compare these is to verify the efficiency of deep learning methods depending on graphs, especially the graphs of code property in the defect predicting software. During our research, 16 sets of experiments were conducted for the defect predictions on those given in Table 1, and in each of the two versions were used from a similar project. As we have mentioned that the older versions are used for training the predictive models and the latest versions are used for evaluating the trained models as test sets. The results are displayed in Table 9 indicating the F-measure of the file-level within-project defect predicting experiments. The maximum F1 in all the six sets of features is exhibited in bold.

By considering the example of synapse project, after using synapse 1.1 as the training set, and using synapse 1.2 as the test set, the F1 of defect prediction is 87.9%, while the F1 is only 56.3%, 55.6%, 51.5%, 55.46, and 67.2% from the baselines DBN, DP-CNN, Seml, MPT-embedding, and node2defect, respectively. Moreover, it can be observed in Table 9 that mostly out CPG based approaches get the maximum F measures, indicating that the CPG-based DGCNN is valuable in defect predicting field.

For comparing this, we observed that the only difference is in the learning of features, i.e., the similar parameters and the similar training, as well as test sets, was used. But the various classifying algorithms are used to apply, and various techniques of code representation have been used.

Overall, our CPG-based features acquire an F1 of 75.6%, whereas the F1 of 65.57% has been achieved by the node2defect features, and an F1 of 62.08% has been acquired by MPT-embedding feature. Regarding the other baselines, we have an F1 of 61.5, 60.8%, and 58.83% have been reached by DBN features, DP-CNN features, and Seml, respectively. All these outcomes demonstrated that automatically learned CPG- based semantic features involving data and the control flow; and the detailed learning algorithm DGCNN can improve the defect prediction F1 by almost 14%, 14.8%, 16.77%, 13.52%, and 10.0% as compared to DBN, the DP-CNN, Seml, the MPT-embedding, and node2defect, respectively.

We conducted extra experiments on GitHub projects to extensively evaluate our approach on different size of projects. These projects are used to evaluate the effect of Seml approach [213]. Table 10 shows the F1 values obtained by our approach and Seml with the GitHub projects. As it is shown in the Table 10, our approach outperforms significantly Seml on all projects in term of F1 score. On average, our approach achieves an F1 score of 77.68% which is 45.16% higher than the one of Seml approach. Finally, we carried out a third experiment in which we compare our approach with the baseline Node2defect on other Java projects. Table 11 details the experiment results obtained in our experiment. We can observe that our approach exceeds considerably the approach node2defect on every Java project in term of F1 score. Therefore, our approach reaches an F1 of 78.76% while the Node2defect has an F1 of 65.56%. So, it improves the Node2defect by 13.2%.

Table 9: a comparison among F1 scores of the developed CPG- based features and the baselines of traditional features (DBN and DP-CNN, Seml, MPT-embedding, and Node2vec) is set for the defect prediction within-project. The F1 is calculated in percent and the highest scores of F1 are presented in

Project	Versions (Tr→T)	CPG-DGCNN F1	DBN F1[10]	DP-CNN F1	Seml	MPT-embedding	Node2defect
ANT	1.5→1.6 1.6→1.7	78.2	**92.8**	N/A	N/A	59.7	61.5
CAMEL	1.2→1.4 1.4→1.6	**79.0**	57.8	50.8	48.95	48.35	53.5
JEDIT	3.2→4.0 4.0→4.1	**80.1**	59.4	58.0	59.95	63.76	59.6
LOG4J	1.0→1.1	**82.1**	70.1	N/A	68.5	74.6	N/A
LUCENE	2.0→2.2 2.2→2.4	60.1	71.2	**76.1**	68.05	76.54	**80**
XALAN	2.4→2.5	49.7	59.5	69.6	59.4	56.51	**76.7**
XERCES	1.2→1.3	**80.2**	41.1	37.4	35.4	31.51	N/A
IVY	1.4→2.0	**86.5**	35.0	N/A	N/A	N/A	40
SYNAPSE	1.0→1.1 1.1→1.2	**87.9**	56.3	55.6	51.5	55.46	67.2
POI	1.5→2.5 2.5→3.0	72.2	72.1	78.4	78.9	83.34	**86.1**
Average		**75.6**	61.5	60.8	58.83	62.08	65.57

Table 10: F1 scores obtained by our approach and the baseline Seml approach in GitHub projects

Project	Versions (Tr→T)	Our approach	Seml
BROADLEAFCOMMERCE	2013-09→2014-10	88.4	23.7
ELASTICSEARCH	2013-08→2014-02	58.8	38.3
HAZELCAST	2013-11→2014-11	79.6	36.5
NETTY	2013-02→2014-02	77.3	31.2
ORIENTDB	2013-06→2014-06	84.3	32.9
Average		77.68	32.5

Table 11: F1 scores obtained by our approach and the baseline Node2defect

Project	Versions (Tr→T)	Our approach	Node2defect
DrJava	2013-09→2014-10	81.9	66.5
Genoviz	2013-08→2014-02	73.5	68.2
Jmri	2013-11→2014-11	84.9	74.6
Jmol	2013-02→2014-02	75.7	60
JPPF	2013-06→2014-06	77.8	58.5
Average		78.76	65.56

Effort-aware evaluation

In this type of evaluation, a new experiment was conducted on the same dataset and 16 pairs of defect prediction were rerun within the project as presented in Table 12, by computing the metric PofB20. This leads to the analysis of the first 20% lines of code.

As mentioned earlier, the PofB20 indicates the identified number of defects by monitoring the first 20% lines of code standardized to the number of defects in the dataset of the project. In Table 12, the PofB20 values of the defect predicting models for file-level within-project are displayed with the CPG-based features. Our CPG-based features are compared with the featured based on DBN, or CNN. In Table 12, the improvement has been on an average of 18.8% points.

Our code property graph-based features automatically learned from DGCNN enhance the performance of file-level within project defect prediction models under both non-effort-aware and effort-aware evaluation.

Table 12: the PofB20 values of both CPG-based features and the features based on baseline DBN are displayed for file-level within-project defect prediction. The highest PofB20 scores are presented in bold.

Project	*Versions (Tr à T)*	*CPG-DGCNN F1*	*DBN F1[10]*
ANT	1.5 à 1.6	18.9	**44.3**
	1.6 à 1.7	**59.5**	50.2
CAMEL	1.2 à 1.4	**49.4**	33.2
	1.4 à 1.6	26.1	**30.1**
JEDIT	3.2 à 4.0	**75.5**	40.1
	4.0 à 4.1	**60.8**	32.6
LOG4J	1.0 à 1.1	**60.0**	25.0
LUCENE	2.0 à 2.2	**58.4**	32.1
	2.2 à 2.4	**75.1**	37.9
XALAN	2.4 à 2.5	14.8	**24.5**
XERCES	1.2 à 1.3	**17.3**	09.1
IVY	1.4 à 2.0	15.0	**28.3**
SYNAPSE	1.0 à 1.1	**67.3**	29.6
	1.1 à 1.2	**76.1**	32.5
POI	1.5 à 2.5	**80.9**	38.7
	2.5 à 3.0	**58.8**	25.5
Average		**50.9**	32.1

4.4.3.2 RQ2: Do code property graph-based features learned from DGCNN outperform traditional features for file-level cross-project defect prediction?

Non-effort-aware evaluation

This question is specifically answered to validate the efficiency of our approach for the file-level cross-project defect prediction. Our techniques were compared with two state-of-the-art software cross-DP models as F1 on the standard datasets i.e. the DBN-CP [32] and TCA+ [41]. DBN-CP develops the semantic features, whereas the TCA+ is a model based on metric, using the PROMISE features.

To conduct an un-biased comparison, a similar approach as that of Wang [32] was applied to conduct the experiment on CPDP. As described in section 4.4.2.6, to considerably examine the performance of our DGCNN-based features extracted from CPG, we select the training project from all projects for each target project. This means that for each version from one project, we use all the versions from other projects for training to form all test pairs in the cross project. This experiment involves 606 test pairs.

The results of the average F1 scores of our approach CPG-based features and the baseline DBN are displayed in Table 13. Our approach delivers better performance than DBN in almost all projects (9 out of 10), with an improvement of the F1 score of 13.8%. This outcome demonstrates that our proposed approach improves the baseline DBN significantly on file-level cross-project defect prediction.

Effort-aware evaluation

During this evaluation, we conduct the same experiment as Wang [32] to calculate PofB20 on CPDP for our proposed approach. Table 14 presents the scores of PofB20 for the cross-project DP. For every target project, we applied the other whole source project in the form of a training set and computed the PofB20, hence almost 606 runs were performed.

Table 13: F1 scores of our CPG-based features are compared with the baselines DBN-CP of file-level cross-project for all projects. Where the F1 is calculated in percent and the highest F1 scores are presented in bold

Source	Target	CPG-DGCNN F1	DBN F1[10]
All Others	ANT	**69 .77**	57.3
	CAMEL	**61.08**	46.1
	JEDIT	**79.04**	49.7
	LOG4J	**61.2**	56.2
	LUCENE	**48.7**	43.9
	XALAN	**59.65**	46.2
	XERCES	**76.55**	39 .7
	IVY	**70.41**	41.4
	SYNAPSE	**61.07**	50.2
	POI	45.29	**63.2**
Average		**63.28**	49.39

As presented in Table 14 the scores of PofB20 range from 29.3 to 57.8 % across the experiments.

During the comparison of cross-project CPG based features with the features based on DBN-CP; we concluded that our approach achieved a better PofB20 in every experiment. This improvement depicts an average of 16% points.

Table 14: presents PofB20 values for CPG based features and the features based on DBN-baseline for the cross-project DP. The maximum PofB20 score is indicated in bold.

Source	*Target*	*CPG-DGCNN F1*	*DBN F1[10]*
All Others	ANT	**57.8**	28.3
	CAMEL	29.3	**32.7**
	JEDIT	**55.2**	23.2
	LOG4J	**45.3**	28.6
	LUCENE	**55.1**	30.5
	XALAN	**44.8**	37.6
	XERCES	**41.7**	29.1
	IVY	**52.2**	26.5
	SYNAPSE	**36.0**	21.8
	POI	**38.0**	36.7
Average		**45.5**	29.5

Our code property graph-based features automatically learned from DGCNN enhance the performance of file-level cross-project defect prediction models under both non-effort-aware and effort-aware evaluation. Therefore, our approach is able to capture common characteristics of defects including syntax, semantics, and intra-procedural dependencies across projects.

4.4.3.3 RQ3: What is the improvement made by the code property graph?

To respond to this question, we performed an ablation study to explore the contribution introduced by CPG. Therefore, we conducted different experiments and used separately different combinations to train the neural network as follows: we removed the PDG, and we carried out experiment with only AST and CFG. Then, we also removed the CFG and performed experiment with only AST. For each experiment, we check the prediction results. Table 15 shows the effect of code property graph. Taking as example the project Synapse, when using only AST to represent the code, an F1 score of 74.6 is reached. By combining AST with CFG, an F1 score of 83.9 is achieved. Thus, more types of defects are covered. By further integrating the three code graphs, an F1 score of 87.9 is attained. So, more different types of defective patterns are detected.

The ablation study carried out confirms our idea of having a rich graph to better predict bugs. Indeed, the poorer the graph is in structural and semantic information, the poorer the quality of the prediction. Therefore, this study demonstrates the great impact of code property graphs to improve the prediction results and detect different types of defective patterns.

Table 15: F1 score of three different experiments: AST based features, AST+CFG based features and AST+CFG+PDG based features

Project	Versions (Tr→T)	AST F1	Ast+CFG F1	AST+CFG+PDG
ANT	1.5→1.6 1.6→1.7	**67.7**	**70.8**	**78.2**
CAMEL	1.2→1.4 1.4→1.6	**70.2**	73.7	79
JEDIT	3.2→4.0 4.0→4.1	**59.4**	66.5	80.1

LOG4J	1.0→1.1	**64.53**	58.7	82.1
LUCENE	2.0→2.2 2.2→2.4	**54.6**	55.1	60.1
XALAN	2.4→2.5	42.78	**46.84**	**49.7**
XERCES	1.2→1.3	**59.9**	67.2	80.2
IVY	1.4→2.0	60.9	**66.96**	**86.5**
SYNAPSE	1.0→1.1 1.1→1.2	**74.6**	83.9	87.9
POI	1.5→2.5 2.5→3.0	**68.7**	70.1	72.2
Average		**62.33**	65.98	75.6

4.4.3.4 RQ4: Do code property graph-based features learned from DGCNN outperform traditional features for change-level within-project defect prediction?

