8-2020

Statistical and deep learning models for software engineering corpora

Van Duc Thong HOANG

STATISTICAL AND DEEP LEARNING MODELS FOR SOFTWARE ENGINEERING CORPORA

HOANG VAN DUC THONG

SINGAPORE MANAGEMENT UNIVERSITY

2020

Statistical and Deep Learning Models for Software Engineering Corpora

Hoang Van Duc Thong

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

David Lo (Supervisor/Chair)
Associate Professor
Singapore Management University

Hady Wirawan Lauw
Associate Professor
Singapore Management University

Lingxiao Jiang
Associate Professor
Singapore Management University

Julia Lawall
Senior Research Scientist
Sorbonne University/Inria/LIP6, France

Singapore Management University
2020

I hereby declare that this dissertation is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in this dissertation.

This dissertation has also not been submitted for any degree in any university previously.

Hoang Van Duc Thong

August 2020, Singapore

Statistical and Deep Learning Models for Software Engineering Corpora

Hoang Van Duc Thong

Abstract

This dissertation focuses on proposing statistical and deep learning models for software engineering corpora to detect bugs in software system. The dissertation aims to solve three main software engineering problems, i.e., bug localization (locating the potential buggy source files in a software project given a bug report or failing test cases), just-in-time defect prediction (identifying the potential defective commits as they are introduced into a version control system), and bug fixing patch identification (identifying commits repairing bugs for their propagation to parallelly maintained versions) to save developers' time and effort in improving software system quality. Moreover, I also propose a neural network model learning a vector representation of code changes based on their commit messages. The vector representation of code changes contains its semantic intent and can be used to improve the performance of just-in-time defect prediction and bug fixing patch identification. This vector can also be applicable for potentially many other software engineering problems related to code changes, such as tangled change prediction, the recommendation of a code reviewer for a patch, etc.

My dissertation develops one statistical model and three deep learning models for various software engineering tasks. The first one introduces a statistical model which is a novel multi-modal approach for bug localization problem. The multi-modal approach is built by utilizing information from both bug reports and program spectra (or program elements) to effectively localize bugs in programs. Different from other multi-modal approaches for bug localization that treat bug reports (or program elements) as independent, my approach

considers similarities between bug reports (or program elements). Hence, similar bugs should have model parameters that are close together. My novel multi-modal approach employs *network Lasso* regularization to incentivize the model parameters of similar bug reports (or program elements) to be close together.

The second one presents a novel deep learning framework to find likely defective code early; the problem is commonly referred to as *Just-In-Time* (JIT) defect prediction. While most existing JIT defect prediction approaches involve a manual feature engineering step, where researchers propose a number of features extracted from commits (e.g., the number of deleted and added lines, number of files, information of authors and code reviewers, etc.), I introduce an end-to-end deep learning framework, namely DeepJIT, which automatically extracts features from commit messages and code changes in the commits, and then uses them to identify defects.

The third one introduces a hierarchical deep learning-based approach, namely PatchNet, to find bug fixing patches in the Linux kernel. Bug fixing patch identification and JIT defect prediction are pretty similar as they take as input the same type of data (i.e., commits to version control systems). While DeepJIT simply merges the removed and added code in the code changes together, PatchNet separates the removed and added code and takes into account the hierarchical structure of the removed and added code.

Finally, the last one presents a neural network model, namely CC2Vec, that learns a representation of code changes based on the semantic information in commit messages. Unlike DeepJIT or PatchNet which only solve a specific software engineering task (i.e., just-in-time defect prediction or bug fixing patch identification), the vector representation represents the semantic meaning of the code changes and can be used to solve a number of software engineering problems related to commits (i.e., just-in-time defect prediction, identification of bug fixing patches, and tangled change prediction, etc.).

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor Prof. David Lo for his support, guidance, and encouragement during my Ph.D. journey. I would also like to thank the Singapore Management University (SMU) Information Systems postgraduate programmes officers who offered me a great opportunity to start my studies. I also want to thank Dr. Julia Lawall for her advice during the last four years. I also want to send a deep thankfulness to Prof. Ee Peng Lim for offering me a chance to be a research engineer at LARC in 2015 and motivating me to pursue postgraduate studies. Many thanks to Prof. Tran Minh Quang for writing a recommendation letter for me.

Next, I would like to express my gratitude to all the committee members of my Ph.D. thesis, i.e., Prof. David Lo, Prof. Hady Wirawan Lauw, Prof. Lingxiao Jiang, and Prof. Julia Lawall for your precious time and effort to read my thesis and provide a lot of helpful feedback.

I am very fortunate to be a part of the Software Analytics Research (SOAR) group at SMU. I would like to thank all the current and former members of our group: Dr. Duy, Dr. Ferdian, Dr. Yuan, Dr. Pavneet, Dr. Bach, Dr. Abhishek, Dr. Agus, Dr. Lingfeng, Dr. Zhiyuan, Dr. Weiqin, Artha, Bowen, and HongJin.

During the last four years in Singapore, I have had an opportunity to meet many great friends, i.e., Dr. Tuan Anh, Dr. Nam, Dr. Dung, Dr. Tuan, Dr. Vu, Minh, Tam, Quang, Hoang, Nghi, Sinh, Jajie, Lin Jan, etc. Please accept my deep gratitude for all the support and motivation.

I would like to express my sincerest gratitude to my family in Vietnam; for my mom and my passed away father, Phuc and Thinh respectively, for my sister Thao who I grew up with and spent an amazing childhood, for my parents-in-law (Tien and Phong) for taking care of my wife and my son. Thank you for your unconditional love, encouragement, kindness, and support. Last but not least, I would also like to give a million thanks to my lovely wife Hai

An and my son Hoang Bach for being in my life. Thank you for supporting me and for the joy in every moment of my life. I love you with all my heart and soul for all eternity.

Hoang Van Duc Thong

May 2020, Singapore

Chapter 1

Introduction

1.1 Motivation

Software engineering corpora, collected from large software systems (i.e., MacOS [190], Ubuntu [74], Firefox [60], etc.), differ from natural language corpora. Specifically, software engineering corpora not only include natural languages used by humans but also contain programming languages used by machines. Software engineering corpora have been heavily studied in the last decade and used to solve many software engineering problems, e.g., tag recommendation [223, 212], duplicate bug report detection [205], profiling Android applications [161], etc.

In this dissertation, I focus on analyzing software engineering corpora to detect bugs in software systems to save developers' time and effort in improving software quality. Specifically, I aim to propose solutions that address three software engineering tasks: bug localization, just-in-time defect prediction, and bug fixing patch identification. Moreover, I introduce a neural network model learning a vector representation of code changes based on their commit messages. The vector representation can be used in addressing various software engineering problems related to code changes, such as just-in-time defect prediction, bug fixing patch identification and more (e.g., tangled change prediction, the recommendation of a code reviewer for a patch, etc.).

For the first problem (i.e., bug localization), two main research approaches (i.e., information retrieval-based and spectrum-based bug localization) have been proposed to tackle it. While information retrieval (IR)-based techniques [182, 191] process textual information in bug reports, spectrum-based techniques [91, 11] process program spectra (i.e., a record of which methods are executed for each test case). The two approaches return a ranked list of methods that most likely contains bugs. However, those techniques fail to leverage the information of bug report similarity and method similarity graphs. Typically, some bug reports (or methods) may be more similar to certain bug reports (or methods) than to others. Hence, similar bugs should have model parameters that are close together.

For the second and third problems (i.e., just-in-time defect prediction and identification of bug fixing patches), a common theme of existing work [156, 99, 103, 112] is manually building a set of features to represent a code change and using them for the training process. Those features are constructed based on properties of code changes, such as the number of removed or added lines, the number of files modified, the number of directories modified, etc. The set of features is used as an input to a machine learning classifier [172] to predict the defectiveness of code changes. However, the metric-based features may not fully capture the semantic and syntactic structure of the actual code changes.

Previous studies have shown the advantages of deeply analyzing syntactic structures of source code in order to uncover semantic information for many software engineering tasks, such as code completion, bug detection, or defect prediction [215, 207, 163, 75]. Unfortunately, there has not been prior work that extracts semantic information from code changes (prior work only considers extracting information from a piece of code). As there are many software engineering problems related to code changes (i.e., just-in-time defect prediction, identification of bug fixing patches, tangled change prediction, etc.), there is also a need to design an approach to extract semantic representation of code changes to improve the performance of automated solutions designed for these problems.

1.2 Contributions

The contributions of work done for this dissertation are as follows:

- **Bug localization:** I propose a new approach, namely Network clustered Multi-modal Bug Localization (NetML), which uses multi-modal information from both bug reports and program spectra to localize bugs [80]. NetML facilitates effective bug localization by carrying out a joint optimization of bug localization error and clustering of both bug reports and program elements (i.e., methods). The clustering is achieved through the incorporation of *network Lasso* regularization [69], which incentivizes the model parameters of similar bug reports and similar program elements to be close together. To estimate the model parameters of both bug reports and methods, NetML employs an adaptive learning procedure based on the Newton method [101] that updates the parameters on a per-feature basis. Extensive experiments on 355 real bugs from seven software systems have been conducted to benchmark NetML against various state-of-the-art localization methods. The results show that NetML surpasses the best-performing baseline by 31.82%, 22.35%, 19.72%, and 19.24%, in terms of the number of bugs successfully localized when a developer inspects the top 1, 5, and 10 methods and Mean Average Precision (MAP), respectively. The paper was published at IEEE Transactions on Software Engineering (IEEE TSE) in 2018.

- **Just-in-time defect prediction:** I propose an end-to-end deep learning framework, named DeepJIT, that automatically extracts features from commit messages and code changes and use them to identify defects [78]. I run the experiments on two popular software projects, namely QT and OPENSTACK. Among 25,150 commits in the QT dataset, there are 2,002 defect commits (8%). In the OPENSTACK dataset, there are 1,616 defect commits (13%) in 12,374 commits. The results show that the best variant of DeepJIT (i.e., DeepJIT-Combined), compared with the best performing state-of-the-art approach, achieves improvements of 10.36-11.02% for the project QT and 9.51-13.69% for the project OPENSTACK in terms of the Area Under the Curve (AUC) on three evaluation settings (i.e., cross-validation, short-period, and long-period). The paper was published at the Mining Software Repositories Conference (MSR) in 2019.

- **Bug fixing patch identification:** I propose a hierarchical deep learning-based approach, named PatchNet, capable of automatically extracting features from commit messages and commit code and using them to identify stable patches [79]. Unlike DeepJIT, PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of commit code, making it distinctive from the existing deep learning models on source code. Experiments on 82,403 recent Linux kernel patches, including 42,408 stable patches and 39,995 non-stable patches, confirm the superiority of PatchNet against various state-of-the-art baselines, including the one recently adopted by Linux kernel maintainers. Specifically, PatchNet achieves a 14.9% higher recall at a high precision level and a 41.2% higher precision at a high recall level compared to the best-performing baseline. The paper was published at IEEE Transactions on Software Engineering (IEEE TSE) in 2019.

- **Code changes representation:** I propose a novel deep learning model, namely CC2Vec, that learns distributed representations of code changes guided by the semantic meaning contained in commit messages. I evaluate the effectiveness of CC2Vec in three software engineering tasks:

1) commit message generation 2) bug fixing patch identification and 3) just-in-time defect prediction. In the first task of commit message generation, CC2Vec can be used to improve over the best baseline by 24.73% in terms of BLEU score (an accuracy measure that is widely used to evaluate machine translation systems). For the task of identifying bug fixing patches, CC2Vec helps to improve the best performing baseline by 5.22%, 9.18%, 4.36%, and 6.51% in terms of accuracy, precision, F1 score, and Area Under the Curve (AUC). For just-in-time defect prediction, CC2Vec helps to improve the AUC metric by 7.03% and 7.72% on the QT and OPENSTACK datasets as compared to the best baseline. The paper was published at the International Conference on Software Engineering (ICSE) in 2020.

1.3 Dissertation Structure

The rest of this dissertation is organized as follows. I first review related work in Chapter 2. Chapter 3 presents my work on bug localization that utilizes multi-modal information from both bug reports and program spectra; this work also takes advantage of the information from bug report similarity and method similarity graphs. Chapter 4 describes my work on just-in-time defect prediction using a deep learning framework. Chapter 5 presents a hierarchical deep learning-based framework for detecting stable patches in the Linux kernel. Chapter 6 introduces a deep learning model used to construct distributed representations of code changes based on commit messages. Finally, I summarize the contributions of this thesis and point to future directions (Chapter 7).

Chapter 2

Related Work

In this section, we highlight several research studies that are closely related to our works for the different software engineering tasks: i.e., bug localization, just-in-time defect prediction, bug fixing patch identification, and code representation.

2.1 Bug Localization

2.1.1 Multi-Modal Feature Location

Multi-modal feature location takes as input a feature description and a program spectra, and finds program elements that implement the corresponding feature. Several multi-modal feature location techniques have been proposed in the literature [175, 53, 137].

Poshyvanyk et al. proposed PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution (LSI [149]) and a spectrum-based solution (Tarantula [91]), and rank program elements based on their corresponding weighted sums [175]. Then, Liu et al. proposed an approach named SITIR that filters program elements returned by an IR-based feature location solution (LSI [149]) if they are not executed in a failing execution trace [137]. Later, Dit et al. used HITS, a popular algorithm that ranks the importance of nodes in a graph, to filter program elements returned by SITIR [53]. They describe several variants and the best performing ones are $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. I refer to these two as DIT^{A} and DIT^{B} in Chapter 3, respectively. They have showed that these variants outperform SITIR, although they have never been compared with PROMESIR.

2.1.2 IR-Based Bug Localization

Various IR-based bug localization approaches that employ information retrieval techniques to calculate the similarity between a bug report and a program element (e.g., a method or a source code file) have been proposed [121, 145, 182, 186, 191, 210, 211, 230, 238].

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization [145]. Then, Rao and Kak evaluated the use of many standard IR methods for bug localization including VSM and Smoothed Unigram Model (SUM) [182]. In the IR community, VSM has a long history, as it was proposed four decades ago by Salton et al. [187]. It has been followed by many other IR methods including SUM and LDA, which address the limitations of VSM.

More recently, a number of approaches that consider information aside from text in bug reports to better locate bugs have been proposed. Sisman and Kak proposed a version history-aware bug localization method that considers past buggy files to predict the likelihood of a file to be buggy and uses this likelihood along with VSM to localize bugs [191]. Around the same time, Zhou et al. [238] proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs. Next, Saha et al. [186] developed an approach that considers the structure of source code files and bug reports and employs structured retrieval for bug localization; this approach outperforms BugLocator. Wang and

Lo proposed an approach that integrates the approaches by Sisman and Kak, Zhou et al. and Saha et al. for more effective bug localization [210]. Most recently, Ye et al. devised an approach named LR that combines multiple ranking features using learning-to-rank to localize bugs. The considered features include surface lexical similarity, API-enriched lexical similarity, collaborative filtering, class name similarity, bug fix recency, and bug fix frequency [230].

2.1.3 Spectrum-Based Bug Localization

Various spectrum-based bug localization approaches have been proposed [11, 20, 21, 44, 91, 133, 136, 140, 144, 141, 234, 235]. These approaches analyze a program spectra which is a record of program elements that are executed in failed and successful executions, and generate a ranked list of program elements. Many of these approaches propose various formulas that can be used to compute the suspiciousness of a program element given the number of times it appears in failing and successful executions.

Jones and Harrold proposed Tarantula that uses a suspiciousness score formula to rank program elements [91]. Later, Abreu et al. proposed another suspiciousness formula called Ochiai [11], which outperforms Tarantula. Then, Lucia et al. investigated 40 different association measures and found that some of them including Klosgen and Information Gain are promising for spectrum-based bug localization [140, 144]. Recently, Xie et al. conducted a theoretical analysis and found that several families of suspiciousness score formulas outperform other families [225]. Next, Yoo proposed to use genetic programming to generate new suspiciousness score formulas that can perform better than many human designed formulas [232]. Subsequently, Xie et al. theoretically compared the performance of the formulas produced by genetic programming and identified the best performing ones [226]. Most recently, Xuan and Monperrus combined 25 different suspiciousness score formulas into a composite formula using their proposed algorithm named MULTRIC, which performs its task by making use of an off-the-shelf learning-to-rank algorithm named RankBoost [227]. MULTRIC has been shown to outperform the best performing formulas studied by Xie et al. [225] and the best performing formula constructed by genetic programming [232, 226].

Wong et al. [222] provided a comprehensive literature review of a large number of spectrum-fault localization techniques, and pointed out avenues for future work. Perez et al. [173] proposed DUU, a new metric for evaluating the diagnosability of a test-suite when applying spectrum-based fault localization approaches. Sohn et al. [194] presented FLUCCs, a fault localization technique that learns to rank program elements based on existing spectrum-fault localization techniques and source code metrics such as age, code churn, and complexity. Li et al. [130] proposed TraPT, another learning-to-rank approach that transforms programs and test outputs/messages in order to localize faults effectively. Pearson et al. [171] highlighted that results found by evaluating spectrum-based and mutation-based fault localization techniques on artificial faults are significantly different than when they are evaluated on real faults. They thus recommended that fault localization techniques should be evaluated using real faults. Moreover, they introduced several new variants of a mutation-based fault localization technique that also use coverage information (in addition to mutation information).

2.1.4 Other Related Studies

There are many studies that compose multiple methods together to achieve better performance. For example, Kocaguneli et al. [109] combined several single software effort estimation models to create more powerful multi-modal ensembles. Also, Rahman et al. [177] used static bug-finding to improve the performance of statistical defect prediction (and vice versa). Le et al. [119] proposed SpecForge that combines different automaton based specification miners using model fission and model fusion in order to create a more effective specification miner. Kellogg et al. [102] presents *N-Prog* that combines static bug detection and test case generation to avoid unnecessary human effort. In particular, *N-Prog* produces no false alarms, by construction, since its output alarm is either a new test case or a bug in a program.

Different from the previous studies assuming that bug reports (or program elements) are independent, my approach (NetML) takes advantage of the relationship between similar bug reports (or program elements). Specifically, NetML employs a *network Lasso* regularization to incentivize the model parameters of similar bug reports (or program elements) to be close together, the model parameters of both bug reports (or program elements) are then used to effectively localize bugs in software.

2.2 Just-in-time Defect Prediction

Some previous studies focus on change-level defect prediction (i.e. JIT defect prediction). For example, Mockus and Weiss [156] predict whether commits are buggy in an industrial project. They use metric-based features, such as the number of subsystems touched, the number of files modified, the number of lines of code added, and the number of modification requests. Motivated by their previous work, Kamei et al. [99] built upon the set of code change features, reporting that the addition of a variety of features that were extracted from the Version Control System (VCS) and the Issue Tracking System (ITS) helped to improve the prediction accuracy. They conduct an empirical study of the effectiveness of JIT defect prediction on a set of six open source and five commercial projects and also evaluate their findings by considering the effort required to review the changes.

Aversano *et al.* [22] and Kim *et al.* [103] used source code change logs to predict whether commits are buggy. For example, Kim *et al.* [103] used the identifiers in added and deleted source code and the words in change logs. The experimental results on the dataset collected from 12 open source software projects show that the proposed approach achieved 78 percent accuracy and a 60 percent recall.

Kononenko et al. [112] find that the addition of code change features that were extracted from code review databases contributed a significant amount of explanatory power to JIT models. McIntosh and Kamei also used 5 families of code and review features in the context of JIT defect prediction. Through a case study of 37,524 changes from QT and OpenStack systems, the paper shows that the importance of impactful families of code change features are consistently under or overestimated in the studied systems.

Deep learning has recently attracted increasing interest in software defect prediction. Deep Belief Network (DBN) [76] has been commonly used in previous work. For example, a recent work [228] used the Deep Belief Network to build JIT defect prediction models. Their approach still however relies on the same set of metric-based features that are manually engineered as in earlier work. Other studies (e.g., [215, 214]) also used Deep Belief Network to automatically learn features for defect prediction. Unlike our approach, their models are not end-to-end trainable, i.e., features are learned separately (not using the defect ground-truths) and are then input to a separate traditional classifier. This approach has also been used in previous work (e.g. [50, 129]) where two other well-known deep learning architectures (Long Short Term Memory in [50] and Convolutional Neural Network in [129]) were leveraged to automatic feature learning for defect prediction. There is a risk in those approaches that the learned features may not correlate with defect outcomes.

To address this issue, I propose an end-to-end deep learning framework, named DeepJIT, that automatically extracts features from commit messages and code changes. These features are then put into a fully connected layer to train a model to predict whether a given commit is buggy. Extensive experiments show DeepJIT outperforms the best performing state-of-the-art approach.

2.3 Bug Fixing Patches

Previous studies identify bug fixing commits based on keywords, such as “bug” and “fix” in commit messages [49, 104, 155, 192]. Bird et al. [29] showed that the previous studies lack understanding of the bug fixing patches themselves, which has caused potential bias in detecting bug fixing patches. Tian et al. [204] proposed a method (namely LPU+SVM) to automatically identify bug fixing patches by combining LPU (Learning from Positive and Unlabeled Examples) [127] and SVM (Support Vector Machine) [196]. Unlike those previous approaches, which are strongly based on keywords, LPU+SVM relies on thousands of word features extracted from commit messages and 52 features manually extracted from code changes.

Unlike the existing approaches which manually extract features from commits, I propose a hierarchical deep learning-based approach (PatchNet), which automatically extracts features from commit messages and code changes in the commits to find bug fixing patches in the Linux kernel. Different from DeepJIT which simply merges the removed and added code in the code changes together, PatchNet separates the removed and added code and takes into account the hierarchical structure of the removed and added code.

2.4 Code Representation

There are many studies on the representation of source code, including recent studies proposing distributed representations for identifiers [57], APIs [165, 166], and software libraries [201]. A comprehensive survey of learning the representation of source code has been done by Allamanis et al. [13].

Some studies transform the source code into a different form, such as control-flow graphs [52] and symbolic traces [73], or collect runtime execution traces [209], before learning distributed representations. DeFreez et al. [52] found function synonyms by learning embeddings through random walks of the interprocedural control-flow graph of a program. These embeddings are then used in a single downstream task of mining error-handling specifications. Henkel et al. [73] described a toolchain to produce abstracted intraprocedural symbolic traces for learning word embeddings. They experimented on a downstream task to find and repair bugs related to incorrect error codes. Wang et al. [209] used execution traces to learn embeddings. They integrate their embeddings into a program repair system in order to produce fixes to correct student errors in programming assignments. These studies differ from our work as we leverage natural language data as well as source code.

There have been other studies using deep learning of both source code and natural language data, for example, joint learning of embeddings for both text and source code to improve code search [67]. Other studies proposed approaches to learn distributed representations of source code on prediction tasks with natural language output. Iyer et al. [85] proposed a model using LSTM networks with attention for code summarization, and Yin et al. [231] trained a model to align source code to natural language text from StackOverflow posts. However, unlike our work, these studies do not use structural information of the source code.

Several studies [82, 15, 14, 114] account for structural information but differ from our work. Hu et al. [82] proposed an approach to use Sequence-to-Sequence Neural Machine Translation to generate method-level code comments. By prefixing the AST node type in each token and traversing the AST of methods such that the original AST can be unambiguously reconstructed, they convert the AST of each method into a sequence that preserves structural information. Alon et al. proposed code2vec [15], which represents code as paths in an AST, learning the vector representation of each AST path. They trained their model on the task of predicting a label, such as the method name, of the code snippet. In a later work, they proposed code2seq [14]. Instead of predicting a single label, they generate a sequence of natural language words. Similar to our work, structural information of the input source code is encoded in the model's architecture, however, in these studies, the input code snippet is required to be parseable to build an AST.

As our work focuses on the representation of software patches, we deliberately designed CC2Vec to not require parseable code in its input. This is done for two reasons. Firstly, a small but still significant proportion of patches may have compilation errors. A study by Beller et al. on Travis CI build failures revealed that about 4% of Java project build failures are due to compilation errors [27]. CC2Vec is designed to be usable even for these patches. Secondly, parsing will require the entire file with the changed code. Retrieving this information and parsing the entire file will be time consuming. Furthermore, all the studies above proposed general representations of source code. The representations they learn, with the exception of DeFreez et al. [52], are of source code contained in a single function. In contrast, we learn representations of code changes, which can contain modifications to multiple different functions, across multiple files.

Chapter 3

Network-Clustered Multi-Modal Bug Localization

Developers often spend much effort and resources to debug a program. To help the developers debug, numerous information retrieval (IR)-based and spectrum-based bug localization techniques have been devised. IR-based techniques process textual information in bug reports, while spectrum-based techniques process program spectra (i.e., a record of which program elements are executed for each test case). While both techniques ultimately generate a ranked list of program elements that likely contain a bug, they only consider one source of information—either bug reports or program spectra—which is not optimal. In light of this deficiency, this chapter presents a new approach dubbed Network-clustered Multi-modal Bug Localization (NetML), which utilizes multi-modal information from both bug reports and program spectra to localize bugs. NetML facilitates an effective bug localization by carrying out a joint optimization of bug localization error and clustering of both bug reports and program elements (i.e., methods). The clustering is achieved through the incorporation of *network Lasso* regularization [69], which incentivizes the model parameters of similar bug reports and similar program elements to be close together. To estimate the model parameters of both bug reports and methods, NetML employs an adaptive learning procedure based on the Newton method [101] that updates the parameters on a per-feature basis. Extensive experiments on 355 real bugs from seven software systems have been conducted to benchmark NetML against various state-of-the-art localization methods. The results show that NetML surpasses the best-performing baseline by 31.82%, 22.35%, 19.72%, and 19.24%, in terms of the number of bugs successfully localized when a developer inspects the top 1, 5, and 10 methods and Mean Average Precision (MAP), respectively.

3.1 Introduction

Debugging software programs, which often come in high volume [18], has proved to be a difficult task that takes any resources and much time [200]. Various techniques have been devised to help developers locate buggy program elements from their symptoms. These symptoms could be in the form of description of a bug experienced by a user, or a failing test case. These techniques—often collectively referred to as bug (or fault) localization—analyze the symptoms of a bug and produce a list of program elements ranked based on their likelihood to contain the bug. In general, a program element can be defined at three levels of granularity, i.e., source file level, method/function level, and line of code level.

3.1.1 The Need for Multi-modal Bug Localization

Existing bug localization techniques broadly fall into two major categories: *information retrieval* (IR)-based techniques [182, 191, 238, 186], and *spectrum*-based bug localization techniques [91, 11, 185, 235, 234, 44, 132, 133, 136]. The IR-based bug localization techniques typically analyze textual descriptions contained in bug reports and identifier names and comments in source code files. They then return a ranked list of program elements (typically program files) that are the most similar to the bug textual description. The spectrum-based bug localization techniques typically analyze program spectra that correspond to program elements that are executed by failing and successful execution traces. Likewise, they return a ranked list of program elements (typically program blocks or statements) that are executed more often in the failing traces than in the correct traces.

The above-mentioned approaches, however, only consider one kind of symptom or one source of information, i.e., only bug reports or only execution traces. This is a limiting factor since hints of the location of a bug may be spread in both the bug report and the execution traces; and some hints may only appear in one but not the other. In this work, we put forward a bug localization approach that addresses the deficiency of existing methods by jointly utilizing both bug reports and execution traces. We refer to this approach as *multi-modal bug localization*, as we consider multiple modes of inputs (i.e., bug reports and program spectra). Such an approach fits well with developers' debugging activities as illustrated by the following scenarios:

1. Developer D is working on a bug report that was submitted to Bugzilla. One of the first tasks that he needs to do is to replicate the bug based on the description in the report. If the bug can be successfully replicated, he will proceed to the debugging step; otherwise, he will mark the bug report as "WORKSFORME" and will not continue further [109]. After D replicates the bug, he has one or a few failing execution traces. He also has a set of regression tests that he can run to get successful execution traces. Thus, after the replication process, D has *both* the textual description of the bug and program spectra that characterize the bug. With this, D can proceed to use multi-modal bug localization.

2. Developer D runs a regression test suite and some test cases fail. Based on his experience, D has some idea why the test cases fail. D can create a textual document describing the bug. At the end of this step, D has *both*

program spectra and a textual bug description, and can proceed to use multi-modal bug localization which will leverage not only the program spectra but also D's domain knowledge to locate the bug.

It is worth noting that our work focuses on localizing a bug to the *method* that contains it. Historically, most IR-based bug localization techniques aim at finding buggy files [182, 191, 238, 186], while most spectrum-based techniques find buggy lines [91, 11, 185]. Localization at the method level can be a good tradeoff. That is, a method is not as big as a file, but it often contains sufficient context needed to help developers understand a bug. On the other hand, by just looking at a line of code, developers often cannot determine whether it is the location of the bug or understand the bug well enough to fix it [170]. Admittedly, if the methods are long, a finer granularity (e.g., basic blocks) may be preferred. Nevertheless, a recent study by Kochhar et al. [110] highlights that out of the 386 practitioners they surveyed, the majority indicates method-level as the preferred granularity.

In this chapter, we present a new approach called the Network-clustered Multi-modal Bug Localization (NetML), which works based on three main intuitions:

1. Firstly, it is recognized that a large variety of bugs exist, and different bugs need different treatments [203, 224]. A bug report written by a developer provides a unique description of a bug. Thus, different bugs require separate model parameters to capture their individual characteristics. Similarly, different program elements (or methods in this work) are of different nature, and should be characterized by separate model parameters.

2. A recent study by Parnin and Orso [170] also showed that some words are more useful in localizing bugs, and suggested that "future research could also investigate ways to automatically suggest or highlight terms that

might be related to a failure". Our NetML provides such a capability by incorporating a *method suspiciousness* feature, which allows us to automatically highlight suspicious terms and use them to localize bugs.

3. We also observe that bugs and program elements are *not* completely independent, and some bug reports (or methods) may be more similar to certain other bug reports (or methods) than to others. As such, similar bugs (or methods) should have model parameters that are close together. This enforcement of clustering of model parameters would enable similar bug reports (or methods) to share information and reinforce one another.

The first two intuitions have been captured in our recent work—dubbed Adaptive Multi-Modal Bug Localization (AML) [120]—which we extend in this chapter. In particular, AML already incorporates the ideas of adaptively computing separate model parameters for each bug report, and of computing the method suspiciousness feature.[1] However, AML exhibits two main shortcomings. Firstly, AML only has the concept of model parameters for bug reports, but not for program elements (or methods). As such, it is not able to capture variation in the inherent characteristics of different program elements (methods), which may limit its effectiveness in localizing a bug. Secondly, the model parameters of each bug report are learned independently of those of other bug reports. As a result, AML is unable to take advantage of the clustering/similarity traits of different bug reports in the localization process.

The proposed NetML method addresses these shortcomings by performing joint optimization of the localization loss function and clustering of both bug reports and methods. Specifically, it generalizes AML in two important ways:

1. NetML provides a richer model that has two sets of (model) parameters—one for bug reports and the other for methods. The addition of the

[1] To understand the concept of feature and model parameters, we can draw an analogy to a linear model $y = \sum_i w_i x_i$. A feature refers to the (independent) input variable x_i, while a model parameter refers to the weight coefficient w_i for each feature x_i. In this case, the model parameters w_i need to be learned/estimated from data.

Bug 30798	**Bug 43969**
Summary: JUnit shows output implementation grabs System.out and System.err later than it should. Description: **What steps will reproduce the problem?** JUnitTestRunner creates the junit.framework.Test instance before grabbing System.out and System.err. As a result, anything printed to System.out or System.err in the constructor . . .	Summary: JUnit4 tests marked @Ignore do not appear in XML output Description: **What steps will reproduce the problem?** Run a JUnit 4 test marked with the @Ignore annotation. The test will not appear at all in the XML output.

Figure 3.1: Example of two bug reports which have the same faulty method in project Apache-Ant [1]. The colored text indicates some common word tokens that these two bugs share.

method parameters (in contrast to AML that has only bug report parameters) provides NetML with a higher degree of freedom to characterize the different variety of bug reports and methods more accurately.

2. NetML incorporates network Lasso regularization [69] into its parameter learning procedure, which forces similar bug reports (and methods) to have similar (or even identical) model parameters. This clustering enforcement would allow similar bug reports (or methods) to reach a consensus on the model parameters, leading to a simpler "policy" for bug localization. This enables the models of bug reports (or methods) to complement and borrow strength from one another. In turn, this would improve robustness and the generalization of the performance to new/unseen bug reports.

It is noteworthy that, deviating from the conventional network Lasso [69] which deals with only a single network (graph), we impose regularization over two networks, i.e., bug report similarity and method similarity graphs. This allows us to achieve simultaneous clustering of both bug reports and methods, and exploits their similarity traits so as to achieve a more effective bug localization.

To illustrate how the network Lasso regularization in NetML can benefit bug localization, Fig. 3.1 shows two bug reports from Apache-Ant [1] project, namely Bug 30798 and Bug 43969. These two bug reports describe issues with the "showoutput" option for Apache Ant's JUnit task and the corresponding bugs both reside in the `run` method of the `JUnitTestRunner.java` file. Bug 30798 mentions the names of a few source code files and one of them is the name of the buggy file (i.e., `JUnitTestRunner`), while no such hint is included in Bug 43969. AML manages to successfully localize the buggy method for Bug 30798, by ranking it high in the returned ranked list. However, due to the limited information in Bug 43969, it is not able to do the same for it. Upon a closer investigation, we can see that Bug 30798 and Bug 43969 are similar, since they share a number of common word tokens (i.e., "JUnit", "text", "output", etc.). The network Lasso regularization is able to take advantage of this similarity by enforcing similar bug reports to have similar model parameters. In such way, NetML leverages the similarity of Bug 30798 and Bug 43969 to guide/reinforce the prediction for Bug 43969, which leads to successful localization for both bugs.

3.1.2 Contributions

To evaluate the efficacy of the NetML approach, we conducted experiments using a dataset of 355 real bugs from seven medium to large software systems: Ant, AspectJ, Lang, Lucene, Math, Rhino, and Time. All real bug reports and real test cases were collected from these systems. The test cases were run to generate program spectra. We compare NetML with our previous AML method. Additionally, we evaluate our approach against a wide range of state-of-the-art approaches, including two multi-modal feature localization techniques (i.e., PROMESIR [175], DIT$^{\mathrm{A}}$ and DIT$^{\mathrm{B}}$ [53]), four spectrum-based bug localization techniques ([12, 221, 227, 23]), and an IR-based bug localization technique (i.e., LR^{A} and LR^{B} [230]). We use two well-known evaluation metrics to estimate the performance of our approach: number of bugs localized by inspecting the top N program elements (Top N) and mean average precision (MAP). Top N and MAP have been widely used in past bug localization studies, e.g., [182, 191, 238, 186].

Our results demonstrate that, among the 355 bugs, NetML can successfully localize 116, 219, and 255 bugs when developers only inspect the Top 1, Top 5, and Top 10 methods in the lists that NetML produces, respectively. These constitute 31.82%, 22.35%, 19.72%, and 19.24% improvements over AML (which is the second best method in our benchmark), in terms of Top 1, Top 5 , Top 10, and MAP results respectively.

We summarize the key contributions of this chapter below:

1. We present a novel multi-modal bug localization method that adaptively learns two sets of model parameters that characterize each bug report and method, respectively. We are also the first to incorporate the network Lasso regularization on both bug report and method similarity networks, which facilitates an effective joint optimization of bug localization quality and clustering of both bug reports and methods.

2. We develop an adaptive learning procedure based on the Newton method [101] to jointly update the model parameters of bug reports and methods on a per-feature basis. The procedure is based on the formulation of a strict convex loss function, which provides a theoretical guarantee that any minimum found will be globally optimal.

3. We have extensively evaluated NetML on a dataset of 355 real bugs from seven software systems using real bug reports and test cases. Our statistical significance tests reveal that NetML improves upon state-of-the-art bug localization approaches by a substantial margin.

3.2 Background

In this section, we present some background material on IR-based and spectrum-based bug localization.

3.2.1 IR-Based Bug Localization

IR-based bug localization techniques consider an input bug report (i.e., the text in the summary and description of the bug report as a query, and program elements in a code base as documents, and employ IR techniques to sort the program elements based on their relevance to the bug report. The intuition behind these techniques is that program elements sharing many common words with the input bug report are likely to be relevant to the bug. By using text retrieval models, IR-based bug localization computes the similarities between various program elements and the input bug report. Then, program elements are sorted in descending order of their textual similarities to the bug report, and sent to developers for manual inspection.

All IR-based bug localization techniques need to extract textual content from source code files and preprocess textual content (either from bug reports or source code files). First, comments and identifier names are extracted from source code files. These can be extracted by employing a simple parser. In this work, we use JDT [7] to recover the comments and identifier names from source code. Next, after the textual contents from source code and bug reports are obtained, we need to preprocess them. The purpose of text preprocessing is to standardize words in source code and bug reports. There are three main steps: text normalization, stopword removal, and stemming:

1. Text normalization breaks an identifier into its constituent words (tokens), following camel casing convention. Following the work by Saha et al. [186], we also keep the original identifier names.

2. Stopword removal removes punctuation marks, special symbols, number literals, and common stopwords [9]. It also removes programming keywords such as *if*, *for*, *while*, etc., which usually appear too frequently to be useful to differentiate between documents.

3. Stemming simplifies English words into their root forms. For example,

”processed“, ”processing“, and ”processes“ are all simplified to ”process“. This increases the chance that the query and document share some common words. We use the popular Porter Stemming algorithm [92].

Numerous IR techniques have been employed for bug localization. We highlight a popular IR technique namely *Vector Space Model* (VSM). In VSM, queries and documents are represented as vectors of weights, where each weight corresponds to a term. The value of each weight is usually the *term frequency—inverse document frequency* (TF-IDF) [181] of the corresponding word. Term frequency refers to the number of times a word appears in a document. Inverse document frequency refers to the number of documents in a corpus (i.e., a collection of documents) that contain the word. The higher the term frequency and inverse document frequency of a word, the more important the word would be. In this work, given a document d and a corpus C, we compute the TF-IDF weight of a word w as follows:

$$\begin{aligned} weight(w, d) &= \text{TF-IDF}(w, d, C) \\ &= \log(f(w, d) + 1) \times \log \frac{|C|}{|d_i \in C : w \in d_i|} \end{aligned}$$

where $f(w, d)$ is the number of times w appears in d.