Non-effort-aware evaluation

To address this question and validate the effectiveness of our proposed structural and semantic features based on graphs, we need to compare it with the baseline methods. As explained in the experiment setup, we apply the same settings as Wang‟ paper experiment to collect multiple runs and make the training set more balanced [23].

Then, for each project, we use the training data from each run to train and construct the DGCNN based predictive model and evaluate it on the test data in this run. Finally, we indicate the overall performance by computing the average of these runs. As the code source of both baselines is not available, we take the values from their experiment results provided in their papers and we consider only Java datasets. Thus, we compare our approach with DBN and CBS+ on the available Java datasets (Jackrabbit, Lucene and JDT) and (JDT and Plateform) respectively; and pick the available values of DBN and CBS+.

Table 16 shows the F1 results of both. It can be observed that our CPG- based features outperform significantly all the baseline methods in each project, indicating that deep semantic and structural features learning based on DGCNN is valuable in defect prediction on change-level within-project. It can improve DBN based change features on average of 20.86 percentage points and CBS+ on average of 34.1 percentage points.

Table 16: F1 scores of our approach are compared with the baseline methods for change-level defect prediction where the F1 is calculated in percent and the highest F1 scores are presented in bold.

Project	Approach	F1 score
Jackrabbit	DBN	49.9
	CPG-based	**74.55**
Lucene	DBN	39.7
	CPG-based	**61.55**
JDT	CBS+	32.9
	DBN	41.4
	CPG-based	**57.48**
Platform	CBS+	35.1
	CPG-based	**78.72**
Average (Jackrabbit, Lucene, JDT)	DBN	43.66
	CPG-based	**64.52**
Average (JDT Platform)	CBS+	34
	CPG-based	**68.1**

Effort-aware evaluation

We further conducted a new experiment for change-level within-project defect prediction by computing the PofB20 metric. In Table 17, the PofB20 values of the defect prediction models related to code change are displayed with the CPG-based features as well as with the baseline DBN-based features. The PofB20 score varies from 33 to 49 percentage points. Compared to DBN, our approach achieves an improvement on average of 11.8 percentage points.

Our code property graph-based features automatically learned from DGCNN enhance the performance of change-level within project defect prediction models under both non-effort-aware and effort-aware evaluation.

Table 17: PofB20 values of our approach are compared with the baseline methods for change-level within-project defect prediction where the PofB20 are calculated in percent and the highest PofB20 scores are presented in bold**.**

Project	CPG-based features F1	DBN-based features F1
Lucene	**33.3**	28.1
Jackrabbit	**33**	27.9
JDT	**49**	23.8
Average	**38.4**	26.6

4.4.3.5 RQ5: Do code property graph-based features learned from DGCNN outperform traditional features for change-level cross-project defect prediction?

Non-Effort-aware evaluation

We answer this question to validate the efficiency of our approach for change-level cross-project defect prediction. We compare our technique with the baselines DBN-CPP [32] and CBS+ [214]. To conduct an un-biased comparison, a similar approach as that of Wang [32] was applied and which is also similar to the CBS+. Therefore, we select the data of the training set of one run from a source project and the test set of one run from a different project to prepare the trial pairs. For example, to build a prediction model to the target project Jackrabbit, we select the training set from the source projects Lucene and JDT.

During this evaluation, we compute the PofB20 metric on change-level cross-project defect prediction for our proposed approach as well as the DBN-CPP.

Table 18: F1 scores of our CPG-based features DBN-based features for change-level cross-project defect prediction. The F1 metrics are calculated in percent.

Source Project	Target Project	CPG-based features F1	DBN-based features F1
All projects	Jackrabbit	**72.69**	44.4
	Lucene	**63.84**	31.3
	JDT	**70.94**	33.3
Average		**69.15**	36.3

Table 19: F1 scores of our CPG-based features and traditional features CBS+ for change-level cross-project defect prediction. The F1 metrics are calculated in percent.

Source Project	Target Project	CPG-based features F1	CBS+-based features F1
All projects	JDT	**70.94**	30.8
	Platform	**73.05**	33.3
Average		**71.99**	32.05

Table 18 presents the average F1 scores of the CPG based features with those of DBN-CCP on three projects. The higher score of F1 among them is displayed in bold. The results show that our approach significantly improves the average of F1 by 32.85 percentage points for three projects. Moreover, we provide comparison results of CPG-based features and CBS+ for change-level cross-validation in Table 19. Compared to CBS+ on two projects, our approach achieves a better F1 score on average of 39.94 percentage points.

Effort-aware evaluation

Table 20 presents the scores of PofB20 for the change-level cross-project DP. For every target project, we applied the other whole source project as a training set and computed the PofB20. As presented in Table 20, the scores of PofB20 range from 39.6 to 43.7 % across the experiments. During the comparison of cross-project CPG based features with the features based on DBN-CP; we concluded that our approach achieved a better PofB20 in every experiment. This improvement depicts an average of 20.7 points.

Our code property graph-based features automatically learned from DGCNN enhance the performance of cross-project defect prediction models under both non-effort-aware and effort-aware evaluation. Therefore, our approach is able to capture common characteristics of defects including syntax, semantics, and intra-procedural dependencies within code changes across projects.

Table 20: PofB20 scores our CPG-based features for change-level cross-project defect prediction. The PofB20 metrics are calculated in percent. The best values are in bold.

Source Project	Target Project	CPG-based features	DBN-based features
,All projects	Jackrabbit	**42.0**	19.3
	Lucene	**39.6**	18.1
	JDT	**43.7**	25.6
Average		**41.7**	21

4.4.3.6 Time cost of the deep learning approach based on code property graph

This question leads to the study of the efficiency of our approach which is an important indicator to assess whether the approach is good enough.

We measure therefore the time taken for file-level DGCNN-based features generation process described in the sections 4.3.2.4 and 4.3.2.5. Moreover, we keep track of the time cost for tuning the DGCNN parameters in our experiments. The other operations, involving parsing source code, merging into code property graphs, mapping token graphs and predicting defects, are all common operations, so we do not examine their costs.

As mentioned in section 4.4.2.5, we tune the three parameters (the number of nodes in each layer, the number of hidden layers, and the number of iterations) for training the DGCNN. To identify the best combination among the three parameters, we performed 6*7*10 experiments. Thus, the time cost of the tuning process is about 33 hours.

Table 21 presents our method‟ time cost on the ten datasets for generating features process. By considering the example of the two experiments performed on the two sets of the project Lucene which is lucene 2.0 →2.2 and lucene 2.2→ 2.4, the calculated average execution time is of value 120 seconds.

For every project, the execution time automatically developed features based on DGCNN lies in the range of 26 sec (jedit) to the 417 sec (xalan).

Table 21: Time cost of generating features involving the semantics and the intra-procedural dependencies of the source code

Project	Generating features process Time (s)
LUCENE	120
LOG4J	45
IVY	132
JEDIT	26
CAMEL	242
SYNAPSE	78,5
ANT	116
XERCES	257
XALAN	417
POI	172
Average	160.5

Moreover, we monitor the time cost for generating features for change-level defect prediction. Contrary to file-level defect prediction, changes always have fewer lines than source files. Thus, the time cost for changes is smaller than those for files. The average time of our experiments performed for change-level is about 26.6 seconds.

Our CPG based features learned automatically from the DGCNN is applicable in practice

4.4.4 Threats to validity

4.4.4.1 Internal validity

Threats to internal validity involve potential errors that may have occurred in the code implementation of our proposed approach and study settings. Hence, to develop the semantic feature with the dependency information, we must present the source code within the data structure known as CPG involving the AST, PDG, and CFG. As the original implementation of CPG is not released, so we have implemented a new CPG version. Though we have followed the methods given in previous studies [22], yet the newly developed CPG version may not reflect each detail of the actual CPG.

Therefore, we have consulted with the writer of PROGEX[3] by email; about the basic details of implementation and this was the beginning of our framework implementation. We are

[3] https://github.com/ghaffarian/progex

confident that the CPG implementation is quite close to the original CPG, because the PROGEX includes the basic features which were useful for us to implement the merge of graphs. Moreover we don't possess the basic source code to copy the technique of Wang et al. [12], Jian et al.[26], and Huang [115] therefore we have allowed ourselves to consider the results they gave in their papers. For change-level defect prediction, we have followed the same experiment settings just as it is applied in [32] in carrying out a comparison with our approach and we have realized a supplementary comparison by retrieving the results of [115].

Furthermore, we have relied on the results of the SZZ algorithm for labelling data. SZZ is the most widely used algorithm and available in literature. It is known that this technique may introduce intrinsic imprecisions [218]. Thus, the mislabelled data may affect the accuracy of our results. To mitigate this threat, we use Google java format[4] to ensure that any source code differences considered are based on unified format rules.

4.4.4.2 External validity

The external validity indicates the normalization of our research outcomes. In this study, we conducted our experiments only on java open source projects among 38 projects, initially collected by Jureczko and Madeyski [33] witch are very used by almost papers that deal with file-level SDP studies [12], [26], [133]. This can influence the generalizability of our results as these datasets do not represent all the software projects. So, our approach could produce better or weaker performance for other projects that are not applied in the experiments. To mitigate this threat, we select projects that vary considerably in their domains, complexity, popularity, sized and average defects rate. Yet, as the performance of our approach is considered as un-known in projects composed in any other language, further studies are required to make our proposed approach more common in the future; by performing more experiments on a variety of projects whether propriety software or commercial one written in other languages for example PHP, C++, and Python.

In the context of cross-project analysis, the pairwise do not mimic any strategy to select the training dataset from the available projects (i.e., select only the projects which have same characteristics as the target project whose data are insufficient). We mitigate this threat by

[4] https://github.com/google/google-java-format

forming all the possible test pairs from the available projects in the experiments for cross project to extensively analyze the performance of our approach.

4.4.4.3 Construct validity

In terms of the construct validity regarding suitability of evaluation criteria for performance, a standard measure of performance is used for predicting the defects, commonly used in previous studies [219]–[221], including the PD, precision, PF, precision, Balance, F1 measure, MCC, G-measure, AUC, G-mean1, G-mean2 and G-measure. We used only three performance measures in our experiments that are precision, recall, and the F1 measure. All of these measures cannot be used and in-fact these measures have not been used in any studies to analyze the SDP classifier. So, it may lead to any threats for constructing the validity. Furthermore, we admit that the statistical significance for our results can be verified by using several statistical analyses [222] and we have planned this for the future.

4.5 Conclusion

This chapter proposes an end-to-end deep learning algorithm to learn meaningful features involving syntactic and semantic information as well as intra-procedural dependencies. Typically, we employ DGCNN to automatically learn the features from token graphs extracted from the program files or code changes to construct a predictive classifier of a high quality. The key insight underlying the representation of the code by the code property graph is to provide a suitable and robust representation exploring deeply the program files/code changes and express patterns linked to different types of bugs. Then, the designing features are fed to the DGCNN to build the defect prediction model.

We conduct evaluations on ten open-source projects for program files from the dataset Promise; and four open-source projects for codes changes. Both of them is performed under two different scenarios: non-effort-aware and effort-aware evaluation scenarios. The experiment results of program files proved that our approach significantly improves the existing works on average of 14.04 in F1 in the task of within-project defect prediction. Besides, it improves the cross-defect prediction techniques TCA+ on average of 10.68 in F1. Also, our approach can outperform traditional features un-der the effort-aware evaluation context.

Concerning the experiment results related to the code changes, the experiment values proved that our approach significantly improves the existing work DBN-based features and CBS+ on average of 20.86 and 34.1 in F1, respectively in the task of within-project defect prediction. Besides, it improves the cross-defect prediction technique DBN-CPP and CBS+ on average of 32.85 and 39.95 respectively in F1. Also, our approach can outperform it under the effort-aware evaluation context.

In the future, we would like to extend our work and generate expressive features that include the semantics and the dependencies among program entities within multiple methods at other levels, such as module level and package level. In addition, it would also be interesting to generalize the performance of our framework proposed in this manuscript to open-source projects written in different languages, such as Python, C/C++, etc., and confirm its efficiency by performing statistical tests such as (Wilcoxon signed-rank test and Cliff's Delat Effect Size analysis). To be able to apply the statistics, we plan to implement the baselines considered in our experiments as their source codes are not available. Finally, we would like to address the CPDP challenges by proposing a strategy to select relevant training projects that have similar characteristics as the target project (this is the subject of the next chapter).

Fifth CHAPTER

5 A source project selection framework for cross-project defect prediction

This chapter presents our proposed three-phase methodology to select relevant source projects that have same characteristics as the target project to conduct the cross-project prediction task and improve its performance. In the first phase, we computed high-level similarity by performing a pair-wise qualification matching of the project 'model. The qualification model of the project characterized the project in terms of organizational, reliability, etc. In the second phase, we computed the low-level similarity between projects by matching graphs representing the structural and semantic information of both source and target project. Finally, in the third phase, we performed our selection by considering both high- and low-level similarities. In the rest of this chapter, we present the motivation for the building of the framework to automatically choose source projects in section 1. We describe the implementation details of our framework in section 2. The experiment settings are presented in section 3. The experimental results are discussed in section 4 followed by the outline of the threads for validity in section 5. Finally, we conclude and propose perspectives to our work in section 6.

5.1 Motivation

Recently, several studies have proposed prediction methods to construct the prediction model based on a given training set. Later, they applied the prediction model to predict on a given

test set. To achieve a prediction of high quality, most of the research work requires meeting two assumptions: 1) the training set and the test set should be from the same project as they have similar data distribution; 2) there is enough historical data to be able to learn different defective patterns and construct an effective prediction model. Therefore, the prediction cannot work well on a new project or a project that has insufficient data. To resolve this issue, a promising solution, called cross-project defect-prediction is proposed in the literature. This method allows training the prediction model by using a training set from other projects (i.e., source projects) with enough historical data, and apply this model to predict the project lacking data (i.e., target project). The main challenges of CPDP reside primarily in the difference of data distribution between the source and target projects [16], [41]. For example, a new project may involve several bugs in large-sized modules, while a stable project may have more bugs in small-sized modules following modifications made to update functions.

From the state-of-the-art analysis, we derived two main conclusions in the field of cross-project: 1) the metrics fail to extract meaningful properties linked to defective patterns from the code, and 2) selecting carefully valuable source projects which have similar data distribution as the target project, instead of considering all the projects, could improve considerably the prediction quality of cross-project.

Moreover, as reported in the previous chapter, deep learning algorithms based on structural and semantic features have identified successfully defected files and changes. The experiment results have shown that our proposed approach has improved the F1 score measurement compared to other traditional methods under both within-project defect prediction and cross-project defect prediction setting.

Based on the above, we propose a novel framework of project selection by computing the similarities between project instances and extracting knowledge from the source projects. The similarity identification is established on the mismatch between the low-level details reflected in the structural and semantic information as along with the dependencies of the code and the high-level purpose reflected in the qualification and description of these source projects. This setting can help us to automatically detect closely related source projects for a given project.

The key idea behind this framework can be briefly summarized as follows:

- To the best of our knowledge, we are the first to propose a novel approach to select suitable application via structural and semantic information hidden in the code and

global knowledge of the applications. This approach performs a graph-matching representing structural and semantic aspect of the code to keep relevant instances in the source projects. These selected instances can effectively predict the target project and identify diverse types of bugs as they involve similar defective patterns as the target project.

- To confirm the effectiveness of our framework in selecting similar source projects, we performed an experiment on 10 large-scale Java project from Promise and evaluated our approach by comparing it with previously succeeded CPDP baselines and our approach without making any selection of source projects beforehand.

5.2 The proposed approach

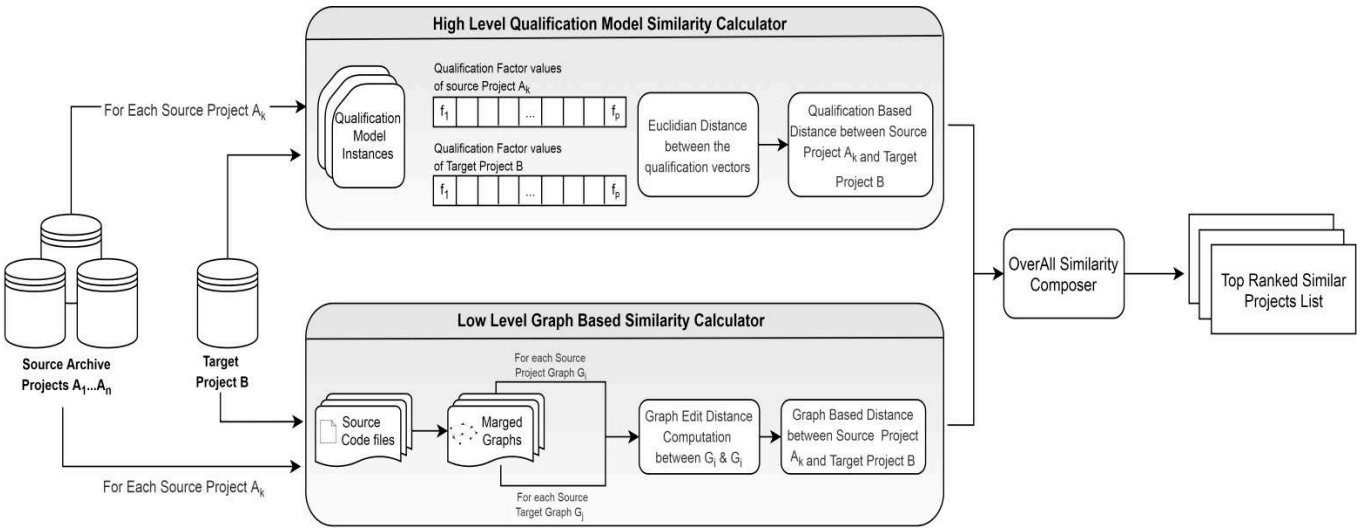

Figure 23: Framework of source projects selection for cross-project defect prediction

In this section, we introduce the overall architecture of our proposed framework. Then, we describe the details of its components.

5.2.1 Overall architecture

Figure 23 illustrates the overall process of the selection source project which operates in three steps: 1) computing the high-level similarity, 2) computing the low-level similarity, and 3) selecting the three best candidates.

Phase-I: Computing high-level similarity: Initially, we have a collection of candidate source project as input of our approach. First, we use a qualification model based on metrics to assess all the candidate source software systems and evaluate their states related to different dimensions. For the target project, we use the same qualification model to characterize it. Then, to compute the similarity between a pair of source and target projects, we use the Euclidian distance. The details are showed in Section 5.2.1.1.

Phase-II: Computing low-level similarity: In this phase, we compute the similarity between a pair of source and target project using source code. In the source project, each class is represented by the code property graph to consider simultaneously structural and semantic information as well as the dependencies in the code. In the target project, we extract the code property graph of each class as in the source projects. Then, we perform a pair-wise comparison between source project's graphs and target project's graphs by applying the graph edit distance.

Phase-III: Selecting the best three source projects: The global similarity between a pair of each source and target project is obtained by a function defined as the weighted-sum of both high-level similarity and low-level similarity. We assign an important weight to the low-level similarity compared to the high-level similarity.

5.2.1.1 Computing high-level similarity

As mentioned before, the aim of this phase is to extract knowledge from source projects. We therefore used a qualification model based on the identification of the business, technical, data and organizational quality profiles. The high-level aspects/dimensions are described as follows:

- Business: represents the system value from the points of view of its direct and indirect users; where the direct and indirect users can be managers, end users, customers and so on.
- Technical: evaluates the quality of the information systems architectures of the organization, the structure, performance, and interactions of the applications.
- Data: describes the structure and interaction of the organization's sources of data, logical data assets and data management resources.
- Technology: depicts the technology architecture layer of the organization. It represents the structure and interaction of the platform services and physical technology components.

The identification of the characteristics and the related quality metrics describing the aspects are adopted from the software engineering process and the ISO 9126 standard [223]–[225]. Table 22 lists the characteristics and the related quality attributes used to assess them in each aspect.

Table 22: The qualification model

ASPECT	FACTOR	ATTRIBUTE
BUSINESS	Economic	Time to market
		Profitability
		ROI
		Maintenance Cost
	Specialisation	Pourcetage of high specialized functions
		Pourcetage of generic functions
	Understanding	Pourcetage of business logic understanding
		Pourcetage of business logic prioritizing
	Usage	Pourcetage of Business Function Coverage rate
		Usage frequency
		User satisfaction
	Organizational	Technical maturity
		Skill levels
		Response to change
		Training procedures
		Transparency

APPLICATION	Flexibility	Modularity
		Simplicity
		Self-documentation
		Lack of technical documentation
	Application Performance	Average responsiveness
		Average availability
		Batch SLAs met
	Reliability	Software reliability
	Maintenability	Correcteness
		Testability
		Modularity
		Expandability
	Application Obsolescence	SW obsolescence
	Application Interoperability	Layout appropriatness
		Degree of standadization
DATA	Data Integrity	Data Consistency
		Data Security
		Auditability
	Data Usage	Data Accessibility
		Data Availability
		Data Quality
		Data dependance
	Data Interoperability	Data Commonality
	Data Obsolescence	DB Obsolescence
TECHNOLOGY	Operational Performance	Storage performance
	Obsolescence	OS obsolescence
		Hardware obsolescence
	Portability	Software System Independence
		Machine Independence

The metrics values are used for evaluating the attributes. Let M_i be a generic metric, X_i is the value assumed by M_i and X_1, X_2,...., X_n are the values of the metrics M_1, M_2,...., M_n related to the attribute A_t. All the values are in interval [0.2, 0.8]. In this work, the metric values are affected arbitrarily, instead of being identified by an expert. A_t is computed as the average of the corresponding metrics. The same formulas are used for aggregating the values of the attributes and evaluating the values of factors or characteristics.

Each source project as well as the target project is represented by a vector containing the values of the factors. To compute the similarity between the two qualifications models corresponding to a pair of source project and target project (A_i ; B), we used the Euclidian distance of their related vectors. A_i represents the source project i; with i ∈ [1...k], and k is the number of source projects; and B is regarded as the target project.

5.2.1.2 Computing low-level similarity

The implementation details of this sub-section can be found in the following algorithm.

Algorithm 1 Pseucode for the method of the computation of low-level similarity

Input:
The number of source projetcs K
The number of instances in th target project B S'
The minimum number of instances in all source projects A_i M
Output:
Generate code property graphs of each instance in project B G'
Generate code property graphs of each instance in each source project G
For i from 1 to K **do**
For j from 1 to M **do**
For l from 1 to S' **do**
Graph-edit-distance (G_j , G'_l)
end for
end for
average-edit-distance (A_i , B)
end for

Initially, we have the number of candidate source projects K, the number of instances or files S' in the target project B, and the minimum number of files M in all candidate source projects. We choose the minimum number of files in all source projects to have a same number of files in all projects and consequently a fair comparison between each source project with the target project.

For each instance (i.e., file) of each source project, we generated the code property graph G by merging the three representations AST, CFG and PDG as explained in the previous chapter. We do the same for the target project and extract the CPG G' for each instance.

Then, to compute the similarity between each source project with the target project, we compared each source graph G in M with each graph G' in S' by using the common graph similarity called Graph Edit distance. Therefore, the similarity between each source project A_i in K and target project B is obtained by averaging the edit distance results.

Applying a mapping between CPG of each source project with the target project enabled to compute the similarity by considering the structural and the semantics of the code. More details of the graph edit distance are available in the following sub-section.

Graph Edit Distance. It is the base of inexact graph matching and is widely used by researchers to pattern recognition and analysis. It is a way of measuring the similarity between pairwise graphs by transforming a graph into another one by a finite sequence of graph edit operations. The edit operations may include edge insertion, node substitution, node deletion, etc. A cost is given to each edit operations. The cost of edit operation sequence is the sum of costs given to all operations in the sequence. The number of changes and its cost required for transforming a graph into another graph is not unique, but the minimum cost is retained. Thus, the graph edit distance between these two graphs corresponds to the least cost.