After computing a vector of weights for the query and each document in the corpus, we calculate the cosine similarity of the query and document vectors. The cosine similarity between query q and document d is given by:

$$sim(q, d) = \frac{\sum\limits_{w \in (q \bigcap d)} weight(w, q) \times weight(w, d)}{\sqrt{\sum\limits_{w \in q} weight(w, q)^2} \times \sqrt{\sum\limits_{w \in d} weight(w, d)^2}} \tag{3.1}$$

where $w \in (q \bigcap d)$ means word w appears both in the query q and document d. Also, $weight(w, q)$ refers to the weight of word w in the query q's vector. Similarly, $weight(w, d)$ refers to the weight of word w in the document d's vector.

Table 3.1: Raw Statistics for Program Element e.

	e is *executed*	e is *not executed*
unsuccessful test	$n_f(e)$	$n_f(\bar{e})$
successful test	$n_s(e)$	$n_s(\bar{e})$

Table 3.2: Raw Statistic Description.

Notation	**Description**
$n_f(e,p)$	Number of unsuccessful test cases executing program element e in program spectra p
$n_f(\bar{e},p)$	Number of unsuccessful test cases that do not execute program element e in program spectra p
$n_s(e,p)$	Number of successful test cases that execute program element e in program spectra p
$n_s(\bar{e},p)$	Number of successful test cases that do not execute program element e in program spectra p
$n_f(p)$	Total number of unsuccessful test cases
$n_s(p)$	Total number of successful test cases

3.2.2 Spectrum-Based Bug Localization

Spectrum-based bug localization (SBBL)—also known as spectrum-based fault localization (SBFL)—takes as input a faulty program and two sets of test cases. One is a set of failed test cases, and the other one is a set of passed test cases. SBBL then instruments the target program, and records program spectra that are collected when the set of failed and passed test cases are run on the instrumented program. Each of the collected program spectrum contains information about the program elements that are executed by a test case. Various tools can be used to collect program spectra as a set of test cases are run. In this work, we use Cobertura [6].

Based on these spectra, SBBL typically computes some raw statistics for every program element. Tables 3.1 and 3.2 summarize some raw statistics that can be computed for a program element e, given program spectra p. These statistics are the counts of unsuccessful (i.e., failed), and successful (i.e., passed) test cases that execute or do not execute e. If a successful test case executes program element e, then we increase $n_s(e,p)$ by one unit. Similarly, if an unsuccessful test case executes program element e, then we increase $n_f(e,p)$ by one unit. SBBL uses these statistics to calculate the suspiciousness scores of each program element. The higher the suspiciousness score, the more likely the corresponding program element is the faulty element. After the suspiciousness scores of all program elements are computed, program elements are then sorted in descending order of their suspiciousness scores, and sent to developers for manual inspection.

Different SBBL techniques have used different formulas to calculate the suspiciousness scores. Among these techniques, Tarantula is a popular one [91]. Using the notation in Table 3.2, the following is the formula that Tarantula uses to compute the suspiciousness score of program element e, given program spectra p:

$$Tarantula(e,p) = \frac{\frac{n_f(e,p)}{n_f(p)}}{\frac{n_f(e,p)}{n_f(p)} + \frac{n_s(e,p)}{n_s(p)}} \tag{3.2}$$

The main idea of Tarantula is that program elements that are executed by failed test cases are more likely to be faulty than those that are not executed. Thus, Tarantula assigns a non-zero score to program element e that has $n_f(e,p) > 0$.

3.3 Proposed Framework

An overview of our NetML framework is given in Fig. 3.2 (enclosed in the dashed box). NetML takes as input a new bug report, the program spectra corresponding to it, and a method corpus. It also takes as input *historical* bug reports that have been localized before. For each *historical* bug report, we have its corresponding program spectra and ground truth labels. If a method contains a root cause of the bug, it is labeled as *faulty*, otherwise it is labeled as *non-faulty*. Given these inputs, NetML eventually produces a list of methods, ranked based on their likelihood to contain the root cause of the new bug

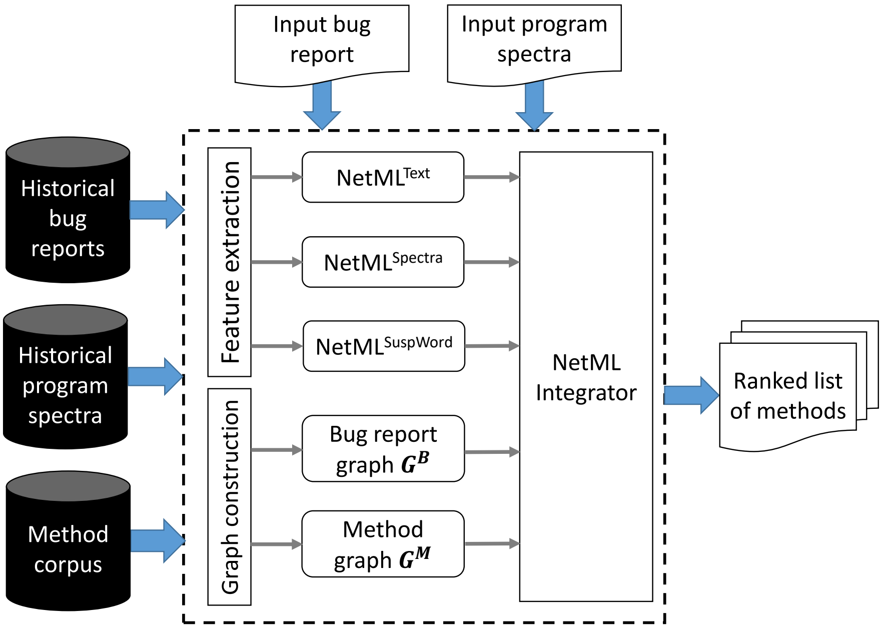


Figure 3.2: The proposed NetML framework.

report.

NetML has three main components, namely: *feature extraction*, *graph construction*, and *integrator*. The feature extraction component serves to extract multi-modal input features that quantify different perspectives on the degree of relevancy between a bug report and a method [120]. Meanwhile, the graph construction component computes the similarity graphs among the bug reports ($\mathcal{G}_B$) and methods ($\mathcal{G}_M$).

Finally, the integrator component is the heart of NetML and constitutes the primary contribution of this work. It integrates both the input features and the similarity graph information in order to produce a ranked list of methods based on their relevancy score. In particular, the integrator performs adaptive learning that aims at jointly minimizing the bug localization errors and fostering clustering of the model parameters of similar bug reports and/or methods.

In Sections 3.3.1 and 3.3.2, we first elaborate the feature extraction and graph construction components respectively. We then describe the NetML integrator component in greater detail in Sections 3.3.3–3.3.5, including the formulation of our new integrator model as well as the corresponding objective function and adaptive learning procedure.

3.3.1 Feature Extraction

The first component of the NetML framework is the feature extraction module, which generates features $\mathbf{X} = \{x_{b,m.j}\}$ to be fed as inputs to the NetML integrator (see Fig. 3.2). In line with our earlier AML work [120], for each bug report–method pair (b, m), we compute a feature vector $\vec{x}_{b,m}$ that consists of three elements:

$$\vec{x}_{b,m} = \left[\text{NetML}^{\text{Text}}_{b,m}, \text{NetML}^{\text{Spectra}}_{b,m}, \text{NetML}^{\text{SuspWord}}_{b,m}\right] \tag{3.3}$$

The three features are elaborated in turn below.

$\text{NetML}^{\text{Text}}_{b,m}$ makes use of the TF-IDF method [181] to estimate the similarity between methods and bug reports. In particular, given a method m and a bug report b, $\text{NetML}^{\text{Text}}_{b,m}$ computes the cosine similarity between the TF-IDF representation of the bug report text and that of the method code, which is akin to the IR-based bug localization method (cf. Section 3.2.1). That is, $\text{NetML}^{\text{Text}}_{b,m}$ is given by:

$$\text{NetML}^{\text{Text}}_{b,m} = sim(b, m) \tag{3.4}$$

where $sim(b, m)$ is the cosine similarity as defined in (3.1).

$\text{NetML}^{\text{Spectra}}_{b,m}$ processes only the program spectra information using the spectrum-based bug localization technique described in Section 3.2.2. Given program spectra p corresponding to bug report b and a method m, $\text{NetML}^{\text{Spectra}}_{b,m}$ gives a score that quantifies how suspicious m is given p. By default, $\text{NetML}^{\text{Spectra}}_{b,m}$ uses the Tarantula formula as described in Section 3.2.2 (cf. equation (3.2)):

$$\text{NetML}^{\text{Spectra}}_{b,m} = Tarantula(m, p) \tag{3.5}$$

Finally, $\text{NetML}_{b,m}^{\text{SuspWord}}$ processes both bug reports and program spectra, and computes the suspiciousness scores of words to rank different methods. It breaks a method into its constituent words, computes the suspiciousness scores of these words, and then aggregates these scores in order to arrive at the suspiciousness score of the method. $\text{NetML}_{b,m}^{\text{SuspWord}}$ aims to integrate both macro view of method suspiciousness (which considers direct execution of a method in the failing and correct execution traces) and micro view of method suspiciousness (which considers the executions of its constituent words in the execution traces) [120]. Given a bug report b, a program spectra p, and a method m in a corpus C, $\text{NetML}_{b,m}^{\text{SuspWord}}$ measures how suspicious m is considering b and p, as follows:

$$\begin{aligned}\text{NetML}_{b,m}^{\text{SuspWord}} &= \text{NetML}_{b,m}^{\text{Spectra}} \times \\ &\left(\frac{\sum_{w \in b \cap m} \text{SSTFIDF}(w,p,b,C) \times \text{SSTFIDF}(w,p,m,C)}{\sqrt{\sum_{w \in b} \text{SSTFIDF}(w,p,b,C)^2} \times \sqrt{\sum_{w \in m} \text{SSTFIDF}(w,p,m,C)^2}} \right)\end{aligned} \tag{3.6}$$

where $\text{SSTFIDF}(w,p,b,C)$ is the weight of a word w in document (i.e., bug report or method) d with corpus C given program spectra p:

$$\begin{aligned}\text{SSTFIDF}(w,p,d,C) = &\ \text{SS}_{\text{word}}(w,p) \times \ln(f(w,d)+1) \\ &\times \ln \frac{|C|}{|d_i \in C : w \in d_i|}\end{aligned} \tag{3.7}$$

where $\text{SS}_{\text{word}}(w,p)$ is the suspiciousness score of a word w:

$$\text{SS}_{\text{word}}(w,p) = \frac{\frac{|EF(w,p)|}{|p.FAIL|}}{\frac{|EF(w,p)|}{|p.FAIL|} + \frac{|ES(w,p)|}{|p.SUCCESS|}} \tag{3.8}$$

In the above equation, $EF(w,p)$ is the set of execution traces in $p.FAIL$ that contain a method in which the word w appears, while $ES(w,p)$ is the set of execution traces in $p.SUCCESS$ that contain a method in which the word w appears.

3.3.2 Graph Construction

The second component of the NetML framework is the graph construction module, which serves to compute the similarity graphs among bug reports and methods, to be used in the K-nearest neighbor retrieval as well as the network Lasso regularization. In this work, we define the bug report similarity graph $\mathcal{G}_B$ as comprising edge weights that reflect the textual similarity between two bug reports. For a pair of bug reports b and b', we define the edge weight $e_{b.b'}$ as follows:

$$e_{b,b'} = sim(b, b') \tag{3.9}$$

where $sim(b, b')$ is the cosine similarity between the TF-IDF weights of the textual descriptions of b and b', as per (3.1).

Similarly, the method similarity graph $\mathcal{G}_M$ comprises a set of edge weights $e_{m,m'}$ that reflect the textual similarity between two methods m and m'. This is given by:

$$e_{m,m'} = sim(m, m') \tag{3.10}$$

where $sim(b, b')$ is the cosine similarity between the TF-IDF representations of the source code of m and m'.

3.3.3 Integrator Model

The new integrator model proposed in this work characterizes the relevancy of a method m to a given bug report b as an interaction between two types of model parameters, namely: *bug report parameters* $\vec{u}_b = [u_{b,1}, \ldots, u_{b,j}, \ldots, u_{b,J}]$ and *method parameters* $\vec{v}_m = [v_{m,1}, \ldots, v_{m,j}, \ldots, v_{m,J}]$, where J is the total number of features. Note that $J = 3$ in this case, i.e., $\text{NetML}_{b,m}^{\text{Text}}$, $\text{NetML}_{b,m}^{\text{Spectra}}$, $\text{NetML}_{b,m}^{\text{SuspWord}}$. More specifically, the integrator model computes the relevancy score $\hat{f}_{b,m}$ as follows:

$$\hat{f}_{b,m} = \hat{f}(\vec{x}_{b,m}, \vec{u}_b, \vec{v}_m) = \sum_{j=1}^{J} (u_{b,j} + v_{m,j}) x_{b,m,j} \tag{3.11}$$

where $\vec{x}_{b,m} = [x_{b,m,1}, \ldots, x_{b,m,j}, \ldots, x_{b,m,J}]$ is the feature vector corresponding to a bug report–method pair (b, m).

It is worth mentioning that the above model constitutes a generalization of the AML integrator model that we previously developed [120]. In AML, the final relevancy score is computed based solely on the bug report parameters, and this set of parameters is shared by all methods for a given bug report. On the other hand, the NetML integrator model accounts for not only the bug report parameters but also the method parameters. The addition of the latter parameters provides a greater degree of freedom and flexibility in quantifying the contribution of different methods to the localization of a given bug report.

3.3.4 Objective Function

Based on the above model formulation, we devise an objective function that guides the learning process of our integrator model. Specifically, we consider a joint optimization of bug localization quality and clustering of similar bug reports and methods, expressed by the loss function $\mathcal{L}$:

$$\mathcal{L} = \mathcal{L}_{\text{Entropy}} + \mathcal{L}_{\text{Ridge}} + \mathcal{L}_{\text{NetLasso}} \tag{3.12}$$

The loss function $\mathcal{L}$, used to learn the model's parameters, consists three parts: $\mathcal{L}_{\text{Entropy}}$ is used to measure the difference between the predicted and the true label, $\mathcal{L}_{\text{Ridge}}$ is regularization function to avoid the overfitting during the training process, and $\mathcal{L}_{\text{NetLasso}}$ is employed to force similar bugs and methods to have similar parameters in latent space. These three components are presented as follows:

$$\begin{aligned}
\mathcal{L}_{\text{Entropy}} =& -\sum_{b\in\mathcal{B}}\sum_{m\in\mathcal{M}} w_{b,m}\Big[y_{b,m}\ln(\sigma(\hat{f}_{b,m})) \\
&+ (1-y_{b,m})\ln(1-\sigma(\hat{f}_{b,m}))\Big] && (3.13)\\
\mathcal{L}_{\text{Ridge}} =& \frac{\alpha}{2}\sum_{j=1}^{J}\left[\sum_{b\in\mathcal{B}} u_{b,j}^2 + \sum_{m\in\mathcal{M}} v_{m,j}^2\right] && (3.14)\\
\mathcal{L}_{\text{NetLasso}} =& \frac{\beta}{2}\sum_{j=1}^{J}\Bigg[\sum_{(b,b')\in\mathcal{G}^B} e_{b,b'}(u_{b,j}-u_{b',j})^2 \\
&+ \sum_{(m,m')\in\mathcal{G}^M} e_{m,m'}(u_{m,j}-u_{m',j})^2\Bigg] && (3.15)
\end{aligned}$$

where $\mathcal{B}$ and $\mathcal{M}$ are the sets of bug reports and methods respectively, $y_{b,m}$ is a binary label that indicates whether method m is relevant to bug report b ($y_{b,m} = 1$) or not ($y_{b,m} = 0$), and $\sigma(\hat{f}_{b,m}) = \frac{1}{1+\exp(-\hat{f}_{b,m})}$ is the logistic function [45]. Also, $w_{b,m}$ denotes the instance weight of a bug report–method pair (b, m), while $e_{b,b'}$ and $e_{m,m'}$ are the edge weights reflecting the degree of similarity between two bug reports b and b', and two methods m and m', respectively. Finally, $\alpha > 0$ and $\beta > 0$ are the user-defined parameters that control the strength of the ridge and network Lasso regularization, respectively.

Note that $\mathcal{L}_{\text{Entropy}}$ refers to the so-called cross-entropy loss [158], which provides an error measure of the bug localization process. Here $\mathcal{L}_{\text{Entropy}}$ can be interpreted as the discrepancy between the probability distribution of the predictive model $\hat{f}_{b,m}$ and that of the true label $y_{b,m}$ [158]. We also introduce the instance weight[2] $w_{b,m}$ in (3.13) to cater for the extremely *skewed* distribution of the relevant vs. irrelevant methods for a given bug report, which is a major challenge in the bug localization process. That is, the number of relevant (faulty) methods is much smaller than that of irrelevant (non-faulty) ones. To address this, we configure $w_{b,m}$ in such a way that it imposes a greater

[2] An instance refers to a specific bug report–method pair (b, m)

penalty for relevant instances being incorrectly predicted/classified than that for irrelevant ones. Specifically, we set $w_{b,m}$ as:

$$w_{b,m} = \begin{cases} \frac{1}{N_{\text{faulty}}}, & \text{if } y_{b,m} = 1 \\ \frac{1}{N - N_{\text{non-faulty}}}, & \text{if } y_{b,m} = 0 \end{cases} \tag{3.16}$$

where N is the total number of instances observed in the historical data, and N_{faulty} is the number of faulty instances.

Meanwhile, the ridge regularization $\mathcal{L}_{\text{Ridge}}$ serves to penalize large values of the model parameters [158], which in turn helps mitigate the risk of data overfitting. From a probabilistic perspective, this corresponds to the Gaussian prior distribution for the model parameters $u_{b,j}$ and $v_{m,j}$, with zero mean and inverse variance of α [120]. Finally, $\mathcal{L}_{\text{NetLasso}}$ refers to the network Lasso regularization [69], which enforces clustering of the model parameters of bug reports and methods. The intuition is straightforward—the more similar two bug reports or two methods are (as quantified by $e_{b,b'}$ and $e_{m,m'}$), the closer their model parameters $\vec{u}_b$ and $\vec{v}_m$ should be. This combination of $\mathcal{L}_{\text{Entropy}}$, $\mathcal{L}_{\text{Ridge}}$ and $\mathcal{L}_{\text{NetLasso}}$ facilitates a robust model that can simultaneously optimize the bug localization quality and cluster the model parameters of similar bug reports and methods.

Next, in order to minimize the joint loss $\mathcal{L}$, we employ a Newton method [101] that is derived from a second-order Taylor series expansion of the loss function $\mathcal{L}$:

$$\mathcal{L}(\theta) = \mathcal{L}(\theta_0) + \nabla\mathcal{L}(\theta_0)(\theta - \theta_0) + \frac{\nabla^2\mathcal{L}(\theta_0)}{2}(\theta - \theta_0)^2 \tag{3.17}$$

The minima of $\mathcal{L}$ can be obtained by taking the partial derivative of $\mathcal{L}(\theta)$ and equating it to zero:

$$\begin{aligned} 0 &= \nabla \mathcal{L}(\theta_0) + \nabla^2 \mathcal{L}(\theta_0)(\theta - \theta_0) \\ \theta &= \theta_0 - \frac{\nabla \mathcal{L}(\theta_0)}{\nabla^2 \mathcal{L}(\theta_0)} \end{aligned} \tag{3.18}$$

If we take θ_0 as the old estimate of $u_{b,j}$ or $v_{m,j}$, this leads to the following update formulae:

$$u_{b,j} \leftarrow u_{b,j} - \frac{\nabla \mathcal{L}(u_{b,j})}{\nabla^2 \mathcal{L}(u_{b,j})} \tag{3.19}$$

$$v_{m,j} \leftarrow v_{m,j} - \frac{\nabla \mathcal{L}(v_{m,j})}{\nabla^2 \mathcal{L}(v_{m,j})} \tag{3.20}$$

In turn, we need to compute the first and second derivatives of each model parameter $u_{b,j}$ and $v_{m,j}$. For the bug report parameter $u_{b,j}$, the first and second derivatives are respectively given by:

$$\begin{aligned} \nabla \mathcal{L}(u_{b,j}) = &\sum_{m \in \mathcal{M}} \left[w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m}) x_{b,m,j} \right] \\ &+ \alpha u_{b,j} + \beta \sum_{b'} \left[e_{b,b'} \left(u_{b,j} - u_{b',j} \right) \right] \end{aligned} \tag{3.21}$$

$$\begin{aligned} \nabla^2 \mathcal{L}(u_{b,j}) = &\sum_{m \in \mathcal{M}} \left[w_{b,m} \sigma(\hat{f}_{b,m})(1 - \sigma(\hat{f}_{b,m})) x_{b,m,j}^2 \right] \\ &+ \alpha + \beta \sum_{b'} e_{b,b'} \end{aligned} \tag{3.22}$$

Similarly, we can compute the first and second derivatives w.r.t each method parameter $v_{m,j}$ as:

$$\begin{aligned}\nabla\mathcal{L}(v_{m,j}) &= \sum_{b\in\mathcal{B}}\left[w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m})x_{b,m,j}\right]\\ &\quad + \alpha v_{m,j} + \beta\sum_{m'}\left[e_{m,m'}\left(v_{m,j} - v_{m',j}\right)\right] \qquad (3.23)\\ \nabla^2\mathcal{L}(v_{m,j}) &= \sum_{b\in\mathcal{B}}\left[w_{b,m}\sigma(\hat{f}_{b,m})(1-\sigma(\hat{f}_{b,m}))x^2_{b,m,j}\right]\\ &\quad + \alpha + \beta\sum_{m'} e_{m,m'} \qquad (3.24)\end{aligned}$$

Finally, the update formula for $u_{b,j}$ can be obtained by substituting equation (3.21) and (3.22) into equation (3.19). Likewise, we can substitute (3.23) and (3.24) into (3.20) to arrive at the update formula for $v_{m,j}$. To learn the model parameters, we use a *Newton method* that updates the parameters on a per-feature j basis. This will be elaborated in Section 3.3.5.

3.3.5 Adaptive Learning

Algorithm 1 summarizes the adaptive learning procedure of the NetML integrator for computing the relevancy scores of a new bug report (i.e., a new query) to different methods (i.e., documents). Given a new bug report b^*, the set of K relevant bug reports $\mathcal{B}_K$ in the historical data, the set of all methods $\mathcal{M}$, and the similarity graphs $\mathcal{G}_B$ and $\mathcal{G}_M$, the learning procedure appends b^* into $\mathcal{B}_K$ and then updates the model parameters on a per-feature basis. That is, for each feature j, it performs Newton updates on the bug report parameters $u_{b,j}$ (steps 14–16) and method parameters $v_{m,j}$ (steps 23–25), in accordance with equations (3.19) and (3.20) respectively. The key idea here is to alternatingly update the parameter for one feature while keeping the parameters of the remaining features fixed. The procedure is repeated until a maximum iteration T_{max} is reached. Afterwards, the final prediction score $\hat{f}_{b^*,m}$ of the new bug report b^* for each method m is computed via equation (3.11).

Note that the selection of relevant bug report set B_K is based on the K-

Algorithm 1 Adaptive learning of the NetML integrator

```
Require:
    Set of $K$ relevant historical bug reports $\mathcal{B}_K$ (i.e., $|\mathcal{B}_K| = K$)
    Set of all methods $\mathcal{M}$, where $|\mathcal{M}| = M$
    New bug report query $b^*$ along with its features $\mathbf{X}_{b^*} = \{x_{b^*,m,j}\} \in \mathbb{R}^{1\times M\times J}$
    Features: $\mathbf{X} = \{x_{b,m,j}\} \in \mathbb{R}^{K\times M\times J}$. Labels: $\mathbf{Y} = \{y_{b,m}\} \in \mathbb{R}^{K\times M}$
    Bug report similarity graph $\mathcal{G}_B$, $\mathbf{E}_B = \{e_{b,b'}\}$
    Method similarity graph $\mathcal{G}_M$, $\mathbf{E}_M = \{e_{m,m'}\}$
Ensure:
    Relevancy scores $\hat{f}_{b^*,m} \in \mathbb{R}^{1\times M}$ of the new bug report $b^*$ to all methods $m$
    Bug report parameters $\mathbf{U} = \{u_{b,j}\} \in \mathbb{R}^{(K+1)\times J}$
    Method parameters $\mathbf{V} = \{v_{m,j}\} \in \mathbb{R}^{M\times J}$

Compute the union set of bug reports $\mathcal{B} \leftarrow \mathcal{B}_K \cup \{b^*\}$
Initialize: $u_{b,j} \leftarrow 0$ and $v_{m,j} \leftarrow 0$, $\forall b \in \mathcal{B}, m \in \mathcal{M}, j \in \{1, \dots, J\}$
Precompute: $q_b \leftarrow \sum_{b'} e_{b,b'}$ and $q_m \leftarrow \sum_{m'} e_{m,m'}$, $\forall b \in \mathcal{B}, m \in \mathcal{M}$
Compute the bug probabilities $\sigma(\hat{f}_{b,m})$ for all $(b, m)$ pairs
$\mathcal{L}_{\text{curr}} \leftarrow -\sum_b \sum_m w_{b,m} \big[ y_{b,m} \ln \big(\sigma(\hat{f}_{b,m})\big) + \big(1 - y_{b,m}\big) \ln \big(1 - \sigma(\hat{f}_{b,m})\big) \big]$
repeat
    $\mathcal{L}_{\text{prev}} \leftarrow \mathcal{L}_{\text{curr}}$
    for each $j \in \{1, \dots, J\}$ do
        for each $b \in \mathcal{B}$ do
            $p_b \leftarrow \sum_{b'} e_{b,b'} u_{b',j}$
        end for
        for each $b \in \mathcal{B}$ do
            $u_{\text{numer}} \leftarrow \sum_m \big[ w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m}) x_{b,m,j} \big] + \beta \big[ u_{b,j} q_b - p_b \big] + \alpha u_{b,j}$
            $u_{\text{denom}} \leftarrow \sum_m \big[ w_{b,m} \sigma(\hat{f}_{b,m})(1 - \sigma(\hat{f}_{b,m})) x^2_{b,m,j} \big] + \beta q_b + \alpha$
            $u_{b,j} \leftarrow u_{b,j} - \eta \left( \frac{u_{\text{numer}}}{u_{\text{denom}}} \right)$
        end for
        for each $m \in \mathcal{M}$ do
            $p_m \leftarrow \sum_{m'} e_{m,m'} v_{m',j}$
        end for
        for each $m \in \mathcal{M}$ do
            $v_{\text{numer}} \leftarrow \sum_b \big[ w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m}) x_{b,m,j} \big] + \beta \big[ v_{m,j} q_m - p_m \big] + \alpha v_{m,j}$
            $v_{\text{denom}} \leftarrow \sum_b \big[ w_{b,m} \sigma(\hat{f}_{b,m})(1 - \sigma(\hat{f}_{b,m})) x^2_{b,m,j} \big] + \beta q_m + \alpha$
            $v_{m,j} \leftarrow v_{m,j} - \eta \left( \frac{v_{\text{numer}}}{v_{\text{denom}}} \right)$
        end for
    end for
    Compute the updated bug probabilities $\sigma(\hat{f}_{b,m})$ via equation (3.11)
    $\mathcal{L}_{\text{curr}} \leftarrow -\sum_b \sum_m w_{b,m} \big[ y_{b,m} \ln \big(\sigma(\hat{f}_{b,m})\big) + \big(1 - y_{b,m}\big) \ln \big(1 - \sigma(\hat{f}_{b,m})\big) \big]$
    $\eta \leftarrow \begin{cases} \frac{\eta}{2}, & \text{if } \mathcal{L}_{\text{curr}} > \mathcal{L}_{\text{prev}} \\ \min(1, 2\eta), & \text{otherwise} \end{cases}$
until $T_{max}$ iterations
Compute the relevancy scores $\hat{f}_{b^*,m}$ using equation (3.11)
```

nearest neighbor retrieval from the bug report similarity graph $\mathcal{G}_B$, as follows:

$$\mathcal{B}_K = \text{Top}_K(\{e_{b^*,b} \mid \forall b \neq b^*\}) \tag{3.25}$$

where Top_K is a function that returns bug reports with the highest similarity $e_{b^*,b}$ to the query bug report b^*. The calculation of the similarity graphs is based on the VSM model and is described in Section 3.3.2.

It is also worth mentioning that the magnitude of the Newton update is downscaled by an adaptive learning rate η (where $0 < \eta \leq 1$). We introduce this scaling factor as a way to address the problem of *overshooting* in the Newton method [26], whereby the update $\frac{\nabla\mathcal{L}(u_{b,j})}{\nabla^2\mathcal{L}(u_{b,j})}$ or $\frac{\nabla\mathcal{L}(v_{m,j})}{\nabla^2\mathcal{L}(v_{m,j})}$ is overestimated—possibly by many orders of magnitude. This may lead to oscillations and sometimes divergence in the loss function. To alleviate this issue, we compare the loss function $\mathcal{L}$ before and after a Newton iteration (step 30), and then adjust η accordingly depending on whether $\mathcal{L}$ increases or not. If it increases, then we reduce η by half in order to dampen the update magnitude; otherwise, the value of η gets doubled, up to a maximum limit of 1.

For computational efficiency, we precompute the constant terms $q_b = \sum_{b'} e_{b,b'}$ and $q_m = \sum_{m'} e_{m,m'}$ before the Newton iterations begins. Additionally, during each Newton iteration, we have separate loops to compute the terms $\sum_{b'} e_{b,b'}u_{b',j}$ (step 11) and $\sum_{m'} e_{m,m'}v_{m',j}$ (step 20) for each feature j, prior to updating $u_{b,j}$ and $v_{m,j}$. The purpose is to make sure that, during the parameter updates (steps 14 and 23), the computation of $\sum_{b'} e_{b,b'}u_{b',j}$ in equation (3.19) and $\sum_{m'} e_{m,m'}v_{m',j}$ in equation (3.20) is based on the old parameter values from the previous iteration, and not affected by the ordering of b or m in the update loops.

We additionally highlight that the loss function $\mathcal{L}$ is *strictly convex.* This provides a nice theoretical guarantee that there is only one unique minimum in the loss function surface, and this minimum is globally optimal [184]. The convexity trait can be proven by looking at the curvatures (i.e., second derivatives) with respect to the bug report and method parameters, as per equations (3.22) and (3.24) respectively. Clearly, since $0 \leq \sigma(\hat{f}_{b,m}) \leq 1$, $w_{b,m}$, $x^2_{b,m,j}$, $e_{b,b'}$ and $e_{m,m'}$ are non-negative, while α and β are positive, the curvatures will be positive. The positive curvatures correspond to the so-called *positive definite* Hessian matrix—a well-known property of a strictly convex function [184].

3.4 Experiments

In this section, we first describe the datasets and evaluation settings used in our experiments. We then present a list of research questions we want to address, and accordingly elaborate our experiment results.

3.4.1 Dataset

To evaluate our approach, we use a dataset of 355 bugs from seven popular software projects. The seven projects are Ant [1], AspectJ [5], Lang [2], Lucene [4], Math [3], Rhino [10], and Time [8]. All seven projects are medium-large scale and implemented in Java. Ant, AspectJ, and Lucene each contain more than 300 KLOC. Math, Rhino, and Time each contain almost 100 KLOC, while Lang only contains more than 50 KLOC. The Ant, Lang, Lucene, and Math projects use Jira as the issue tracking system, from which we retrieve their bug reports. Bissyande et al. found that in Jira bugs are generally well linked to the commits that fix them [33]. AspectJ and Rhino use Bugzilla whereas Time uses Github as the issue tracking system, from which we collect their bug reports. Table 3.3 presents an overview of the seven projects considered in our study.

Following the procedure of Dallmeir and Zimmermann [49], we collected 116 bugs from Ant, Lucene, and Rhino. For each bug, we collected the pre-fix version, post-fix version, a set of successful test cases, and at least one failing test case. A failing test case is often included as an attachment to a

Table 3.3: Summary of the datasets used in this work. We use the short names of projects for brevity; "Ant" stands for "Apache-Ant", "Lang" stands for "Apache-Commons-Lang", "Math" stands for "Apache-Commons-Math", and "Time" stands for "Joda-Time".

Project	**#Bugs**	**Time Period**	**Average # Methods**
Ant	53	12/2001 – 09/2013	9,624.66
AspectJ	41	03/2005 – 02/2007	14,218.39
Lang	65	10/2002 – 04/2016	2,151.1
Lucene	37	06/2006 – 01/2011	10,220.14
Math	106	12/2004 – 03/2016	4,792.3
Rhino	26	12/2007 – 12/2011	4,839.58
Time	27	05/2004 – 03/2017	4,083.5

bug report or committed along with the fix in the post-fix version. When a developer receives a bug report, he/she first may want to replicate the error described in the report [157]. In this process, he is creating a failing test case. Unfortunately, not all test cases are documented and saved in the version control system. The 41 AspectJ bugs are from the iBugs dataset which was collected by Dallmeier and Zimmermann [49]. Each bug in the iBugs dataset comes with the code before the fix (pre-fix version), the code after the fix (post-fix version), and a set of test cases. The iBugs dataset contains more than 41 AspectJ bugs, but not all of them come with failing test cases. Test cases provided in the iBugs dataset are obtained from the various versions of the regression test suite that comes with AspectJ. We collected the remaining 198 bugs from Lang, Math, and Time from the Defects4J benchmark [94], a database of real, isolated, reproducible software faults from real-world open-source Java projects. The three projects include a large number of test cases, and there exists at least one failing test case per bug. Defects4J also contains two other projects, namely JFreechart and Closure-Compiler. We omit these projects since we are unable to fully collect all their bug reports.

3.4.2 Evaluation Metrics and Settings

To assess the effectiveness of a bug localization method, we employ two key metrics, namely: Top N and mean average precision (MAP). They are respectively described below:

- **Top N**: Given a bug report, if one of its corresponding faulty methods is in the top-N results, we consider that the bug is successfully localized. The Top N score of a bug localization method is the number of bugs it can successfully localize [238, 186]

- **Mean Average Precision (MAP)**: MAP is an IR metric to evaluate ranking approaches [148], and is computed by taking the mean of the *average precision* scores across all bug reports. The average precision of a single bug report is computed as:

  $$AP = \frac{\sum_{k=1}^{M} P(k) \times pos(k)}{\sum_{k=1}^{M} pos(k)}$$

  where k is a rank in the returned ranked methods, M is the number of ranked methods, and $pos(k)$ indicates whether the k^{th} method is faulty or not. Here $P(k)$ is the precision at a given top k methods, which is computed as follows:

  $$P(k) = \frac{\#faulty\ methods\ in\ the\ top\ k}{k}.$$

  Note that the MAP scores of existing bug localization methods are typically low [182, 191, 238, 186].

Our evaluation procedure is based on 10-fold *cross validation* (CV). That is, for each project, we divide the bug reports into ten (mutually exclusive) sets. Then, for each fold, we take 1 set as new bug report queries (i.e., the testing set) and treat the remaining 9 sets as historical bug reports (i.e., the training set). We repeat this 10 times, and then collate the results to get the aggregated Top N and MAP scores.

In all our experiments, the hyper-parameters of the NetML method were configured as follows. Firstly, the regularization parameters α and β were chosen by performing 10 fold cross validation on the training set. Next, the maximum number of iterations T_{max} was fixed to 30. We use $K = 10$ as default value for the number of nearest neighbors. Note that the NetML parameters K and T_{max} follow the settings used in AML [120], so as to ensure fair comparisons. All experiments were conducted on an Intel(R) Xeon 2.9GHz server running a Linux operating system.

In order to assess whether NetML substantially outperforms other bug localization methods, we apply *Wilcoxon signed-rank test* [219]; it is a non-parametric statistical significance test for comparing two related or matched samples, whereby the population cannot be assumed to be normally distributed. The Wilcoxon test was applied to two types of metric (i.e., Top N and MAP). For every evaluation metric, we collated the 10-fold results of a bug localization technique across the four software projects (i.e., AspectJ, Ant, Lucene, and Rhino) and then performed the Wilcoxon test to compare the collated results of different techniques. For this test, our null hypothesis is that NetML performs *worse* than or *equal* to the other method, and so we used a one-sided/tail p-value to validate this hypothesis. Moreover, we also apply the Benjamini-Hochberg (BH) [28] procedure to control the effect of multiple comparisons. If the p-value is sufficiently small (say, below a significance level of 0.05), we can confidently reject the null hypothesis and conclude that NetML is significantly better than the other method.

3.4.3 Research Questions

Our empirical study seeks to answer several research questions (RQ), as described in the following subsections.

3.4.3.1 RQ1: How Effective is NetML Compared to Other State-of-the-Art Techniques?

We compare our NetML approach with its predecessor, i.e., AML [120], and several other state-of-the-art techniques. Previously, Le et al. proposed Savant [23], a state-of-the-art bug localization approach that employs a learning-to-rank [89] strategy, using likely invariant *diffs* and suspiciousness scores as features. Ochiai [12] and Dstar [221] are well-known statistical formulas to detect suspicious locations for bug localization without requiring any prior information on program structure or semantics. PROMESIR [175], SITIR [137], and several variants developed by Dit et al. [53] were state-of-the-art multi-modal feature location techniques. Among the variants proposed by Dit et al. [53], the best performing ones were $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. We refer to these variants as $\mathrm{DIT^A}$ and $\mathrm{DIT^B}$ respectively. Dit et al. had shown that these two variants outperform SITIR, and so we exclude SITIR from our study. We also compare NetML with a state-of-the-art IR-based bug localization method named LR [230], and a state-of-the-art spectrum-based bug localization method named MULTRIC [227]. Note that, unlike PROMESIR, $\mathrm{DIT^A}$, $\mathrm{DIT^B}$, and MULTRIC which locate buggy methods, LR locates buggy files. Thus, we convert the list of files that LR produces into a list of methods by using two heuristics: (1) to return methods in a file in the same order that they appear in the file; and (2) to return methods based on their similarity to the input bug report as computed using a VSM model. We refer to the two variants of LR as LR^A and LR^B respectively.

For all the above-mentioned techniques, we used the same parameters and settings as described in the respective papers, with the following exceptions that we justify. For $\mathrm{DIT^A}$ and $\mathrm{DIT^B}$, the threshold used to filter methods using HITS was decided “such that at least one gold set method remained in the results for 66% of the [bugs]” [53]. In this chapter, since we used 10-fold

CV, rather than using 66% of all bugs, we used all bugs in the training data (i.e., 90% of all bugs) to tune the threshold. For PROMESIR, we also used 10-fold CV and applied a brute force approach to tune PROMESIR's component weights using a step of 0.05.

3.4.3.2 RQ2: Do Feature Components of NetML Contribute toward Its Overall Performance?

To answer this question, we conducted an ablation test by dropping one feature component (i.e., $\text{NetML}^{\text{Text}}$, $\text{NetML}^{\text{SuspWord}}$, or $\text{NetML}^{\text{Spectra}}$) one-at-a-time and evaluating the performance. In the process, we created three variants of NetML: $\text{All}^{-\text{Text}}$, $\text{All}^{-\text{SuspWord}}$, and $\text{All}^{-\text{Spectra}}$. That is, we excluded Text, SuspWord, and Spectra from all feature components, respectively (see also Fig. 3.2). We used the default value of $K = 10$, and applied the NetML adaptive learning procedure (i.e., Algorithm 1) to tune the model parameters of these variants. As our baseline, we performed the same ablation test to the feature components of the AML method (i.e., AML^{Text}, $\text{AML}^{\text{SuspWord}}$, or $\text{AML}^{\text{Spectra}}$).

3.4.3.3 RQ3: How Effective is the NetML Integrator?

Instead of using the NetML integrator component (see Section 3.3.3), one may consider using a standard machine learning algorithm, such as the learning-to-rank method, to combine the scores produced by the three feature components. Indeed, state-of-the-art IR-based and spectrum-based bug localization techniques such as LR and MULTRIC are based on the learning-to-rank method. As such, we conduct an experiment to compare our NetML integrator model with an off-the-shelf learning-to-rank model called SVM^{rank} [89], which was also used by LR [230]. To do so, we simply replace the NetML integrator model in Fig. 3.2 with SVM^{rank}, and then compare the resulting performance. For completeness, we also compare our NetML integrator with the integrator model used by the AML algorithm.

3.4.3.4 RQ4: What is the Effect of Varying the Number of Neighbors K on the Performance of NetML?

The most important parameter in our NetML approach is the number of nearest neighbors K (while the regularization parameters α and β were chosen via cross-validation—see Section 3.4.2). By default, we set the number of neighbors to $K = 10$, but the effect of varying this value is unclear. To answer this research question, we vary the value of K and investigate its effects on the performance of NetML. In particular, we wish to investigate if the performance remains relatively stable with varying values of K. For each K value, we also compare the performance of NetML against its predecessor (i.e., AML) using the same value.

3.4.3.5 RQ5: How Effective is NetML in Cross-Project Bug Localization?

To evaluate the robustness of our approach, we also conducted an empirical study on cross-project bug localization. That is, we first use a source project as training data to build a bug localization model, and then employ the model to predict a method that likely contains a bug in a (different) target project [215]. In this study, we compare NetML with its predecessor (i.e., AML) [120], Savant [23], Ochiai [12] and Dstar [221]. We use the same evaluation metrics as per Section 3.4.2 to assess the effectiveness of the different techniques. To configure the hyper-parameters of NetML, we adopt the same parameter tuning procedure as described in Section 3.4.2. Meanwhile, the hyper-parameters of the remaining localization techniques follow the parameter settings stated in their respective papers. We also apply *Wilcoxon signed-rank test* with the BH procedure to determine whether NetML performs substantially better than the other techniques.

Table 3.4: Top N (N $\in \{1, 5, 10\}$) results of NetML vs. AML, Savant, Ochiai, Dstar, and PROMESIR. The percentage in parentheses indicates the proportion of bug reports whose faulty methods are correctly localized.

Top N	NetML	AML	SAVANT	OCHIAI	DSTAR	PROMESIR
1	**116 (32.68%)**	88 (24.79%)	67 (21.34%)	48 (13.52%)	43 (12.11%)	61 (17.18%)
5	**219 (61.69%)**	179 (50.42%)	122 (38.85%)	94 (26.48%)	88 (24.79%)	139 (39.15%)
10	**255 (71.83%)**	213 (60.00%)	152 (48.41%)	124 (34.93%)	106 (29.86%)	174 (49.01%)

3.4.4 Results

This section presents the results of our experiments and discussion in relation to the research questions raised in Section 3.4.3.

3.4.4.1 RQ1: Comparisons of NetML with Other Techniques

Tables 3.4 and 3.5 show the Top N results of NetML as well as the other baseline methods including AML. Out of the 355 bugs, NetML is able to successfully localize 116, 219, and 255 bugs when the developers inspect the Top 1, Top 5, and Top 10 methods respectively. This implies that NetML can successfully localize 31.82%, 22.35%, and 19.72% more bugs than the best baseline (i.e., AML) by examining the Top 1, Top 5, and Top 10 methods respectively. Note that we encountered java.lang.UnsupportedClassVersionError when running Savant for AspectJ bugs. These AspectJ bugs are from iBugs dataset [49]. We have investigated and found that according to iBugs' documentation,[3] the AspectJ's faulty versions work with Java Virtual Machine (JVM) version 1.4. However, Savant employs Daikon [58] which requires Java 7 or later[4]. Therefore, we exclude AspectJ's bugs from the experiments for Savant.

Table 3.6 shows the MAP score of NetML along with those of the state-of-the-art multi-modal localization methods. Averaging across the seven projects, NetML achieves an overall MAP score of 0.347, which outperforms all the other baselines. In particular, NetML improves the average MAP of AML,

[3] https://www.st.cs.uni-saarland.de/ibugs/

[4] http://plse.cs.washington.edu/daikon/download/doc/daikon.html

Table 3.5: Top N (N $\in \{1, 5, 10\}$) results of NetML vs. DIT$^{\text{A}}$, DIT$^{\text{B}}$, LR^{A}, LR^{B}, and MULTRIC. The percentage in parentheses indicates the proportion of bug reports whose faulty methods are correctly localized.

Top N	**NetML**	**DIT$^{\text{A}}$**	**DIT$^{\text{B}}$**	**LR^{A}**	**LR^{B}**	**MULTRIC**
1	**116 (32.68%)**	41 (11.55%)	37 (10.42%)	12 (3.38%)	66 (18.59%)	68 (19.15%)
5	**219 (61.69%)**	88 (24.79%)	78 (21.97%)	67 (18.87%)	137 (38.59%)	133 (37.46%)
10	**255 (71.83%)**	117 (32.96%)	109 (30.7%)	116 (32.68%)	181 (50.99%)	162 (45.63%)

Table 3.6: Mean Average Precision (MAP) results of different bug localization methods.

Project	**NetML**	**AML**	**Savant**	**Ochiai**	**Dstar**	**PROMESIR**	**DIT$^{\text{A}}$**	**DIT$^{\text{B}}$**	**LR^{A}**	**LR^{B}**	**MULTRIC**
Ant	0.270	0.234	0.188	0.179	0.127	0.206	0.12	0.120	0.070	0.218	0.077
Aspectj	0.219	0.187	–	0.117	0.007	0.121	0.092	0.071	0.006	0.004	0.016
Lang	0.638	0.542	0.535	0.147	0.146	0.394	0.198	0.184	0.167	0.424	0.564
Lucene	0.290	0.284	0.178	0.133	0.136	0.204	0.169	0.166	0.063	0.184	0.188
Math	0.358	0.255	0.261	0.14	0.139	0.271	0.179	0.176	0.165	0.303	0.391
Rhino	0.302	0.243	0.243	0.137	0.127	0.203	0.092	0.09	0.034	0.103	0.172
Time	0.354	0.294	0.166	0.115	0.115	0.148	0.062	0.062	0.051	0.142	0.282
Overall	**0.347**	0.291	0.262	0.138	0.114	0.221	0.130	0.124	0.079	0.197	0.241

Savant, Ochiai, Dstar, PROMESIR, DIT$^{\text{A}}$, DIT$^{\text{B}}$, LR^{A}, LR^{B}, and MULTRIC by 19.24%, 32.44%, 151.45%, 204.39%, 57.01%, 166.92%, 62.15%, 339.24%, 76.14% and 43.98% respectively. Considering the individual projects, NetML remains the best performing approach in terms of MAP. That is, NetML achieves MAP scores of 0.270, 0.219, 0.638, 0.290, 0.358, 0.302, and 0.354 for the Ant, AspectJ, Lang, Lucene, Math, Rhino, and Time projects respectively. With respect to the best performing baseline (i.e., AML), these respectively constitute of 15.38%, 17.11%, 17.71%, 2.11%, 40.39%, 24.28%, and 20.41% improvements.

We finally performed the Wilcoxon test to compare the Top N and MAP results of different techniques. As we are unable run Savant on AspectJ, we omit this project and run the Wilcoxon test on the results collated over the remaining six software projects for each metric (i.e., Top 1, Top 5, Top 10, and MAP). Table 3.7 presents the p-values for the four metrics, evaluated at the significance levels of 0.05 and 0.01. The results show that NetML significantly outperforms AML on all the four metrics. Compared to the remaining techniques, NetML also performs significantly better in terms of Top 1, Top 5, Top 10 methods and MAP. Altogether, these results demonstrate the efficacy

Table 3.7: The p-values of the Wilcoxon test applying the BH procedure on various pairs of bug localization methods.

Method Comparision	**Top 1**	**Top 5**	**Top 10**	**MAP**
NetML vs. AML	3×10^{-7} (**)	4×10^{-5} (**)	0.008 (**)	5×10^{-8} (**)
NetML vs. Savant	6×10^{-8} (**)	1×10^{-5} (**)	9×10^{-4} (**)	1×10^{-8} (**)
NetML vs. Ochiai	2×10^{-7} (**)	4×10^{-10} (**)	6×10^{-10} (**)	6×10^{-12} (**)
NetML vs. Dstar	1×10^{-7} (**)	8×10^{-8} (**)	1×10^{-15} (**)	1×10^{-11} (**)
NetML vs. PROMESIR	8×10^{-9} (**)	1×10^{-8} (**)	5×10^{-6} (**)	4×10^{-10} (**)
NetML vs. $\mathrm{DIT}^{\mathrm{A}}$	4×10^{-14} (**)	2×10^{-16} (**)	3×10^{-16} (**)	8×10^{-21} (**)
NetML vs. $\mathrm{DIT}^{\mathrm{B}}$	4×10^{-15} (**)	8×10^{-17} (**)	1×10^{-20} (**)	3×10^{-27} (**)
NetML vs. LR^{A}	1×10^{-18} (**)	5×10^{-22} (**)	4×10^{-20} (**)0	8×10^{-22} (**)
NetML vs. LR^{B}	4×10^{-16} (**)	2×10^{-21} (**)	2×10^{-20} (**)	1×10^{-24} (**)
NetML vs. MULTRIC	3×10^{-16} (**)	1×10^{-21} (**)	1×10^{-20} (**)	1×10^{-28} (**)

(**): smaller than 0.01

Table 3.8: Contributions of feature components in NetML and AML. The percentage in parentheses indicates the propotion of bug reports whose faulty methods are correctly localized.

Approach	**Top 1**		**Top 5**		**Top 10**		**MAP**	
	NetML	**AML**	**NetML**	**AML**	**NetML**	**AML**	**NetML**	**AML**
$\mathrm{All}^{-\mathrm{Text}}$	68 (19.15%)	61 (17.18%)	144 (40.56%)	130 (36.62%)	179 (50.42%)	165 (46.48%)	0.228	0.212
$\mathrm{All}^{-\mathrm{Spectra}}$	56 (15.77%)	49 (13.80%)	128 (36.06%)	112 (31.65%)	172 (48.45%)	157 (44.23%)	0.215	0.210
$\mathrm{All}^{-\mathrm{SuspWord}}$	74 (20.85%)	65 (18.31%)	156 (43.94%)	136 (38.31%)	196 (55.21%)	182 (51.27%)	0.211	0.229
All	**116 (36.62%)**	88 (24.79%)	**219 (61.69%)**	179 (50.42%)	**255 (71.83%)**	213 (60.00%)	**0.347**	0.291

of the proposed NetML approach.

3.4.4.2 RQ2: Contribution of Feature Components

Table 3.8 summarizes the results of our ablation test on both NetML and AML, each comparing the full model and three reduced variants (i.e., $\mathrm{All}^{-\mathrm{Text}}$, $\mathrm{All}^{-\mathrm{Spectra}}$ and $\mathrm{All}^{-\mathrm{SuspWord}}$). It is evident that, for both NetML and AML, the full model always performs better than the reduced variants. This suggests that each feature component plays an important role, and omitting one of them may greatly affect the modelling performance. Among the three variants, it can be seen that $\mathrm{All}^{-\mathrm{SuspWord}}$ yields the smallest Top 1, Top 5, Top 10, and MAP scores for both NetML and AML. This suggests that the SuspWord feature component is more important than the other two (i.e., Text and Spectra).

Furthermore, comparing the results of NetML and AML, we can also observe that the former always gives a better, or at least equal, result than the latter. This suggests that the model parameterization using two sets of model

Table 3.9: Comparisons among different integrator models. The percentage in parentheses indicates the proportion of bug reports whose faulty methods are correctly localized.

Metrics	**Project**	**NetML**	**AML**	**SVM**^{rank}
Top 1	Ant	13 (24.53%)	9 (16.98%)	7 (13.21%)
	AspectJ	11 (26.83%)	7 (17.07%)	4 (9.76%)
	Lang	30 (46.15%)	28 (43.08%)	27 (41.54%)
	Lucene	12 (32.43%)	11 (29.73%)	10 (27.03%)
	Math	32 (30.19%)	25 (23.58%)	26(24.53%)
	Rhino	10 (38.46%)	4 (15.38%)	4 (15.38%)
	Time	8 (32.77%)	4 (24.79%)	5 (23.38%)
	Overall	**116 (32.68%)**	88 (24.79%)	83 (23.38%)
Top 5	Ant	24 (45.28%)	22 (41.51%)	24 (45.28%)
	AspectJ	15 (36.59%)	13 (31.71%)	11 (26.83%)
	Lang	55 (84.62%)	48 (73.85%)	45 (69.23%)
	Lucene	25 (67.57%)	22 (59.46%)	23 (62.16%)
	Math	69 (65.09%)	47 (44.34%)	46 (43.40%)
	Rhino	18 (69.23%)	14 (53.85%)	13 (50.00%)
	Time	13 (48.15%)	13 (48.15%)	13 (48.15%)
	Overall	**219 (61.69%)**	179 (50.42%)	175 (49.30%)
Top 10	Ant	35 (66.04%)	31 (58.49%)	31 (58.49%)
	AspectJ	16 (39.02%)	13 (31.71%)	14 (34.15%)
	Lang	62 (95.38%)	53 (81.54%)	54 (83.08%)
	Lucene	30 (81.08%)	29 (78.38%)	26 (70.27%)
	Math	75 (70.75%)	53 (50.00%)	55 (51.89%)
	Rhino	19 (73.08%)	19 (73.08%)	16 (61.54%)
	Time	18 (66.67%)	15 (55.56%)	16 (59.26%)
	Overall	**255 (71.83%)**	213 **(60.00%)**	212 **(59.72%)**
MAP	Ant	0.270	0.234	0.234
	AspectJ	0.219	0.187	0.131
	Lang	0.638	0.542	0.540
	Lucene	0.290	0.284	0.267
	Math	0.358	0.255	0.269
	Rhino	0.302	0.243	0.227
	Time	0.354	0.294	0.287
	Overall	**0.347**	0.291	0.279

parameters (instead of one in AML), along with the objective function formulation that jointly optimizes bug localization error and fosters clustering of similar bug reports and methods, contribute to the better overall performance of NetML.

3.4.4.3 RQ3: Comparisons among Integrator Models

Table 3.9 compares the performance of the NetML integrator model with that of the AML integrator and SVM^{rank}. We can observe that for all projects (i.e., AspectJ, Ant, Lucene, and Rhino) and metrics, the NetML integrator

Table 3.10: The p-values of the Wilcoxon test applying the BH procedure on various pairs of integrator model.

Metrics	**NetML vs. SVM^{rank}**	**NetML vs. AML**
Top 1	3×10^{-7} (**)	2×10^{-5} (**)
Top 5	2×10^{-3} (**)	1×10^{-3} (**)
Top 10	4×10^{-4} (**)	2×10^{-3} (**)
MAP	9×10^{-10} (**)	2×10^{-8} (**)

(**): smaller than 0.01

outperforms both the AML integrator and SVM^{rank}. With respect to SVM^{rank}, NetML achieves 39.76%, 25.15%, 20.28%, and 24.37% improvements, in terms of Top 1, Top 5, Top 10 and MAP scores across the four software projects, respectively. This can again be attributed to our NetML approach taking advantage of two sets of model parameters and performing a joint optimization of bug localization error and clustering of similar bug reports and methods.

We also conducted the Wilcoxon test to examine whether the improvements over the AML integrator and SVM^{rank} are statistically significant. The resulting p-values are summarized in Table 3.10. As before, the NetML integrator significantly outperforms the AML integrator in terms of Top 1, Top 5, Top 10, and MAP scores. Moreover, the NetML integrator is significantly better than SVM^{rank} in all evaluation metrics (i.e., Top 1, Top 5, Top 10, and MAP). All in all, these justify the effectiveness of our NetML integrator component.

3.4.4.4 RQ4: Effect of Varying Number of Neighbors

To address this research question, we varied the number of nearest neighbors from $K = 5$ to all bug reports in the training set (i.e., $K = \infty$) for both NetML and AML. The results are shown in Table 3.11. We can see that, as we increase K, the performance of both multi-modal techniques improves until a certain point (i.e., $K = 15$), and decreases beyond that. This suggests that including more neighbors can improve performance to some extent. However, an overly large number of neighbors may lead to an increased level of noise (i.e., the number of irrelevant neighbors), resulting in a degraded performance. Nevertheless, the differences in the Top N and MAP scores are fairly marginal,

Table 3.11: Effect of varying the number of nearest neighbors on NetML and AML. The percentage in parentheses indicates the proportion of bug reports whose faulty methods are correctly localized.

#Neighbors	**Top 1**		**Top 5**		**Top 10**		**MAP**	
	NetML	**AML**	**NetML**	**AML**	**NetML**	**AML**	**NetML**	**AML**
$K = 5$	112 (31.55%)	84 (23.66%)	224 (63.10%)	181 (50.99%)	254 (71.55%)	212 (59.72%)	0.342	0.289
$K = 10$	116 (32.68%)	88 (24.79%)	219 (61.69%)	179 (50.42%)	255 (71.83%)	213 (60.00%)	0.347	0.291
$K = 15$	117 (32.96%)	86 (24.23%)	223 (62.82%)	175 (49.30%)	255 (71.83%)	212 (59.72%)	0.347	0.291
$K = 20$	115 (32.39%)	86 (24.23%)	210 (61.97%)	173 (48.73%)	251 (70.70%)	210 (59.15%)	0.345	0.290
$K = 25$	110 (30.99%)	81 (22.81%)	210 (61.97%)	173 (48.73%)	251 (70.70%)	209 (58.87%)	0.331	0.285
$K = \infty$	110 (30.99%)	79 (22.26%)	208 (58.59%)	169 (47.61%)	251 (70.70%)	205 (57.75%)	0.329	0.283

Table 3.12: Overall Top N (N $\in \{1, 5, 10\}$ and Mean Average Precision (MAP) results in cross-project setting. The percentage in parentheses indicates the proportion of bug reports whose faulty methods are correctly localized.

Methods	**Top 1**	**Top 5**	**Top 10**	**MAP**
NetML	**74 (20.85%)**	**157 (44.23%)**	**197 (55.49%)**	**0.218**
AML	58 (16.34%)	131 (36.90%)	170 (47.89%)	0.174
Savant	45 (14.33%)	106 (33.76%)	135 (42.99%)	0.133
Ochiai	48 (12.11%)	94 (26.48%)	124 (34.93%)	0.138
Dstar	43 (13.52%)	88 (24.79%)	106 (29.86%)	0.114

which justifies the robustness of our NetML approach. Looking at Table 3.11, it is also clear that NetML consistently outperforms AML for all K values (i.e., from $K = 5$ to $K = \infty$).

3.4.4.5 RQ5: How Effective is NetML in Cross-Project Bug Localization?

Table 3.12 shows the overall performance of NetML and the baseline methods (i.e., AML, Savant, Ochiai, and Dstar) for the cross-project setting, in terms of the Top N and MAP scores respectively. Ochiai and Dstar are unsupervised learning methods, which do not depend on training labels. In this case, they give the same result for both cross-project and within-project settings. Hence, we reuse the results in Table 3.4. For the remaining techniques (i.e., NetML, AML, and Savant), we use the source project that has the best MAP score for the target project. The results show that NetML outperforms the best baseline (i.e., AML) by 27.59%, 19.85%, and 15.88% in terms of the Top 1, Top 5, and Top 10 methods, respectively. In terms of MAP, NetML outperforms AML, Savant, Ochiai, and Dstar by 25.29%, 63.91%, 91.23%, and 57.97% respectively.

Table 3.13: The p-values of the Wilcoxon test applying the BH procedure on various pairs of integrator model in cross-project setting.

Method Comparison	**Top 1**	**Top 5**	**Top 10**	**MAP**
NetML vs. AML	0.012 (*)	0.048 (*)	0.040 (*)	0.029 (*)
NetML vs. Savant	0.003 (**)	0.001 (**)	0.007 (**)	0.003 (**)
NetML vs. Ochiai	1×10^{-5} (**)	0.003 (**)	8×10^{-7} (**)	1×10^{-18} (**)
NetML vs. Dstar	4×10^{-4} (**)	7×10^{-5} (**)	6×10^{-7} (**)	7×10^{-17} (**)

(*): smaller than 0.05, (**): smaller than 0.01

We also perform Wilcoxon test to compare the overall results of the different techniques in the cross-project setting. Table 3.13 shows the p-values for different evaluation metrics (i.e., Top 1, Top 5, Top 10, and MAP) and pairs of techniques. The results indicate that NetML significantly outperforms all the baseline techniques (i.e., AML, Savant, Ochiai, and Dstar) for all the four metrics, thus demonstrating the superior performance of NetML in the cross-project setting.

3.5 Results Analysis and Discussion

In this section, we present a detailed analysis of the results obtained in Section 3.4.4. Firstly, we present some examples to understand the scenarios in which NetML performs well or poorly. We then present an analysis of how NetML can improve the MAP.

3.5.1 Successful Cases

We first present two examples of successful bug localization, with the goal of showing how NetML can take advantage of two types of similarity: 1) similarity among bug reports, and 2) similarity among methods.

Bug report similarity. Our first example involves Bug 30798[5] and Bug 43969[6] from project Ant – see Fig. 3.1. It has been briefly described in Section 1. NetML can outperform AML in identifying the buggy method of Bug

[5] https://bz.apache.org/bugzilla/show_bug.cgi?id=30798

[6] https://bz.apache.org/bugzilla/show_bug.cgi?id=43969

43969 by taking advantage of similarity among bug reports. To confirm that indeed these two bug reports are similar, we can apply the *Vector Space Model* (VSM) [148]. We represent each bug report as a TF-IDF vector [181], and then compute the cosine similarity between the TF-IDF vector of Bug 30798 and that of the remaining bug reports. We find that Bug 43969 is ranked at position #3. Likewise, we compute the cosine similarity between Bug 43969 and the other bug reports. Here, Bug 30798 is ranked at position #5. This shows that Bug 30798 and Bug 43969 are indeed very similar. AML assumes that the bug reports are independent and, owing to the lack of information on the textual description of Bug 43969, it fails to localize the faulty method. In contrast, we found that NetML learns similar model parameters (i.e., $\vec{u}_b$) for the two bug reports, and exploits this to compensate for the insufficient information when localizing Bug 43969.

Additionally, we find that none of the other baselines perform as well as NetML. Savant can localize the faulty method of Bug 30798 in its top-10 list, but it fails to do so for Bug 43969. For the other baselines (i.e., Ochiai, Dstar, PROMESIR, DIT$^{\text{A}}$, DIT$^{\text{B}}$, LR^{A}, LR^{B}, and MULTRIC), none of them is able to localize the faulty method for both bug reports. Among them, the two best performers (i.e., Ochiai and Dstar) give a high suspiciousness score to the faulty method, but there are more than 100 methods sharing this score.

Method similarity. Fig. 3.3 presents the description of Bug 31389[7] in project Ant. The bug resides in the `throwNotSupported` and `getElementType` methods of `IntrospectionHelper.java`. NetML is able to localize both methods at positions #1 and #9 respectively, all within the top 10 list. Meanwhile, AML is able to put the `throwNotSupported` method in the top 10 list, but it ranks the `getElementType` method at position #17. Ochiai, Dstar, PROMESIR and MULTRIC localize the `throwNotSupported` method in the top 10 list, but they fail to put the `getElementType` into the top 10 list. The other

[7] `https://bz.apache.org/bugzilla/show_bug.cgi?id=31389`

Program IntrospectionHelper.java	**Program IntrospectionHelper.java**

public void throwNotSupported(*final* Project project,
final Object parent, *final String* elementName) {
final String msg = project.getElementName)(parent)
+ NOT_SUPPORTED_CHILD_PREFIX
+ elementName
+ NOT_SUPPORTED_CHILD_POSTFIX;
throw new *UnsupportedElementException*(msg, elementName);
}
	public Class⟨?⟩ getElementType(*final String* elementName)
	throws BuildException {
	final Class⟨?⟩ nt = nestedTypes.get(elementName);
	if (nt == null) {
	throw new *UnsupportedElementException*("Class" +
	bean.getName() + "doesn't support the nested \" +
	elementName + "\" element.", elementName); }
	return nt;
	}

Bug 31389
`Summary`: incorrect error text with invalid "javac" task after a "presetdef" `Description`: **What steps will reproduce the problem?** See below for the build.xml that was used and the faulty error message. 1. I made a preset definition containing a javac task 2. I made a normal target (not using the preset definition) containing a javac task with an illegal tag name 3. When running ant, the error message says that the error is in the preset definition instead of the javac task. (The line number in the message is good.) . . .

Figure 3.3: Example of successful bug localization of two methods in project Ant that need to be resolve the same bug report. The two methods have high cosine similarity score. The colored text indicates some common word tokens occurring in the two methods.

baselines give low relevancy scores to the two methods, and exclude them from the top 10 list.

As with the previous example, we try to analyze this further by computing the cosine similarity of the TF-IDF representation of the methods' source code. Specifically, we compute the cosine similarity between `throwNotSupported` and remaining methods. The result shows that the `getElementType` method is ranked at position #4. Looking at the content of these two methods, it can again be seen that they share many common word tokens (e.g., "elementName", etc.). Accordingly, NetML would cause the corresponding method parameters to be similar. As such, NetML manages to successfully to localize the `getElementType` method at position #9. In contrast, AML assumes that the methods are independent, and thus fails to leverage the strength of similar methods to localize the `getElementType` method.

To see how typical the successful cases are in our dataset, we randomly select 75 out of 183 successful cases, in which NetML manages to localize a faulty method within the top 10 list whereas the other baseline methods (i.e., AML, Savant, Ochiai, Dstar, PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, LR^{B}, and MULTRIC) fail to do so. Among these cases, in total, we find that 63 successful

Bug 338	**Bug 358**
`Summary:` Wrong parameter for first step size guess for Embedded Runge Kutta methods `Description:` **What steps will reproduce the problem?** In a space application using DOP853 i detected what seems to be a bad parameter in the call to the method initializeStep of class AdaptiveStepsizeIntegrator. . . .	`Summary:` ODE integrator goes past specified end of integration range `Description:` **What steps will reproduce the problem?** End of integration range in ODE solving is handled as an event. In some cases, numerical accuracy in events detection leads to error in events location. . . .

Figure 3.4: Example of unsuccessful bug localization of two bug reports which have the same faulty method in the project Math. The two bug reports have low cosine similarity score.

cases, which constitute the majority (84%) of our samples, are similar to the first (17 cases) and second (46 cases) examples we presented earlier.

3.5.2 Unsuccessful Cases

Next, we present two examples whereby NetML fails to localize a bug. These examples provide an understanding of cases in which NetML may not perform well.

Bug report similarity. We first consider Bug 338[8] and Bug 358[9] from project Math shown in Fig. 3.4. The faulty method for these two bug reports is the `integrate` method in `EmbeddedRungeKuttaIntegrator.java`. Interestingly, Ochiai and Dstar manage to localize this faulty method for these two bug reports within the top 10 list. On the other hand, NetML, AML, and Savant fail to localize the faulty `integrate` method for Bug 358. Specifically, NetML, AML, and Savant rank the faulty method at positions #14, #19, and #23 respectively. MULTRIC assigns a high suspiciousness score to the `integrate` method for both Bug 338 and Bug 358. However, there are around 30 methods sharing this score. Also note that the remaining baselines (i.e., PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, and LR^{B}) fail to localize the faulty method for both bug reports.

Similar to Section 3.5.1, we calculate the cosine similarity between Bug 338 and the remaining bug reports. We found that Bug 358 is ranked at position

[8] `https://issues.apache.org/jira/browse/MATH-338`

[9] `https://issues.apache.org/jira/browse/MATH-358`

Program ChangeLogParser.java

```
private Date parseDate(final String date) {
  try {
    return c_inputDate.parse(date);
  } catch (ParseException e) {
    //final String message = REZ.getString(
    //"changelog.bat-date.error", date);
    //getContext().error( message );
    return null;
    }
}
```

Program ChangeLogParser.java

```
private void processGetPreviousRevision(final String line)
{
  if (!line.startsWith("revision")){
    throw new IllegalStateException("Unexpected line
    from CVS:" + line);
  }
  m_previousRevision = line.substring(9);
  saveEntry();
  m_revision = m_previousRevision;
  m_status = GET_DATE;
}
```

Bug 30962

`Summary`: cvschangelog crashes with NullPointerException

`Description`:

What steps will reproduce the problem? I try to make cvschangelog running and face a strange problem that nobody else seems to have: cvschangelog crashes with a NullPointerException. My task looks like:

⟨target name="cvs.changelog"⟩
⟨cvschangelog dir="somedir" destfile="changelog.xml"⟩
...

Figure 3.5: Example of unsuccessful bug localization of two methods in project Ant that need to be resolved. The two methods have low cosine similarity score.

#53, suggesting that the two bug reports are dissimilar. As such, there is less incentive for NetML to leverage the strength of common words shared by the two bug reports, which potentially explains why it fails to localize the faulty method for Bug 358. This also suggests that, when the data contain bug reports that are largely dissimilar (i.e., share very few common word tokens), our NetML approach may not work as well as some spectrum-based fault localization techniques such as Ochiai and Dstar.

Method similarity. Fig. 3.5 shows the description of Bug 30962[10] in the project Ant. The bug is associated with two faulty methods, i.e., `parseDate` and `processGetPreviousRevision` in `ChangeLogParser.java`. We find that Ochiai and Dstar successfully localize these two methods in the top 10 list. NetML and AML are able to localize the `parseDate` method within the top 10 list. However, they fail to localize the faulty `processGetPreviousRevision` method for Bug 30962. In particular, NetML and AML place this faulty method at positions #17 and #15 respectively. The remaining techniques (i.e., Savant, PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, LR^{B}) fail to localize these two methods in the top 10 list.

To better understand this, we again compute the cosine similarity between

[10] https://bz.apache.org/bugzilla/show_bug.cgi?id=30962

the `parseDate` method and the remaining methods in project Ant. In this case, the `processGetPreviousRevision` method is ranked at position #478. This suggests that these two methods have low proximity, which gives less incentive for NetML to utilize their common words in the localization of the `processGetPreviousRevision` method. It also suggests that, when the data contain methods that are mostly dissimilar, spectrum-based fault localization techniques (e.g., Ochiai and Dstar) may perform better than NetML.

To again evaluate how typical the unsuccessful cases are in seven projects, we randomly select 75 (out of 80) unsuccessful cases whereby NetML fails to localize a faulty method within the top 10 list, but one of the baseline method (i.e., AML, Savant, Ochiai, Dstar, PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, LR^{B}, and MULTRIC) succeeds. Among them, in total, we discover that 70 unsuccessful cases, which constitute 93% of our samples, are similar to the first (21 cases) and second (49 cases) unsuccessful examples presented earlier.

3.5.3 Improved vs. Deteriorated Bug Reports

To understand how the MAP results improve due to NetML, following Chaparro et al. [40], we perform a finer-grained analysis in terms of the number of bug reports. We compare our approach against the best baseline method (i.e., AML). A bug report is *improved* if the rank of the top faulty method produced by NetML is better than the rank of the top faulty method produced by AML. On the other hand, a bug report is *deteriorated* if the rank of the top faulty method produced by NetML is worse than that produced by AML. Otherwise, a bug report is *unchanged*. Ideally, we wish to have a higher number of *improved* bug reports than that of *deteriorated* bug reports. To measure the relative magnitude of improvement or deterioration for each bug report, we adopt the approach described by Chaparro et al. [40]. In particular, for *improved* and *deteriorated* bug reports, we compute the expected average precision (AP) difference $E[\Delta AP]$ and expected rank difference $E[\Delta Rank]$ as

Table 3.14: Comparison of number of samples, expected average precision difference, and expected rank difference between NetML and AML.

Project	**Improved**			**Deteriorated**			**Unchanged**
	No. of samples	$E[\Delta AP]$	$E[\Delta Rank]$	**No. of samples**	$E[\Delta AP]$	$E[\Delta Rank]$	**No. of samples**
Ant	35 (66.04%)	12.57%	186.86	3 (5.66%)	-8.47%	-54.27	15 (28.3%)
Aspectj	32 (78.05%)	23.02%	59.31	3 (7.32%)	-39.67%	-39.67	6 (14.63%)
Lang	37 (56.92%)	29.94%	26.93	5 (7.69%)	-36.11%	-35.52	23 (35.38%)
Lucene	14 (37.84%)	11.94%	372.64	10 (27.03%)	-14.97%	-297.2	13 (35.14%)
Math	70 (66.04%)	37.37%	16.45	10 (9.43%)	-6.33%	-13.07	26 (24.53%)
Rhino	22 (84.62%)	36.39%	54.54	2 (7.69%)	-15.51%	-15.32	2 (7.69%)
Time	16 (59.26%)	24.15%	60.5	6 (22.22%)	-7.54%	-10.12	5 (18.52%)
Overall	**229 (63.66%)**	**27.75%**	**212.91**	**39 (10.98%)**	**-14.36%**	**-69.09**	**90 (25.35%)**

follows:

$$E[\Delta AP] = \frac{1}{|\mathcal{B}|}\sum_{b=1}^{\mathcal{B}}(AP_b^{NetML} - AP_b^{AML}) \tag{3.26}$$

$$E[\Delta Rank] = \frac{1}{|\mathcal{B}|}\sum_{b=1}^{\mathcal{B}}(Rank_b^{AML} - Rank_b^{NetML}) \tag{3.27}$$

where $|\mathcal{B}|$ is the number of bug reports, AP_b^{NetML} and AP_b^{AML} are the average precision produced by NetML and AML for bug report b, and $Rank_b^{NetML}$ and $Rank_b^{AML}$ are the rank produced by NetML and AML, respectively. Intuitively, if NetML is better than AML, we expect the $E[\Delta AP]$ and $E[\Delta Rank]$ for *improved* bug reports to be larger than those of *deteriorated* bug reports.

Table 3.14 shows the number of *improved*, *deteriorated*, and *unchanged* bug reports in our seven projects. Additionally, Table 3.14 presents the $E[\Delta AP]$ and $E[\Delta Rank]$ for *improved* and *deteriorated* cases of different projects. The results show that the number of *improved* bug reports is indeed higher than the number of *deteriorated* bug reports for all different projects. It is also evident that the overall $E[\Delta AP]$ and $E[\Delta Rank]$ of *improved* bug reports are higher than those of *deteriorated* bug reports. This implies that MAP improvement comes from improvements across the board and not due to a few outlier bug reports or projects.

3.6 Threats to Validity

This section presents a number of threats that may potentially impact the validity of our study.

3.6.1 Number of Failed Test Cases and Its Impact

In our experiments with 355 bugs, most of the bugs were found to come with few failed test cases (average = 2.155). We have investigated whether the number of failed test cases impacts the effectiveness of our approach. To this end, we computed the difference between the average number of failed test cases for bugs that are successfully localized at Top-N positions (N = 1,5,10) and bugs that are not successfully localized. We found that the differences are small (-0.362 to 0.055 test cases). These indicate that the number of test cases does not impact the effectiveness of our approach significantly and typically 1 to 3 failed test cases are sufficient for our approach to be effective.

3.6.2 Threats to Internal Validity

Threats to internal validity relate to implementation and dataset errors. We have checked our implementations and datasets. However, there could still be errors that we do not notice. Threats to external validity relate to the generalizability of our findings. In this work, we have analyzed 355 real bugs from seven medium-large software systems. In the future, we plan to reduce the threats to external validity by investigating more real bugs from additional software systems, written in various programming languages.