5.2.1.3 Selecting the three best source projects

In this phase, we measured the global similarity between each candidate source project and target project and select the best three candidates. For this, we assigned weights to both high and low similarities, but the low-level similarity value is given the greater weight. In this work, we gave 0.2 to the high-level similarity value and 0.8 to the low-level similarity value.

5.3 Experiment setting

In this section, we evaluated the performance of our candidate source project selection framework and compared it with the stat-of-the-art cross-project defect prediction baselines as along with our proposed file-level cross-defect prediction framework, but without any strategy for selecting the best candidate source projects.

Initially, we present the standard datasets and then the experiment setup. After this the baseline approaches are described and the evaluation criteria for used performance are explained.

.

5.3.1 Dataset

Similarly to the file-level defect prediction, we evaluated our framework of source project selection using the defect datasets Promise repository collected by Jureczko an Madeyski [33], as listed in Table 5 in the sub-section 4.4.2. We have, therefore 10 open-source Java projects. Each project contains releases and in total we have 32 distinct releases. Each instance in respective project release corresponds to a Java class or Java file and one instance is represented as a graph, specifically, as a code property graph. Each file is linked to a label, i.e., clean, or defective.

5.3.2 Experiment setup

We performed the cross-project practical usage as described in previous studies [13], [100], [190], [226]. Specifically, for one cross-defect prediction execution, we considered one version from a project as a target project, and all versions of all other projects as the candidate source projects, except the release chosen for the target project.

5.3.3 Baselines

In this study, we evaluated our proposed cross-project framework including a source project selection strategy by comparing it with some succeeded CPDP approaches such as TCA+, TPTL, TDS, and our cross- project framework without any source project selection strategy.

TDS. [189] performed distance-based strategies to select the best suitable training data based on distributional characteristics of the target data. They first use the EM algorithm to create meaningful clusters whose data characteristics are close to the target's data. Then, they apply the Euclidean distance for determining the most similar candidates. TDS is considered as the most related work for source project selection.

TCA+. This baseline is proposed by Nam et al. [41]. The core objective of this method is to reduce the data distribution between the target project and the available candidate source projects. Firstly, they proposed the basic TCA which normalize the both data of source

projects and target project by selecting one method among normalization methods such as min-max normalization or Z-score normalization, and then it learns a nonlinear function to apply it on the normalization data to map the source project and the target project into a latent space.

TPTL. This baseline is proposed by Liu et al. [190]. The method two-phase transfer learning model selects two source projects whose data distribution are very similar to the target. The chosen projects are considered as he best projects as it is estimated that they have the highest performance in terms of F1 score and PofB20 indicators. After that, they leverage TCA + to construct two prediction models based on the two chosen projects and further improve the prediction performance by combining the two models.

DBN-based CPDP. This method is proposed by Wang [32]. Different from standard CPDP baselines which rely on metrics to capture meaningful defective patterns from projects, this approach is based on semantic information provided by AST. However, they do not mimic any practical use case, without a strategy to select considerable projects from all projects; and perform the cross-project by taking all the combinations possible compound of the target project and all the available source projects.

5.3.4 Evaluation criteria

We adopted the same evaluation metrics used in the previous chapter that are the F1-score measure and the cost-effectiveness measure PofB20. Moreover, as explained in the previous chapter, these metrics are widely applied by the literature. We refer to the sub-section 4.4.2.4 for more details.

5.4 Result analysis

In this section, we mainly investigate whether our proposed project selection framework is effective or not. We check the performance of our framework by comparing it with our file-level cross defect prediction without any strategy for selecting the source projects. Also, we evaluate how much improvement can achieve over the baselines. As we have mentioned that

the target project (a version of a project) is used as test set while the versions of the selected projects are used for training the prediction model to the target project.

Table 23 and Table 24 show the F1-score and the PofB20 of our CPDP with project selection versus our CPDP without any project selection strategy and 4 baselines.

Table 23 indicates that our CPDP with project selection achieves a significant improvement over our CPDP without any project selection strategy and the other baseline models. As it is shown in the table, our method always obtains the best value of F1 score which range from 63.66 to 85.94 through 42 datasets. Our approach outperforms our CDPD without project selection, DBN-based CPDP, TPTL, TCA+, and TDS by 11.08%, 24.97%, 26.94%, 27.63%, 34.72%, respectively. We can notice that the structural and semantic aspects are important to take them into account in selecting useful defective patterns from either same project or external projects. Also, carefully selecting projects can ameliorate the results even the approaches are based on metrics. This can be clearly observed from the results since our method shows large improvement in comparison with the two baselines TCA+ and TDS which are based on metrics and slightly less with the method TPTL which relies on metrics but includes a strategy for selecting source projects.

Table 23: F1-score comparison of our CPDP with project selection versus our CPDP without project selection and 4 baselines (DBN-based approach, TPTL, TCA+, and TDS)

Target		Selected projects	DGCNN With project selection	DGCNN without Selection	DBN-based approach	TPTL	TCA+	TDS
ANT	ANT_1,5	IVY_1.4; XALAN_2.4; SYNAPS_1.2 Xerces_1.2; LUCENE_2.2	81.60	69,77	57,30	42,40	42.5	38,08
	ANT_1,6	CAMEL_1.6; XALAN_2.4; JEDIT_3.2 JEDIT_4.1; SYNAPSE_1.0						
	ANT_1,7	SYNAPSE_1.0; JEDIT_3.2; SYNAPSE_1.1; SYNAPSE_1.2; LUCENE_2.2						
SYNAPSE	SYNAPSE_1,0	LUCENE_2.4; IVY_1.4; LUCENE_2.0 LOG4J_1.1;	72.03	61,07	50,20	43,30	44,83	50,76

		LOG4J_1.0						
	SYNAPSE_1,1	LUCENE_2.0; CAMEL_1.4; POI_1.5; LOG4J_1.1; LUCENE_2.2						
	SYNAPSE_1,2	LUCENE_2.2; POI_2.5; JEDIT_3.2; LOG4J_1.1; IVY_1.4						
CAMEL	CAMEL_1,2	ANT_1.7; SYNAPSE_1.0; IVY_1.4 XALAN_2.4; LOG4J_1.0	70.85	61,08	46,10	24,60	31,47	27,06
	CAMEL_1,4	ANT_1.5; XALAN_2.4; POI_2.5; ANT_1.7; IVY_1.4						
	CAMEL_1,6	ANT_1.6; LOG4J_1.0; JEDIT_4.1; ANT_1.7; IVY_1.4						
IVY	IVY_1,4	ANT_1.7; JEDIT_3.2; XALAN_2.4; SYNAPSE_1.2; ANT_1.5	77.93	70,41	41,40	41,56	43,00	31,56
	IVY_2,0	LUCENE_2.2; LUCENE_2.4; SYNAPSE_1.1; JEDIT_3.2; JEDIT_4.1						
LOG4J	LOG4J_1,0	ANT_1.6; LUCENE_2.2; ANT_1.7; CAMEL_1.6; SYNAPSE_1.2	69.88	61,20	56,20	64,73	57,43	46,00
	LOG4J_1,1	POI_1.5; CAMEL_1.4; SYNAPSE_1.0; SYNAPSE_1.2; CAMEL_1.6						
JEDIT	JEDIT_3,2	ANT_1.7; CAMEL_1.4; ANT_1.5; XALAN_2.4; ANT_1.6	85,94	79,04	49,70	38,50	39,20	23,38
	JEDIT_4,0	ANT_1.5; ANT_1.6; XALAN_2.4 ANT_1.7; CAMEL_1.4						
	JEDIT_4,1	CAMEL_1.6; CAMEL_1.2; ANT_1.6; XALAN_2.4; ANT_1.5						
LUCENE	LUCENE_2,0	SYNAPSE_1.1; LOG4J_1.0; IVY_1.4; SYNAPSE_1.2; SYNAPSE_1.0	63,66	48,70	43,90	64,36	59,70	60,76

	LUCENE_2,4	ANT_1.7; CAMEL_1.2; LOG4J_1.1 LOG4J_1.0; POI_1.5						
POI	POI_1,5	SYNAPSE_1.1; ANT_1.7; CAMEL_1.4; LUCENE_2.0; LUCENE_2.4	65,15	45,29	63,20	61,15	55,90	51,75
	POI_2,5	JEDIT_3.2; SYNAPSE_1.1; SYNAPSE_1.0 IVY_1.4; LOG4J_1.0						
	POI_3,0	CAMEL_1.4; LUCENE_2.2; SYNAPSE_1.0 LOG4J_1.1; LUCENE_2.0						
XALAN	XALAN_2,4	CAMEL_1.4; ANT_1.5; ANT_1.7 CAMEL_1.6; ANT_1.6 POI_3.0; POI_1.5; SYNAPSE_1.0; LUCENE_2.0; LOG4J_1.1	72,05	59,65	46,20	51,60	49,47	44,62
	XALAN_2,5							
XERCES	XERCES_1,2	LUCENE_2.4; JEDIT_4.1; ANT_1.5; CAMEL_1.6 ANT_1.5; ANT_1.6; JEDIT_4.1; CAMEL_1.6; XALAN_2.4	84,52	76,55	39,70	41,96	39,60	22,43
	XERCES_1,3							
Average			74,36	63,28	49,39	47,42	46,73	39,64

Our CPDP with project selection always shows the best cost-effectiveness in terms of PofB20 in Table 24. The Table 24 shows the PofB20 scores which vary from 39.12 to 65.14 with an average of 54.16. The DGCNN based approach with project selection outperforms the DGCNN approach without project selection and the baselines DBN-based approach, TPTL, TCA+ by 8.62, 24.66, 34.5, and 34.96, respectively.

Table 24: PofB20 comparison of our CPDP with project selection versus our CPDP without project selection and 4 baselines (DBN-based approach, TPTL, TCA+, and TDS)

Target		Selected projects	DGCNN with project Selection	DGCNN Without selection	DBN-based approach	TPTL	TCA+	TDS
ANT	ANT_1,5	IVY_1.4; XALAN_2.4; SYNAPS_1.2 Xerces_1.2; LUCENE_2.2	65,14	57,83	28.3	24.32	28.1	13.58
	ANT_1,6	CAMEL_1.6; XALAN_2.4; JEDIT_3.2 JEDIT_4.1; SYNAPSE_1.0 SYNAPSE_1.0; JEDIT_3.2; SYNAPSE_1.1; SYNAPSE_1.2;						
	ANT_1,7	LUCENE_2.2						
SYNAPSE	SYNAPSE_1,0	LUCENE_2.4; IVY_1.4; LUCENE_2.0 LOG4J_1.1; LOG4J_1.0	44,23	36,01	21.8	23.83	19.2	22.73
	SYNAPSE_1,1	LUCENE_2.0; CAMEL_1.4; POI_1.5; LOG4J_1.1; LUCENE_2.2						
	SYNAPSE_1,2	LUCENE_2.2; POI_2.5; JEDIT_3.2; LOG4J_1.1; IVY_1.4						
CAMEL	CAMEL_1,2	ANT_1.7; SYNAPSE_1.0; IVY_1.4 XALAN_2.4; LOG4J_1.0	39,12	29,34	32.7	21.77	14.8	9.8
	CAMEL_1,4	ANT_1.5; XALAN_2.4; POI_2.5; ANT_1.7; IVY_1.4						
	CAMEL_1,6	ANT_1.6; LOG4J_1.0; JEDIT_4.1; ANT_1.7; IVY_1.4						
IVY	IVY_1,4	ANT_1.7; JEDIT_3.2; XALAN_2.4; SYNAPSE_1.2; ANT_1.5	58,30	52,19	26.5	14.76	20.1	11.5
	IVY_2,0	LUCENE_2.2; LUCENE_2.4;						