3.7 Chapter Summary

In this chapter, we put forward a novel multi-modal bug localization approach named Network-clustered Multi-modal Bug Localization (NetML). Deviating from the contemporary multi-modal localization approaches, NetML is able to achieve an effective bug localization through the interplay of two sets of model parameters characterizing both bug reports and methods. It also features an adaptive learning procedure that stems from a strictly convex objective function formulation, thereby provides a sound theoretical guarantee on the uniqueness of the optimal solution.

We have extensively evaluated NetML on 355 real bugs from seven different software projects (i.e., Ant, AspectJ, Lang, Lucene, Math, Rhino, and Time). Among the 355 bugs, NetML is able to successfully localize 116, 219, and 255 bugs when developers inspect the Top 1, Top 5, and Top 10 methods, respectively. Compared to the best performing baseline (i.e., AML), NetML can successfully localize 31.82%, 22.35%, and 19.72% more bugs when developers inspect the Top 1, Top 5, and Top 10 methods, respectively. Furthermore, in terms of MAP, NetML outperforms the other baselines by 19.24%. Based on the Wilcoxon signed-rank test using BH procedure, we show that the results of NetML are significantly better across the seven projects, in terms of Top 1, Top 5, Top 10, and MAP scores.

Chapter 4

Deep Learning Framework for Just-in-time Defect Prediction

Software quality assurance efforts often focus on identifying defective code. To find likely defective code early, change-level defect prediction – aka. *Just-In-Time* (JIT) defect prediction – has been proposed. JIT defect prediction models identify likely defective changes and they are trained using machine learning techniques with the assumption that historical changes are similar to future ones. Most existing JIT defect prediction approaches make use of manually engineered features. Unlike those approaches, in this chapter, we propose an end-to-end deep learning framework, named DeepJIT, that automatically extracts features from commit messages and code changes and uses them to identify defects. Experiments on two popular software projects (i.e., QT and OPENSTACK) in three evaluation settings (i.e., cross-validation, short-period, and long-period) show that the best variant of DeepJIT (DeepJIT-Combined), compared with the best performing state-of-the-art approach, achieves improvements of 10.36-11.02% for the project QT and 9.51-13.69% for the project OPENSTACK in terms of the Area Under the Curve (AUC).

4.1 Introduction

As software systems are becoming the backbone of our economy and society, defects existing in those systems may substantially affect businesses and people's lives in many ways. For example, Knight Capital,[1] a company that executes automated trading for retail brokers, lost $440 million in only one morning in 2012 due to an overnight faulty update to its trading software. A flawed code change, introduced into OpenSSL's source code repository, caused the infamous Heartbleed[2] bug which affected billions of Internet users in 2014. As software grows significantly in both size and complexity, finding defects and fixing them has become increasingly difficult and costly.

One common best practice for cost saving is identifying defects and fixing them as early as possible, ideally before new code changes (i.e. *commits*) are introduced into codebases. Emerging research [98, 56] has thus developed *Just-In-Time* (JIT) defect prediction models and techniques that help software engineers and testers to quickly narrow down the most likely defective commits to a software codebase. JIT defect prediction tools provide early feedback to software developers to allow them to prioritize and optimize their effort for inspection and (regression) testing, especially when facing with deadlines and limited resources. They have therefore been integrated into the development practice at large software organizations such as Avaya [156], Blackberry [189], and Cisco [199].

Machine learning techniques have been widely used in existing work for building JIT defect prediction models. A common theme of existing work [156, 99, 103, 112] is carefully crafting a set of features to represent a code change, and using them as defectiveness predictors. Those features are mostly derived from properties of code changes, such as change size (e.g. lines deleted or added), change scope (e.g. number of files or directories modified), history of

[1] https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/

[2] http://heartbleed.com

changes (e.g. number of prior changes to the updated files), track record of the author and code reviewers, and activeness of the code review of the change. This set of features can then be used as an input to a traditional classifier (e.g. Random Forests or Logistic Regression) to predict the defectiveness of code changes.

The aforementioned metric-based features however do not represent the semantic and syntactic structure of the actual code changes. In many cases, two different code changes that have exactly the same metrics (e.g. the same number of lines deleted and added) may generate different behaviour when executed, and thus have a different likelihood of defectiveness. Previous studies have showed the usefulness of harvesting the semantic information and syntactic structure hidden in source code to perform various software engineering tasks such as code completion, bug detection and defect prediction [215, 207, 163, 75, 131]. This information may enrich representations for defective code changes, and thus improve JIT defect prediction.

A recent work [228] used a deep learning model (i.e. Deep Belief Network) to improve the performance of JIT defect prediction models. However, their approach does not leverage the true notions of deep learning as they still employ the same set of features that are manually engineered as in previous work implying that their model is *not* end-to-end trainable.

To more fully explore the power of deep learning for JIT defect prediction, in this chapter, we present a new model (named DeepJIT) which is built upon the well-known deep learning technique, namely Convolutional Neural Network (CNN) [123]. CNN has produced many breakthroughs in Natural Language Processing (NLP) [105, 54, 95, 236, 90]. Our DeepJIT model processes both a commit message (in natural language), if available and the associated code changes (in programming languages) and automatically extracts features that represent the "meaning" of the commit. Unlike commit messages, code changes are more complex as they include a number of deleted and added lines across multiple files. Our model automatically learns the semantic features of each deleted or added line in each changed file. Those features are then aggregated to generate a new representation of the changed file, which is used to construct the features of the code changes in a given commit. This approach removes the need for software practitioners to manually design and extract features, as was done in previous work [151]. The features extracted from commit messages and code changes are then collectively used to train a model to predict whether a given commit is buggy or not.

The main contributions of our work include:

- An end-to-end deep learning framework (DeepJIT) to automatically extract features from both commit messages and code changes in a given commit.

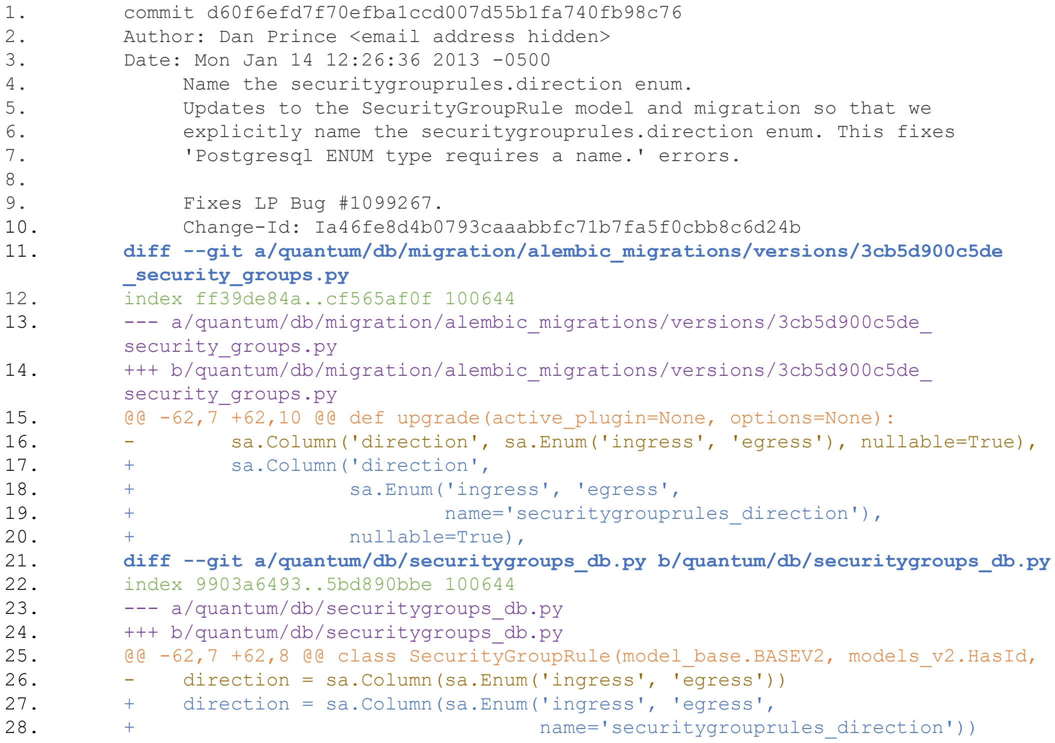
- An evaluation of DeepJIT on two software projects (i.e., QT and OPENSTACK). This dataset was originally collected by McIntosh and Kamei to evaluate their proposed technique for JIT defect prediction [151] that we use as one of the baselines. The experiments show the superiority of DeepJIT compared to state-of-the-art baselines.

4.2 Background

In this section, we first present an example of a buggy change and briefly describe a typical buggy change identification process that is followed by QT and OPENSTACK. We then introduce background knowledge about Convolutional Neural Network (CNN).

4.2.1 Buggy Changes and Their Identification

Figure 4.1 shows an example of a buggy commit in OPENSTACK. The buggy commit contains many pieces of information, i.e., a commit id (line 1), an author name (line 2), a commit date (line 3), a commit message (line 4-10)



```
1.      commit d60f6efd7f70efba1ccd007d55b1fa740fb98c76
2.      Author: Dan Prince <email address hidden>
3.      Date: Mon Jan 14 12:26:36 2013 -0500
4.           Name the securitygrouprules.direction enum.
5.           Updates to the SecurityGroupRule model and migration so that we
6.           explicitly name the securitygrouprules.direction enum. This fixes
7.           'Postgresql ENUM type requires a name.' errors.
8.
9.           Fixes LP Bug #1099267.
10.          Change-Id: Ia46fe8d4b0793caaabbfc71b7fa5f0cbb8c6d24b
11.     diff --git a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de
        _security_groups.py
12.     index ff39de84a..cf565af0f 100644
13.     --- a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
        security_groups.py
14.     +++ b/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
        security_groups.py
15.     @@ -62,7 +62,10 @@ def upgrade(active_plugin=None, options=None):
16.     -        sa.Column('direction', sa.Enum('ingress', 'egress'), nullable=True),
17.     +        sa.Column('direction',
18.     +                  sa.Enum('ingress', 'egress',
19.     +                          name='securitygrouprules_direction'),
20.     +                  nullable=True),
21.     diff --git a/quantum/db/securitygroups_db.py b/quantum/db/securitygroups_db.py
22.     index 9903a6493..5bd890bbe 100644
23.     --- a/quantum/db/securitygroups_db.py
24.     +++ b/quantum/db/securitygroups_db.py
25.     @@ -62,7 +62,8 @@ class SecurityGroupRule(model_base.BASEV2, models_v2.HasId,
26.     -    direction = sa.Column(sa.Enum('ingress', 'egress'))
27.     +    direction = sa.Column(sa.Enum('ingress', 'egress',
28.     +                                  name='securitygrouprules_direction'))
```

Figure 4.1: An example of a buggy commit change in OPENSTACK.

and a set of code changes (i.e., 11-28). A set of code changes includes changes to multiple files and each file includes a number of deleted and added lines representing the change. In Figure 4.1, line 16 (starting with -) and lines 17-20 (starting with +) indicate the deleted and added lines of a changed file (namely `3cb5d900c5de_security_groups.py`), respectively. The commit message also plays an important role as a good commit message can help maintainers to speed up the reviewing process and write a good release note.

To review a commit, QT and OPENSTACK use Gerrit,[3] which is a code review tool for `git`-based software projects. The process of reviewing code changes is as follows:

- Upload change revision: An author of a code change submits a new change to Gerrit and invites reviewers to comment on it.

- Execute sanity tests: Sanity tests verify that the code changes are compliant with the coding style conventions before sending the changes to

[3] https://code.google.com/p/gerrit/

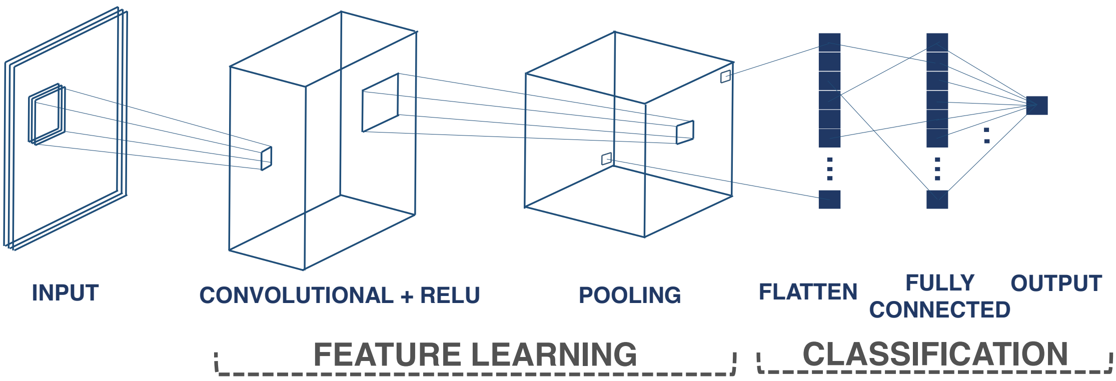


Figure 4.2: A simple convolutional neural network architecture.

the reviewers.

- Solicit peer feedback: The reviewers are asked to examine the code changes after it passes the sanity tests.

- Initiate an integration request: Teams are allowed to verify the code changes before integrating it into `git` repositories.

- Execute integration tests: The integration testing system is run to ensure that the code changes that are put in the `git` repository is clean.

- Final integration: After passing the integration testing, Gerrit automatically commits the code changes into the `git` repository.

4.2.2 Convolutional Neural Network

One of the most promising neural networks is the Convolutional Neural Network (CNN) [123]. CNNs have been widely used for many problems (i.e., image pattern recognition, natural language processing, information retrieval, etc.) and demonstrated to achieve promising results [100, 118, 115]. CNNs receive an input and perform a product operation followed by a nonlinear function. The last layer is the output layer containing objective functions [237] associated with the labels of the input.

Figure 4.2 illustrates a simple CNN for a classification task. The CNN includes an input layer, a convolutional layer, followed by the application of the rectified linear unit (RELU) which is a nonlinear activation function, a pooling layer, a fully-connected layer, and an output layer. We briefly explain these layers in the following paragraphs.

The input layer typically takes as an input a 2-dimensional matrix and passes it through a series of convolutional layers. The convolutional layers play a vital role in CNN and these layers takes advantage of the use of learnable filters. These filters are small in spatial dimensionality, but they are applied along the entirety of the depth of the input data. For example, given an input data $\mathbf{I} \in \mathbb{R}^{\mathrm{H \times W \times D}}$ and a filter $\mathbf{K} \in \mathbb{R}^{\mathrm{h \times w \times D}}$, we produce an activation map $\mathbf{A} \in \mathbb{R}^{\mathrm{(H-h) \times (W-w) \times 1}}$. The RELU, which outperforms other activation functions [123], is then applied to each value of the activation map as follows:

$$f(x) = max(0, x) \tag{4.1}$$

The pooling layer aims to reduce the dimensionality of the activation map and the number of parameters in order to control overfitting [206]. The pooling layer operates on the activation map and scales its dimensionality. There are three different types of pooling layers:

- Max pooling takes the largest element from each region of the activation map.

- Average pooling constructs the average value from each region of the activation map.

- Sum pooling sums all the elements from each region of the activation map.

In practice, max pooling has often been found to achieve a better performance compared to the other two pooling techniques [233]. The output of the pooling layer is flattened and directly passed to a fully connected layer. The output of the fully connected layer is passed to the output layer to calculate an objective

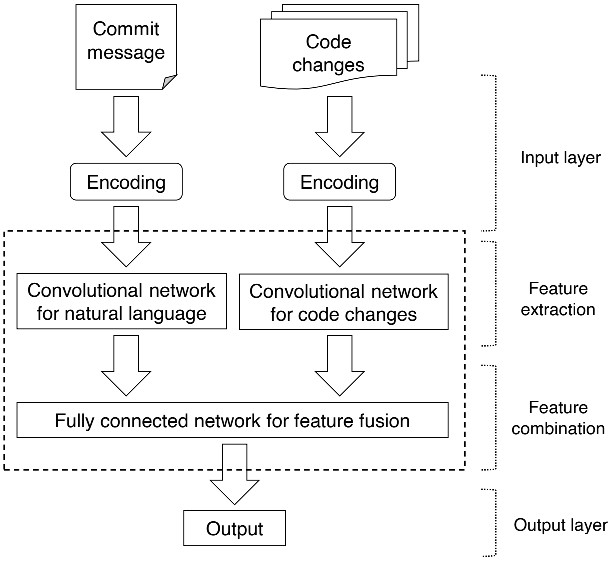

Commit message
Code changes
Encoding
Encoding
Input layer
Convolutional network for natural language
Convolutional network for code changes
Feature extraction
Fully connected network for feature fusion
Feature combination
Output
Output layer

Figure 4.3: The general framework of the *Just-In-Time* defect prediction model.

function (or a loss function). The objective function is normally optimized using stochastic gradient descent (SGD) [34].

4.3 Proposed Approach

In this section, we first formulate the *Just-In-Time* (JIT) defect prediction problem and provide an overview of our framework. We then describe the details of each part inside the framework. Finally, we present an algorithm for learning effective settings of our model's parameters.

4.3.1 Framework Overview

The goal of a JIT defect prediction model is to automatically classify a commit change as buggy or clean. This helps software teams prioritize the effort and optimize testing and inspection. We consider the JIT defect prediction problem as a learning task to construct prediction function $\mathbf{f} : \mathcal{X} \longmapsto \mathcal{Y}$, where $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether a commit change $x_i \in \mathcal{X}$ is clean ($y_i = 0$) or contains a buggy code ($y_i = 1$). The prediction function $\mathbf{f}$ can be learned by minimizing the following objective function:

$$\min_{\mathbf{f}} \sum_{i} \mathcal{L}(\mathbf{f}(x_i), y_i) + \lambda\Omega(\mathbf{f}) \tag{4.2}$$

where $\mathcal{L}(.)$ is the empirical loss function measuring the difference between the predicted and the output label, $\Omega(\mathbf{f})$ is a regularization function to prevent over fitting, and λ the trade-off between $\mathcal{L}(.)$ and $\Omega(\mathbf{f})$. Figure 4.3 gives an overview of the framework of the JIT defect prediction model (namely DeepJIT). The model consists of four parts: the input layer, the feature extraction layer, the feature combination layer, and the output layer. We explain the details of each part in the following subsections.

4.3.2 Parsing a Commit to Input Layer

To feed the raw textual data to the convolutional layers for feature learning, we first encode a commit message and code changes into arrays and feed them in the input layer. For the commit message, we use NLTK [142], which is a suite of libraries for natural language processing (NLP), to extract a sequence of words from it. We employ PorterStemmer [220] to produce the root forms of words. We also remove stop words and rare words (e.g. those occurring fewer than three times in the commit messages).

We then again use NLTK for parsing the code changes of a given commit. In particular, each change file in the code changes is parsed into a set of deleted and added lines, and each line is parsed into a sequence of words. We ignore comments and blank lines in the change file (see Figure 4.5). Following a previous work [218], we replace a number (i.e., an integer, real number, or hexadecimal number) with a special `<num>` token. We also replace rare code tokens (e.g. those occurring fewer than three times in the commit codes) and tokens existing in test data but absent in the training data with a special `<unk>` token. We add a `<deleted>` token or a `<added>` token at the beginning

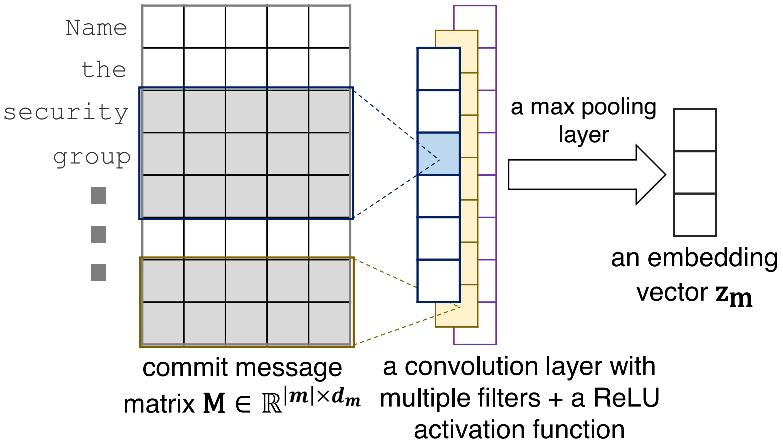

Name
the
security
group
a max pooling layer
an embedding vector $\mathbf{z_m}$
commit message matrix $\mathbf{M} \in \mathbb{R}^{|m| \times d_m}$
a convolution layer with multiple filters + a ReLU activation function

Figure 4.4: A convolutional network architecture for commit message.

of a deleted or added line respectively so that DeepJIT recognizes whether this code line is a deleted line or an added line.

We represent each word in the commit message and code changes as a dimensional vector. After the preprocessing step, $\mathcal{X}_i^m$ and $\mathcal{X}_i^c$, which are the encoded data of the commit message and code changes respectively, are passed to the convolutional layers to generate the commit message and code change features. In the convolutional layers, the commit messages and code changes are processed independently to extract the features based on each type of textual information. These features are then combined into a unified feature representation, and followed by a linear hidden layer connected to the output layer used to produce the output label $\mathcal{Y}$ indicating whether the commit change x_i is clean or contains a buggy code.

The core of the DeepJIT lies in the convolutional network layers for code changes (see Section 4.3.4) and the feature combination layers (see Section 4.3.5). In the following subsections, we firstly discuss the convolutional layers for the commit message and then present the core parts of DeepJIT in Section 4.3.4 and Section 4.3.5.

4.3.3 Convolutional Network Architecture for Commit Message

CNN was first used to automatically learn the salient features in the images from raw pixel values [115]. Recently, CNN has also generated multiple breakthroughs in various Natural Language Processing (NLP) applications [105, 54, 95, 236, 90]. The architecture of CNN allows it to extract the structural information features from the raw text data of a word embedding. Figure 4.4 presents an architecture of CNN for commit messages. The architecture includes a convolutional layer with multiple filters and a nonlinear activation function (i.e., RELU). We briefly explain it in the following paragraphs.

Given a commit message $\mathbf{m}$, which is essentially a sequence of words $[w_1, \ldots, w_{|m|}]$, we aim to obtain its matrix representation $\mathbf{m} \rightarrow \mathbf{M} \in \mathbb{R}^{|m| \times d_m}$, where the matrix $\mathbf{M}$ comprises a set of words $w_i \rightarrow W_i$, $i = 1, \ldots, |m|$ in the given commit message. Each word w_i now is represented by an embedding vector, i.e., $W_i \in \mathbb{R}^{d_m}$, where d_m is a d_m-dimensional matrix of a word appearing in the commit message.

Following previous works [105, 236], the d_m-dimensional embedding vector is extracted from an embedding matrix that is randomly initialized and jointly learned during the training process. Hence, the matrix representation $\mathbf{M}$ of the commit message $\mathbf{m}$ with a sequence of $|m|$ words can be represented as follows:

$$\mathbf{M} = [W_1, \ldots, W_{|m|}] \tag{4.3}$$

For the purpose of parallelization, all commit messages are padded or truncated to the same number of words $|m|$.

To extract the commit message's salient features, a filter $f \in \mathbb{R}^{k \times d_m}$, followed by a non-linear activation function $\alpha(.)$, is applied to a window of k

words to produce a new feature as follows:

$$c_i = \alpha(f * M_{i:i+k-1} + b_i) \tag{4.4}$$

where $*$ is the sum of the element-wise product, and $b_i \in \mathbb{R}$ is the bias value. We choose the rectified linear unit (RELU) as our activation function since it has been found to achieve a better performance compared to other activation functions [160, 48, 72]. The filter f is applied to every k-words of the commit message, these results of this process are then concatenated to product output vector $\mathbf{c}$ such that:

$$\mathbf{c} = [c_1, \ldots, c_{|m|-k+1}] \tag{4.5}$$

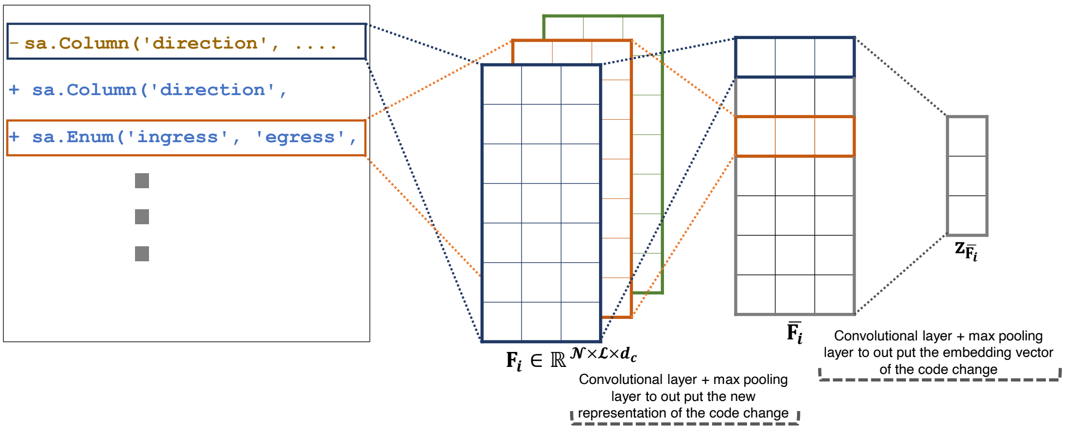
By applying the filter f on every k-words of the commit message, the CNN is able to exploit the semantic information of its input. In practice, the CNN model may include multiple filters with different k. These hyperparameters need to be set by the user before starting the training process. To characterize the commit message, we apply a max pooling operation [123] over the output vector $\mathbf{c}$ to obtain the highest value as follows:

$$\underset{1 \leq i \leq |m|-k+1}{max} \quad c_i \tag{4.6}$$

The results of the max pooling operation from each filter are then used to form an embedding vector (i.e., $\mathbf{z_m}$) of the commit message (see Figure 4.3).

4.3.4 Convolutional Network Architecture for Code Changes

In this section, we focus on building convolutional networks for code changes to solve the *Just-In-Time* defect prediction problem. A code change, although it can be viewed as a sequence of words, differs from natural language mainly because of its structure. Natural language carries sequences of words, and the semantics of a natural language sequence of words can be inferred from



```
- sa.Column('direction', ....
+ sa.Column('direction',
+ sa.Enum('ingress', 'egress',
```

$\mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$
Convolutional layer + max pooling layer to out put the new representation of the code change
$\overline{\mathbf{F}}_i$
Convolutional layer + max pooling layer to out put the embedding vector of the code change
$\mathbf{z}_{\overline{\mathbf{F}}_i}$

Figure 4.5: The overall structure of convolutional neural network for each change file in code change. The first convolutional and pooling layers use to learn the semantic features of each added or removed code line based on the words within the added or removed line, and the subsequent convolutional and pooling layers aim to learn the interactions between added or removed code lines with respect to the code change structure. The output of the convolutional neural network is the embedding vector $\mathbf{z}_{\overline{\mathbf{F}}_i}$ representing the features of the each changed.

a bag of words [162]. On the other hand, a code change includes changes in different files and different kinds of changes (removals or additions) for each file. Hence, to extract salient features from the code changes, the convolutional networks should obey the code changes structure. Based on the aforementioned considerations, we propose a deep learning framework for extracting features from code changes based on convolutional neural networks.

Given a code change $\mathcal{C}$ including a change in different source code files $[\mathrm{F}_1, \ldots, \mathrm{F}_n]$, where n is a number of files in the code change, we aim to extract features for each different file F_i. The features of each file are then concatenated to each other to represent the features for the given code change. In the rest of this section, we explain how the convolutional networks can extract the features for each file in the code change and how these features are concatenated.

Suppose F_i represents a change in each diff file. F_i contains a number of lines (removals or additions) in a code change file. We also have a sequence of words in each line in F_i. As described in Section 4.3.3, we first aim to obtain the matrix representation $\mathrm{F}_i \rightarrow \mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$, where $\mathcal{N}$ is the number of lines in a code change file, $\mathcal{L}$ presents a sequence of words in each line, and

d_c is a number of dimension of a word appearing in the F_i. For the purposed of parallelization, all the source code changes are padded or truncated to the same $\mathcal{N}$ and $\mathcal{L}$.

For each line $\mathcal{N}_i \in \mathbb{R}^{\mathcal{L} \times d_c}$, we follow the convolutional network architecture for a commit message described in Section 4.3.3 to extract an embedding vector, $\mathbf{z}_{\mathcal{N}_i}$. The embedding vector $\mathbf{z}_{\mathcal{N}_i}$ represents the features or the semantic of a code line based on the words within the code line. These features $\mathbf{z}_{\mathcal{N}_i}$ are then stacked to produce the new representation of the code change file F_i as follows:

$$\overline{\mathbf{F}}_i = [\mathbf{z}_{\mathcal{N}_1}, \ldots, \mathbf{z}_{\mathcal{N}_{|\mathcal{N}|}}] \tag{4.7}$$

We again apply the convolutional layer and pooling layer on the new representation of the code change (i.e., $\overline{\mathbf{F}}_i$) to extract its embedding vector, namely $\mathbf{z}_{\overline{\mathbf{F}}_i}$. The vector $\mathbf{z}_{\overline{\mathbf{F}}_i}$ represents the features or the semantics conveyed by the interactions between deleted or added lines. Figure 4.5 presents the overall convolutional network architecture for each change file F_i in code changes. The first convolutional and pooling layers aim to learn a new representation of the file, and the subsequent convolutional and pooling layers aim to extract the salient features from the new representation of the change file.

For each change file $\mathrm{F}_i \in C$, we build its embedding vector $\mathbf{z}_{\overline{\mathbf{F}}_i}$. These embedding vectors are then concatenated to build a new embedding vector representing the salient features of the code change C as follows:

$$\mathbf{z}_C = \mathbf{z}_{\overline{\mathbf{F}}_1} \oplus \cdots \oplus \mathbf{z}_{\overline{\mathbf{F}}_n} \tag{4.8}$$

where $\oplus$ is the concatenation operator.

4.3.5 Feature Combination

Figure 4.6 shows the details of the architecture of the feature combination. The inputs of this architecture are the two embedding vectors $\mathbf{z}_{\mathbf{m}}$ and $\mathbf{z}_C$

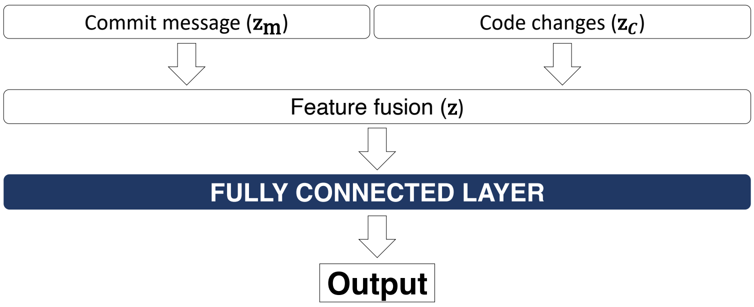

Commit message ($\mathbf{z_m}$)
Code changes ($\mathbf{z}_C$)
Feature fusion (z)
FULLY CONNECTED LAYER
Output

Figure 4.6: The structure of our fully-connected network for feature combination. The embedding vector of the commit message $\mathbf{z_m}$ and the code change $\mathbf{z}_C$ are concatenated to generate a single vector (i.e., $\mathbf{z}$).

which represent the salient features extracted from the commit message and the code change, respectively.

These vectors are then concatenated to generate a unified feature representation, i.e., a new vector ($\mathbf{z}$), representing the commit change:

$$\mathbf{z} = \mathbf{z_m} \oplus \mathbf{z}_C \tag{4.9}$$

The new vector then feed into a fully-connected (FC) layer, which outputs a vector $\mathbf{h}$ as follows:

$$\mathbf{h} = \alpha(\mathbf{w_h} \cdot \mathbf{z} + b_{\mathbf{h}}) \tag{4.10}$$

where $\cdot$ is a dot product, $\mathbf{w}_h$ is a weight matrix of the vector $\mathbf{h}$ and the FC layer, $b_{\mathbf{h}}$ is the bias value, and $\alpha(\cdot)$ is the RELU activation function. The vector $\mathbf{h}$ is passed to an output layer to compute a probability score for a given commit:

Finally, the vector $\mathbf{h}$ is passed to an output layer, which computes a probability score for a given patch:

$$p(y_i = 1 | x_i) = \frac{1}{1 + \exp(-\mathbf{h} \cdot \mathbf{w_o})} \tag{4.11}$$

where $\mathbf{w_o}$ is the weight matrix between the FC layer and the output layer.

4.3.6 Parameter Learning

In the training process, DeepJIT aims to learn the following parameters: the word embedding matrices of the commit message and the commit code in a given commit, the convolutional layer matrices, the weights and bias of the fully connected layer, and the output layer.

Just-In-Time defect prediction datasets often suffer from the imbalance problem: only a few commits contain a buggy code while a large number of commits are clean. This imbalance increases the difficulty in learning a prediction function [42]. Specifically, the imbalance problem may affect the performance of a defect prediction model as the overall accuracy is biased to the majority class (e.g., commits containing buggy code), leading to misclassification of the minority class. Inspired by Zhou and Liu [239] and Kukar et al. [116], we propose an unequal misclassification loss function that specifically aims to reduce the negative influence of the imbalanced data. Unlike traditional methods, this "cost-senstive" learning technique does *not* treat all misclassifications equally. As our datasets is imbalance, we impose a higher cost on misclassifications of the minority class (i.e., buggy commits) than we do with misclassifications of the majority class (i.e., clean commits). Details of this technique is as follows.

Let $\mathbf{w_n}$ and $\mathbf{w_p}$ denote the cost of incorrectly associating a commit change and the cost of missing a buggy commit change, respectively. The parameters of DeepJIT can be learned by minimizing the following objective function:

$$\begin{aligned}\mathcal{O} &= -\log\left(\prod_{i=1} p(y_i|x_i)\right) + \frac{\lambda}{2}\|\theta\|_2^2 \\ &= -\sum_{i=1}[\mathbf{w_n}(1-y_i)\log(1-p(y_i|x_i)) \\ &\quad + \mathbf{w_p} y_i \log(p(y_i|x_i))] + \frac{\lambda}{2}\|\theta\|_2^2\end{aligned} \tag{4.12}$$

where $p(y_i|x_i)$ is the probability score from the output layer and θ contains all parameters our model. The term $\frac{\lambda}{2}\|\theta\|_2^2$ is used to mitigate data overfitting in training deep neural networks [38]. We also apply the dropout technique [195] to improve the robustness of our model.

We use Adam [107], which is a variant of stochastic gradient descent (SGD) [34], to minimize the objective function in the equation 4.12. We choose Adam due to its computational efficiency and low memory requirements compared to other optimization techniques [107, 17, 19]. To efficiently compute the gradients in linear time (with respect to the neural network size), we use back-propagation [68], which is a simple implementation of the chain rule of partial derivatives.

4.4 Experiments

In this section, we first describe the dataset used in our experiments. We then introduce all baselines and the evaluation metric. Finally, we present our research questions and results.

4.4.1 Dataset

We used two well-known software projects (i.e., QT and OPENSTACK) to evaluate the performance of *Just-In-Time* (JIT) models. QT,[4] developed by the Qt Company, is a cross-platform application framework and allows contributions from individual developers and organizations. OPENSTACK [5] is an open-source software platform for cloud computing and is deployed as an infrastructure-as-a-service which allows customers to access its resources.

Table 4.1: Summary of the dataset used in this work

Dataset	**Timespan**		**Commits**	
	Start	**End**	**Total**	**Defective**
QT	06/2011	03/2014	25,150	2,002 (8%)
OPENSTACK	11/2011	02/2014	12,374	1,616 (13%)

[4] https://www.qt.io/

[5] https://www.openstack.org/

Table 4.1 briefly summarizes the dataset. This dataset was originally collected and cleaned by McIntosh and Kamei [151] for *Just-In-Time* defect prediction. After their cleaning process, the QT dataset contains 25,150 commits, while the OPENSTACK dataset contains 12,374 commits. McIntosh and Kamei stratified the dataset into six month periods for time-sensitive training-and-testing settings.

4.4.2 Baselines

We compared DeepJIT with two state-of-the-art baselines for *Just-In-Time* (JIT) defect prediction:

- JIT: This method for identifying buggy code changes was proposed by McIntosh and Kamei [151]. The method used a nonlinear variant of multiple regression modeling [61] to build a classification model for automatically identifying defects in commits. McIntosh and Kamei manually designed a set of code features, using six families of code change properties, which were primarily derived from prior studies [99, 103, 112, 156]. These properties were: the magnitude of changes, the dispersion of the changes, the defect proneness of prior changes, the experience of the author, the code reviewers, and the degree of participation in the code review. Table 4.2 summarizes the code features extracted from code change properties.

- DBNJIT: This approach adopted Deep Belief Network (DBN) [76] to generate a more expressive set of features from an initial feature set [228]. The generated feature set, which is a nonlinear combination of the initial features, was put into a machine learning classifier [31] to predict buggy commits. For a fair comparison, we used McIntosh and Kamei [151]'s features as the initial feature set for DBNJIT.

For all the above-mentioned techniques, we employ the same parameters

Table 4.2: A summary of McIntosh and Kamei's code features [151].

	Property	**Description**	**Rationale**
Size	Lines deleted	The number of deleted lines.	The more deleted or added code, the more likely that defects may appear [159, 97].
	Lines added	The number of added lines.	
Diffusion	Subsystems	The number of modified subsystems.	Scattered changes may have more defects compared to focused one [51, 70].
	Directories	The number of modified directories.	
	Files	The number of modified files.	
	Entropy	The spread of modified lines across file.	
History	Unique changes	The number of prior changes to the modified files.	More changes may lead to have defects since developers need to track many previous changes [99].
	Developers	The number of developers who have changed the modified files in the past.	Files touched by many developers may include defects [150].
	Age	The time interval between the last and current changes.	More recently changed code likely contains defects compared to older code [65].
Author/Rev. Experience	Prior changes	The number of prior changes that an actor has participated in.	Changes produced by novices are likely to be more defective than changes produced by experienced developers [156].
	Recent changes	The number of prior changes that an actor has participated in weighted by the age of the changes (older changes are given less weight than recent ones).	
	Subsystem changes	The number of prior changes to the modified subsystem(s) that an actor has participated in.	
	Awareness	The proportion of the prior changes to the modified subsystem(s) that an actor has participated in.	Changes made by developers who are aware of the prior changes in the impacted subsystems are likely to be less risky.
Review	Iterations	The number of times that a change was revised prior to integration.	The quality of a change likely improves with each iteration. Hence, changes that undergo iterations prior to integration may be less risky [174, 202].
	Reviewers	The mumber of reviewers who have voted on whether a change should be integrated or abandoned.	Changes observed by many reviewers are likely to be less risky [183].
	Comments	The number of non-automated, non-owner comments posted during the review of a change.	Changes with short discussions may be more risky [152, 153].
	Review window	The length of time between the creation of a review request and its final approval for integration.	Changes with shorter review windows may be more risky [174, 202].

and settings as described in the respective papers.