		SYNAPSE_1.1; JEDIT_3.2; JEDIT_4.1						
LOG4J	LOG4J_1,0	ANT_1.6; LUCENE_2.2; ANT_1.7; CAMEL_1.6; SYNAPSE_1.2	53,86	45,27	28.6	22.66	19.1	15.7
	LOG4J_1,1	POI_1.5; CAMEL_1.4; SYNAPSE_1.0; SYNAPSE_1.2; CAMEL_1.6						
JEDIT	JEDIT_3,2	ANT_1.7; CAMEL_1.4; ANT_1.5; XALAN_2.4; ANT_1.6	62,47	55,21	23.2	23.88	21.8	7.64
	JEDIT_4,0	ANT_1.5; ANT_1.6; XALAN_2.4 ANT_1.7; CAMEL_1.4						
	JEDIT_4,1	CAMEL_1.6; CAMEL_1.2; ANT_1.6; XALAN_2.4; ANT_1.5						
LUCENE	LUCENE_2,0	SYNAPSE_1.1; LOG4J_1.0; IVY_1.4; SYNAPSE_1.2; SYNAPSE_1.0	67,62	55,13	30.5	20.96	15.6	6.26
	LUCENE_2,4	ANT_1.7; CAMEL_1.2; LOG4J_1.1 LOG4J_1.0; POI_1.5						
POI	POI_1,5	SYNAPSE_1.1; ANT_1.7; CAMEL_1.4; LUCENE_2.0; LUCENE_2.4	48,62	37,99	36.7	13.55	14.92	6.12
	POI_2,5	JEDIT_3.2; SYNAPSE_1.1; SYNAPSE_1.0 IVY_1.4; LOG4J_1.0						
	POI_3,0	CAMEL_1.4; LUCENE_2.2; SYNAPSE_1.0 LOG4J_1.1; LUCENE_2.0						
XALAN	XALAN_2,4	CAMEL_1.4; ANT_1.5; ANT_1.7 CAMEL_1.6; ANT_1.6	54,88	44,79	37.6	19.1	15.5	13.45

	XALAN_2,5	POI_3.0; POI_1.5; SYNAPSE_1.0; LUCENE_2.0; LOG4J_1.1						
XERCES	XERCES_1,2	LUCENE_2.4; JEDIT_4.1; ANT_1.5; CAMEL_1.6	47,36	41,67	29.1	15.1	22.5	5.93
	XERCES_1,3	ANT_1.5; ANT_1.6; JEDIT_4.1; CAMEL_1.6; XALAN_2.4						
Average			54,16	45,54	29.5	19.99	19.2	

5.5 Threats to validity

5.5.1 Internal validity

Threats to internal validity refer to potential errors that may have occurred in the code implementation and the replication of code property graph algorithm and study settings. Hence, to select the best candidate source projects involving same structural and semantic defective patterns as the target project, we applied a graph matching based on the CPG representation to compute the low-level similarity between the concerned projects. As explained in the sub-section 4.4.4.1, the original implementation of CPG is not released, so we re-implemented a CPG version by ourselves by following the methods given in previous studies [22]. Nonetheless, the newly developed CPG version can have errors that we are unknown of.

However, we have double checked with the writer of PROGEX[5] by email that our CPG implementation involves the basic concepts. Moreover, we have taken into consideration the results of the baseline models from their related papers [32], [41], [189], [190]. Besides, to qualify the projects and compute the high-level similarity, we needed experts from related projects to assign values to the metrics. But, since we do not have these experts, we identified an interval ranging from 0.2 to 0.8 to fulfill the metric values arbitrarily. These could bias the results.

[5] https://github.com/ghaffarian/progex

5.5.2 External validity

The external validity is related to the generalization of our research outcomes. In this research, we conducted our experiments only on Java open source projects from Promise repository collected by Jureczko and Madeyski [33], which were extensively used in previous CPDP studies. However, our approach performance could vary if we used different datasets from different repositories or closed projects. To mitigate this threat, we planned to examine our selection of project framework on varied datasets written on different languages besides Java language in future studies. Moreover, the graph matching algorithm can be the threat to validity. We selected graph edit distance as the graph matching algorithm to compute the low-level similarity between a pair of source project and target project. However, the choice of graph matching algorithm can affect the performance of our proposed selection project approach. Therefore, our approach should be investigated by other graph matching algorithms. Another threat is the choice of baselines. We selected these baselines due to their superior performance compared to other CPDP approaches in latest studies or their broad usage as baselines in previous CPDP research. Thus, selecting these baselines can reflect the state-of-the-art of existing CPDP research.

5.5.3 Construct validity

In this study, we only considered the two commonly applied evaluation metrics, F1 score and PofB20, to analyze the performance of our approach.

In future studies, we plan to use other performance indicators such as AUC, balance, etc. to evaluate our CPDP framework including project selection. Furthermore, other statistical performance indicators should be explored in all our experiments to better rank our approach as well as baselines.

Sixth CHAPTER

6 Conclusion and perspectives

6.1 Conclusion

In the field of software defect prediction, the researchers aim to construct a prediction model of high quality. As a result, they attempt to provide advance knowledge of code and apply the best learning models based on the data extracted from the code. Consequently, many contributions have been made to ameliorate the code representation. Accordingly, deep learning models are becoming increasingly popular in improving current software defect prediction practices. In this dissertation, our research work ensures the continuity of these initiatives.

In chapter 3, we saw that a considerable number of defect prediction studies draw on either handcrafted traditional metrics or either tree-based representation or graph-based representation to characterize the software program and extract useful features from it. However, all these traditional representations often fail to capture the intra-procedural dependencies into a program, and such a capability is required for constructing a more powerful classifier. Indeed, the accuracy of approaches is widely influenced by the quality of the input data no matter which data the deep learning model used. Contrary to the classic machine learning algorithms, one of the main advantages of deep learning model, specifically of the graph convolutional network is the lack of requirement for any handcrafted features. This algorithm can also perform the learning by automatically exploring graph-based features with complex structures.

To bridge the gap between program' dependencies and defect prediction features, we proposed in chapter 4 a framework that leverages graph based deep learning techniques to learn simultaneously semantic and syntactic representation including dependency information

automatically from source code, and further construct and train defect prediction classifiers based on these complex features. This framework is proposed for two levels, file-level, and change-level, under two settings, within-project, and cross-project. We evaluated the effectiveness of the graph based deep learning defect prediction approaches on open-source projects. Our experiment results confirm that the learned complex features including semantic, syntactic and dependency information can significantly outperform the current defect prediction models. In the case of cross project, we examined our framework by combining all the pairs of source projects and target projects. In fact, the pair-wise cross-project predictions do not mimic any strategy to select the source training project from all source projects. However, as it was concluded in chapter 3, carefully selecting the candidate source projects instead of randomly choosing one or several source projects, can significantly ameliorate the quality of cross-project.

Many approaches have been proposed to tackle the CPDP challenges. Their main aim is to alleviate the large data distribution between source projects and target project by proposing different strategies. However, all these strategies are based on traditional metrics, and none of them consider the semantics and the dependencies in the software program. Furthermore, none of them qualify the source projects in terms of various aspects such as organizational, technical, and functional aspects, etc.

To solve this issue, we propose in chapter 5, a framework of source project selection which leverages a selection of best candidate source projects based on semantic and structural representation of code to detect meaningful defective patterns; and the external qualification of projects in terms of distinct aspects to extract a global knowledge of projects. Evaluations on open-source projects demonstrate that our defect prediction framework including a source project selection strategy can improve our defect prediction framework without any strategy for selecting source project and the existing defect cross-project approaches.

6.2 Future Work

The application of deep learning algorithms based on a solid code representation like the code property graph show promising results to improve the software defect prediction. Based on the findings published in this thesis, we have determined eventual directions for future research.

Further improvements and generalizing the proposed end-to-end deep learning framework based on code property graph is needed. As our proposed framework shows successful results in this thesis compared to the existing defect prediction frameworks, we aim to improve the code representation by considering inter-procedural dependencies in program (i.e., the dependencies between classes). This help to detect more complex bugs which are linked to inter-dependencies. Moreover, we aim to implement our framework on other

Leveraging code property graph and deep learning method to tackle the automatic program challenges. Previous research have proved that the use of deep learning with a powerful representation of code could solve several software analytics issues such as software defect prediction (the subject of this thesis), malware classification [152], [227], software traceability [153], test report classification [156], etc. Along this direction, we plan to explore the application of code property graph and deep learning on the problem of automatic program. The main objective of this research field is to automatically find a solution to software bugs without any human intervention. This topic has been investigated by many researchers over the years [228], [229]. But most of current methods cannot fix complex bugs [230]. The basic insight of this topic is that the open-source projects have thousands of bug-fixing history records (i.e., patches). We can extract important knowledge from these patches, and then exploit them to automatically fix new bugs. Therefore, the application of deep learning could automatically explore the past efforts of developers and then help to fix similar bugs in a new project.

References

[1] « Gartner Says Worldwide IT Spending on Pace to Grow 3.2 Percent in 2014 », *Gartner*. https://www.gartner.com/en/newsroom/press-releases/2014-04-02-gartner-says-worldwide-it-spending-on-pace-to-grow-3-percent-in-2014 (consulté le 11 mars 2021).

[2] « World Quality Report 2013-14 », *Capgemini Worldwide*, 12 septembre 2013. https://www.capgemini.com/resources/world-quality-report-2013-14/ (consulté le 11 mars 2021).

[3] S. Planning, « The economic impacts of inadequate infrastructure for software testing », *National Institute of Standards and Technology*, 2002.

[4] « Faulty software can lead to astronomic costs », *ComputerWeekly.com*. https://www.computerweekly.com/opinion/Faulty-software-can-lead-to-astronomic-costs (consulté le 11 mars 2021).

[5] L. Pelayo et S. Dick, « Applying novel resampling strategies to software defect prediction », in *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*, 2007, p. 69-72.

[6] C. Catal et B. Diri, « A systematic review of software fault prediction studies », *Expert systems with applications*, vol. 36, n° 4, Art. n° 4, 2009.

[7] M. D'Ambros, M. Lanza, et R. Robbes, « Evaluating defect prediction approaches: a benchmark and an extensive comparison », *Empirical Software Engineering*, vol. 17, n° 4, p. 531-577, 2012.

[8] Y. Kamei et E. Shihab, « Defect prediction: Accomplishments and future challenges », in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, 2016, vol. 5, p. 33-45.

[9] T. Jiang, L. Tan, et S. Kim, « Personalized defect prediction », in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, p. 279-289.

[10] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, et J. Liu, « Dictionary learning based software defect prediction », in *Proceedings of the 36th International Conference on Software Engineering*, 2014, p. 414-423.

[11] C. Catal, « Software fault prediction: A literature review and current trends », *Expert systems with applications*, vol. 38, n° 4, Art. n° 4, 2011.

[12] S. Wang, T. Liu, et L. Tan, « Automatically learning semantic features for defect prediction », in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, p. 297-308.

[13] Q. Yu, J. Qian, S. Jiang, Z. Wu, et G. Zhang, « An empirical study on the effectiveness of feature selection for cross-project defect prediction », *IEEE Access*, vol. 7, p. 35710-35718, 2019.

[14] S. Hosseini, B. Turhan, et D. Gunarathna, « A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction », *IEEE Transactions on Software Engineering*, vol. 45, n° 2, p. 111-147, févr. 2019, doi: 10.1109/TSE.2017.2770124.

[15] S. Herbold, A. Trautsch, et J. Grabowski, « A comparative study to benchmark cross-project defect prediction approaches », *IEEE Transactions on Software Engineering*, vol. 44, n° 9, p. 811-833, 2017.

[16] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, et B. Murphy, « Cross-project defect prediction: a large scale experiment on data vs. domain vs. process », in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, p. 91-100.