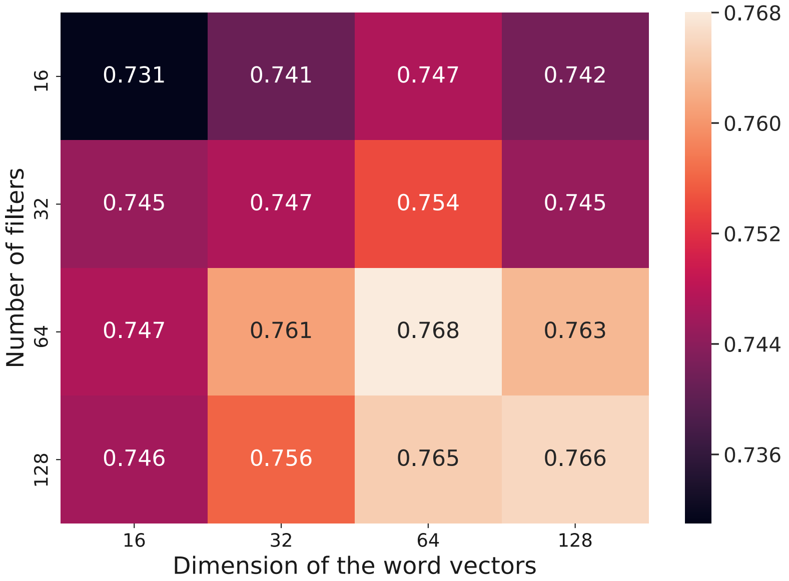


Figure 4.7: The AUC results of DeepJIT across two different hyperparameters in QT project.

4.4.3 Evaluation Metric

To evaluate the accuracy of *Just-In-Time* (JIT) models, we calculate threshold-independent measures of model performance. Since our dataset is imbalanced, we avoid using threshold-dependent measures (i.e., precision, recall, or F1) since these measures strongly depend on arbitrary thresholds [164, 66]. Following the previous work by McIntosh and Kamei [151], we use the Area Under the receiver operator characteristics Curve (AUC) to measure the discriminatory power of DeepJIT, i.e., its ability to differentiate between defective or clean commits. AUC computes the area under the curve plotting the true positive rate against the false positive rate, while applying multiple thresholds to determine if a commit is buggy or not. The values of AUC range between 0 (worst discrimination) and 1 (perfect discrimination).

4.4.4 Training and hyperparameters

One of the key challenges in training DeepJIT is how to select the dimensions of the word vectors for the commit message (d_m) and code changes (d_c), and the size of the convolution layers (i.e., see Section 4.3.3 and Section 4.3.4). We

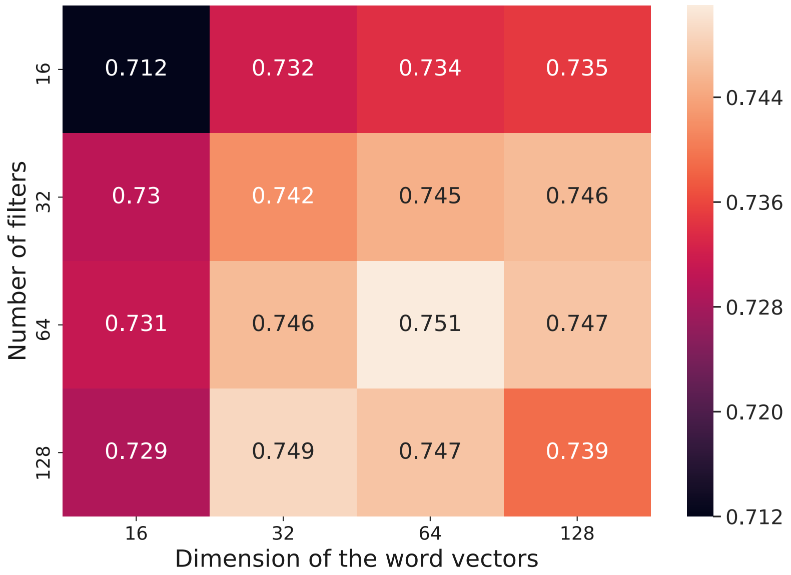


Figure 4.8: The AUC results of DeepJIT across two different hyperparameters in OPENSTACK project.

evaluated the performance of DeepJIT, using 5-fold cross validation, across different word dimensions and number of filters. Figure 4.7 and Figure 4.8 present the AUC results of DeepJIT for these hyperparameters. The figures show that DeepJIT achieves the best AUC results when the dimension of word vectors and the number of filters are set to 64. We set the other hyperparameters as follows: The batch size was set to 32. The size of DeepJIT's fully-connected layer described in Section 4.3.5 was set to 512. These hyperparameter settings are commonly used in prior deep learning work [188, 84, 83, 77].

We trained DeepJIT using the Adam method [107] with shuffled mini-batches. We also trained DeepJIT for 100 epochs. We applied an early stopping strategy [176, 38] to avoid overfitting during the training process. We stopped the training if the value of the objective function (see Equation 5.12) has not been updated in the last 5 epochs.

4.4.5 Research Questions and Results

We evaluated the accuracy of a trained JIT model in predicting buggy changes using test data. In particular, we considered three evaluation settings:

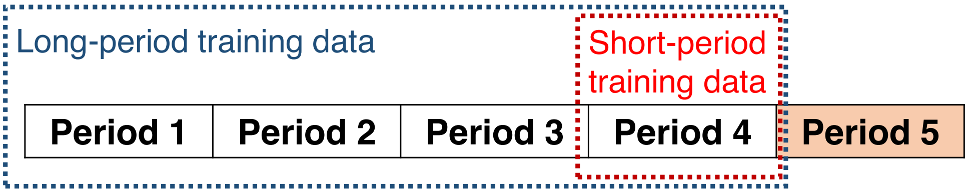


Figure 4.9: An example of choosing the training data for short-period and long-period models. The last period is used as testing data.

- **Cross-validation:** To evaluate the machine learning algorithm, most people use k-fold cross-validation [111] in which a dataset is randomly divided to k folds, and each fold is considered as testing data for evaluating JIT model while $k-1$ folds are considered as training data. In this case, the JIT model is trained on a mixture of past and future data. In our experiments, we set $k = 5$.

- **Short-period:** The JIT model is trained using commits that occurred at one time period. We assume that older commit changes have characteristics that are different from those of the latest commits.

- **Long-period:** Inspired by Rahman et al. [178], suggesting that larger amounts of training data tend to achieve better performance in defect prediction problems, we train the JIT model using all commits that occurred before a particular period. We discover whether additional data may improve the performance of the JIT model.

Figure 4.9 describes how the training data is selected to train models following the short-period and long-period settings. We used the last period (i.e., period 5) as the testing data. While the short-period model was trained using the commits that occurred during period 4, the long-period model was trained using the commits that occurred from period 1 to 4. After training the short-period and long-period models, we measured their performance using the AUC evaluation metric described in Section 4.4.3.

RQ1: How effective is DeepJIT compared to the state-of-the-art baseline?

Table 4.3: The AUC results of DeepJIT vs. with other baselines in three types of JIT models: cross-validation, short-period, and long-period.

Settings	**Models**	QT	OPENSTACK
Cross-validation	JIT	0.701	0.691
	DBNJIT	0.705	0.694
	DeepJIT	**0.768**	**0.751**
Short-Period	JIT	0.703	0.711
	DBNJIT	0.714	0.716
	DeepJIT	**0.764**	**0.781**
Long-period	JIT	0.702	0.706
	DBNJIT	0.708	0.712
	DeepJIT	**0.765**	**0.771**

Table 4.3 shows the AUC results of DeepJIT as well as baselines considering the three evaluation settings: cross-validation, short-period, and long-period. The difference between the results obtained using cross-validation, short-period, and long-period settings is relatively small (i.e., below 2.2%) which suggests that there is no difference between training on past or future data. In the QT project, DeepJIT achieved AUC scores of 0.768, 0.764, and 0.765 in three different evaluation settings: cross-validation, short-period, and long-period, respectively. We compare DeepJIT to the best performing baseline (i.e., DBNJIT), DeepJIT achieved improvements of 8.96%, 7.00%, and 8.05% in terms of AUC. In the OPENSTACK project, DeepJIT also achieved improvements of 8.21%, 9.08%, and 8.29% in terms of AUC compared to DBNJIT (the best performing baseline). We also employed the Scott-Knott test [63] on the cross-validation evaluation setting to statistically compare the differences between the three considered JIT models. The results show that DeepJIT consistently appears in the top Scott-Knott ESD rank in terms of AUC (i.e, DeepJIT > DBNJIT > JIT).

RQ2: Does the proposed model benefit from both the commit message and the code changes?

To answer this question, we employed an ablation test [113, 138], by ignoring the commit message and the code change in a commit and then evaluate the AUC . Specifically, we created two different variants of DeepJIT, namely

Table 4.4: Contribution of feature components in DeepJIT.

Settings	**Models**	QT	OPENSTACK
Cross-validation	DeepJIT-Msg	0.641	0.689
	DeepJIT-Code	0.738	0.729
	DeepJIT	**0.768**	**0.751**
Short-Period	DeepJIT-Msg	0.609	0.583
	DeepJIT-Code	0.734	0.769
	DeepJIT	**0.764**	**0.781**
Long-period	DeepJIT-Msg	0.638	0.659
	DeepJIT-Code	0.727	0.738
	DeepJIT	**0.765**	**0.771**

DeepJIT-Msg and DeepJIT-Code. DeepJIT-Msg only considers commit message information while DeepJIT-Code only uses commit code information. We again used the three evaluation settings (i.e., cross-validation, short-period, and long-period) and the AUC scores to evaluate the performance of our models. Table 4.4 shows that the performance of DeepJIT degrades if we ignore either of the considered types of information (i.e. commit messages or code changes). The AUC scores dropped by 19.81%, 28.45%, and 19.01% in the project QT and by 9.00%, 33.96%, and 16.00% in the project OPENSTACK for the three evaluation settings if we the ignore commit messages. The AUC scores dropped by 4.07%, 4.09%, and 5.23% in the project QT and by 3.02%, 1.56%, and 4.47% in the project OPENSTACK for the three evaluation settings if we ignore the code change information. It suggests that each information type contributes to DeepJIT's performance. Moreover, it also indicates that code changes are more important to detect buggy commits than commit messages.

RQ3: Does DeepJIT benefit from the manually extracted code changes features?

To address this question, we incorporated the code features, derived by McIntosh and Kamei et al. [151], into our proposed model. Specifically, the code features, namely $\mathbf{z_r}$, are concatenated with the two embedding vectors $\mathbf{z_m}$ and $\mathbf{z}_C$, representing the features of the commit message and code change

Table 4.5: Combination of DeepJIT with the manually crafted code features extracted by McIntosh and Kamei et al. [151].

Settings	**Models**	QT	OPENSTACK
Cross-validation	DeepJIT	0.768	0.751
	DeepJIT-Combined	**0.779**	**0.76**
Short-Period	DeepJIT	0.764	0.781
	DeepJIT-Combined	**0.788**	**0.814**
Long-period	DeepJIT	0.765	0.771
	DeepJIT-Combined	**0.786**	**0.799**

Table 4.6: Training time of DeepJIT

Dataset	Cross-validation	Short-period	Long-period
QT	5 hours 43 mins	17.2 mins	1 hours 18 mins
OPENSTACK	12 hours 15 mins	10.1 mins	2 hours 37 mins

(see Section 4.3.5), to build a new single vector **z** as follows:

$$\mathbf{z} = \mathbf{z_m} \oplus \mathbf{z}_C \oplus \mathbf{z_r} \tag{4.13}$$

where $\oplus$ is the concatenation operator. Table 4.5 shows the AUC results of a DeepJIT variant (referred to as DeepJIT-Combined) that also leverages McIntosh and Kamei [151]'s manually crafted features. We find that the AUC scores increased by 1.43%, 3.14%, and 2.75% in the project QT and by 1.20%, 4.23%, and 3.63% in the project OPENSTACK for the three evaluation settings (i.e. cross-validation, short-period, long-period). DeepJIT-Combined improved the best baseline model (i.e. DBNJIT) by 10.50%, 10.36%, and 11.02% in the project QT and by 9.51%, 13.69%, 12.22% in the project OPENSTACK for the there evaluation settings. This suggests that the manually extracted code features are complementary and can be used to slightly improve the performance of our proposed approach.

RQ4: What are the time costs of DeepJIT?

We trained and tested DeepJIT on an NVIDIA DGX1 server with Tesla P100 [62]. Table 4.6 shows the time cost of training DeepJIT for the three evaluation settings (i.e., cross-validation, short-period, and long-period) on the QT and OPENSTACK. The cross-validation setting requires the longest training time since we performed 5-fold cross-validation to evaluate the performance of DeepJIT. The long-period setting requires more training time than the short-period setting since it considers all the commits occurring before a particular period. Once DeepJIT has been trained, it only takes a few milliseconds to generate the prediction score for a given commit.

4.5 Threats to Validity

We mitigated concerns related to construct validity by evaluating our approach on a publicly available dataset that has been used in previous work. This dataset contains commits extracted from real projects (QT and OPENSTACK) and buggy/no-bug labels on those commits. Threats to conclusion validity were also minimized by using Area Under the Curve (AUC), a standard performance measure recommended for assessing the predictive performance of defect prediction models [198].

We have compared our approach against two baselines which have been proposed and implemented in existing work. Since the source code of their original implementations were not made publicly available, we needed to re-implement our own versions of those techniques. Although our implementation closely follows the description of their work, it might not have all of the details of the original implementation, specifically those not explicitly presented in their papers. Our study considers two large open source projects which are significantly different in size, complexity and revision history. However, due to small sample sizes, our findings may not generalize to all software projects. Further studies are needed to confirm our results for other types of software projects.

4.6 Chapter Summary

In this chapter, we propose an end-to-end deep learning model (namely DeepJIT) for *Just-In-Time* defect prediction problem. For a given commit, DeepJIT automatically extracts features from the commit message and the set of code changes. These features are then combined to evaluate how likely the commit is buggy. DeepJIT also allows users to add their manually crafted features to make it more robust. We evaluate DeepJIT on two popular software projects (i.e. QT and OPENSTACK) on three evaluation settings (i.e. cross-validation, short-period, and long-period). The evaluation results show that compared to the best performing state-of-the-art baseline (DBNJIT), the best variant of DeepJIT (DeepJIT-Combined) achieves improvements of 10.50%, 10.36%, and 11.02% in the project QT and 9.51%, 13.69%, 12.22% in the project OPENSTACK in terms of the Area Under the Curve (AUC).

Chapter 5

Hierarchical Deep Learning-Based Stable Patch Identification

Linux kernel stable versions serve the needs of users who value stability of the kernel over new features. The quality of such stable versions depends on the initiative of kernel developers and maintainers to propagate bug fixing patches to the stable versions. Thus, it is desirable to consider to what extent this process can be automated. A previous approach relies on words from commit messages and a small set of manually constructed code features. This approach, however, shows only moderate accuracy. In this chapter, we investigate whether deep learning can provide a more accurate solution. We propose PatchNet, a hierarchical deep learning-based approach capable of automatically extracting features from commit messages and commit code and using them to identify stable patches. Unlike DeepJIT which simply merges the removed and added code in the code changes, PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of the removed and added code, making it distinctive from the existing deep learning models on source code. Experiments on 82,403 recent Linux patches confirm the superiority of PatchNet against various state-of-the-art baselines, including the one recently-adopted by Linux kernel maintainers.

5.1 Introduction

The Linux kernel follows a two-tiered release model in which a *mainline* version, accepting bug fixes and feature enhancements, is paralleled by a series of stable versions that accept only bug fixes [125]. The mainline serves the needs of users who want to take advantage of the latest features, while the stable versions serve the needs of users who value stability, or cannot upgrade their kernel due to hardware and software dependencies. To ensure that there is as much review as possible of the bug fixing patches and to ensure the highest quality of the mainline itself, the Linux kernel requires that all patches applied to the stable versions pass through the mainline first. A mainline subsystem developer or maintainer may identify a patch as a bug fixing patch appropriate for stable kernels and add to the commit message a Cc: stable tag (stable@vger.kernel.org). Stable-kernel maintainers then extract such annotated commits from the mainline commit history and apply the resulting patches to the stable versions that are affected by the bug.

A patch consists of a commit message followed by the code changes, expressed as a unified diff [146]. The diff consists of a series of changes (removed and added lines of code), separated by lines beginning with `@@` indicating the number of the line in the affected source file at which the subsequent change should be applied. Each block of code starting with an `@@` line is referred to as a *hunk*. Fig. 5.1 shows three patches to the Linux kernel. The first patch changes various return values of the function `csum_tree_block`. The commit message is on lines 1-10 and the code changes are on lines 11-25. The code changes consist of multiple hunks, only the first of which is shown in detail (lines 15-23). In the shown hunk, the function called just previously to the

```
commit 342da5cefddbf818e1cb59537e021cdad9744e93
Author: Alex Lyakas <...>
Date:   Thu Mar 10 13:09:46 2016 +0200

    btrfs: csum_tree_block: return proper errno value

    commit 8bd98f0e6bf792e8fa7c3fed709321ad42ba8d2e upstream.

    Signed-off-by: Alex Lyakas <...>
    Reviewed-by: Filipe Manana <...>
    Signed-off-by: David Sterba <...>
    Signed-off-by: Greg Kroah-Hartman <...>

diff --git a/fs/btrfs/disk-io.c b/fs/btrfs/disk-io.c
index d8d68af..87946c6 100644
--- a/fs/btrfs/disk-io.c
+++ b/fs/btrfs/disk-io.c
@@ -303,7 +303,7 @@ static int csum_tree_block(struct btrfs_fs_info *fs_info,
                err = map_private_extent_buffer(buf, offset, 32,
                                        &kaddr, &map_start, &map_len);
                if (err)
-                       return 1;
+                       return err;
                cur_len = min(len, map_len - (offset - map_start));
                crc = btrfs_csum_data(kaddr + offset - map_start,
                                      crc, cur_len);
@@ -313,7 +313,7 @@ static int csum_tree_block(struct btrfs_fs_info *fs_info,
...
```

(a) A fix of a bug that can impact the user level.

```
commit 7b0692f1c60a9551f8ad5fe706b79a23720a196c
Author: Andy Shevchenko <...>
Date:   Wed Aug 14 11:07:11 2013 +0300

    HID: hid-sensor-hub: change kmalloc + memcpy by kmemdup

    The patch substitutes kmemdup for kmalloc followed by memcpy.

    Signed-off-by: Andy Shevchenko <...>
    Acked-by: Srinivas Pandruvada <...>
    Signed-off-by: Jiri Kosina <...>

diff --git a/drivers/hid/hid-sensor-hub.c b/drivers/hid/hid-sensor-hub.c
index 1877a2552483..e46e0134b0f9 100644
--- a/drivers/hid/hid-sensor-hub.c
+++ b/drivers/hid/hid-sensor-hub.c
@@ -430,11 +430,10 @@ static int sensor_hub_raw_event(struct hid_device *hdev,
                        ...
-                       pdata->pending.raw_data = kmalloc(sz, GFP_ATOMIC);
-                       if (pdata->pending.raw_data) {
-                               memcpy(pdata->pending.raw_data, ptr, sz);
+                       pdata->pending.raw_data = kmemdup(ptr, sz, GFP_ATOMIC);
+                       if (pdata->pending.raw_data)
                                pdata->pending.raw_size = sz;
-                       } else
+                       else
                                pdata->pending.raw_size = 0;
                        ...
```

(b) A refactoring.

```
commit: 501bcbd1b233edc160d0c770c03747a1c4aa14e5
Author: Thierry Reding <...>
Date:   Wed Apr 14 09:52:31 2014 +0200

    drm/tegra: dc - Do not touch power control register

    Setting the bits in this register is dependent on the output type driven
    by the display controller. All output drivers already set these properly
    so there is no need to do it here again.

    Signed-off-by: Thierry Reding <...>

diff --git a/drivers/gpu/drm/tegra/dc.c b/drivers/gpu/drm/tegra/dc.c
index 8b21e20..33e03a6 100644
--- a/drivers/gpu/drm/tegra/dc.c
+++ b/drivers/gpu/drm/tegra/dc.c
@@ -743,10 +743,6 @@ static void tegra_crtc_prepare(struct drm_crtc *crtc)
            WIN_A_OF_INT | WIN_B_OF_INT | WIN_C_OF_INT;
        tegra_dc_writel(dc, value, DC_CMD_INT_POLARITY);
-       value = PW0_ENABLE | PW1_ENABLE | PW2_ENABLE | PW3_ENABLE |
-           PW4_ENABLE | PM0_ENABLE | PM1_ENABLE;
-       tegra_dc_writel(dc, value, DC_CMD_DISPLAY_POWER_CONTROL);
        /* initialize timer */
        value = CURSOR_THRESHOLD(0) | WINDOW_A_THRESHOLD(0x20) |
            WINDOW_B_THRESHOLD(0x20) | WINDOW_C_THRESHOLD(0x20);
```

(c) A fix of a minor performance bug.

Figure 5.1: Example patches to the Linux kernel.

return site, `map_private_extent_buffer` (line 16), can return either 1 or a negative value in case of an error. So that the user can correctly understand the reason for any failure, it is important to propagate such return values up the call chain. The patch thus changes the return value of `csum_tree_block` in this case from `1` to the value returned by the `map_private_extent_buffer` call. The remaining hunks contain similar changes. The Linux kernel documentation [59] stipulates that a patch should be applied to stable kernels if it fixes a real bug that can affect the user level and satisfies a number of criteria, such as containing fewer than 100 lines of code and being obviously correct. This patch fits those criteria. The patch was first included in the Linux mainline version v4.6, and was additionally applied to the stable version derived from the mainline release v4.5, first appearing in v4.5.5 (the fifth release based on Linux v4.5) as commit 342da5cefddb.

The remaining patches in Fig. 5.1 should not be propagated to stable kernels. The patch in Fig. 5.1b performs a refactoring, replacing some lines of code by a function call that has the same behavior. As the behavior is unchanged, there is no impact on the user level. The patch in Fig. 5.1c addresses a minor performance bug, in that it removes some code that performs a redundant operation. The performance improvement should not be noticeable at the user level, and thus this patch is not worth propagating to stable kernels. Note that none of the patches shown in Fig. 5.1 contains keywords such as “bug” or “fix”, or links to a bug tracking system. Instead, the stable kernel maintainer has to study the commit message and the code changes, to understand the impact of the changes on the kernel code.

As patches for stable kernels contain fixes for bugs that can impact the user level, the quality of the stable kernels critically relies on the effort that the developers and subsystem maintainers put into identifying and labeling such patches, which we refer to as *stable patches*. This manual effort represents a potential weak point in the Linux kernel development process, as the developers and maintainers may forget to label some relevant patches, and apply different criteria for selecting them. While the stable-kernel maintainers can themselves additionally pick up relevant patches from the mainline commits, there are hundreds of mainline commits per day, and many will likely slip past. This task can thus benefit from automated assistance.

One way to provide such automated assistance is to build a tool that learns from historical data how to differentiate stable from non-stable patches. However, building such a tool poses some challenges. First, a patch contains both a commit message (in natural language) and some code changes. While the commit message is a sequence of words, and is thus amenable to existing approaches on classifying text, the code changes have a more complex structure. Indeed, a single patch may include changes to multiple files, the changes in each file consist of a number of hunks, and each hunk contains zero or more removed and added code lines. As the structure of the commit message and code changes differs, there is a need to extract their features separately. Second, the historical information is noisy since stable kernels do not receive only bug fixing patches, but also patches adding new device identifiers and patches on which a subsequent bug fixing patches depends. Moreover, patches that should have been propagated to stable kernels may have been overlooked. Finally, as illustrated by Fig. 5.1c, there are some patches that perform bug fixes but should not be propagated to stable kernels for various reasons (e.g., lack of impact on the user level or complexity of the patch).

A first step in the direction of automatically identifying patches that should be applied to stable Linux kernels was proposed by Tian et al. [204] who combine LPU (Learning from Positive and Unlabeled Examples) [127] and SVM (Support Vector Machine) [196] to learn from historical information how to identify bug-fixing patches. Their approach relies on thousands of word features extracted from commit messages and 52 features extracted from code changes. The word features are obtained automatically by representing each commit message as a bag of words, *i.e.*, a multiset of the words found in the commit, whereas the code features are defined manually. The bag-of-words representation of the commit message implies that the temporal dependencies (ordering) of words in a commit message are ignored. The manual creation of code features might overlook features that are important to identify stable patches.

To address the limitations of the work of Tian et al. and to focus on stable patches, we propose a novel hierarchical representation learning architecture for patches, named PatchNet. Like the LPU+SVM work, PatchNet focuses on the commit message and code changes, as this information is easily available and stable-kernel maintainers have reported to us that they use one or both of these elements in assessing potential stable patches. Deviating from the previous LPU+SVM work, however, which requires human effort to construct code features, PatchNet aims to automatically learn two embedding vectors for representing the commit message and the set of code changes in a given patch, respectively. While the first embedding vector encodes the semantic information of the commit message to differentiate between similar commit messages and dissimilar ones, the latter embedding vector captures the sequential nature of the code changes in the given patch. The two embedding vectors are then used to compute a prediction score for a given patch, based on the similarity of the patch's vector representation to the information learned from other stable or non-stable patches. The key challenge is to accurately represent the structure of code changes, which are not contiguous text like the commit message, but rather amount to scattered fragments of removed and added code across multiple files, within multiple hunks. Thus, different from existing deep learning techniques working on source code [217, 83, 215, 117], PatchNet constructs separate embedding vectors representing the removed code and the added code in each hunk of each affected file in the given patch. The information about a file's hunks are then concatenated to build an embedding vector for the affected file. In turn, the embedding vectors of all the affected files are used to build the representation of the entire set of code changes in the given patch.

PatchNet has already attracted some industry attention. Inspired by the work of Tian et al. and by our work on PatchNet, the Linux kernel stable maintainer Sasha Levin has adopted a machine-learning based approach for identifying patches for stable kernels, which we use as a baseline for our evaluation (Section 5.4.3). Recently Wen et al. [216] of ZTE Corporation have also adapted PatchNet to the needs of their company. These works show the potential usefulness of PatchNet in an industrial setting.

The main contributions of this chapter include:

- We study the manual process of identifying patches for Linux stable versions. We explore the potential benefit of automatically identifying stable patches and summarize the challenges in using machine learning for this purpose.

- We propose a novel framework, PatchNet, to automatically learn a representation of a patch by considering both its commit message and corresponding code changes. PatchNet contains a novel deep learning model to construct an embedding vector for the code changes made by a patch, based on their sequential content and hierarchical structure. The two embedding vectors, representing the commit message and the set of code changes, are combined to predict whether a patch should be propagated to stable kernels.

- We evaluate PatchNet on a new dataset that contains 82,403 recent Linux patches. The results show the superiority of PatchNet compared to state-of-the-art baselines. PatchNet also achieves good performance on the complete set of Linux kernel patches.

5.2 Background

In this section, we present background information about the maintenance of Linux kernel stable versions, the potential benefits of introducing automation into the stable kernel maintenance process, and the challenges posed for automation via machine learning.

5.2.1 Context

The Linux kernel, developed by Linus Torvalds in 1991, is a free and open-source, monolithic, and Unix-like operating system kernel [143]. It has been deployed on both traditional computer systems, i.e., personal computers and servers, and on many embedded devices such as routers, wireless access points, smart TVs, etc. Many devices, i.e., tablet computers, smartphones, smartwatches, etc. that have the Android operating system also use the Linux kernel.

The Linux kernel includes a two tiered release model comprising a mainline version and a set of stable versions. The mainline version, often released every two to three months, is the version where all new features are introduced. After a mainline version is released, it is considered to be "stable". Any bug fixing patches for a stable version are backported from the mainline version.

Linux kernel development is carried out according to a hierarchical model, with Linus Torvalds—who has ultimate authority about which patches are accepted into the kernel—at the root and patch *authors* at the leaves. A patch author is anyone who wishes to make a contribution to the kernel, fix a bug, add a new functionality, or improve the coding style. Authors submit their patches by email to *maintainers*, who commit the changes to their git trees and submit pull requests up the hierarchy. In this work, we are mostly concerned with the maintainers, who are responsible for assessing the correctness and usefulness of the patches that they receive. Part of this responsibility involves determining

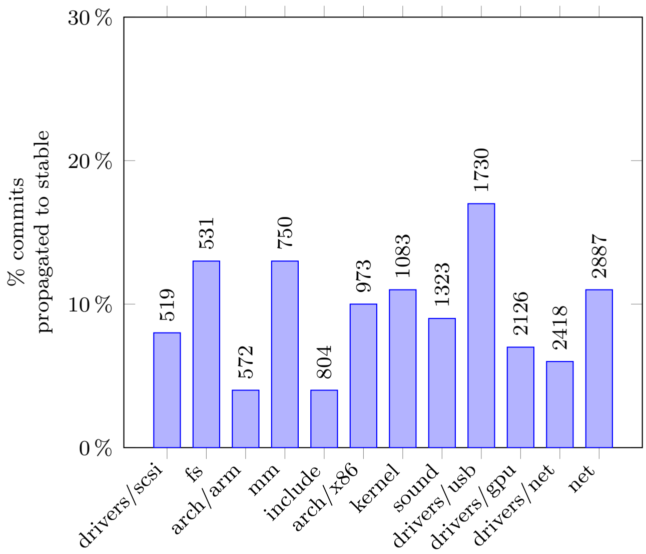


Figure 5.2: Percentage of mainline commits propagated to stable kernels for the 12 directories with more than 500 mainline commits being propagated to stable kernels between Linux v3.0 (July 2011) and Linux v4.12 (July 2017). The number above each bar indicates the number of propagated commits.

whether a patch is stable, and ensuring that it is annotated accordingly.

The Linux kernel provides a number of guidelines to help maintainers determine whether a patch should be annotated for propagation to stable kernels [59]. They are summarized as follows:

- It cannot be bigger than 100 lines.
- It must fix a problem that causes a build error, an oops, a hang, data corruption, a real security issue, or some “oh, that’s not good” issue.

These criteria may be simple, but are open to interpretation. For example, even the criterion about patch size, which seems unambiguous, is only satisfied by 93% of the patches found in the stable versions based on Linux v3.0 to v4.13, as of September 2017.

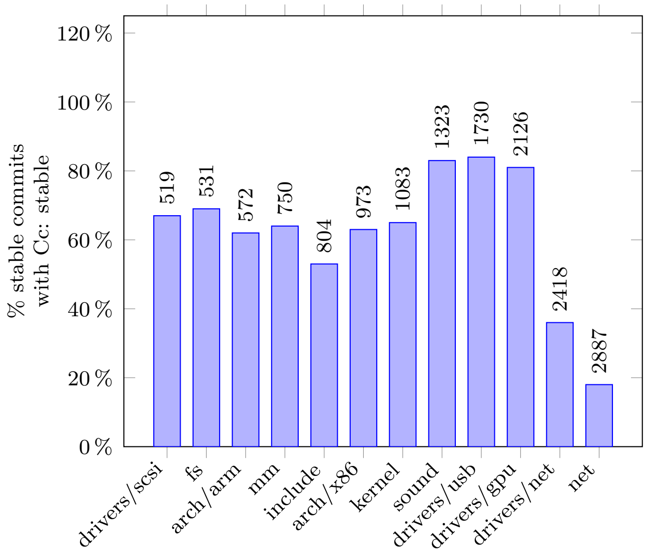


Figure 5.3: Percentage of mainline commits propagated to stable kernels that contain a Cc: stable tag for the 12 directories with more than 500 mainline commits being propagated to stable kernels between Linux v3.0 (July 2011) and Linux v4.12 (July 2017). The number above each bar indicates the number of propagated commits.

5.2.2 Potential Benefits of Automatically Identifying Stable Patches

To understand the potential benefit of automatically identifying stable patches, we examine the percentage of all mainline commits that are propagated to stable kernels across different kernel subsystems and the percentage of these that are annotated with the Cc: stable tag. We focus on the 12 directories for which more than 500 mainline commits were propagated to stable kernels between Linux v3.0 (July 2011) and Linux v4.12 (July 2017). Fig. 5.2 shows the percentage of all mainline commits that are propagated to stable kernels for these 12 directories. We observe that there is a large variation in these values. Comparing directories with similar purposes, 4% of `arch/arm` (ARM hardware support) commits are propagated, while 10% of `arch/x86` (x86 hardware support) commits are propagated, and 6-8% of the `scsi`, `gpu` and `net` driver commits are propagated, while 17% of `usb` driver commits are propagated.[1] If we make the assumption that the rate of bug introduction is roughly constant

[1] The `usb` driver maintainer is also a stable kernel maintainer.

across similar kinds of code, the wide variation in the propagation rates for similar kinds of code suggests that relevant commits may be being missed.

Fig. 5.3 shows the percentage of mainline commits propagated to stable kernels that contain the Cc: stable tag, for the same set of kernel directories. The rate is very low for `drivers/net` and `net`, which are documented to have their own procedure [59]. The others mostly range from 60% to 85%. Commits in stable kernels that do not contain the tag are commits that the stable kernel maintainers have identified on their own or that they have received via other non-standard channels. This represents work that can be saved by an automatic labeling approach.

5.2.3 Challenges for Machine Learning

Stable patch identification poses some unique challenges for machine learning. These include the kind of information available in a Linux kernel patch and the different reasons why patches are or are not selected for stable kernels.

First, patches contain a combination of text, represented by the commit message, and code, represented by the enumeration of the changed lines. Code is structured differently than text, and thus we need to construct a representation that enables machine learning algorithms to detect relevant properties.

Second, the available labeled data from which to learn is somewhat noisy. The only available source of labels is whether a given patch is already in a stable kernel. However, stable kernels in practice do not receive only bug-fixing patches, but also patches that add new device identifiers (structure field values that indicate some properties of a supported device) and patches on which a subsequent bug-fixing patch depends, as long as these patches are small and obviously correct. On the other hand, our results in the previous section suggest that not all patches that should be propagated to stable kernels actually get propagated. These sources of noise may introduce apparent inconsistencies into the machine learning process.

Finally, although some patches perform bug fixes, not propagating them to stable kernels is the correct choice. One reason is that some parts of the code change so rapidly that the patch does not apply cleanly to any stable version. Another reason is that the bug was introduced since the most recent mainline release, and thus does not appear in any stable version.

As the decision of whether to apply a patch to a stable kernel depends in part on factors external to the patch itself, we cannot hope to achieve a perfect solution based on applying machine learning to patches alone. Still, we believe that machine learning can effectively complement existing practice by orienting stable-kernel maintainers towards likely stable commits that they may have overlooked, even though the above issues introduce the risk of some false negatives and false positives.

5.3 Proposed Approach

In this section, we first formulate the problem and provide an overview of PatchNet. We then describe the details of each module inside PatchNet. Finally, we present an algorithm for learning effective values of PatchNet's parameters.

5.3.1 Framework Overview

The goal of PatchNet is to automatically label a patch as stable or non stable in order to reduce the manual effort for the stable-kernel maintainers. We consider the identification of stable patches as a learning task to construct a prediction function $f : \mathcal{X} \longmapsto \mathcal{Y}$, where $\mathcal{Y} = \{0, 1\}$. Then, $x_i \in \mathcal{X}$ is identified as a stable patch when $f(x_i) = 1$.

As illustrated in Fig. 5.4, PatchNet consists of three main modules: (1) a *commit message module*, (2) a *commit code module*, and (3) a *classification module*. The first two are built upon a convolutional neural network (CNN)

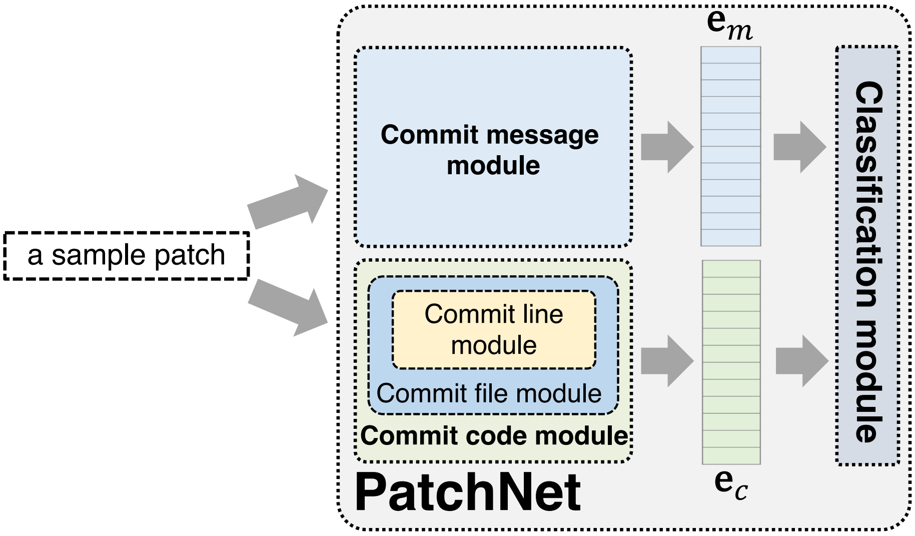


Figure 5.4: The proposed PatchNet framework. $\mathbf{e}_m$ and $\mathbf{e}_c$ are embedding vectors collected from the commit message module and commit code module, respectively.

architecture [124, 115], and aim to learn a representation of the textual commit message (cf. Fig. 5.1a, lines 5-12) and the set of diff code elements (cf. Fig. 5.1a, lines 14-28) of a patch, respectively. The *commit message module* and the *commit code module* transform the commit message and the code changes into embedding vectors $\mathbf{e}_m$ and $\mathbf{e}_c$, respectively. The two vectors are then passed to the *classification module*, which computes a prediction score indicating the likelihood of a patch being a stable patch.