[17] M. H. Halstead, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[18] T. J. McCabe, « A complexity measure », *IEEE Transactions on software Engineering*, n° 4, Art. n° 4, 1976.
[19] S. R. Chidamber et C. F. Kemerer, « A metrics suite for object oriented design », *IEEE Transactions on software engineering*, vol. 20, n° 6, p. 476-493, 1994.
[20] T. Zimmermann et N. Nagappan, « Predicting defects using network analysis on dependency graphs », in *Proceedings of the 30th international conference on Software engineering*, 2008, p. 531-540.
[21] T. Mikolov, K. Chen, G. Corrado, et J. Dean, « Efficient estimation of word representations in vector space », *arXiv preprint arXiv:1301.3781*, 2013.
[22] F. Yamaguchi, N. Golde, D. Arp, et K. Rieck, « Modeling and discovering vulnerabilities with code property graphs », in *2014 IEEE Symposium on Security and Privacy*, 2014, p. 590-604.
[23] M. D'Ambros, M. Lanza, et R. Robbes, « An extensive comparison of bug prediction approaches », in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, p. 31-41.
[24] A. E. Hassan, « Predicting faults using the complexity of code changes », in *2009 IEEE 31st international conference on software engineering*, 2009, p. 78-88.
[25] N. Nagappan et T. Ball, « Use of relative code churn measures to predict system defect density », in *Proceedings of the 27th international conference on Software engineering*, 2005, p. 284-292.
[26] J. Li, P. He, J. Zhu, et M. R. Lyu, « Software defect prediction via convolutional neural network », in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, p. 318-328.
[27] A. V. Phan, M. Le Nguyen, et L. T. Bui, « Convolutional neural networks over control flow graphs for software defect prediction », in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2017, p. 45-52.
[28] Y. Li, S. Wang, T. N. Nguyen, et S. Van Nguyen, « Improving bug detection via context-based code representation learning and attention-based neural networks », *Proceedings of the ACM on Programming Languages*, vol. 3, n° OOPSLA, p. 1-30, 2019.
[29] S. Meilong, P. He, H. Xiao, H. Li, et C. Zeng, « An Approach to Semantic and Structural Features Learning for Software Defect Prediction », *Mathematical Problems in Engineering*, vol. 2020, 2020.
[30] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, et N. Ubayashi, « DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction », in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, mai 2019, p. 34-45. doi: 10.1109/MSR.2019.00016.
[31] H. Tong, B. Liu, et S. Wang, « Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning », *Information and Software Technology*, vol. 96, p. 94-111, 2018.
[32] S. Wang, T. Liu, J. Nam, et L. Tan, « Deep semantic feature learning for software defect prediction », *IEEE Transactions on Software Engineering*, 2018.
[33] M. Jureczko et L. Madeyski, « Towards identifying software project clusters with regard to defect prediction », in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, New York, NY, USA, sept. 2010, p. 1-10. doi: 10.1145/1868328.1868342.
[34] R. N. Charette, « Why software fails [software failure] », *IEEE spectrum*, vol. 42, n° 9, p. 42-49, 2005.
[35] G. Rajkumar et K. Alagarsamy, « The most common factors for the failure of software development project », *The International Journal of Computer Science & Applications (TIJCSA)*, vol. 1, n° 11, p. 74-77, 2013.

[36] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, et R. A. Paul, « Empirical assessment of machine learning based software defect prediction techniques », *International Journal on Artificial Intelligence Tools*, vol. 17, n° 02, p. 389-400, 2008.
[37] S. Dalal et R. S. Chhillar, « Case studies of most common and severe types of software system failure », *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 2, n° 8, 2012.
[38] T. Lee, J. Nam, D. Han, S. Kim, et H. P. In, « Micro interaction metrics for defect prediction », in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, p. 311-321.
[39] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, et A. Bener, « Defect prediction from static code features: current results, limitations, new approaches », *Automated Software Engineering*, vol. 17, n° 4, p. 375-407, 2010.
[40] J. Nam, W. Fu, S. Kim, T. Menzies, et L. Tan, « Heterogeneous defect prediction », *IEEE Transactions on Software Engineering*, vol. 44, n° 9, p. 874-896, 2017.
[41] J. Nam, S. J. Pan, et S. Kim, « Transfer defect learning », in *2013 35th international conference on software engineering (ICSE)*, 2013, p. 382-391.
[42] M. Pinzger, N. Nagappan, et B. Murphy, « Can developer-module networks predict failures? », in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, p. 2-12.
[43] E. Giger, M. D'Ambros, M. Pinzger, et H. C. Gall, « Method-level bug prediction », in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012, p. 171-180.
[44] S. McIntosh et Y. Kamei, « Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction », *IEEE Transactions on Software Engineering*, vol. 44, n° 5, p. 412-428, 2017.
[45] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, et S. Li, « The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction », *IEEE Transactions on Software Engineering*, 2019.
[46] S. Kim, T. Zimmermann, K. Pan, et E. James Jr, « Automatic identification of bug-introducing changes », in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, 2006, p. 81-90.
[47] A. Mockus et D. M. Weiss, « Predicting risk of software changes », *Bell Labs Technical Journal*, vol. 5, n° 2, p. 169-180, 2000.
[48] M. Tan, L. Tan, S. Dara, et C. Mayeux, « Online defect prediction for imbalanced data », in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 2, p. 99-108.
[49] X. Yang, D. Lo, X. Xia, Y. Zhang, et J. Sun, « Deep learning for just-in-time defect prediction », in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, p. 17-26.
[50] P. Lam, E. Bodden, O. Lhoták, et L. Hendren, « The Soot framework for Java program analysis: a retrospective », in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011, vol. 15, n° 35.
[51] R. Pawlak, C. Noguera, et N. Petitprez, « Spoon: Program analysis and transformation in java », PhD Thesis, Inria, 2006.
[52] E. Söderberg, T. Ekman, G. Hedin, et E. Magnusson, « Extensible intraprocedural flow analysis at the abstract syntax tree level », *Science of Computer Programming*, vol. 78, n° 10, p. 1809-1827, 2013.
[53] A. Robinson et C. Bates, « APRT–Another Pattern Recognition Tool », *GSTF Journal on Computing*, vol. 5, n° 2, 2017.
[54] T. Parr et K. Fisher, « LL (*) the foundation of the ANTLR parser generator », *ACM Sigplan Notices*, vol. 46, n° 6, p. 425-436, 2011.
[55] « JastAdd.org ». https://jastadd.cs.lth.se/web/ (consulté le 23 mars 2021).

[56] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
[57] F. E. Allen, « Control flow analysis », *ACM Sigplan Notices*, vol. 5, n° 7, p. 1-19, 1970.
[58] D. Bruschi, L. Martignoni, et M. Monga, « Detecting self-mutating malware using control-flow graph matching », in *International conference on detection of intrusions and malware, and vulnerability assessment*, 2006, p. 129-143.
[59] B. Anderson, D. Quist, J. Neil, C. Storlie, et T. Lane, « Graph-based malware detection using dynamic analysis », *Journal in computer Virology*, vol. 7, n° 4, p. 247-258, 2011.
[60] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, et L. Xie, « Detecting code reuse in android applications using component-based control flow graph », in *IFIP international information security conference*, 2014, p. 142-155.
[61] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, et E. G. Im, « Software plagiarism detection: a graph-based approach », in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, p. 1577-1580.
[62] J. Ferrante, K. J. Ottenstein, et J. D. Warren, « The program dependence graph and its use in optimization », *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, n° 3, p. 319-349, 1987.
[63] A. V. Aho, R. Sethi, et J. D. Ullman, « Compilers, principles, techniques », *Addison wesley*, vol. 7, n° 8, p. 9, 1986.
[64] M. A. Rodriguez et P. Neubauer, « The graph traversal pattern », in *Graph data management: Techniques and applications*, IGI Global, 2012, p. 29-46.
[65] L. Gerling et K. Schmid, « Variability-Aware Semantic Slicing Using Code Property Graphs », in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, New York, NY, USA, sept. 2019, p. 65-71. doi: 10.1145/3336294.3336312.
[66] Q. Meng, C. Feng, B. Zhang, et C. Tang, « Assisting in auditing of buffer overflow vulnerabilities via machine learning », *Mathematical Problems in Engineering*, vol. 2017, 2017.
[67] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, et H. Changyu, « CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning », in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, 2018, p. 184-188.
[68] F. Yamaguchi, A. Maier, H. Gascon, et K. Rieck, « Automatic inference of search patterns for taint-style vulnerabilities », in *2015 IEEE Symposium on Security and Privacy*, 2015, p. 797-812.
[69] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, et A. Morari, « Learning to map source code to software vulnerability using code-as-a-graph », *arXiv preprint arXiv:2006.08614*, 2020.
[70] S. Chakraborty, R. Krishna, Y. Ding, et B. Ray, « Deep learning based vulnerability detection: Are we there yet », *IEEE Transactions on Software Engineering*, 2021.
[71] X. Li, H. Jiang, Z. Ren, G. Li, et J. Zhang, « Deep learning in software engineering », *arXiv preprint arXiv:1805.04825*, 2018.
[72] F. Ferreira, L. L. Silva, et M. T. Valente, « Software Engineering Meets Deep Learning: A Literature Review », *arXiv preprint arXiv:1909.11436*, 2019.
[73] J.-T. Huang, J. Li, et Y. Gong, « An analysis of convolutional neural networks for speech recognition », in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, p. 4989-4993.
[74] L. Deng *et al.*, « Recent advances in deep learning for speech research at Microsoft », in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, p. 8604-8608.
[75] Y. Wei *et al.*, « HCP: A flexible CNN framework for multi-label image classification », *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, n° 9, p. 1901-1907, 2015.
[76] G. E. Hinton, « Learning multiple layers of representation », *Trends in cognitive sciences*, vol. 11, n° 10, p. 428-434, 2007.
[77] W. Yin, K. Kann, M. Yu, et H. Schütze, « Comparative study of CNN and RNN for natural language processing », *arXiv preprint arXiv:1702.01923*, 2017.
[78] T. Young, D. Hazarika, S. Poria, et E. Cambria, « Recent trends in deep learning based natural language processing », *ieee Computational intelligenCe magazine*, vol. 13, n° 3, p. 55-75, 2018.

[79] H. Sak, A. Senior, et F. Beaufays, « Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition », *arXiv preprint arXiv:1402.1128*, 2014.
[80] Y. Wang, « A new concept using LSTM Neural Networks for dynamic system identification », in *2017 American Control Conference (ACC)*, 2017, p. 5324-5329.
[81] A. Krizhevsky, I. Sutskever, et G. E. Hinton, « Imagenet classification with deep convolutional neural networks », *Advances in neural information processing systems*, vol. 25, p. 1097-1105, 2012.
[82] X. Zhang, J. Zou, K. He, et J. Sun, « Accelerating very deep convolutional networks for classification and detection », *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, nº 10, p. 1943-1955, 2015.
[83] C. Dong, C. C. Loy, K. He, et X. Tang, « Image super-resolution using deep convolutional networks », *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, nº 2, p. 295-307, 2015.
[84] S. Ren, K. He, R. Girshick, et J. Sun, « Faster R-CNN: towards real-time object detection with region proposal networks », *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, nº 6, p. 1137-1149, 2016.
[85] J. You, R. Ying, et J. Leskovec, « Position-aware graph neural networks », in *International Conference on Machine Learning*, 2019, p. 7134-7143.
[86] M. Defferrard, X. Bresson, et P. Vandergheynst, « Convolutional neural networks on graphs with fast localized spectral filtering », *arXiv preprint arXiv:1606.09375*, 2016.
[87] J. Bruna, W. Zaremba, A. Szlam, et Y. LeCun, « Spectral networks and locally connected networks on graphs », *arXiv preprint arXiv:1312.6203*, 2013.
[88] T. Komorowski, C. Landim, et S. Olla, *Fluctuations in Markov processes: time symmetry and martingale approximation*, vol. 345. Springer Science & Business Media, 2012.
[89] H.-H. Lin, J.-H. Chuang, et T.-L. Liu, « Regularized background adaptation: a novel learning rate control scheme for Gaussian mixture modeling », *IEEE Transactions on Image Processing*, vol. 20, nº 3, p. 822-836, 2010.
[90] K. Simonyan et A. Zisserman, « Very deep convolutional networks for large-scale image recognition », *arXiv preprint arXiv:1409.1556*, 2014.
[91] N. Verma, E. Boyer, et J. Verbeek, « Dynamic filters in graph convolutional networks », *arXiv preprint arXiv:1706.05206*, vol. 2, nº 6, 2017.
[92] M. Zhang, Z. Cui, M. Neumann, et Y. Chen, « An end-to-end deep learning architecture for graph classification », 2018.
[93] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, et K. M. Borgwardt, « Weisfeiler-lehman graph kernels. », *Journal of Machine Learning Research*, vol. 12, nº 9, 2011.
[94] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, et N. Ubayashi, « An empirical study of just-in-time defect prediction using cross-project models », in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, p. 172-181.
[95] Y. Kamei *et al.*, « A large-scale empirical study of just-in-time quality assurance », *IEEE Transactions on Software Engineering*, vol. 39, nº 6, p. 757-773, 2012.
[96] S. Kim, E. J. Whitehead Jr, et Y. Zhang, « Classifying software changes: Clean or buggy? », *IEEE Transactions on Software Engineering*, vol. 34, nº 2, p. 181-196, 2008.
[97] « Adapting a fault prediction model to allow inter languagereuse | Proceedings of the 4th international workshop on Predictor models in software engineering ». https://dl.acm.org/doi/abs/10.1145/1370788.1370794 (consulté le 4 mai 2020).
[98] Y. Ma, G. Luo, X. Zeng, et A. Chen, « Transfer learning for cross-company software defect prediction », *Information and Software Technology*, vol. 54, p. 248-256, mars 2012, doi: 10.1016/j.infsof.2011.09.007.