5.3.2 Commit Message Module

The commit message module is the same as the one proposed by Kim [105] and Kalchbrenner *et al.* [95] for sentence classification, and was introduced in Section 4.3.3. The module involves an input message, represented as a two-dimensional matrix, a set of filters for identifying features in the message, and a means of combining the results of the filters into an *embedding vector* that represents the most salient features of the message, to be used as a basis for classification.

Message representation. We encode a commit message as a two-dimensional matrix by viewing the message as a sequence of vectors where each vector represents one word appearing in the message. The embedding vectors of the individual words are maintained using a lookup table, the *word embedding matrix*, that is shared across all messages.

Given a message m as a sequence of words $[\mathtt{w}_1, \ldots, \mathtt{w}_{|m|}]$ and a word embedding matrix $\mathbf{W}_m \in \mathbb{R}^{|V_m| \times d_m}$, where V_m is the vocabulary containing all words in commit messages and d_m is the dimension of the representation of a word, the matrix representation $\mathbf{M} \in \mathbb{R}^{|m| \times d_m}$ of the message is:

$$\mathbf{M} = [\mathbf{W}[\mathtt{w}_1], \ldots, \mathbf{W}[\mathtt{w}_{|m|}]] \tag{5.1}$$

For parallelization, all messages are padded or truncated to the same length.

Convolutional layer. The role of the convolutional layer is to apply filters to the message, in order to identify the message's salient features. A filter $\mathbf{f} \in \mathbb{R}^{k \times d_m}$ is a small matrix that is applied to a window of k words to produce a new feature. A feature t_i is generated from a window of words $\mathbf{M}_{i:i+k-1}$ starting at word $i \leq |m| - k + 1$ by:

$$t_i = \alpha(\mathbf{f} * \mathbf{M}_{i:i+k-1} + b_i) \tag{5.2}$$

where $*$ is the sum of the element-wise products, $b_i \in \mathbb{R}$ is a bias value, and $\alpha(\cdot)$ is a non-linear activation function. For $\alpha(\cdot)$, we choose the *rectified linear unit* (ReLU) activation function [160], as it has been shown to have better performance than its alternatives [16, 48].

The filter $\mathbf{f}$ is applied to all windows of size k in the message resulting in a *feature vector* $\mathbf{t} \in \mathbb{R}^{|m|-k+1}$:

$$\mathbf{t} = [t_1, t_2, \cdots, t_{|m|-k+1}] \tag{5.3}$$

Max pooling. To characterize the commit message, we are interested in the degree to which it contains various features, but not where in the message those features occur. Accordingly, for each filter, we apply max pooling [46] over the feature vector $\mathbf{t}$ to obtain the highest value:

$$\max_{1 \leq i \leq |m|-k+1} t_i \tag{5.4}$$

The results of applying max pooling to the feature vector resulting from applying each filter are then concatenated to form an embedding vector ($\mathbf{e}_m$) representing the meaning of the message.

5.3.3 Commit Code Module

Like the commit message, the commit code can be viewed as a sequence of words. This view, however, overlooks the structure of code changes, as needed to distinguish between changes to different files, changes in different hunks, and changes of different kinds (removals or additions). To incorporate this structural information, PatchNet contains a *commit code module* that takes as input the code changes in a given patch and outputs an embedding vector that represents the most salient features of the code changes. The commit code module contains a *commit file module* that automatically builds an embedding vector representing the code changes made to a given file in the patch. The embedding vectors of code changes at the file level are then concatenated into a single vector representing all the code changes made by the patch.

5.3.3.1 Commit File Module

The commit file module builds an embedding vector for each file in the patch that represents the changes to the file.

As shown in Fig. 5.5, the commit file module takes as input two matrices (denoted by "–" and "+" in Fig. 5.5) representing the removed code and added code for the affected file in a patch, respectively. These two matrices are passed to the *removed code module* and the *added code module*, respectively, to construct corresponding embedding vectors. The two embedding vectors

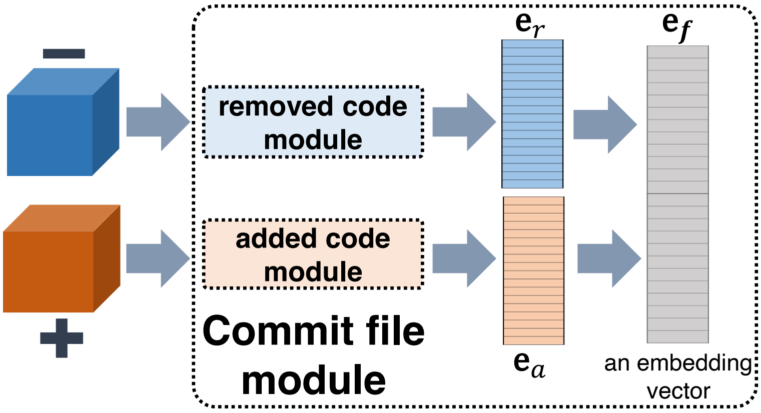


Figure 5.5: Architecture of the *Commit File Module* for mapping a file in a given patch to an embedding vector. The input of the module is the removed code and added code of the affected file, denoted by "–" and "+", respectively.

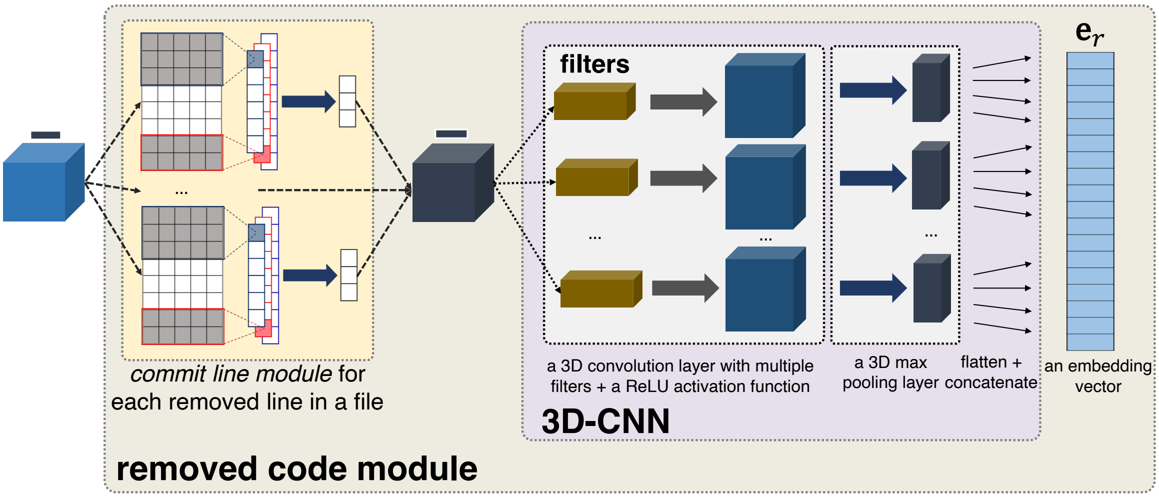

−
...
commit line module for each removed line in a file
−
filters
...
a 3D convolution layer with multiple filters + a ReLU activation function
a 3D max pooling layer
flatten + concatenate
3D-CNN
$\mathbf{e}_r$
an embedding vector
removed code module

Figure 5.6: Architecture of the *removed code module* used to build an embedding vector for the code removed from an affected file.

are then concatenated to represent the code changes in each affected file. We present the removed code module and the added code module below.

Removed code module. Fig. 5.6 shows the structure of the *removed code module*. The input of this module is a three-dimensional matrix, indicating the removed code in a file of a given patch, denoted by $\mathcal{B}_r \in \mathbb{R}^{\mathcal{H}\times\mathcal{N}\times\mathcal{L}}$, where $\mathcal{H}$, $\mathcal{N}$, and $\mathcal{L}$ are the number of hunks, the number of removed code lines for each hunk, and the number of words in each removed code line in the affected file, respectively. This module takes advantage of a *commit line module* and a 3D convolutional layer (*3D-CNN*) to construct an embedding vector (denoted by $\mathbf{e}_r$ in Fig. 5.5) representing the removed code in the affected file. We describe the *commit line module* and the *3D-CNN* in the following sections.

a) Commit line module. Each line of removed code in $\mathcal{B}_r$ is processed by the *commit line module* to obtain a list of embedding vectors representing the removed code lines. This module has the same structure as the commit message module, but maintains a code-specific vocabulary and word embedding matrix, as a word may have different meanings in a textual message and in source code.

The obtained commit line vectors are used to construct a new three-dimensional matrix, $\hat{\mathcal{B}}_r \in \mathbb{R}^{\mathcal{H} \times \mathcal{N} \times E}$. $\hat{\mathcal{B}}_r$ represents a sequence of $\mathcal{H}$ hunks; each hunk has a sequence of removed lines, where each line is now represented as a E-dimensional embedding vector ($e_{ij} \in \mathbb{R}^E$) extracted by the *commit line module*. $\hat{\mathcal{B}}_r$ is then passed to the 3D convolutional neural network (3D-CNN), described below, to construct an embedding vector for the code removed from a file by a given patch.

b) 3D-CNN. The 3D convolutional layer is used to extract features from the code removed from the affected file, as represented by $\hat{\mathcal{B}}_r$. This layer applies each filter $\mathbf{F} \in \mathbb{R}^{k \times \mathcal{N} \times E}$ to a window of k hunks $\mathbf{H}_{i:i+k-1}$ to build a new feature as follows:

$$f_i = \alpha(\mathbf{F} * \mathbf{H}_{i:i+k-1} + b_i) \tag{5.5}$$

$*$ is the sum of element-wise products, $\mathbf{H}_{i:i+k-1} \in \mathbb{R}^{|i:i+k-1| \times \mathcal{N} \times E}$ is constructed from the i-th hunk through the $(i+k-1)$-th hunk in the removed code of the affected file, $b_i \in \mathbb{R}$ is the bias value, and $\alpha(\cdot)$ is the ReLU activation function. As for the commit message module (see Section 5.3.3), we choose $k \in \{1, 2\}$. Fig. 5.7 shows an example of a 3D convolutional layer that has one filter.

Applying the filter $\mathbf{F}$ to all windows of hunks in $\hat{\mathcal{B}}_r$ produces a feature vector:

$$\mathcal{F} = [f_1, \ldots, f_{\mathcal{H}-k+1}] \tag{5.6}$$

As in Section 5.3.2, we apply a max pooling operation to $\mathcal{F}$ to obtain the most important feature. The features selected by max pooling with multi-

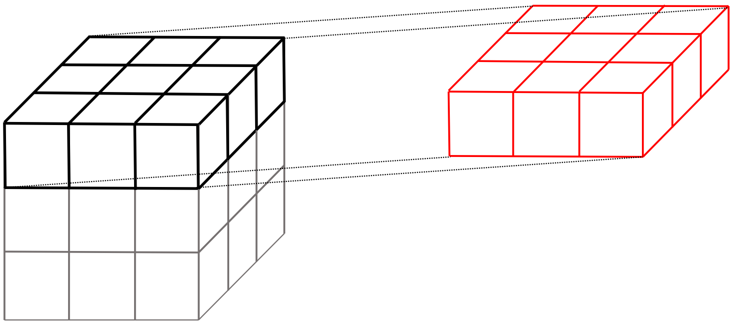


Figure 5.7: A 3D convolutional layer on $3 \times 3 \times 3$ data. The $1 \times 3 \times 3$ red cube on the right is the filter. The dotted lines indicate the sum of element-wise products over all three dimensions. The result is a scalar vector.

ple filters are concatenated to construct an embedding vector $\mathbf{e}_r$ representing information extracted from the removed code changes in the affected file.

Added code module. This module has the same architecture as the removed code module. The changes in the added and removed code are furthermore padded or truncated to have the same number of hunks ($\mathcal{H}$), number of lines for each hunk ($\mathcal{N}$), and the number of words of each line ($\mathcal{L}$), for parallelization. Moreover, both modules also share the same vocabulary and use the same word embedding matrix.

The added code module constructs an embedding vector (denoted by $\mathbf{e}_a$ in Fig. 5.5) representing the added code in a file of a given patch. An embedding vector representing all of the changes made to a given file by a commit is constructed by concatenating the two embedding vectors representing the removed code and added code as follows:

$$\mathbf{e}_{\mathtt{f}} = \mathbf{e}_r \oplus \mathbf{e}_a \tag{5.7}$$

5.3.3.2 Embedding Vector for Commit Code

The embedding vector for all the changes performed by a given patch is constructed as follows:

$$\mathbf{e}_c = \mathbf{e}_{\mathtt{f}_1} \oplus \cdots \oplus \mathbf{e}_{\mathtt{f}_v} \tag{5.8}$$

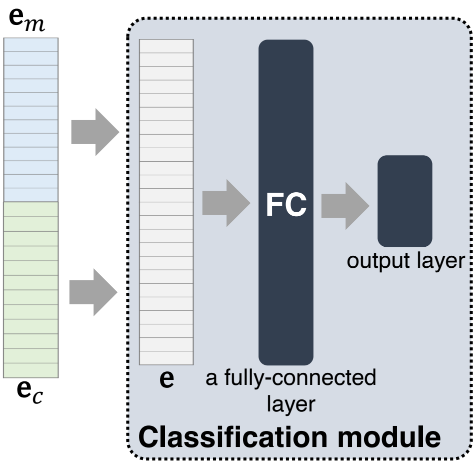


Figure 5.8: Architecture of the *classification module*, comprising a fully connected layer (FC), and an output layer.

where $\oplus$ is the concatenation operator used to concatenate the embedding vector of each changed file, $\mathtt{f}_i$ denotes the i-th file affected by the given commit, v is the number of affected files, and $\mathbf{e}_{\mathtt{f}_i}$ denotes the vector constructed by applying the *commit file module* to the affected file $\mathtt{f}_i$.

5.3.4 Classification Module

Fig. 5.8 shows the architecture of the *classification module*. It takes as input the commit message embedding vector $\mathbf{e}_m$ and the commit code embedding vector $\mathbf{e}_c$ discussed in Sections 5.3.2 and 5.3.3, respectively. The patch is represented by their concatenation as follows:

$$\mathbf{e} = \mathbf{e}_m \oplus \mathbf{e}_c \tag{5.9}$$

We then feed the concatenated vector $\mathbf{e}$ into a fully-connected (FC) layer, which outputs a vector $\mathbf{h}$ as follows:

$$\mathbf{h} = \alpha(\mathbf{w}_h \cdot \mathbf{e} + b_h) \tag{5.10}$$

where $\cdot$ is a dot product, $\mathbf{w}_h$ is a weight matrix associated with the concatenated vector, b_h is the bias value, and $\alpha(\cdot)$ is a non-linear activation function.

Again, we use ReLU to implement $\alpha(\cdot)$. Note that both $\mathbf{w}_h$ and b_h are learned during our model's training process.

Finally, the vector $\mathbf{h}$ is passed to an output layer, which computes a probability score for a given patch:

$$z_i = p(y_i = 1|x_i) = \frac{1}{1 + \exp(-\mathbf{h} \cdot \mathbf{w}_o)} \tag{5.11}$$

where $\mathbf{w}_o$ is a weight matrix that is also learned during the training process.

5.3.5 Parameter Learning

During the training process, PatchNet learns the following parameters: the word embedding matrices for commit messages and commit code, the filter matrices and bias of the convolutional layers, and the weights and bias of the fully connected layer and the output layer. The training aims to minimize the following regularized loss function [71]:

$$\begin{aligned} \mathcal{O} &= -\log\left(\prod_{i=1}^{N} p(y_i|x_i)\right) + \frac{\lambda}{2}\|\theta\|_2^2 \\ &= -\sum_{i=1}^{N} \left[y_i \log(z_i) + (1 - y_i)\log(1 - z_i)\right] + \frac{\lambda}{2}\|\theta\|_2^2 \end{aligned} \tag{5.12}$$

where z_i is the probability score from the output layer and θ contains all the (learnable) parameters as mentioned before.

The term $\frac{\lambda}{2}\|\theta\|_2^2$ is used to mitigate data overfitting by penalizing large model parameters, thus reducing the model complexity. To further improve the robustness of our model, we also apply the dropout technique [195] on all the convolutional and fully-connected layers in PatchNet.

To minimize the regularized loss function (5.12), we employ a variant of stochastic gradient descent (SGD) [34] called *adaptive moment estimation* (Adam) [107]. We choose Adam over SGD due to its computational efficiency and low memory requirements [107, 17, 19].

5.4 Experiments

We first describe our dataset and how we preprocess it. We then introduce the baselines and evaluation metrics. Finally, we present our research questions and results.

5.4.1 Dataset

We take our data from the patches that have been committed to mainline Linux kernel[2] v3.0, released in July 2011, through v4.12, released in July 2017. We additionally collect information from the stable kernels[3] that had been released as of October 2017 building on Linux kernels v3.0 through v4.13. We consider a mainline commit to be stable if it is duplicated in at least one stable version. To increase the set of commits that can be used for training, we furthermore include in the training set of stable patches other Linux kernel commits that are expected by convention to be bug-fixing patches. Indeed, a Linux kernel release is created by first collecting a set of commits for the coming release into a preliminary release called a "release candidate", named *rc1*, that may include new features and bug fixes. This is followed by a succession of further release candidates, named *rc2* onwards, that should include only bug fixes. We thus also include the commits added for release candidates rc2 onwards in our set of stable patches.

We refer to patches that are propagated to stable kernels or are found in later release candidates as *stable patches* and patches that are not propagated to stable kernels or found in later release candidates as *non-stable patches*. To avoid biasing the learning process towards either stable or non stable patches, we construct our training datasets such that the number of patches in each category is roughly balanced. While this situation does not reflect the number of stable and non-stable patches that confront a stable kernel maintainer each

[2] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[3] git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

day, it allows effective training and interpretation of the experimental results.

5.4.1.1 Identifying Stable Patches

The main challenge in constructing the datasets is to determine which mainline patches have been propagated to stable kernels. Indeed, there is no required link information connecting the two. Many stable patches explicitly mention the corresponding mainline commit in the commit message, which we refer to as a *back link*. For others, we rely on the author name and the subject line. Subject lines typically contain information about both the change made and the name of the file or directory in which the change is made, and should be unique. We first collect from the patches in the stable kernels a list of back links and a list of pairs of author name and subject line. A commit from the mainline whose commit id is mentioned in a back link or whose author name and subject line are the same as one found in a patch to a stable kernel is considered to be a stable patch.

5.4.1.2 Collecting the Dataset

We collect our dataset from the mainline Linux kernel. In order to focus on patches that are challenging for stable maintainers to classify, we drop in advance all patches that do not meet the stable-kernel size guidelines,[4] *i.e.*, those that exceed 100 code lines, including both changed lines and context as reported by `diff`. We subsequently keep all identified stable patches for our dataset and select an equal number of non-stable patches. Whenever possible, we select non-stable patches that have a similar number of changed lines as the stable patches, again to create a dataset that reflects the cases that cannot be excluded by size alone and thus are challenging for stable kernel maintainers. These patches are then subject to a preprocessing step that is detailed in the next section. We do not use the dataset studied by Tian *et al.* [204], because

[4] https://www.kernel.org/doc/html/v4.15/process/stable-kernel-rules.html

it is seven years old and unclean, including labeling as bug-fixing patches the results of tools that may report coding style issues or faults whose impact is not visible in practice.

Our dataset comes from Linux kernel mainline versions 3.0 (July 2011) through 4.12 (July 2017). There were 424,380 commits during that period. We consider only those commits that are not merge commits, that modify a file as opposed to only adding or removing files, and that affect at least one `.c` or `.h` file. This leaves 346,570 commits (82%). Of these 346,570 commits, 79,319 (23%) are not considered because they contain more than 100 changed lines, leaving 267,251 commits. Of these, to have a balanced training dataset, we pick the 42,408 stable patches for which the preprocessing step is successful (see below,) and 39,995 non-stable patches, *i.e.*, 82,403 patches in all. In RQ4 described below, we consider the full set of Linux kernel patches in versions v3.0-v4.12 that are accepted by our preprocessing step.

5.4.2 Patch Preprocessing

Our approach applies some preprocessing steps to the patches before they are given to PatchNet.

5.4.2.1 Preprocessing of Commit Messages

Our approach applies various standard natural language techniques to the commit messages, such as stop word elimination and stemming [208, 36], to reduce message length and eliminate irrelevant information. Subsequently, we pad or truncate all commit messages to the same size, specifically 512 words, covering the complete commit message for all patches, for parallelism. Because we are interested in cases that are challenging for the stable kernel maintainer, we drop tags such as Cc: stable and Fixes, whose goal is to indicate that a given patch is a stable or a bug fixing patch. We also drop tags indicating who has approved the patch, as the set of developers and their work profiles can change in the future.

5.4.2.2 Preprocessing of Code Changes

Diff code elements, as illustrated in Fig. 5.1a, may have many shapes and sizes, from a single word to multiple lines spread out over multiple hunks. To describe changes in terms of meaningful syntactic units and to provide context for very small changes, we collect differences at the granularity of atomic statements. These may be, *e.g.*, simple assignment statements, return statements, if headers, etc. For example, in the patch illustrated in Fig. 5.1a, the only change is to replace `1` on line 22 by `err` on line 23. Nevertheless, we represent the change as a change in the complete return statement, *i.e.*, `return 1;` that is transformed into `return err;`. We also distinguish changes in error checking code (code to detect whether an error has occurred, *e.g.*, line 21 in Fig. 5.1a) and in error handling code (code to clean up after an error has occurred, *e.g.*, lines 22 and 23 in Fig. 5.1a) from changes in other code, which we refer to as *normal code*. Error handling code is considered to be any code that is in a conditional with only one branch, where the conditional ends in a `return` with an argument other than `0` (0 is typically the success indicator) or a `goto`, as well as any code following a label that ends in a `return` with an argument other than `0` or a `goto`. Error checking code is considered to be the header of a conditional that matches the former pattern. These criteria are not completely reliable, as such code can sometimes represent the success case rather than a failure case, but they are typically followed and are actively promoted by Linux kernel developers. Error checking code and error handling code are very common in the Linux kernel, which must be robust, and they are disjoint in structure and purpose from the implementation of the main functionality.

For a given commit, the first step is to extract the names of the affected files and to extract the state of those files before and after the commit. Analogous to the stemming and stop word elimination performed at the commit message level, for each before and after file instance, we remove comments and the contents of strings, as changes in comments and within strings are not likely to be needed in stable kernels. For a given pair of before and after files, we then compute the difference using the command “git diff -U0 old new”, giving the changed lines with no lines of surrounding context. For each “–” or “+” line in the diff output, we then collect a record indicating the sign (“–” or “+”), the category (error-handling code, etc.), the hunk number, the line number in the old or new version, respectively, and the starting and ending columns of the non-space changes on the line. We furthermore keep the names of called functions, when these are not defined in the same file and are used at least 5 times, but drop other identifiers, *i.e.* field names and variable names, as these may be too diverse to allow effective learning and may unnecessarily slow down the training time. Indeed, adding just the frequently used function names increases the code vocabulary size from 43 to 3,616 unique tokens, which increases the training time.

To extract changes at the level of atomic statements, rather than the individual lines obtained by diff, we parse each file as it exists before and after the change and keep the atomic statements that intersect with a changed line observed by diff. For this, we use the parser of the C program transformation system Coccinelle [168], which uses heuristics to parse around compiler directives and macros [167]. This makes it possible to reason about patches in terms of the way they appear to the user, without macro expansion, but comes with some cost, as some patches must be discarded because the parsing heuristics are not sufficient to parse all of the code affected by the changed lines.

By following the above-mentioned steps, we collect the files affected by a given patch. For each removed or added code line of an affected file, denoted by “–” and “+”, we collect the corresponding hunk number and line number. Each word in a line is a pair of the associated token and the annotation indicating whether the word occurs on a line of as error-checking code, error-handling code, or normal code. This information is used to build the two three-dimensional matrices representing the removed code and the added code for the affected file (see Fig. 5.5).

5.4.3 Baselines

We compare PatchNet with several baselines:

- *Keyword*: As a simple but frequently used heuristic [204], we select all commits in which the commit message includes “bug”, “fix”, or “bug-fix” after conversion of all words to lowercase and stemming. While not all bug fixes are relevant for stable kernels, as some bugs may have very low impact or the fix may be too large or complex to be considered clearly correct, the problem of identifying bug fixes is close enough to that of recognizing stable patches to make comparison with our model valuable.

- *LPU+SVM*: This method was proposed by Tian et al. [204] and combines Learning from Positive and Unlabeled Examples (LPU) [88, 127, 135] and Support Vector Machine (SVM) [39, 47], to build a classification model for automatically identifying bug fixing patches. The set of code features considered was manually selected. In Tian *et al.*'s work, stable kernels were considered as a source of bug-fixing patches in the training and testing data.

- *LS-CNN*: Huo *et al.* [83] combined LSTM [81] and CNN [124] to localize potential buggy source files based on bug report information. They used CNN to learn a representation of the bug report and a combination of LSTM and CNN to learn the structure of the code. To assess the ability of LS-CNN to classify patches as stable, for a given patch, we give the commit message and the code changes (i.e., the result of concatenating the lines changed in the various files and hunks) as input to LS-CNN in place of the bug report and the potential buggy source file, respectively. To make a fair comparison, the CNN used to learn the representation of the commit message in LS-CNN has the same architecture (i.e., number of convolutional layer, filter size, activation function, etc.) as the CNN used to learn the representation of the commit message in PatchNet.

- *Feed-forward fully connected neural network* (F-NN): Inspired by PatchNet and the work of Tian *et al.* on LPU+SVM, a Linux stable kernel maintainer, Sasha Levin, has developed an approach to identifying stable patches [128] based on a feed-forward fully connected neural network [32, 64] and a set of manually selected features, including frequent commit message words, author names, and some code metrics. Levin actively uses this approach in his work on the Linux kernel.

For LPU-SVM and LS-CNN, we used the same parameters and settings as described in the respective papers. For F-NN, we asked Levin to train the tool on our training data and test it with our testing data. We use 50% as the cut off for considering a patch as stable for PatchNet and all baselines.

5.4.4 Experimental Settings

PatchNet has several hyperparameters (i.e., the sizes of the filters, the number of convolutional filters, the size of the fully-connected layer, etc.) that we instantiate them in the following paragraph.

For the sizes of the filters described in Section 5.3, we choose $k \in \{1, 2\}$, making the associated windows analogous to a 1-gram or 2-gram as used in natural language processing [93, 37]. Using 2-grams allows our approach to take into account the temporal ordering of words, going beyond the bag of words used by Tian et al. [204]. The number of convolutional filters is set to 64. The size of the fully-connected layer described in Section 5.3.4 is set to 100. The dimensions of the word vectors in commit message d_m and code changes d_c are set to 50. PatchNet is trained using Adam [107] with shuffled mini-batches. The batch size is set to 32. We train PatchNet for 50 epochs and apply the early stopping strategy [176, 38], i.e., we stop the training if there has been no update to the loss value (see Equation 5.12) for the last 5 epochs. All these hyperparameter values are widely used in the deep learning community [188, 84, 83, 77]. For parallelization, the number of changed files, the number of hunks for each file, the number of lines for each hunk, the number of words of each removed or added code are set to 5, 8, 10, and 120, respectively.

In our experiments, we run PatchNet on Ubuntu 18.04.3 LTS, 64 bit, with a Tesla P100-SXM2-16GB5 GPU.[5] Training takes around 20 hours and testing less than 30 minutes to process 16,481 patches (one of the five folds presented in Section 5.4.6). Note that training only needs to be done periodically (e.g., weekly/monthly) and the trained model can be used to label many patches. In our experiments, on average, the trained PatchNet can assign a label to a single patch in 0.11 seconds.

5.4.5 Evaluation Metrics

To evaluate the effectiveness of a stable patch identification model, we employ the following metrics:

- *Accuracy*: Proportion of stable and non-stable patches that are correctly classified.
- *Precision*: Proportion of patches that are correctly classified as stable.
- *Recall*: Proportion of stable patches that are correctly classified.
- *F1 score*: Harmonic mean between precision and recall

[5] https://www.nvidia.com/en-us/data-center/tesla-p100/

- *AUC*: Area under the Receiver Operating Characteristic curve, measuring if the stable patches tend to have higher predicted probabilities (to be stable) than non stable ones.

5.4.6 Research Questions and Results

Our study seeks to answer several research questions (RQs):

RQ1: Do the properties of stable and non stable patches change over time?

A common strategy for evaluating machine learning algorithms is n-fold cross-validation [111], in which a dataset is randomly distributed among n equal-sized buckets, each of which is considered as test data for a model trained on the remaining $n-1$ buckets. When data elements become available over time, as is the case of Linux kernel patches, this strategy results in testing a model on data that predates some of the data on which the model was trained. Respecting the order of patch submission, however, would limit the amount of testing that can be done, given the fairly small number of stable patches available.

To address this issue, we first assess whether training on future data helps or harms the accuracy of PatchNet. We first sort the patches collected in Section 5.4.1 from earliest to latest based on the date when the patch author submitted the patch to maintainers. Then, we divide the dataset into five mutually exclusive sets by date. Note that the resulting five sets are not perfectly balanced, but they come close, with stable patches making up 45% to 55% of each set. Then, we repeat the following process five times: take one set as a testing set and use the remaining four sets for training. Testing on the first set shows the impact of training only on future data. Testing on the fifth set shows the impact of training only on past data. The other testing sets use models trained on a mixture of past and future data.

Table 5.1 shows the results of PatchNet on the different test sets. The

Table 5.1: The results of PatchNet on the five chronological test sets

	Accuracy	**Precision**	**Recall**	**F1**	**AUC**
Set=1	0.852	0.841	0.886	0.863	0.850
Set=2	0.860	0.833	0.909	0.869	0.859
Set=3	0.866	0.833	0.910	0.870	0.867
Set=4	0.864	0.828	0.912	0.868	0.864
Set=5	0.869	0.860	0.917	0.887	0.862
Std.	**0.007**	**0.013**	**0.012**	**0.009**	**0.007**

standard deviations are quite small (i.e., at most 0.013), hence there is no difference between training on past or future data. Our dataset starts with Linux v3.0, which was released in 2011, twenty years after the start of work on the Linux kernel. The lack of impact due to training on past or future data suggests that in such a mature code base the properties that make a patch relevant for stable kernels are fairly constant over time. This property is indeed beneficial, because it means that our approach can be used to identify stable commits that have been missed in older versions. In the subsequent research questions, we thus retain the same five test and training sets.

RQ2: How effective is PatchNet compared to other state-of-the-art stable patch identification models?

To answer this RQ, we use the five test sets of the dataset described in RQ1. Of these, we take one test set as the testing data and regard the remaining patches as the training data. We repeat this five times, and then average the results to get the aggregated accuracy, precision, recall, F1, and AUC scores. Table 5.2 shows the results for PatchNet and the other baselines. PatchNet achieves accuracy, precision, recall, F1 score, and AUC of 0.862, 0.839, 0.907, 0.871, and 0.860, respectively. Compared to the best performing baseline, F-NN, these constitute improvements of 6.55%, 0.12%, 16.13%, 7.80%, and 6.30%, respectively. PatchNet thus achieves about the same precision as F-NN, but a significant improvement in terms of recall. This is achieved without the feature engineering required for the F-NN approach, but rather by automatically learning the weight of the filters via our hierarchical deep

Table 5.2: PatchNet vs. Keyword, LPU+SVM, LS-CNN, and F-NN.

	Accuracy	**Precision**	**Recall**	**F1**	**AUC**
Keyword	0.626	0.683	0.515	0.587	0.630
LPU+SVM	0.731	0.751	0.716	0.733	0.731
LS-CNN	0.765	0.766	0.785	0.775	0.765
F-NN	0.809	0.838	0.781	0.808	0.809
PatchNet	**0.862**	**0.839**	**0.907**	**0.871**	**0.860**

learning-based architecture.

We also employ Scott-Knott ESD [63] to statistically compare the performance of PatchNet and the four considered approaches (i.e., PatchNet, F-NN, LS-CNN, and LPU-SVM). The results show that PatchNet consistently appears in the top Scott-Knott ESD rank in terms of accuracy, precision, recall, F1 score, and AUC. Specifically, the ranks of the four considered approaches are consistent (i.e., PatchNet > F-NN > LS-CNN > LPU-SVM) except for recall (i.e., PatchNet > LS-CNN > F-NN > LPU-SVM).

Fig. 5.9 compares the precision-recall curves for PatchNet and the baselines. For most values on the curve, PatchNet obtains the highest recall for a given precision and the highest precision for a given recall. For example, for a low false positive rate of 5 percent (precision of 0.95), PatchNet achieves a recall of 0.786 which is 14.9% higher than that of the best performing baseline. Likewise, for a low false negative rate of 5 percent (recall of 0.95), PatchNet achieves a precision of 0.603 which is 41.2% higher than that of the best performing baseline. In addition, considering the sweet spots where both precision and recall are high (larger than 0.8), PatchNet can achieve an F1 score of up to 0.886 which is 10.6% higher than that of the best performing baseline.

Fig. 5.10 shows Venn diagrams indicating the number of patches that PatchNet and each of the baselines correctly recognize as stable. The top diagram compares the Keyword approach to the two approaches, PatchNet and LS-CNN, that automatically learn the relevant features. While there are over 20K patches that all three approaches classify as stable, there are another 11K that are found by both learning-based approaches, showing the advantage of

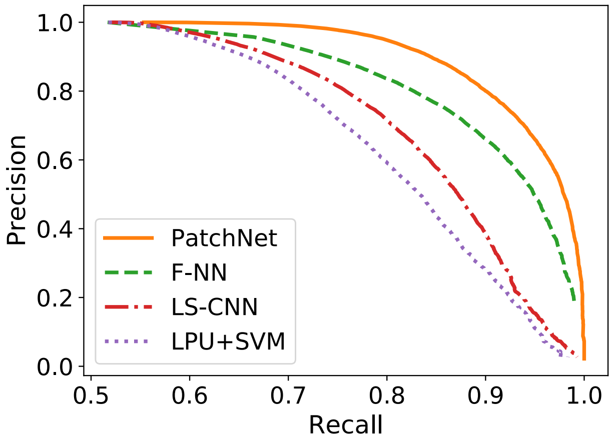


Figure 5.9: Precision-recall curve: PatchNet vs. LPU+SVM, LS-CNN, and F-NN.

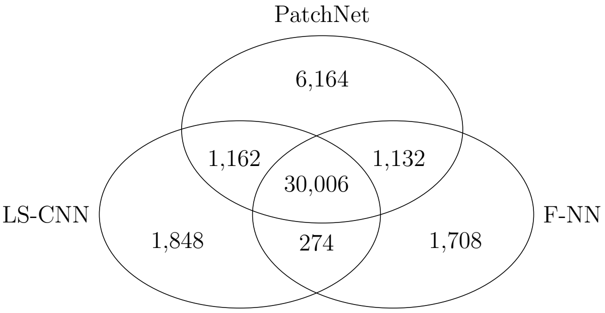


Figure 5.10: Venn diagrams showing the number of stable patches identified by PatchNet and the various baselines

learning-based approach. As compared to Keyword and LS-CNN, there are almost 7,000 patches that are only recognized by PatchNet, while this is the case for fewer than 2,000 patches for LS-CNN, showing the value of an approach that takes the properties of code changes into account. The bottom diagram then compares PatchNet to the two approaches, LPU+SVM and F-NN, in which the code features are hand crafted.

While all three approaches correctly recognize over 27K patches as stable, there are again 3x more patches that only PatchNet correctly detects as stable than there are that only each of the other two approaches recognizes as stable. Examples of PatchNet true positives not found by the other baselines include 5567e989198b[6] and 2e31b4cb895a. Examples of PatchNet false negatives found by at least one other baseline include 03f219041fdb and 56199016e867.

[6] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

All of the above measures of precision and recall assume that the set of patches found in the tested Linux kernel versions is the correct one. Our motivation, however, is that bug fixing patches that should be propagated to the Linux kernel stable versions are being overlooked by the existing manual labeling process. Showing that PatchNet improves on the existing manual process requires collecting a dataset of patches that have not been propagated to stable kernels, but should have been. Collecting such a dataset, however, requires substantial Linux kernel expertise, which is not feasible to harness at a large scale. We have nevertheless been able to carry out two experiments in this direction. First, we randomly selected 200 patches predicted as stable patches by PatchNet, but that were not marked as stable in our dataset. We sent the 200 patches to Sasha Levin (a Linux stable-kernel maintainer and the developer of F-NN) to label. Among the 200 patches, Levin labeled 61 patches (i.e., 30.5%) as stable, highlighting that our approach can find many additional stable patches that were not identified by the existing manual process. Note that these patches predated Levin's used of F-NN on the Linux kernel. Second, we looked at commits that have no Cc stable tag that Sasha Levin selected with the aid of F-NN for the Linux 4.14 stable tree. These commits postdate all of the commits in our dataset. There are over 1,800 of them, showing the false negatives in the existing manual process and the need for automated support. PatchNet detects 91% of them as stable. The relationship between the results of F-NN and PatchNet is similar to that shown in Fig. 5.10 for patches in our original dataset and confirms that PatchNet can find stable patches that were not identified by the existing manual process.

RQ3: Does PatchNet benefit from considering both the commit message and the code changes, and do function names help identify stable patches?

To answer this RQ, we conduct an ablation test [113, 138] by ignoring the commit message, the code changes, or the function names in the code changes

Table 5.3: Contribution of commit messages, code changes and function names to PatchNet's performance

	Accuracy	**Precision**	**Recall**	**F1**	**AUC**
PatchNet-C	0.722	0.727	0.748	0.736	0.741
PatchNet-M	0.737	0.732	0.778	0.759	0.753
PatchNet-NN	0.776	0.745	0.779	0.765	0.768
PatchNet	**0.862**	**0.839**	**0.907**	**0.871**	**0.860**

in a given patch one-at-a-time and evaluating the performance. We create three variants of PatchNet: PatchNet-C, PatchNet-M, and PatchNet-NN. PatchNet-C uses only code change information while PatchNet-M uses only commit message information. PatchNet-NN uses both code change and commit message information, but ignores the function names in the code changes. We again use the five copies of the dataset described in RQ1 and compute the various evaluation metrics.

Table 5.3 shows that the performance of PatchNet degrades if we ignore any one of the considered types of information. Accuracy, precision, recall, F1 score, and AUC drop by 19.39%, 15.41%, 21.26%, 18.34%, and 16.06% respectively if we ignore commit messages. They drop by 16.96%, 14.62%, 16.58%, 14.76%, and 14.21% respectively if we ignore code changes. And they drop by 11.08%, 12.62%, 16.43%, 13.86%, and 11.98% respectively if we ignore function names. Thus, each kind of information contributes to PatchNet's performance. Additionally, the drops are greatest if we ignore commit messages, indicating that they are slightly more important than the other two to PatchNet's performance.

RQ4: What are the results of PatchNet on the complete set of Linux kernel patches?

For RQ1, we use a dataset collected such that the number of stable and non-stable patches is roughly balanced. Among the 267,251 patches that meet the selection criteria, we picked 42,408 stable patches and 39,995 non-stable patches to build our dataset. To investigate the results of PatchNet on the complete set of patches from Linux v3.0-v4.12 having at most 100 lines (and accepted by our preprocessor), we randomly divide the remaining 184,481 non-stable patches into five sets and merge each of them with each of the five test sets described in RQ1. After this process, we have a new collection of five test sets. In each test set, there are around 8.4K stable patches and 44.8K non-stable patches. For each new test set, we use the corresponding model trained for RQ1. We repeat this five times, and then average the results to get the aggregated AUC score. PatchNet achieves an average AUC of 0.808. Since the new five test sets are highly imbalanced (only 15.79% patches are stable patches), we omit the other metrics (i.e., accuracy, precision, recall, and F1) [164, 198, 151]. We also trained PatchNet on a whole training dataset (i.e., 42,408 stable patches and 39,995 non-stable patches) and evaluated it on 184,481 non-stable patches. We find that PatchNet can correctly label them as non-stable 81.32% of the time.

We also check the effectiveness of PatchNet on patches that have more than 100 lines of code (i.e., long patches). As mentioned earlier, we omit those patches from our training dataset as they do not meet the selection criteria of the Linux kernel. We collect 52,415 long patches from July 2011 to July 2017. Among them, there are 3,376 long stable patches and 49,039 long non-stable patches. 21.33% of these patches contain the "Cc: stable" tag. The others may have been manually selected for stable versions despite not having a tag or may come from the release candidates. We again train PatchNet on the whole training dataset and evaluate the effectiveness of PatchNet on the 52,415 long patches. PatchNet achieves an AUC score of 0.805. Again we only use AUC as this dataset is highly imbalanced [164, 198, 151].

Finally, we also check whether there is a difference of performance in classifying patches containing a "Cc: stable" tag and patches that do not containing a "Cc: stable" tag. Among the 42,408 stable patches, there are 15,410 stable patches with a stable tag and 26,998 stable patches with no stable tag. The latter may again have been manually selected for stable versions despite not having a tag or may come from the release candidates. For each test set described in RQ1, we split the stable patches into two groups: tagged stable patches and non-tagged stable patches. We run PatchNet on the stable patches of each test set to predict the stable patches and sum the results of predicting the stable patches. Among 15,410 tagged stable patches, PatchNet predicts 14,578 patches as stable patches (i.e., 94.60%). Among the 26,998 non-tagged stable patches, PatchNet predicts 23,466 patches as stable patches (i.e., 86.92%). We find that PatchNet is more successful at recognizing tagged patches, even when it does not have access to information about the "Cc: stable" tag.

5.5 Qualitative Analysis and Discussion

In this section, we analyze some of the results obtained in Section 5.4.6, considering in detail a patch where PatchNet performs well and another where it performs poorly.

5.5.1 Successful Case

We first present a patch that PatchNet can predict as a stable patch to show the advantages of our model.

Fig. 5.11 shows a patch propagated to stable kernels. The commit message is on line 5 and the code changes are on lines 16-59. The code changes include one changed file, five hunks, 12 removed lines, and 25 added lines. PatchNet is able to predict the patch in Fig. 5.11 as stable patch. We see that the commit message of this patch is quite short and does not contain keywords such as "bug" or "fix". To recognize the patch as a stable patch, the stable kernel maintainer has to study the code changes to understand the impact of the changes in the kernel code. In the code changes, the four variables (i.e, `left`, `right`, `top`, and `bottom`) are defined and used across the multiple hunks in the

```
commit 203dc2201326fa64411158c84ab0745546300310
Author: Jakob Bornecrantz <jakob@vmware.com>
Date:   Mon Sep 17 00:00:00 2001 +0000

    vmwgfx: Do better culling of presents

    Signed-off-by: Jakob Bornecrantz <jakob@vmware.com>
    Reviewed-by: Thomas Hellstrom <thellstrom@vmware.com>
    Signed-off-by: Dave Airlie <airlied@redhat.com>

diff --git a/drivers/gpu/drm/vmwgfx/vmwgfx_kms.c
           b/drivers/gpu/drm/vmwgfx/vmwgfx_kms.c
index ac24cfd..d31ae33 100644
--- a/drivers/gpu/drm/vmwgfx/vmwgfx_kms.c
+++ b/drivers/gpu/drm/vmwgfx/vmwgfx_kms.c
@@ -1098,6 +1098,7 @@ int vmw_kms_present(struct vmw_private *dev_priv,
            ...
+       int left, right, top, bottom;
            ...
+       left = clips->x;
+       right = clips->x + clips->w;
+       top = clips->y;
+       bottom = clips->y + clips->h;
+
+       for (i = 1; i < num_clips; i++) {
+               left = min_t(int, left, (int)clips[i].x);
+               right = max_t(int, right, (int)clips[i].x + clips[i].w);
+               top = min_t(int, top, (int)clips[i].y);
+               bottom = max_t(int, bottom, (int)clips[i].y + clips[i].h);
+       }
+                       return err;
            ...
-       cmd->body.srcRect.left = 0;
-       cmd->body.srcRect.right = surface->sizes[0].width;
-       cmd->body.srcRect.top = 0;
-       cmd->body.srcRect.bottom = surface->sizes[0].height;
+       cmd->body.srcRect.left = left;
+       cmd->body.srcRect.right = right;
+       cmd->body.srcRect.top = top;
+       cmd->body.srcRect.bottom = bottom;
            ...
-               blits[i].left   = clips[i].x;
-               blits[i].right  = clips[i].x + clips[i].w;
-               blits[i].top    = clips[i].y;
-               blits[i].bottom = clips[i].y + clips[i].h;
+               blits[i].left   = clips[i].x - left;
+               blits[i].right  = clips[i].x + clips[i].w - left;
+               blits[i].top    = clips[i].y - top;
+               blits[i].bottom = clips[i].y + clips[i].h - top;
            ...
-               int clip_x1 = destX - unit->crtc.x;
-               int clip_y1 = destY - unit->crtc.y;
-               int clip_x2 = clip_x1 + surface->sizes[0].width;
-               int clip_y2 = clip_y1 + surface->sizes[0].height;
+               int clip_x1 = left + destX - unit->crtc.x;
+               int clip_y1 = top + destY - unit->crtc.y;
+               int clip_x2 = right + destX - unit->crtc.x;
+               int clip_y2 = bottom + destY - unit->crtc.y;
            ...
```

Figure 5.11: Example of a successfully identified stable patch.

changed file (i.e., vmwgfx_kms.c). We also see the difference between removed lines and added lines when the author committed his code. By representing the removed code and the added code as two three-dimensional matrices (each dimension represents the number of hunks, the number of removed or added code lines, and the number of words in each removed or added code line), PatchNet uses the *removed code module* and the *added code module* to construct the embedding vector of the removed code and added code, respectively (see Section 5.3.3). The two embedding vectors are then concatenated to represent the code change information. By doing this process, the distinction between

```
commit  c607f450f6e49f5794f27617bedc638b51044d2e
Author: Al Viro <viro@zeniv.linux.org.uk>
Date:   Sat May 11 12:38:38 2013 -0400

    au1100fb: VM_IO is set by io_remap_pfn_range()

    Signed-off-by: Al Viro <viro@zeniv.linux.org.uk>

diff --git a/drivers/video/au1100fb.c b/drivers/video/au1100fb.c
index 700cac067b46..ebeb9715f061 100644
--- a/drivers/video/au1100fb.c
+++ b/drivers/video/au1100fb.c
@@ -385,8 +385,6 @@ int au1100fb_fb_mmap(struct fb_info *fbi, struct vm_area_struct *vma)
        vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
        pgprot_val(vma->vm_page_prot) |= (6 << 9); //CCA=6

-       vma->vm_flags |= VM_IO;
-
        if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
                                vma->vm_end - vma->vm_start,
                                vma->vm_page_prot)) {
```

Figure 5.12: Example of an unsuccessfully identified stable patch.

removed lines and added lines is preserved. PatchNet automatically learns from this rich representation by updating its parameters during the training process (see Section 5.4.4) to build a model that can predict whether a patch is stable.

On the other hand, we find that none of the other baselines are able to classify the patch in Fig. 5.11 as a stable patch. *Keyword* is a heuristic approach that only looks at whether the content of a commit message includes "bug" or "fix". *LS-CNN* concatenates the removed lines and added lines in the multiple hunks without preserving the code changes information. *LPU+SVM* and *F-NN* define a set of features for the code changes (i.e., the number of removed code lines, the number of added code lines, the number of hunks in a commit, etc.). The manual creation of code changes features may overlook features that are important to identify stable patches, making *LPU+SVM* and *F-NN* unable to classify the patch in Fig. 5.11 as a stable patch.

5.5.2 Unsuccessful Case

Next, we present a patch that PatchNet fails to classify correctly as a stable patch. This example serves to provide an understanding of cases in which PatchNet may not perform well.

Fig. 5.12 shows a stable patch that was not recognized by PatchNet. Its commit message does not contain any keywords (i.e., "bug" or "fix") that suggest whether the patch is a stable patch. The code changes only include one removed line and the removed line contains only three words: `vma`, `vm_flags`, and `VM_IO`. As there is very little information in both the commit message and the code changes, PatchNet is unable to predict the patch in Fig. 5.12 as a stable patch. We find that the other baselines (i.e, *keywords*, *LS-CNN*, and *LPU+SVM*), except *F-NN*, also fail to classify the patch as a stable patch. *F-NN* considers not only the commit message and the code changes of the given patch, but also information such as author name, reviewer information, file names, etc. This suggests that when the information of the commit message and the code changes is limited, an approach that takes advantage of other information in a given patch may perform better than PatchNet.

5.6 Threats to Validity

Internal validity. Threats to internal validity relate to errors in our experiments and experimenter bias. We have double checked our code and data, but errors may remain. In the baseline approach by Tian et al. [204], commits were labeled by an author with expertise in Linux kernel code, which may introduce author bias. In this work, none of the authors label the commits.

External validity. Threats to external validity relate to the generalizability of our approach. We have evaluated our approach on more than 80,000 patches. We believe this is a good number of patches. Still, the results may differ if we consider other sets of Linux kernel patches. Similar to the evaluation of Tian et al. [204], we only investigated Linux kernel patches, although PatchNet can be applied to patches of other systems, if labels are available. In the future, we would like to consider more projects. Still, we note that the Linux kernel represents one of the largest open source projects, with over 16 million lines of C code, and that different kernel subsystems have different developers and very different purposes, resulting in a wide variety of code.

Construct validity. Threats to construct validity relate to the suitability of our evaluation metrics. We use standard metrics commonly used to evaluate classifier performance. Thus, we believe there is little threat to construct validity.

5.7 Chapter Summary

In this chapter, we propose PatchNet, a hierarchical deep learning-based model for identifying stable patches in the Linux kernel. For each patch, our model constructs embedding vectors from the commit message and the set of code changes. The embedding vectors are concatenated and then used to compute a prediction score for the patch. Different from existing deep learning techniques working on the source code [217, 215, 83, 117], our hierarchical deep learning-based architecture takes into account the structure of code changes (*i.e.,* files, hunks, lines) and the sequential nature of source code (by considering each line of code as a sequence of words) to predict stable patches in the Linux kernel.

We have extensively evaluated PatchNet on a new dataset containing 82,403 recent Linux kernel patches. On this dataset, PatchNet outperforms four baselines including two also based on deep-learning. In particular, for a wide range of values in the precision-recall curve, PatchNet obtains the highest recall for a given precision, as well as the highest precision for a given recall. For example, PatchNet achieves a 14.9% higher recall (0.786) at a high precision level (0.95) and a 41.2% higher precision (0.603) at a high recall level (0.95) compared to the best-performing baseline.

Chapter 6

Distributed Representations of Code Changes

Existing work on software patches often use features specific to a single task. These works often rely on manually identified features, and human effort is required to identify these features for each task. In this work, we propose CC2Vec, a neural network model that learns a representation of code changes guided by their accompanying commit messages, which represent the semantic intent of the code changes. CC2Vec models the hierarchical structure of a code change with the help of the attention mechanism and uses multiple comparison functions to identify the differences between the removed and added code.

To evaluate if CC2Vec can produce a distributed representation of code changes that is general and useful for multiple tasks on software patches, we use the vectors produced by CC2Vec for three tasks: commit message generation, bug fixing patch identification, and just-in-time defect prediction. In all tasks, the models using CC2Vec outperform the state-of-the-art techniques.

6.1 Introduction

Patches, used to edit source code, are often created by developers to describe new features, fix bugs, or maintain existing functionality (e.g., API updates, refactoring, etc.). Patches contain two main pieces of information, a commit message and a code change. The commit message, used to describe the semantics of the code changes, is written in natural language by the developers. The code change indicates the lines of code to remove or add across one or multiple files. Research has shown that the study of historical patches can be employed to solve software engineering problems, such as just-in-time defect prediction [96, 78], identification of bug fixing patches [204, 79], tangled change prediction [108], recommendation of a code reviewer for a patch [180], and many more.

Exploring patches to solve software engineering problems requires choosing a representation of the patch data. Most prior work involves manually crafting a set of features to represent a patch and using these features for further processing [204, 96, 156, 99, 103, 228]. These features have mostly been extracted from properties of patches, such as the modifications to source code (e.g., number of removed and added lines, the number of files modified), the history of changes (e.g., the number of prior or recent changes to the updated files), the record of patch authors and reviewers (e.g., the number of developers or reviewers who contributed to the patch), etc. These features can be used as an input to a machine learning classifier (e.g., Support Vector Machine, Logistic Regression, Random Forest, etc.) to address various software engineering tasks [96, 204, 108, 180]. Extracting a suitable vector representation to represent the "meaning" of a patch is certainly crucial. Intuitively, the quality of a patch representation plays a major role in determining the eventual learning outcome.

In this chapter, to boost the effectiveness of existing solutions that employ the properties of patches, we wish to learn vector representations of the code changes in patches that can be used for a number of tasks. We propose a new deep learning architecture named CC2Vec that can effectively embed a code change into a vector space where similar changes are close to each other. As commit messages, written by developers, are used to describe the semantics of the code change, we use them to supervise the learning of code changes' representations from patches. Specifically, CC2Vec optimizes the vector representation of a code change in a patch to predict appropriate words, extracted from the first line of the commit message. We consider only the first line, as it is the focus of many prior works [139, 179], and is considered to carry the most semantic meaning with the least noise.[1]

CC2Vec analyzes the code change, i.e., scattered fragments of removed and added code across multiple files. Code removed or added from a file follows a hierarchical structure (words form line, lines form hunks). Recent work has suggested that the attention mechanism can help in modelling structural dependencies [106, 15], thus, we hypothesize that the attention mechanism may be effective for modelling the structure of a code change. We propose a specialized hierarchical attention network (HAN) to construct a vector representation of the removed code (and another for the added code) of each affected file in a given patch. Our HAN first builds vector representations of lines; these vectors are then used to construct vector representations of hunks; and we then aggregate these vectors to construct the embedding vector of the removed or added code. Next, we employ multiple comparison functions to capture the difference between two embedding vectors representing removed and added code. This produces features representing the relationship between the removed and added code. Each comparison function produces a vector and these vectors are then concatenated to form an embedding vector for the affected file. Finally, the embedding vectors of all the affected files are concatenated to build a vector representation of the code change in a patch. After training is completed, CC2Vec can be used to extract representations of code changes even from patches with empty or meaningless commit messages (which are common in practice [87, 139, 134]). CC2Vec is also programming-language agnostic; one

[1] https://chris.beams.io/posts/git-commit/

can use it to learn vector representations of code changes for any language.

The code change representation enables us to employ the power of (potentially a large number of) unlabeled patch data to improve the effectiveness of supervised learning tasks (also known as semi-supervised learning [41]). We can use the code change representation to boost the effectiveness of many supervised learning tasks (e.g., identification of bug fixing patches, just-in-time defect prediction, etc.), especially on those tasks for which only a limited set of labeled data may be available.

CC2Vec converts code changes into their distributed representations by learning from a large collection of patches. The distributed representation captures pertinent features of the code changes by considering the characteristics of the whole collection of patches. Such distributed representations can be used as additional features for other tasks. Past studies have demonstrated the value of distributed representations to improve text classification [154], action recognition [147], image classification [43], etc. Unfortunately, prior to our work, there is no existing solution that can produce a distributed representation of a code change.

To evaluate the effectiveness of CC2Vec, we employ the representation learned by CC2Vec in three software engineering tasks: 1) commit message generation [139] 2) bug fixing patch identification [79] and 3) just-in-time defect prediction [78]. In the first task of commit message generation, we generate the first line of a commit message given a code change. CC2Vec can be used to improve over the best baseline by 24.73% in terms of BLEU score (an accuracy measure that is widely used to evaluate machine translation systems). For the task of identifying bug fixing patches, CC2Vec helps to improve the best performing baseline by 5.22%, 9.18%, 4.36%, and 6.51% in terms of accuracy, precision, F1, and Area Under the Curve (AUC). For just-in-time defect prediction, CC2Vec helps to improve the AUC metric by 7.03% and 7.72% on the QT and OPENSTACK datasets [151] as compared to the best baseline.

The main contributions of this chapter are as follows:

- We propose a deep learning architecture, namely CC2Vec, that learns distributed representations of code changes guided by the semantic meaning contained in commit messages. To the best of our knowledge, our work is the first work in this direction.

- We empirically investigate the value of integrating the code change vectors generated by CC2Vec and feature vectors used by state-of-the-art approaches on three tasks (i.e., commit message generation, bug fixing patch identification, and just-in-time defect prediction) and demonstrate improvements.

6.2 Approach

In this section, we first present an overview of our framework. We then describe the details of each part of the framework. Finally, we present an algorithm for learning effective settings of our model's parameters.

6.2.1 Framework Overview

Figure 6.1 illustrates the overall framework of CC2Vec. CC2Vec takes the code change of a patch as input and generates its distributed representation. CC2Vec uses the first line of the commit message of the patch to supervise learning the code change representation. Specifically, the framework of CC2Vec includes five parts:

- *Preprocessing*: This part takes information from the code change of the given patch as an input and outputs a list of files. Each file includes a set of removed code lines and added code lines.

- *Input layer*: This part encodes each changed file as a three-dimensional matrix to be given as input to the hierarchical attention network (HAN)

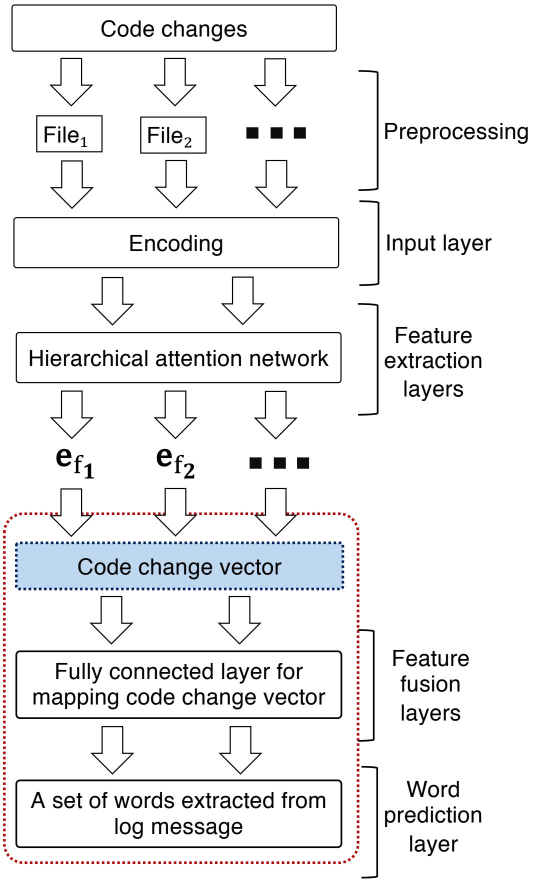

Code changes
File1
File2
Preprocessing
Encoding
Input layer
Hierarchical attention network
Feature extraction layers
ef1
ef2
Code change vector
Fully connected layer for mapping code change vector
Feature fusion layers
A set of words extracted from log message
Word prediction layer

Figure 6.1: The overall framework of CC2Vec. Feature extraction layers are used to construct the embedding vectors for each affected file from a given patch (i.e., $\mathbf{e}_{\mathtt{f}_1}$, $\mathbf{e}_{\mathtt{f}_2}$, etc). The embedding vectors are then concatenated to build a vector representation for the code change in the patch (code change vector). The code change vector is connected to the fully connected layer and is learned by minimizing an objective function of the word prediction layer.

for extracting features.

- *Feature extraction layers*: This part extracts the embedding vector (a.k.a. features) of each changed file. The resulting embedding vectors are then concatenated to form the vector representation of the code change in a given patch.

- *Feature fusion layers and word prediction layer*: This part maps the vector representation of the code change to a word vector extracted from the first line of commit message; the word vector indicates the probabilities that various words describe the patch.

CC2Vec employs the first line of the commit message of a patch to guide the learning of a suitable vector that represents the code change. Words, extracted from the first line of commit message, can be viewed as semantic labels provided by developers. Specifically, we define a learning task to construct a prediction function $\mathbf{f} : P \rightarrow \mathcal{Y}$, where $y_i \in \mathcal{Y}$ indicates the set of words extracted from the first line of the commit message of the patch $p_i \in P$. The prediction function $\mathbf{f}$ is learned by minimizing the differences between the predicted and actual words chosen to describe the patch. After the prediction function $\mathbf{f}$ is learned, for each patch, we can obtain its code change vector from the intermediate output between the feature extraction and feature fusion layers (see Figure 6.1). We explain the details of each part in the following subsections.

6.2.2 Preprocessing

The code change of the given patch includes changes made to one or more files. Each changed file contains a set of lines of removed code and added code. We process the code change of each patch by the following steps:

- **Split the code change based on the affected files.** We first separate the information about the code change to each changed file into a separate code document (i.e., File_1, File_2, etc., see Figure 6.1).

- **Tokenize the removed code and added code lines.** For the changes affecting each changed file, we employ the NLTK library [30] for natural language processing (NLP) to parse its removed code lines or added code lines into a sequence of words. We ignore blank lines in the changed file.

- **Construct a code vocabulary.** Based on the code changes of the patches in the training data, we build a vocabulary $\mathcal{V}^{\text{C}}$. This vocabulary contains the set of code tokens that appear in the code changes of the collection of patches.

At the end of this step, all the changed files of the given patch are extracted from the code changes and they are fed to the input layer of our framework for further processing.

6.2.3 Input Layer

A code change may include changes to multiple files; the changes to each file may contain changes to different hunks; and each hunk contains a list of removed and/or added code lines. To preserve this structural information, in each changed file, we represent the removed (added) code as a three-dimensional matrix, i.e., $\mathcal{B} \in \mathbb{R}^{\mathcal{H} \times \mathcal{L} \times W}$, where $\mathcal{H}$ is the number of hunks, $\mathcal{L}$ is the number of removed (added) code lines for each hunk, and W is the number of words in each removed (added) code line in the affected file. We use $\mathcal{B}_r$ and $\mathcal{B}_a$ to denote the three-dimensional matrix of the removed and added code respectively.

Note that each patch may contain a different number of affected files ($\mathcal{F}$), each file may contain a different number of hunks ($\mathcal{H}$), each hunk may contain a different number of lines ($\mathcal{L}$), and each line may contain a different number of words (W). For parallelization [78, 105], each input instance is padded or truncated to the same $\mathcal{F}$, $\mathcal{H}$, $\mathcal{L}$, and W.

6.2.4 Feature Extraction Layers

The feature extraction layers are used to automatically build an embedding vector representing the code change made to a given file in the patch. The embedding vectors of code changes to multiple files are then concatenated into a single vector representing the code change made by the patch.

As shown in Figure 6.2, for each affected file, the feature extraction layers take as input two matrices (denoted by "-" and "+" in Figure 6.2) representing the removed code and added code, respectively. These two matrices are passed to the *hierarchical attention network* to construct corresponding embedding vectors: $\mathbf{e}_r$ representing the removed code and $\mathbf{e}_a$ representing the added code (see Figure 6.2). These two embedding vectors are fed to the *comparison layers*

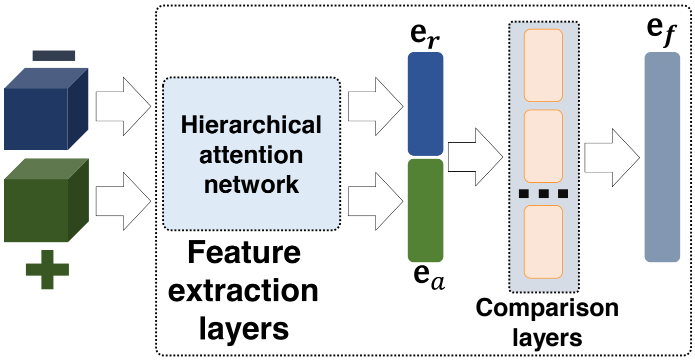


Figure 6.2: Architecture of the feature extraction layers for mapping the code change of the affected file in a given patch to an embedding vector. The input of the module is the removed code and added code of the affected file, denoted by "-" and "+", respectively.

to produce the vectors representing the difference between the removed code and the added code. These vectors are then concatenated to represent the code changes in each affected file. We present the hierarchical attention network and the comparison layers in the following sections.

6.2.4.1 Hierarchical Attention Network

The architecture of our hierarchical attention network (HAN) is shown in Figure 6.3. A HAN takes the removed (added) code of an affected file of a given patch as an input and outputs the embedding vector representing the removed (added) code. Our HAN consists of several parts: a word sequence encoder, a word-level attention layer, a line encoder, a line-level attention layer, a hunk sequence encoder, and a hunk attention layer.

Suppose that the removed (added) code of the affected file contains a sequence of hunks $\mathbf{H} = [t_1, t_2, \ldots, t_{\mathcal{H}}]$, each hunk t_i includes a sequence of lines $[s_{i1}, s_{i2}, \ldots, s_{i\mathcal{L}}]$, and each line s_{ij} contains a sequence of words $[w_{ij1}, w_{ij2}, \ldots, w_{ijW}]$. w_{ijk} with $k \in [1, W]$ represents the word in the $j-$th line in the $i-$th hunk. Now, we describe how the embedding vector of the removed (added) code is built using the hierarchical structure.

Word encoder. Given a line s_{ij} with a sequence of words w_{ijk} and a word embedding matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}^C| \times d}$, where $\mathcal{V}^C$ is the vocabulary containing all

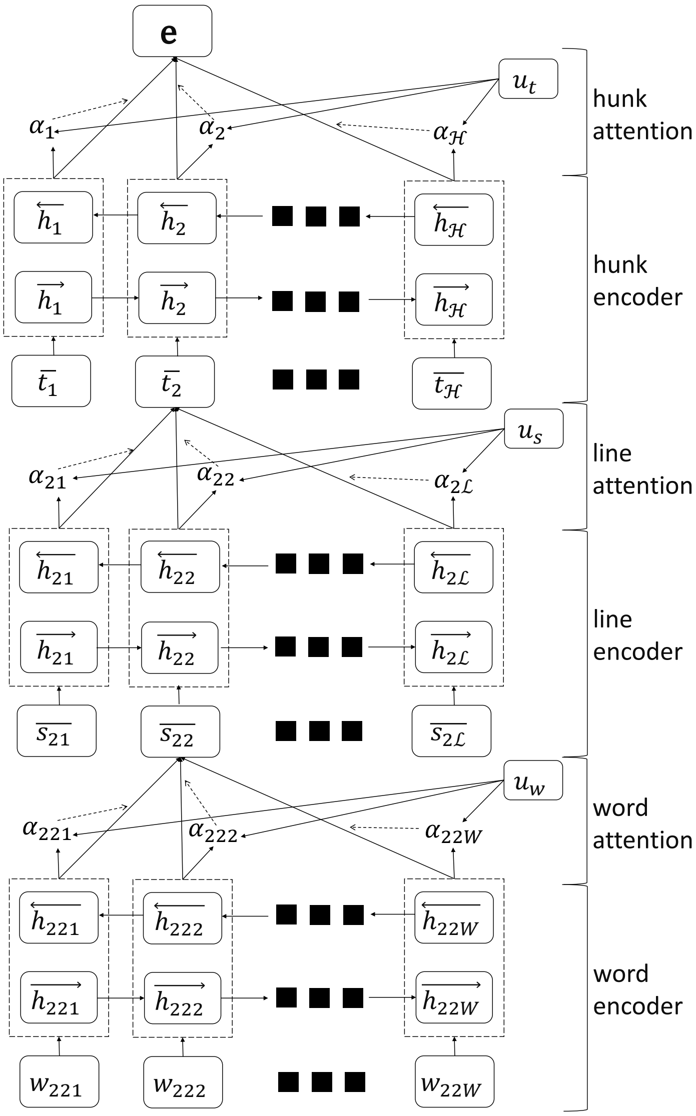

e
u_t
α_1
α_2
$\alpha_{\mathcal{H}}$
hunk attention
$\overleftarrow{h_1}$ $\overleftarrow{h_2}$ $\overleftarrow{h_{\mathcal{H}}}$
$\overrightarrow{h_1}$ $\overrightarrow{h_2}$ $\overrightarrow{h_{\mathcal{H}}}$
hunk encoder
$\overline{t_1}$ $\overline{t_2}$ $\overline{t_{\mathcal{H}}}$
u_s
α_{21} α_{22} $\alpha_{2\mathcal{L}}$
line attention
$\overleftarrow{h_{21}}$ $\overleftarrow{h_{22}}$ $\overleftarrow{h_{2\mathcal{L}}}$
$\overrightarrow{h_{21}}$ $\overrightarrow{h_{22}}$ $\overrightarrow{h_{2\mathcal{L}}}$
line encoder
$\overline{s_{21}}$ $\overline{s_{22}}$ $\overline{s_{2\mathcal{L}}}$
u_w
α_{221} α_{222} α_{22W}
word attention
$\overleftarrow{h_{221}}$ $\overleftarrow{h_{222}}$ $\overleftarrow{h_{22W}}$
$\overrightarrow{h_{221}}$ $\overrightarrow{h_{222}}$ $\overrightarrow{h_{22W}}$
word encoder
w_{221} w_{222} w_{22W}

Figure 6.3: The overall framework of our hierarchical attention network (HAN). The HAN takes as input the removed (added) code of the affected file of a given patch and outputs the embedding vector (denoted by **e**) of the removed (added) code.

words extracted from the code changes and d is the dimension of the representation of word, we first build the matrix representation of each word in the sequence as follows:

$$\overline{w_{ijk}} = \mathbf{W}[w_{ijk}] \tag{6.1}$$

where $\overline{w_{ijk}} \in \mathbb{R}^d$ indicates the vector representation of word w_{ijk} in the word embedding matrix $\mathbf{W}$. We employ a bidirectional GRU to summarize information from the context of a word in both directions [24]. To capture this contextual information, the bidirectional GRU includes a forward GRU that reads the line s_{ij} from w_{ij1} to w_{ijW} and a backward GRU that reads the line s_{ij} from w_{ijW} to w_{ij1}.

$$\begin{aligned} \overrightarrow{h_{ijk}} &= \overrightarrow{GRU}(\overline{w_{ijk}}), k \in [1, W] \\ \overleftarrow{h_{ijk}} &= \overleftarrow{GRU}(\overline{w_{ijk}}), k \in [W, 1] \end{aligned} \tag{6.2}$$

We obtain an annotation of a given word w_{ijk} by concatenating the forward hidden state $\overrightarrow{h_{ijk}}$ and the backward hidden state $\overleftarrow{h_{ijk}}$ of this word, i.e., $h_{ijk} = [\overrightarrow{h_{ijk}} \oplus \overleftarrow{h_{ijk}}]$ ($\oplus$ is the concatenation operator). h_{ijk} summarizes the word w_{ijk} considering its neighboring words.

Word attention. Based on the intuition that not all words contribute equally to extract the "meaning" of the line, we use the attention mechanism to highlight words important for predicting the content of the commit message. The attention mechanism was previously used in source code summarization and was shown to be effective for encoding source code sequences [87, 122]. We also use the attention mechanism to form an embedding vector of the line. We first feed an annotation of a given word w_{ijk} (i.e., h_{ijk}) through a fully connnected layer (i.e., $\mathcal{W}_w$) to get a hidden representation (i.e., u_{ijk}) of h_{ijk} as follows:

$$u_{ijk} = \text{ReLU}(\mathcal{W}_w h_{ijk} + b_w) \tag{6.3}$$

where ReLU is the rectified linear unit activation function [160], as it generally provides better performance in various deep learning tasks [48, 16]. Similar to Yang et al. [229], we define a word context vector (u_w) that can be seen as a high level representation of the answer to the fixed query "what is the most informative word" over the words. The word context vector u_w is randomly initialized and learned during the training process. We then measure the importance of the word as the similarity of u_{ijk} with the word context vector u_w

and get a normalized importance weight α_{ijk} through a softmax function [35]:

$$\alpha_{ijk} = \frac{\exp(u_{ijk}^{\mathrm{T}} u_w)}{\sum_k \exp(u_{ijk}^{\mathrm{T}} u_w)} \tag{6.4}$$

For each line s_{ij}, its vector is computed as a weighted sum of the embedding vectors of the words based on their importance as follows:

$$\overline{s_{ij}} = \sum_k \alpha_{ijk} h_{ijk} \tag{6.5}$$

Line encoder. Given a line vector (i.e., $\overline{s_{ij}}$), we also use a bidirectional GRU to encode the line as follows:

$$\begin{aligned} \overrightarrow{h_{ij}} &= \overrightarrow{GRU}(\overline{s_{ij}}), j \in [1, \mathcal{L}] \\ \overleftarrow{h_{ij}} &= \overleftarrow{GRU}(\overline{s_{ij}}), j \in [\mathcal{L}, 1] \end{aligned} \tag{6.6}$$

Similar to the word encoder, we obtain an annotation of the line s_{ij} by concatenating the forward hidden state $\overrightarrow{h_{ij}}$ and backward hidden state $\overleftarrow{h_{ij}}$ of this line. The annotation of the line s_{ij} is denoted as $h_{ij} = [\overrightarrow{h_{ij}} \oplus \overleftarrow{h_{ij}}]$, which summarizes the line s_{ij} considering its neighboring lines.

Line attention. We use an attention mechanism to learn the important lines to be used to form a hunk vector as follows:

$$u_{ij} = \mathrm{ReLU}(\mathcal{W}_s h_{ij} + b_s) \tag{6.7}$$

$$\alpha_{ij} = \frac{\exp(u_{ij}^{\mathrm{T}} u_s)}{\sum_j \exp(u_{ij}^{\mathrm{T}} u_s)} \tag{6.8}$$

$$\overline{t_i} = \sum_j \alpha_{ij} h_{ij} \tag{6.9}$$

$\mathcal{W}_s$ is the fully connected layer to which we need to feed an annotation of the given line (i.e., s_{ij}). We define u_s as the line context vector that can be seen as a high level representation of the answer to the fixed query “what is the informative line” over the lines. u_s is randomly initialized and learned during the training process. $\overline{t_i}$ is the hunk vector of the i-th hunk in the removed (added) code.

Hunk encoder. Given a hunk vector $\overline{t_i}$, we again use a bidirectional GRU to encode the hunk as follows:

$$\begin{aligned}\overrightarrow{h_i} &= \overrightarrow{GRU}(\overline{t_i}), t \in [1, \mathcal{H}] \\ \overleftarrow{h_i} &= \overleftarrow{GRU}(\overline{t_i}), t \in [\mathcal{H}, 1]\end{aligned} \tag{6.10}$$

An annotation of the hunk t_i is then obtained by concatenating the forward hidden state $\overrightarrow{h_i}$ and the backward hidden state $\overleftarrow{h_i}$, i.e., $h_i = [\overrightarrow{h_i}, \overleftarrow{h_i}]$. h_i summarizes the hunk t_i considering the other hunks around it.