[99] B. Turhan, T. Menzies, A. B. Bener, et J. Di Stefano, « On the relative value of cross-company and within-company data for defect prediction », *Empirical Software Engineering*, vol. 14, n° 5, p. 540-578, 2009.
[100] Z. He, F. Shu, Y. Yang, M. Li, et Q. Wang, « An investigation on the feasibility of cross-project defect prediction », *Automated Software Engineering*, vol. 19, n° 2, p. 167-199, 2012.
[101] A. Bacchelli, M. D'Ambros, et M. Lanza, « Are popular classes more defect prone? », in *International Conference on Fundamental Approaches to Software Engineering*, 2010, p. 59-73.
[102] C. Bird, N. Nagappan, B. Murphy, H. Gall, et P. Devanbu, « Don't touch my code! Examining the effects of ownership on software quality », in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, p. 4-14.
[103] T. Menzies, J. Greenwald, et A. Frank, « Data mining static code attributes to learn defect predictors », *IEEE transactions on software engineering*, vol. 33, n° 1, p. 2-13, 2006.
[104] S. Shivaji, E. J. Whitehead, R. Akella, et S. Kim, « Reducing features to improve code change-based bug prediction », *IEEE Transactions on Software Engineering*, vol. 39, n° 4, p. 552-569, 2012.
[105] S. Kim, H. Zhang, R. wu, et L. Gong, *Dealing with noise in defect prediction*. 2011, p. 490. doi: 10.1145/1985793.1985859.
[106] R. Wu, H. Zhang, S. Kim, et S.-C. Cheung, « ReLink: recovering links between bugs and changes », in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, Szeged, Hungary, 2011, p. 15. doi: 10.1145/2025113.2025120.
[107] J. Ge, J. Liu, et W. Liu, « Comparative study on defect prediction algorithms of supervised learning software based on imbalanced classification data sets », in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2018, p. 399-406.
[108] F. Rahman et P. Devanbu, « How, and why, process metrics are better », in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, p. 432-441.
[109] D. Mao, L. Chen, et L. Zhang, « An extensive study on cross-project predictive mutation testing », in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, p. 160-171.
[110] X. Li, W. Li, Y. Zhang, et L. Zhang, « Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization », in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, p. 169-180.
[111] P. Zhou, J. Liu, X. Liu, Z. Yang, et J. Grundy, « Is deep learning better than traditional approaches in tag recommendation for software information sites? », *Information and software technology*, vol. 109, p. 1-13, 2019.
[112] J. Ott, A. Atchison, P. Harnack, A. Bergh, et E. Linstead, « A deep learning approach to identifying source code in images and video », in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, p. 376-386.
[113] M. K. Daskalantonakis, « A practical view of software measurement and implementation experiences within Motorola », *IEEE Transactions on Software Engineering*, vol. 18, n° 11, p. 998, 1992.
[114] Q. Song, Z. Jia, M. Shepperd, S. Ying, et J. Liu, « A general software defect-proneness prediction framework », *IEEE transactions on software engineering*, vol. 37, n° 3, p. 356-370, 2010.
[115] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, et A. E. Hassan, « Revisiting common bug prediction findings using effort-aware models », in *2010 IEEE International Conference on Software Maintenance*, 2010, p. 1-10.

[116] G. J. Pai et J. B. Dugan, « Empirical analysis of software fault content and fault proneness using Bayesian methods », *IEEE Transactions on software Engineering*, vol. 33, n° 10, p. 675-686, 2007.
[117] H. Zhang et R. Wu, « Sampling program quality », in *2010 IEEE International Conference on Software Maintenance*, 2010, p. 1-10.
[118] V. R. Basili, L. C. Briand, et W. L. Melo, « A validation of object-oriented design metrics as quality indicators », *IEEE Transactions on software engineering*, vol. 22, n° 10, p. 751-761, 1996.
[119] R. Subramanyam et M. S. Krishnan, « Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects », *IEEE Transactions on software engineering*, vol. 29, n° 4, p. 297-310, 2003.
[120] H. Hata, O. Mizuno, et T. Kikuno, « Bug prediction based on fine-grained module histories », in *2012 34th international conference on software engineering (ICSE)*, 2012, p. 200-210.
[121] E. Shihab, A. Mockus, Y. Kamei, B. Adams, et A. E. Hassan, « High-impact defects: a study of breakage and surprise defects », in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, p. 300-310.
[122] R. Shatnawi et W. Li, « The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process », *Journal of systems and software*, vol. 81, n° 11, p. 1868-1882, 2008.
[123] R. Moser, W. Pedrycz, et G. Succi, « A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction », in *Proceedings of the 30th international conference on Software engineering*, 2008, p. 181-190.
[124] L. Madeyski et M. Jureczko, « Which process metrics can significantly improve defect prediction models? An empirical study », *Software Quality Journal*, vol. 23, n° 3, p. 393-422, 2015.
[125] T. L. Graves, A. F. Karr, J. S. Marron, et H. Siy, « Predicting fault incidence using software change history », *IEEE Transactions on software engineering*, vol. 26, n° 7, p. 653-661, 2000.
[126] Q. Yu, S. Jiang, J. Qian, L. Bo, L. Jiang, et G. Zhang, « Process metrics for software defect prediction in object-oriented programs », *IET Software*, vol. 14, n° 3, p. 283-292, 2020.
[127] B. Stanić et W. Afzal, « Process metrics are not bad predictors of fault proneness », in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, p. 493-499.
[128] C. Bird, N. Nagappan, H. Gall, B. Murphy, et P. Devanbu, « Putting it all together: Using socio-technical networks to predict failures », in *2009 20th International Symposium on Software Reliability Engineering*, 2009, p. 109-119.
[129] A. Meneely, L. Williams, W. Snipes, et J. Osborne, « Predicting failures with developer networks and social network analysis », in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, p. 13-23.
[130] Y. Qu *et al.*, « node2defect: Using network embedding to improve software defect prediction », in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, p. 844-849.
[131] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, et Z. Jin, « Building program vector representations for deep learning », in *International Conference on Knowledge Science, Engineering and Management*, 2015, p. 547-553.
[132] K. Shi, Y. Lu, G. Liu, Z. Wei, et J. Chang, « MPT-embedding: An unsupervised representation learning of code for software defect prediction », *Journal of Software: Evolution and Process*, vol. 33, n° 4, p. e2330, 2021.
[133] H. K. Dam *et al.*, « A deep tree-based model for software defect prediction », *arXiv:1802.00921 [cs]*, févr. 2018, Consulté le: 3 mai 2020. [En ligne]. Disponible sur: http://arxiv.org/abs/1802.00921

[134] G. Fan, X. Diao, H. Yu, K. Yang, et L. Chen, « Software defect prediction via attention-based recurrent neural network », *Scientific Programming*, vol. 2019, 2019.
[135] D. Chen, X. Chen, H. Li, J. Xie, et Y. Mu, « DeepCPDP: Deep Learning Based Cross-Project Defect Prediction », *IEEE Access*, vol. 7, p. 184832-184848, 2019, doi: 10.1109/ACCESS.2019.2961129.
[136] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, et H. Rajan, « A study of repetitiveness of code changes in software evolution », in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, p. 180-190.
[137] E. Erturk et E. A. Sezer, « A comparison of some soft computing methods for software fault prediction », *Expert systems with applications*, vol. 42, n° 4, p. 1872-1879, 2015.
[138] M. S. Naidu et N. Geethanjali, « Classification of defects in software using decision tree algorithm », *International Journal of Engineering Science and Technology*, vol. 5, n° 6, p. 1332, 2013.
[139] S. Aleem, L. F. Capretz, et F. Ahmed, « Benchmarking machine learning technologies for software defect detection », *arXiv preprint arXiv:1506.07563*, 2015.
[140] H. A. Al-Jamimi et L. Ghouti, « Efficient prediction of software fault proneness modules using support vector machines and probabilistic neural networks », in *2011 Malaysian Conference in Software Engineering*, 2011, p. 251-256.
[141] L. Guo, Y. Ma, B. Cukic, et H. Singh, « Robust prediction of fault-proneness by random forests », in *15th international symposium on software reliability engineering*, 2004, p. 417-428.
[142] N. Azeem et S. Usmani, « Analysis of data mining based software defect prediction techniques », *Global Journal of Computer Science and Technology*, 2011.
[143] A. Okutan et O. T. Yıldız, « Software defect prediction using Bayesian networks », *Empirical Software Engineering*, vol. 19, n° 1, p. 154-181, 2014.
[144] X.-H. Liu, T. Wang, et Z.-Q. Wu, « Software defect prediction based on classifiers ensemble [J] », *Application Research of Computers*, vol. 6, 2013.
[145] Y. LeCun, Y. Bengio, et G. Hinton, « Deep learning », *nature*, vol. 521, n° 7553, p. 436-444, 2015.
[146] Y. LeCun *et al.*, « Handwritten digit recognition with a back-propagation network », in *Advances in neural information processing systems*, 1990, p. 396-404.
[147] J. Gu *et al.*, « Recent advances in convolutional neural networks », *Pattern Recognition*, vol. 77, p. 354-377, 2018.
[148] I. Goodfellow, Y. Bengio, A. Courville, et Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.
[149] T. Menzies, S. Majumder, N. Balaji, K. Brey, et W. Fu, « 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow) », in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, p. 554-563.
[150] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, et L. Zeng, « Automatic text input generation for mobile testing », in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, p. 643-653.
[151] Z. Yuan, Y. Lu, Z. Wang, et Y. Xue, « Droid-sec: deep learning in android malware detection », in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, p. 371-372.
[152] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, et A. Thomas, « Malware classification with recurrent networks », in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, p. 1916-1920.
[153] J. Guo, J. Cheng, et J. Cleland-Huang, « Semantically enhanced software traceability using deep learning techniques », in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, p. 3-14.
[154] X. Gu, H. Zhang, et S. Kim, « Deep code search », in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, p. 933-944.

[155] X. Gu, H. Zhang, D. Zhang, et S. Kim, « Deep API learning », in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, p. 631-642.
[156] J. Wang, Q. Cui, S. Wang, et Q. Wang, « Domain adaptation for test report classification in crowdsourced testing », in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, p. 83-92.
[157] A. N. Lam, A. T. Nguyen, H. A. Nguyen, et T. N. Nguyen, « Combining deep learning with information retrieval to localize buggy files for bug reports (n) », in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, p. 476-481.
[158] L. Li, H. Feng, W. Zhuang, N. Meng, et B. Ryder, « Cclearner: A deep learning-based clone detection approach », in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, p. 249-260.
[159] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, et A. Grama, « MODE: automated neural network model debugging via state differential analysis and input selection », in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 175-186.
[160] L. Mou, G. Li, Z. Jin, L. Zhang, et T. Wang, « TBCNN: A tree-based convolutional neural network for programming language processing », *arXiv preprint arXiv:1409.5718*, 2014.
[161] V. Raychev, M. Vechev, et E. Yahav, « Code completion with statistical language models », in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, p. 419-428.
[162] M. White, C. Vendome, M. Linares-Vásquez, et D. Poshyvanyk, « Toward deep learning software repositories », in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, p. 334-345.
[163] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, et S. Li, « Predicting semantically linkable knowledge in developer online forums via convolutional neural network », in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, p. 51-62.
[164] M. White, M. Tufano, C. Vendome, et D. Poshyvanyk, « Deep learning code fragments for code clone detection », in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, sept. 2016, p. 87-98.
[165] « Deep API learning | Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering », 3 mai 2020. https://dl.acm.org/doi/abs/10.1145/2950290.2950334 (consulté le 3 mai 2020).
[166] R. Gupta, S. Pal, A. Kanade, et S. Shevade, « DeepFix: Fixing Common C Language Errors by Deep Learning », présenté à Thirty-First AAAI Conference on Artificial Intelligence, févr. 2017. Consulté le: 3 mai 2020. [En ligne]. Disponible sur: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603
[167] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, et A. Ghose, « Automatic feature learning for vulnerability prediction », *arXiv preprint arXiv:1708.02368*, 2017.
[168] F. Dong, J. Wang, Q. Li, G. Xu, et S. Zhang, « Defect prediction in android binary executables using deep neural network », *Wireless Personal Communications*, vol. 102, n° 3, p. 2261-2285, 2018.
[169] Z. Li *et al.*, « Vuldeepecker: A deep learning-based system for vulnerability detection », *arXiv preprint arXiv:1801.01681*, 2018.
[170] M. Wen, R. Wu, et S.-C. Cheung, « How well do change sequences predict defects? sequence learning from software changes », *IEEE Transactions on Software Engineering*, vol. 46, n° 11, p. 1155-1175, 2018.
[171] H. K. Dam, T. Tran, et T. Pham, « A deep language model for software code », *arXiv:1608.02715 [cs, stat]*, août 2016, Consulté le: 3 mai 2020. [En ligne]. Disponible sur: http://arxiv.org/abs/1608.02715

[172] T. Zhou, X. Sun, X. Xia, B. Li, et X. Chen, « Improving defect prediction with deep forest », *Information and Software Technology*, vol. 114, p. 204-216, 2019.
[173] A. Krizhevsky, I. Sutskever, et G. E. Hinton, « ImageNet classification with deep convolutional neural networks », *Communications of the ACM*, vol. 60, nº 6, p. 84-90, 2017.
[174] J. Atwood et D. Towsley, « Diffusion-Convolutional Neural Networks », *arXiv:1511.02136 [cs]*, juill. 2016, Consulté le: 1 avril 2021. [En ligne]. Disponible sur: http://arxiv.org/abs/1511.02136
[175] D. Duvenaud *et al.*, « Convolutional Networks on Graphs for Learning Molecular Fingerprints », *arXiv:1509.09292 [cs, stat]*, nov. 2015, Consulté le: 1 avril 2021. [En ligne]. Disponible sur: http://arxiv.org/abs/1509.09292
[176] J.-C. Vialatte, V. Gripon, et G. Mercier, « Generalizing the Convolution Operator to extend CNNs to Irregular Domains », *arXiv:1606.01166 [cs]*, oct. 2017, Consulté le: 1 avril 2021. [En ligne]. Disponible sur: http://arxiv.org/abs/1606.01166
[177] Y. Liu, T. M. Khoshgoftaar, et N. Seliya, « Evolutionary Optimization of Software Quality Modeling with Multiple Repositories », *IEEE Transactions on Software Engineering*, vol. 36, nº 6, p. 852-864, nov. 2010, doi: 10.1109/TSE.2010.51.
[178] A. Panichella, R. Oliveto, et A. Lucia, *Cross-Project Defect Prediction Models: L'union fait la force*. 2014. doi: 10.1109/CSMR-WCRE.2014.6747166.
[179] « On the relative value of cross-company and within-company data for defect prediction | SpringerLink », 4 mai 2020. https://link.springer.com/article/10.1007/s10664-008-9103-7 (consulté le 4 mai 2020).
[180] S. J. Pan et Q. Yang, « A Survey on Transfer Learning », *IEEE Trans. Knowl. Data Eng.*, vol. 22, nº 10, p. 1345-1359, oct. 2010, doi: 10.1109/TKDE.2009.191.
[181] « Towards logistic regression models for predicting fault-prone code across software projects | IEEE Conference Publication | IEEE Xplore ». https://ieeexplore.ieee.org/abstract/document/5316002 (consulté le 1 avril 2021).
[182] F. Peters, T. Menzies, et A. Marcus, « Better cross company defect prediction », in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, p. 409-418.
[183] L. Peng, B. Yang, Y. Chen, et A. Abraham, « Data gravitation based classification », *Information Sciences*, vol. 179, nº 6, p. 809-819, mars 2009, doi: 10.1016/j.ins.2008.11.007.
[184] « Improving Cross-Project Defect Prediction Methods with Data Simplification | IEEE Conference Publication | IEEE Xplore ». https://ieeexplore.ieee.org/abstract/document/7302438 (consulté le 1 avril 2021).
[185] « Negative samples reduction in cross-company software defects prediction - ScienceDirect ». https://www.sciencedirect.com/science/article/abs/pii/S0950584915000348 (consulté le 1 avril 2021).
[186] Y. Yao et G. Doretto, *Boosting for transfer learning with multiple sources*. 2010, p. 1862. doi: 10.1109/CVPR.2010.5539857.
[187] Z. Yuan, X. Chen, Z. Cui, et Y. Mu, « ALTRA: Cross-project software defect prediction via active learning and tradaboost », *IEEE Access*, vol. 8, p. 30037-30049, 2020.
[188] « Which Is More Important for Cross-Project Defect Prediction: Instance or Feature? | IEEE Conference Publication | IEEE Xplore ». https://ieeexplore.ieee.org/abstract/document/7780200 (consulté le 1 avril 2021).
[189] « Training data selection for cross-project defect prediction | Proceedings of the 9th International Conference on Predictive Models in Software Engineering ». https://dl.acm.org/doi/abs/10.1145/2499393.2499395 (consulté le 1 avril 2021).
[190] « A two-phase transfer learning model for cross-project defect prediction - ScienceDirect », 3 mai 2020. https://www.sciencedirect.com/science/article/abs/pii/S0950584918302416 (consulté le 3 mai 2020).
[191] R. Krishna et T. Menzies, « Bellwethers: A baseline method for transfer learning », *IEEE Transactions on Software Engineering*, vol. 45, nº 11, p. 1081-1105, 2018.

[192] M. Allamanis, E. T. Barr, C. Bird, et C. Sutton, « Learning natural coding conventions », in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, p. 281-293.
[193] E. Arisholm, L. C. Briand, et M. Fuglerud, « Data mining techniques for building fault-proneness models in telecom java software », in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, 2007, p. 215-224.
[194] Z. He, F. Peters, T. Menzies, et Y. Yang, « Learning from open-source projects: An empirical study on defect prediction », in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, p. 45-54.
[195] F. Rahman, D. Posnett, et P. Devanbu, « Recalling the" imprecision" of cross-project defect prediction », in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, p. 1-11.
[196] X. Xia, D. Lo, S. J. Pan, N. Nagappan, et X. Wang, « Hydra: Massively compositional model for cross-project defect prediction », *IEEE Transactions on software Engineering*, vol. 42, n° 10, p. 977-998, 2016.
[197] X. Xia, D. Lo, X. Wang, et X. Yang, « Collective personalized change classification with multiobjective search », *IEEE Transactions on Reliability*, vol. 65, n° 4, p. 1810-1829, 2016.
[198] M. Fischer, M. Pinzger, et H. Gall, « Populating a release history database from version control and bug tracking systems », in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, p. 23-32.
[199] M. Fischer, M. Pinzger, et H. Gall, « Analyzing and relating bug report data for feature tracking », in *WCRE*, 2003, vol. 3, p. 90.
[200] D. Cubranic et G. C. Murphy, « Hipikat: Recommending pertinent software development artifacts », in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, p. 408-418.
[201] E. Murphy-Hill, C. Parnin, et A. P. Black, « How we refactor, and how we know it », *IEEE Transactions on Software Engineering*, vol. 38, n° 1, p. 5-18, 2011.
[202] J. Śliwerski, T. Zimmermann, et A. Zeller, « When do changes induce fixes? », *ACM sigsoft software engineering notes*, vol. 30, n° 4, p. 1-5, 2005.
[203] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, et K. M. Borgwardt, « Weisfeiler-Lehman graph kernels. », *Journal of Machine Learning Research*, vol. 12, n° 9, 2011.
[204] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, et K. Matsumoto, « An empirical comparison of model validation techniques for defect prediction models », *IEEE Transactions on Software Engineering*, vol. 43, n° 1, p. 1-18, 2016.
[205] « A systematic review of unsupervised learning techniques for software defect prediction - ScienceDirect ». https://www.sciencedirect.com/science/article/abs/pii/S0950584920300379 (consulté le 4 mai 2020).
[206] A. Hasanpour, P. Farzi, A. Tehrani, et R. Akbari, « Software Defect Prediction Based On Deep Learning Models: Performance Study », *arXiv:2004.02589 [cs]*, avr. 2020, Consulté le: 4 mai 2020. [En ligne]. Disponible sur: http://arxiv.org/abs/2004.02589
[207] T. Shippey, T. Hall, S. Counsell, et D. Bowes, « So You Need More Method Level Datasets for Your Software Defect Prediction? Voilà! », in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, p. 1-6.
[208] H. He et E. A. Garcia, « Learning from imbalanced data », *IEEE Transactions on knowledge and data engineering*, vol. 21, n° 9, p. 1263-1284, 2009.
[209] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, et A. E. Hassan, « Studying just-in-time defect prediction using cross-project models », *Empirical Software Engineering*, vol. 21, n° 5, p. 2072-2106, 2016.
[210] S. Kim et E. J. Whitehead Jr, « How long did it take to fix bugs? », in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, p. 173-174.

[211] T.-H. Chen, M. Nagappan, E. Shihab, et A. E. Hassan, « An empirical study of dormant bugs », in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, p. 82-91.
[212] J. Eyolfson, L. Tan, et P. Lam, « Do time of day and developer experience affect commit bugginess? », in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, p. 153-162.
[213] H. Liang, Y. Yu, L. Jiang, et Z. Xie, « Seml: A semantic LSTM model for software defect prediction », *IEEE Access*, vol. 7, p. 83812-83824, 2019.
[214] Q. Huang, X. Xia, et D. Lo, « Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction », *Empirical Software Engineering*, vol. 24, nº 5, p. 2823-2862, 2019.
[215] T. Zimmermann, R. Premraj, et A. Zeller, « Predicting Defects for Eclipse », in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, mai 2007, p. 9-9. doi: 10.1109/PROMISE.2007.10.
[216] D. Ciregan, U. Meier, et J. Schmidhuber, « Multi-column deep neural networks for image classification », in *2012 IEEE conference on computer vision and pattern recognition*, 2012, p. 3642-3649.
[217] A. Mohamed, G. E. Dahl, et G. Hinton, « Acoustic modeling using deep belief networks », *IEEE transactions on audio, speech, and language processing*, vol. 20, nº 1, p. 14-22, 2011.
[218] *A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes - IEEE Journals & Magazine*. 2020. Consulté le: 2 juin 2020. [En ligne]. Disponible sur: https://ieeexplore.ieee.org/abstract/document/7588121
[219] B. Ma, H. Zhang, G. Chen, Y. Zhao, et B. Baesens, « Investigating associative classification for software fault prediction: An experimental perspective », *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, nº 01, Art. nº 01, 2014.
[220] Ö. F. Arar et K. Ayan, « Software defect prediction using cost-sensitive neural network », *Applied Soft Computing*, vol. 33, p. 263-277, 2015.
[221] L. Chen, B. Fang, Z. Shang, et Y. Tang, « Negative samples reduction in cross-company software defects prediction », *Information and Software Technology*, vol. 62, p. 67-77, 2015.
[222] A. Arcuri et L. Briand, « A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering », *Software Testing, Verification and Reliability*, vol. 24, nº 3, Art. nº 3, 2014.
[223] E. J. Braude et M. E. Bernstein, *Software engineering: modern approaches*. Waveland Press, 2016.
[224] N. Fenton et J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2019.
[225] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
[226] S. Amasaki, « Cross-version defect prediction: use historical data, cross-project data, or both? », *Empirical Software Engineering*, vol. 25, nº 2, p. 1573-1595, 2020.
[227] Z. Yuan, Y. Lu, Z. Wang, et Y. Xue, « Droid-sec: deep learning in android malware detection », in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, p. 371-372.
[228] C. Le Goues, M. Dewey-Vogt, S. Forrest, et W. Weimer, « A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each », in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, p. 3-13.
[229] M. Monperrus, « Automatic software repair: a bibliography », *ACM Computing Surveys (CSUR)*, vol. 51, nº 1, p. 1-24, 2018.
[230] M. Motwani, S. Sankaranarayanan, R. Just, et Y. Brun, « Do automated program repair techniques repair hard and important bugs? », *Empirical Software Engineering*, vol. 23, nº 5, p. 2901-2947, 2018.