Hunk attention. We again use an attention mechanism to learn important hunks used to form an embedding vector of the removed (added) code as follows:

$$u_i = \text{ReLU}(\mathcal{W}_\text{h} h_i + b_\text{h}) \tag{6.11}$$

$$\alpha_i = \frac{\exp(u_i^\text{T} u_t)}{\sum_i \exp(u_i^\text{T} u_t)} \tag{6.12}$$

$$\mathbf{e} = \sum_i \alpha_i h_i \tag{6.13}$$

$\mathcal{W}_\text{h}$ is the fully connected layer used to feed an annotation of a given hunk (i.e., h_i). u_t is the hunk context vector that can be seen as a high level representation of the answer to the fixed query "what is the informative hunk" over the hunks. Similar to u_w and u_s, u_t is randomly initialized and learned during the training process. $\mathbf{e}$, collected at the end of this part, is the embedding vector of the removed (added) code. For convenience, we denote $\mathbf{e}_r$ and $\mathbf{e}_a$ as the embedding vectors of the removed code and added code, respectively.

6.2.4.2 Comparison Layers

The goal of the comparison layers is to build the vectors that capture the differences between the removed code and added code of the affected file in a given patch. We use multiple comparison functions [213] to represent different

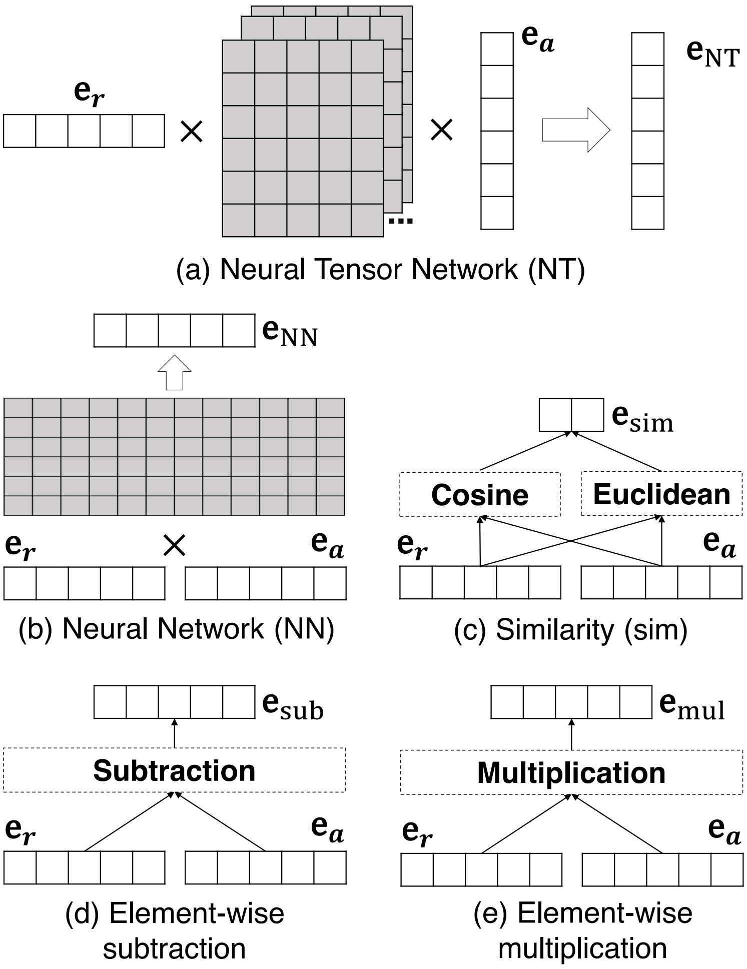


Figure 6.4: A list of comparison functions in the comparison layers.

angles of comparison. These comparison functions were previously used in a question answering task. The comparison layers take as input the embedding vectors of the removed code and added code (denoted by $\mathbf{e}_r$ and $\mathbf{e}_a$, respectively) and output the vectors representing the difference between the removed code and the added code. These vectors are then concatenated to represent an embedding vector of the affected file in the given patch. Figure 6.4 shows the five comparison functions used in the comparison layers to capture the difference between the removed code and added code. We briefly explain these comparison functions in the following paragraphs.

(a) Neural Tensor Network. Inspired by previous works in visual question answering [25], we employ a neural tensor network [193] as follows:

$$\mathbf{e}_{\mathrm{NT}} = \mathrm{ReLU}(\mathbf{e}_r^{\mathrm{T}} \mathbf{T}^{[1,\ldots,n]} \mathbf{e}_a + b_{\mathrm{NT}}) \tag{6.14}$$

$\mathbf{T}^i \in \mathbb{R}^{n \times n}$ is a tensor and b_{NT} is the bias value. These parameters are learned during the training process. Note that both the removed code and added code have the same dimension (i.e., $\mathbf{e}_r \in \mathbb{R}^n$, $\mathbf{e}_a$).

(b) Neural Network. We consider a simple feed forward neural network [197]. The output is computed as follows:

$$\mathbf{e}_{\mathrm{NN}} = \mathrm{ReLU}(\mathbf{W}[\mathbf{e}_a \oplus \mathbf{e}_r] + b_{\mathrm{NN}}) \tag{6.15}$$

$\oplus$ is the concatenation operator, the matrix $\mathbf{W} \in \mathbb{R}^{n \times 2n}$, and the bias value b_{NN} are parameters to be learned.

(c) Similarity. We employ two different similarity measures, euclidean distance and cosine similarity, to capture the similarity between the removed code and added code as follows:

$$\begin{aligned} \mathbf{e}_{\mathrm{sim}} &= \mathrm{EUC}(\mathbf{e}_r, \mathbf{e}_a) \oplus \mathrm{COS}(\mathbf{e}_r, \mathbf{e}_a) \\ \mathrm{EUC}(\mathbf{e}_r, \mathbf{e}_a) &= ||\mathbf{e}_r - \mathbf{e}_a||_2 \\ \mathrm{COS}(\mathbf{e}_r, \mathbf{e}_a) &= \frac{\mathbf{e}_r \mathbf{e}_a}{||\mathbf{e}_r||||\mathbf{e}_a||} \end{aligned} \tag{6.16}$$

$\mathrm{EUC}(\cdot)$ and $\mathrm{COS}(\cdot)$ are the euclidean distance and cosine similarity, respectively. Note that $\mathbf{e}_{\mathrm{sim}}$ is a two-dimensional vector.

(d) Element-wise subtraction. We simply perform a subtraction between the embedding vector of the removed code and the embedding vector of the added code.

$$\mathbf{e}_{\mathrm{sub}} = \mathbf{e}_r - \mathbf{e}_a \tag{6.17}$$

(e) Element-wise multiplication. We perform element-wise multiplication

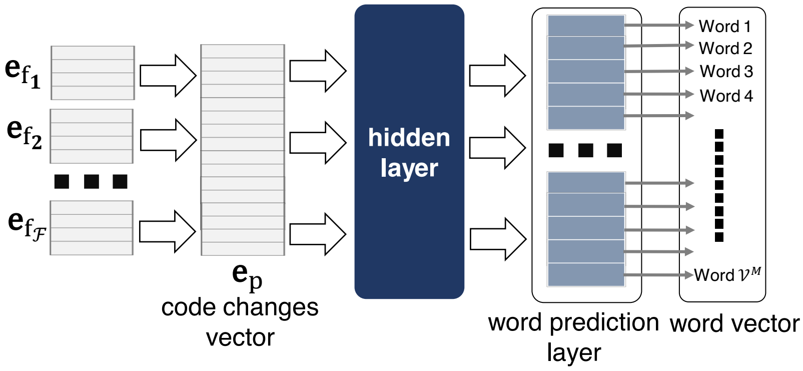


Figure 6.5: The details of the red dashed box in Figure 6.1. It takes as input a list of embedding vectors of the affected files of a given patch (i.e., $\mathbf{e}_{\mathtt{f}_1}$, $\mathbf{e}_{\mathtt{f}_2}$, . . . , $\mathbf{e}_{\mathtt{f}_\mathcal{F}}$). $\mathbf{e}_{\mathrm{p}}$ is the vector representation of the code change and is fed to a hidden layer to produce the word vector (i.e., the probability distribution over words). $\mathcal{V}^M$ is a set of words extracted from the first line of the commit messages.

for the embedding vectors of the removed code and added code.

$$\mathbf{e}_{\mathrm{mul}} = \mathbf{e}_r \odot \mathbf{e}_a \tag{6.18}$$

where $\odot$ is the element-wise multiplication operator.

The vectors resulting from applying these five different comparison functions are then concatenated to represent the embedding vector of the affected file (denoted by $\mathbf{e}_{\mathtt{f}_\mathrm{i}}$) in the given patch as follows:

$$\mathbf{e}_{\mathtt{f}_\mathrm{i}} = \mathbf{e}_{\mathrm{NT}} \oplus \mathbf{e}_{\mathrm{NN}} \oplus \mathbf{e}_{\mathrm{sim}} \oplus \mathbf{e}_{\mathrm{sub}} \oplus \mathbf{e}_{\mathrm{mul}} \tag{6.19}$$

where $\mathtt{f}_\mathrm{i}$ is the i-th file of the code change in the given patch.

6.2.5 Feature Fusion and Word Prediction Layers

Figure 6.5 shows the details of the part of the architecture shown inside the red (dashed) box in Figure 6.1. The inputs of this part are the list of embedding vectors (i.e., $\mathbf{e}_{\mathtt{f}_1}$, $\mathbf{e}_{\mathtt{f}_2}$, . . . , $\mathbf{e}_{\mathtt{f}_\mathcal{F}}$) representing the features extracted from the list of affected files of a given patch. These embedding vectors are concatenated to construct a new embedding vector ($\mathbf{e}_{\mathrm{p}}$) representing the code change in a given patch as follows:

$$\mathbf{e}_{\mathrm{p}} = \mathbf{e}_{\mathbf{f}_1} \oplus \mathbf{e}_{\mathbf{f}_2} \oplus \cdots \oplus \mathbf{e}_{\mathbf{f}_{\mathcal{F}}} \tag{6.20}$$

We pass the embedding vector ($\mathbf{e}_{\mathrm{p}}$) into a hidden layer (a fully connected layer) to produce a vector $\mathbf{h}$:

$$\mathbf{h} = \alpha(\mathbf{w}_{\mathbf{h}}\mathbf{e}_{\mathrm{p}} + b_{\mathbf{h}}) \tag{6.21}$$

where $\mathbf{w}_{\mathbf{h}}$ is the weight matrix used to connect the embedding vector $\mathbf{e}_{\mathrm{p}}$ with the hidden layer and $b_{\mathbf{h}}$ is the bias value. Finally, the vector $\mathbf{h}$ is passed to a word prediction layer to produce the following:

$$\mathbf{o} = -\mathbf{h}\mathbf{w}_{\mathbf{o}} \tag{6.22}$$

where $\mathbf{w}_{\mathbf{o}}$ is the weight matrix between the hidden layer and the word prediction layer, and $\mathbf{o} \in \mathbb{R}^{|\mathcal{V}^M| \times 1}$ ($\mathcal{V}^M$ is a set of words extracted from the first line of commit messages). We then apply the sigmoid function [35] to get the probability distribution over words as follows:

$$\mathbf{p}(o_i|p_i) = \frac{1}{1 + \exp(o_i)} \tag{6.23}$$

where $o_i \in \mathbf{o}$ is the probability score of the i^{th} word and p_i is the patch that we want to assign words to.

6.2.6 Parameter Learning

Our model involves the following parameters: the word embedding matrix of code changes, the hidden states in the different encoders (i.e., the word encoder, line encoder, and hunk encoder), the context vectors of words, lines, and hunks, the weight matrices and the bias values of the neural tensor network and the neural net in the comparison layers, and the weight matrices and the bias values of the hidden layer and the word prediction layer. After these parameters are learned, the vector representation of the code change of each patch can be determined. These parameters are learned by minimizing the following objective function:

$$\begin{aligned}\mathcal{O} = \sum_{y_i \in \mathbf{y}} (y_i \times -\log(\mathbf{p}(o_i|p_i)) + (1 - y_i) \\ \times -\log(1 - \mathbf{p}(o_i|p_i))) + \frac{\lambda}{2}\|\theta\|_2^2\end{aligned} \tag{6.24}$$

where $\mathbf{p}(o_i|p_i)$ is the predicted word probability defined in Equation 6.23, $y_i = \{0, 1\}$ indicates whether the i-th word is part of the commit message of the patch p_i, and θ are all parameters of our model. The regularization term, $\frac{\lambda}{2}\|\theta\|_2^2$, is used to prevent overfitting in the training process [38]. We employ the dropout technique [195] to improve the robustness of CC2Vec. Since Adam [107] has been shown to be computationally efficient and require low memory consumption, we use it to minimize the objective function (i.e., Equation 6.24). We also use backpropagation [68], a simple implementation of the chain rule of partial derivatives, to efficiently update the parameters during the training process.

6.3 Experiments

The goal of this work is to build a representation of code changes that can be applied to multiple tasks. To evaluate the effectiveness of this representation, we employ our framework, namely CC2Vec, on three different tasks, i.e., commit message generation [139], bug fixing patch identification [79] and just-in-time defect prediction [78].

In the first task of commit message generation, we use the vector representation of code changes, extracted by CC2Vec, to find a patch that is most similar to another. For the other two tasks, CC2Vec is used to extract additional features that are input to the models of bug fixing patch identification and just-in-time defect prediction. We compare the resulting performance with and without using our code change vector. We next elaborate on the three tasks, the baselines, and results.

6.3.1 Task 1: Commit Message Generation

6.3.1.1 Problem Formulation

While we learn representations of code changes with the aid of commit messages, we also study the task of generating commit messages from code changes. Developers do not always write high-quality commit messages. Dyer et al. [55] reported that around 14% of commit messages in 23,000 Java projects on SourceForge[2] were empty. commit messages are important for program comprehension and understanding the evolution of software, therefore this motivates the need for the automatic generation of commit messages. In this task, given the code change of a given patch, we aim to produce a brief commit message summarizing it.

6.3.1.2 Prior Approaches

One of the prior approaches is *NNGen* [139], which takes as input a new code change with an unknown commit message and a training dataset (patches), and outputs a commit message for the new code change. NNGen first extracts code changes from the training set. Each code change in the training set and the new code change are represented as vectors in the form of a "bag-of-words" [148]. NNGen then calculates the cosine similarity between the vector of the new code change and the vector of each code change in the training set, and selects the top-k nearest neighbouring code changes in the training dataset. From these k nearest neighbours, the BLEU-4 score [169] is computed between each of the code changes in the top-k and the new code change with an unknown commit message. A commit message of the code change in the top-k with the highest BLEU-4 score is reused as the commit message of the

[2] https://sourceforge.net/

new code change.

The BLEU-4 score is a measure used to evaluate the quality of machine translation systems, measuring the closeness of a translation to a human translation. It is computed as follows:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log\left(p_n\right)\right)$$

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

N is the maximum number of N-grams. Following the previous work [139], we select $N = 4$. p_n is the ratio of length n subsequences that are present in both the output and reference translation. BP is a brevity penalty to penalize short output sentences. Finally, c is the length of the output translation and r is the length of the reference translation.

A deep learning approach was previously proposed for this task by Jiang et al. [87], however, it underperformed the simpler baseline NNGen. In this study, we refer to their work as *NMT*. Their approach modelled this task as a neural machine translation task, translating the code change to a target commit message. Like our work, they proposed an attention-based model, however, our work differs from theirs as ours incorporates the structure of code changes. Liu et al. [139] investigated the performance of Jiang et al.'s attention model; they found that once they remove trivial and automatically-generated messages, the performance of the model decreased significantly, suggesting that this model does not generalize.

6.3.1.3 Our Approach

To use CC2Vec for this task, we propose *LogGen*. Similar to the nearest neighbours approach used by Liu et al. [139], LogGen reuses and outputs a commit message from the training set. However, instead of treating each code change as a bag of words, LogGen uses code change vectors produced by CC2Vec. CC2Vec is first trained over the training dataset. Given a new code change from the test dataset with an unknown commit message, we find the code changes with a known commit message that have the closest CC2Vec vector. Like Liu et al. [139], after identifying the closest code changes, we reuse the commit message as the output.

6.3.1.4 Experimental Setting

The purpose of evaluating CC2Vec on this task is to determine if the code change representations received from CC2Vec outperform the naive representation used by Liu et al. [139]. Jiang et al. [87] originally collected and filtered the commits to construct the original dataset. Another version of the dataset was used by Liu et al. [139], who modified the original dataset.

Jiang et al. extracted a total of 2 million patches from the 1K most starred Java projects. They collected the first line of each commit message. To normalize the dataset, patch ids and issue ids were removed from the code changes and commit messages. Patches were filtered to remove merges, rollbacks, and patches that were too long. The commit messages that do not conform to verb-direct-object pattern, e.g. "delete a method", are also removed. After filtering, the dataset contains 32K patches.

Still, even with all this cleaning, Liu et al. [139] investigated the dataset and found that there were many patches with bot messages and trivial messages. Bot messages refer to messages produced automatically by other development tools, such as continuous integration bots. Trivial messages refer to messages containing only information that can be obtained by looking at the names of the changed files (e.g. "modify dockerfile"). Such messages are of low quality and Liu et al. used regular expressions to locate and remove these patches.

We used the original dataset of Jiang et al. [87] and the cleaned dataset of Liu et al. [139] for evaluation. While the original dataset consists of a training dataset of 30K patches and a testing dataset of 3K patches, the cleaned dataset consists of a training dataset of 22K patches and a testing dataset of 2.5K

Table 6.1: Performance of each approach on the original and cleaned dataset reported in BLEU-4

	LogGen	NNGen	NMT
Original	**43.20**	38.55	31.92
Clean	**20.48**	16.42	14.19

patches. To compare the different approaches, we use BLEU-4 to evaluate each approach since this was used in both previous works.

6.3.1.5 Results

We report the performance of LogGen, NNGen and NMT in Table 6.1. LogGen outperforms both NNGen and NMT. The *Clean* dataset refers to the dataset which Liu et al. filtered out patches with bot and trivial commit messages. On this dataset, LogGen outperforms NNGen and NMT by a BLEU-4 score of 4.06 and 6.29 respectively. LogGen *improves* over the performance of NNGen by 24.75%, a greater improvement than NNGen's improvement over NMT of 15.70% . On the original dataset collected by Jiang et al., LogGen outperforms NNGen and NMT by a BLEU-4 score of 4.65 and 11.28. These results indicate that LogGen can improve over the performance of NNGen and NMT by 12.06% and 2.07% in terms of the BLEU-4 score respectively.

Thus, we conclude that the commit messages retrieved by LogGen are closer in quality to a human translation than those retrieved by NNGen and the commit messages generated by NMT. This suggests that CC2Vec produces vector representations of patches that correlate to the meaning of the patch more strongly than a bag-of-words.

6.3.2 Task 2: Bug Fixing Patch Identification

6.3.2.1 Problem Formulation

Software requires continuous evolution to keep up with new requirements, but this also introduces new bugs. Backporting bugfixes to older versions of a project may be required when a legacy code base is supported. For example, Linux kernel developers regularly backport bugfixes from the latest version to older versions that are still under support. However, the maintainers of older versions may overlook relevant patches in the latest version. Thus, an automated method to identify bug fixing patches may be helpful. We treat the problem as a binary classification problem, in which each patch is labelled as a bug-fixing patch or not. Given the code change and commit message, we produce one of the two labels as the output.

6.3.2.2 Prior Approaches

The prior approach of this problem is *PatchNet* (Chapter 5), which represents the removed (added) code as a three dimensional matrix. The dimensions of the matrix are the number of hunks, the number of lines in each hunk, and the number of words in each line. PatchNet employs a 3D-CNN [86] that automatically extracts features from this matrix. Unfortunately, the 3D-CNN lacks a mechanism to identify important words, lines, and hunks. To address this limitation, we propose a specialized hierarchical attention neural network to quantify the importance of words, lines, and hunks in our model (CC2Vec).

Another approach was proposed by Tian et al. [204] that combines Learning from Positive and Unlabelled examples (LPU) [126] and Support Vector Machine (SVM) [88] to build a patch classification model. Unlike CC2Vec, this approach requires the use of manually selected features. These features include word features, which is a “bag-of-words” extracted from commit messages, and 52 features, manually extracted from the code change (e.g., the number of loops added in a patch and if certain words appear in the commit message).

6.3.2.3 Our Approach

CC2Vec is first used to learn a distributed representation of code changes on the whole dataset. All patches from the training and test dataset are used since the commit messages of the test dataset are not the target of the task. Next, we integrate these vector representations of the code changes with the two existing approaches. To use CC2Vec in PatchNet, we concatenate the vector representation of the code change extracted by CC2Vec with the two embedding vectors extracted from the commit message and code change by PatchNet to form a new embedding vector. The new embedding vector is fed into PatchNet's classification module to predict whether a given patch is a bug fixing patch. For the approach proposed by Tian et al. [204] which uses an SVM as the classifier, we pass the vectors produced by CC2Vec from the code change into the SVM as features.

6.3.2.4 Experimental Setting

The goal of this task is to investigate if CC2Vec helps existing approaches to effectively classify bug-fixing patches. We use the dataset of Linux kernel bug-fixing patches used in the PatchNet paper. This dataset consists of 42K bug-fixing patches and 40K non-bug-fixing patches collected from the Linux kernel versions v3.0 to v4.12, released in July 2011 and July 2017 respectively. Patches in this dataset are limited to 100 lines of changed code, in line with the Linux kernel stable patch guidelines. The non-bug-fixing patches are selected such that they have a similar size, in terms of the number of files and the number of modified lines, as the bug-fixing patches. Following the PatchNet paper, we use 5-fold cross-validation for the evaluation.

To compare the performance of the approaches, we employ the following metrics:

- *Accuracy*: The ratio of correct predictions to the total number of predictions.

Table 6.2: Evaluation of the approaches on the bug-fixing patch identification task

	Acc.	Prec.	Recall	F1	AUC
LPU-SVM	73.1	75.1	71.6	73.3	73.1
LPU-SVM + CC2Vec	**77.1**	**77.2**	**79.8**	**78.5**	**76.2**
PatchNet	86.2	83.9	90.1	87.1	86.0
PatchNet + CC2Vec	**90.7**	**91.6**	**90.1**	**90.9**	**91.6**

- *Precision*: The ratio of correct predictions of bug-fixing patches to the total number of bug-fixing patch predictions

- *Recall*: The ratio of correct predictions of bug-fixing patches to the total number of bug-fixing patches.

- *F1*: Harmonic mean between precision and recall.

- *AUC*: Area under the curve plotting the true positive rate against the false positive rate. AUC values range from 0 to 1, with a value of 1 indicating perfect discrimination.

These metrics were also used in previous studies on this task.

6.3.2.5 Results

We report the performance of the different approaches in Table 6.2. We observe that the best performing approach is PatchNet augmented with CC2Vec. For both Tian et al.'s model (LPU-SVM) and PatchNet, the versions augmented with CC2Vec outperform the original versions. Specifically, CC2Vec helps to improve the best performing baseline (i.e, PatchNet) by 5.22%, 9.18%, 4.37%, and 6.51% in terms of accuracy, precision, F1, and AUC. CC2Vec also helps to improve the performance of LPU-SVM by 5.47%, 2.80%, 11.45%, 7.09%, and 4.24% in accuracy, precision, recall, F1, and AUC. This suggests that CC2Vec can learn patch representations that are general and useful beyond the task it was trained on.

6.3.3 Task 3: Just-in-Time Defect Prediction

6.3.3.1 Problem Formulation

The task of just-in-time (JIT) defect prediction refers to the identification of defective patches. JIT defect prediction tools provide early feedback to software developers to optimize their effort for inspection, and have been used at large software companies [156, 189, 199]. We model the task as a binary classification task, in which each patch is labelled as a patch containing a defect or not. Given a patch containing a code change and a commit message with unknown label, we label the patch with one of the two labels.

6.3.3.2 Prior Approach

The prior approach is *DeepJIT* which is presented in Chapter 4. DeepJIT takes as input the commit message and code change of a given patch and outputs a probability score to predict whether the patch is buggy. DeepJIT employs a Convolutional Neural Network (CNN) [105] to automatically extract features from the code change and commit message of the given patch. However, DeepJIT ignores information about the structure of the removed code or added code, instead relying on CNN to automatically extract such information.

6.3.3.3 Our Approach

Similar to the previous task (i.e., bug fixing patch identification), CC2Vec is first used to learn distributed representations of the code changes in the whole dataset. All patches from the training and test dataset are used since the commit messages of the test dataset are not part of the predictions of the task. We then integrate CC2Vec with DeepJIT. To use CC2Vec with DeepJIT, for each patch, we concatenate the vector representation of the code change extracted by CC2Vec with two embedding vectors extracted from the commit message and code change of the given patch extracted by DeepJIT to form a new embedding vector. The new embedding vector is fed into DeepJIT's

Table 6.3: The AUC results of the various approaches

	QT	OPENSTACK
DeepJIT	76.8	75.1
DeepJIT + CC2Vec	**82.2**	**80.9**

feature combination layers, to predict whether the given patch is defective.

6.3.3.4 Experimental Setting

The purpose of this task is to evaluate if CC2Vec can be used to augment existing approaches in effectively classifying defective patches. Our evaluation is performed on two datasets, the *QT* and *OPENSTACK* datasets, which contain patches collected from the QT and OPENSTACK software projects respectively by McIntosh and Kamei [151]. The QT dataset contains 25K patches over 2 years and 9 months while the OPENSTACK dataset contains 12K patches over 2 years and 3 months. 8% and 13% of the patches are defective in the QT dataset and the OPENSTACK datasets respectively. Like Hoang et al. [78], we use 5-fold cross validation for the evaluation.

To compute the effectiveness of the approaches, we use the Area Under the receiver operator characteristics Curve (AUC), similar to the previous studies.

6.3.3.5 Results

The evaluation results for this task are reported in Table 6.3. The use of CC2Vec with DeepJIT improves the AUCS score of DeepJIT, from 76.8 and 75.1 to 82.2 and 80.9 on the QT and OPENSTACK datasets respectively. Specifically, CC2Vec helps to improve the AUC metric by 7.03% and 7.72% for the QT and OPENSTACK datasets, respectively, as compared to DeepJIT. This indicates that CC2Vec is effective in learning a useful representation of patches that an existing state-of-the-art technique can utilize.

Table 6.4: Results of an ablation study

	Log generation (BLEU-4)		**Bug fix identification (F1)**		**Just-in-time defect prediction (AUC)**			
	Clean	Drops by (%)	BFP	Drops by (%)	QT	Drops by (%)	OPENSTACK	Drops by (%)
All−all	18.30	10.64	87.1	4.18	77.4	5.84	76.7	5.19
All−NT	19.36	5.47	88.7	2.42	79.8	2.92	79.2	2.10
All−NN	19.80	3.32	88.8	2.31	80.1	2.55	79.5	1.73
All−sim	20.41	0.34	90.2	0.77	81.9	0.36	80.5	0.49
All−sub	20.13	1.71	89.6	1.43	80.7	1.82	80.1	0.99
All−mul	20.25	1.12	89.7	1.32	81.1	1.34	80.5	0.49
All	**20.48**	0	**90.9**	0	**82.2**	0	**80.9**	0

6.4 Discussion

6.4.1 Ablation Study

Our approach involves five comparison functions for calculating the difference between the removed code and added code. To estimate the usefulness of comparison functions (see Section 6.2.4.2), we conduct an ablation study on the three tasks: commit message generation, bug fixing patch identification, and just-in-time defect prediction. Specifically, we first remove the comparison functions entirely and then remove these functions one-by-one. For each task, we compare the CC2Vec model and its six reduced variants: All−all (omit all comparison functions), All−NT (omit the neural network tensor comparison function), All−NN (omit the neural network comparison function), All−sim (omit the similarity comparison function), All−sub (omit the subtraction comparison function), and All−mul (omit the multiplication comparison function).

Table 6.4 summarizes the results of our ablation test on three different tasks. We see that CC2Vec model always performs better than the reduced variants for all three tasks. This suggests that each comparison function plays an important role and omitting these comparison functions may greatly affect the overall performance. All−all (CC2Vec model without using any comparison functions) performs the worst. Among the five remaining variants (i.e., All−NT, All−NN, All−sim, All−sub, and All−mul), All-NT performs the worst. This suggests that the *neural network tensor* comparison function is more important the other comparison functions (i.e., *neural network*, *similarity*, *subtraction*, and *multiplication*).

6.4.2 Threats to Validity

Threats to internal validity refer to errors in our experiments and experimenter bias. For each task, we reuse existing implementations of the baseline approaches whenever available. We have double checked our code and data, but errors may remain.

Threats to external validity concern the generalizability of our work. In our experiments, we have studied only three tasks to evaluate the generality of CC2Vec. This may be a threat to external validity since CC2Vec may not generalize beyond the tasks that we have considered. However, each task involves different software projects and different programming languages. As such, we believe that there is minimal threat to external validity. To minimize threats to construct validity, we have used the same evaluation metrics that were used in previous studies.

6.5 Conclusion

We propose CC2Vec, which produces distributed representations of code changes through a hierarchical attention network. In CC2Vec, we model the structural information of a code change and use the attention mechanism to identify important aspects of the code change with respect to the cc2vec message accompanying it. This allows CC2Vec to learn high-quality vector representations that can be used in existing state-of-the-art models on tasks involving code changes.

We empirically evaluated CC2Vec on three tasks and demonstrated that approaches using or augmented with CC2Vec embeddings outperform existing state-of-the-art approaches that do not use the embeddings. Finally, we performed an ablation study to evaluate the usefulness of comparison functions. The results show that the comparison functions play an important role and omitting them in part or in full affects the overall performance.

Chapter 7

Conclusion and Future Work

7.1 Main Contributions

As software engineering corpora grow significantly in both size and complexity, finding bugs becomes challenging, time-consuming, and very costly. My thesis focuses on analyzing software engineering corpora to save developers' time and effort in improving code quality. Specifically, I propose solutions for three software engineering tasks: bug localization, just-in-time defect prediction, and bug fixing patch identification. Moreover, I propose a deep learning framework that can be used to construct a distributed vector representation of code changes. The distributed vector representation can be used for many software engineering tasks related to code changes.

The contributions of my dissertation are as follows:

1. I introduce a multi-modal approach, namely NetML, to utilize the information from both bug reports and program spectra to localize bugs. NetML employs a network Lasso to incentivize the model parameters of similar bug reports (or program elements) to be close together.

2. I propose an end-to-end deep learning framework (DeepJIT) that automatically extracts features from commit messages and code changes in a given commit. These features are then put to a fully-connected

layer to build a model that identifies defective commits. The experiments on two well-known dataset (QT and OPENSTACK) show that our approach outperforms the best performing state-of-the-art approach in term of AUC.

3. I propose a hierarchical deep learning-based approach, named PatchNet, capable of automatically extracting features from commit messages and code changes and using them to identify stable patches. Unlike DeepJIT which ignores the structure of code changes, PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of code changes by separating the removed and added code of the code changes. Extensive experiments on the Linux kernel dataset confirm the superiority of PatchNet.

4. I introduce a novel deep learning model, namely CC2Vec, that learns distributed representations of code changes guided by the semantic meaning contained in commit messages. The experiments on the three software engineering tasks (i.e., commit message generation, bug fixing patch identification, and just-in-time defect prediction) show the effectiveness of CC2Vec.

I have proposed some statistical and deep learning models in various software engineering tasks. However, there are still some limitations of the current approaches as their effectiveness is not optimal yet. The models also have not considered all the pieces of information that are available and can be used to boost effectiveness further. For example, CC2Vec has only considered added and deleted code and does not consider surrounding code that may also be useful to infer the semantics of the change. We plan to address these and other limitations in future work.

7.2 Future Work

In this dissertation, I propose some statistical and deep learning models (i.e., NetML, DeepJIT, PatchNet, and CC2Vec) for some software engineering problems. Despite these approaches have been proved to match or even surpass human performance, they still exhibit unexpected behaviors under some circumstances. For example, in a reported incident,[1] an autonomous driving car slammed into a white truck as the car's radar fails to recognize the white truck on the bright sky. For this reason, there is an urgent need to estimate an error or unexpected behaviors of statistical and deep learning models. As future work, I want to investigate statistical and deep learning models' reliability. Specifically, I wish to know whether the predicted outputs of these models (including those that I have designed and more) given a set of inputs are reliable to make a correct decision.

[1] https://www.wired.com/2017/01/probing-teslas-deadly-crash-feds-say-yay-self-driving/

Bibliography

[1] Apache Ant. http://ant.apache.org/. Accessed: 2015-07-15.

[2] Apache Commons-Lang. https://commons.apache.org/proper/commons-lang/. Accessed: 2015-07-15.

[3] Apache Commons-Math. http://commons.apache.org/proper/commons-math/. Accessed: 2015-07-15.

[4] Apache Lucene. http://lucene.apache.org/core/. Accessed: 2015-07-15.

[5] AspectJ. http://eclipse.org/aspectj/. Accessed: 2015-07-15.

[6] Cobertura: A code coverage utility for Java. http://cobertura.github.io/cobertura/. Accessed: 2015-07-15.

[7] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/. Accessed: 2015-07-15.

[8] Joda-time. http://www.joda.org/joda-time/. Accessed: 2015-07-15.

[9] Mysql 5.6 full-text stopwords. http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html. Accessed: 2015-07-15.

[10] Rhino. http://developer.mozilla.org/en-US/docs/Rhino. Accessed: 2015-07-15.

[11] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[12] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[13] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[14] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[15] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[16] George A Anastassiou. Univariate hyperbolic tangent neural network approximation. *Mathematical and Computer Modelling*, 53(5-6):1111–1132, 2011.

[17] Marios Anthimopoulos, Stergios Christodoulidis, Lukas Ebner, Andreas Christe, and Stavroula Mougiakakou. Lung pattern classification for interstitial lung diseases using a deep convolutional neural network. *IEEE transactions on medical imaging*, 35(5):1207–1216, 2016.

[18] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[19] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *arXiv preprint arXiv:1802.06509*, 2018.

[20] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60. ACM, 2010.

[21] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 265–274. IEEE, 2010.

[22] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26. ACM, 2007.

[23] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[24] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[25] Yalong Bai, Jianlong Fu, Tiejun Zhao, and Tao Mei. Deep attention neural tensor network for visual question answering. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 20–35, 2018.

[26] Randolph E Bank and Donald J Rose. Global approximate newton methods. *Numerische Mathematik*, 37(2):279–295, 1981.

[27] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[28] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995.

[29] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[30] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[31] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[32] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[33] Tegawende F Bissyande, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Reveillere. Empirical evaluation of bug linking. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 89–98. IEEE, 2013.

[34] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[35] Guillaume Bouchard. Efficient bounds for the softmax function and applications to approximate inference in hybrid models. In *NIPS 2007 workshop for approximate Bayesian inference in continuous/hybrid systems*, 2007.

[36] Thorsten Brants. Natural language processing in information retrieval. In *CLIN*, 2003.

[37] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[38] Rich Caruana, Steve Lawrence, and C Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001.

[39] Gert Cauwenberghs and Tomaso Poggio. Incremental and decremental support vector machine learning. In *Advances in neural information processing systems*, pages 409–415, 2001.

[40] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 376–387. IEEE, 2017.

[41] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[42] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter*, 6(1):1–6, 2004.

[43] Brian Cheung. Convolutional neural networks applied to human face classification. In *2012 11th International Conference on Machine Learning and Applications*, volume 2, pages 580–583. IEEE, 2012.

[44] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[45] Michael Collins, Robert E Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

[46] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

[47] Nello Cristianini, John Shawe-Taylor, et al. *An introduction to support vector machines and other kernel-based learning methods.* Cambridge university press, 2000.

[48] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8609–8613. IEEE, 2013.

[49] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[50] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2018.

[51] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[52] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433. ACM, 2018.

[53] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[54] Cicero Dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.

[55] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[56] Marco DAmbros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[57] Vasiliki Efstathiou and Diomidis Spinellis. Semantic source code models using identifier embeddings. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 29–33. IEEE Press, 2019.

[58] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[59] Everything you ever wanted to know about Linux-stable releases. https://www.kernel.org/doc/html/v4.15/process/stable-kernel-rules.html.

[60] Kenneth C Feldt. *Programming Firefox: Building rich internet applications with XUL.* " O'Reilly Media, Inc.", 2007.

[61] John Fox. *Applied regression analysis, linear models, and related methods.* Sage Publications, Inc, 1997.

[62] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. *Future Generation Computer Systems*, 2018.

[63] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[64] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[65] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[66] Qiong Gu, Zhihua Cai, Li Zhu, and Bo Huang. Data mining on imbalanced data sets. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 1020–1024. IEEE, 2008.

[67] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[68] Martin T Hagan and Mohammad B Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE transactions on Neural Networks*, 5(6):989–993, 1994.

[69] David Hallac, Jure Leskovec, and Stephen Boyd. Network lasso: Clustering and optimization in large graphs. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 387–396. ACM, 2015.

[70] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[71] Simon Haykin. *Kalman filtering and neural networks*, volume 47. John Wiley & Sons, 2004.

[72] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[73] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 163–174. ACM, 2018.

[74] Benjamin Mako Hill. *The official Ubuntu book*. Pearson Education India, 2006.

[75] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[76] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[77] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[78] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 34–45. IEEE Press, 2019.

[79] Thong Hoang, Julia Lawall, Yuan Tian, Richard J Oentaryo, and David Lo. Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering*, 2019.

[80] Thong Van-Duc Hoang, Richard J Oentaryo, Tien-Duy Bui Le, and David Lo. Network-clustered multi-modal bug localization. *IEEE Transactions on Software Engineering*, 2018.

[81] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[82] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[83] Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *IJCAI*, pages 1909–1915, 2017.

[84] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, pages 1606–1612, 2016.

[85] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[86] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2012.

[87] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press, 2017.

[88] Thorsten Joachims. Svmlight: Support vector machine. *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund*, 19(4), 1999.

[89] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2002.

[90] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*, 2014.

[91] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[92] Karen Sparck Jones. *Readings in information retrieval.* Morgan Kaufmann, 1997.

[93] Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London, 2014.

[94] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[95] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[96] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[97] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[98] Yasutaka Kamei and Emad Shihab. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 5, pages 33–45. IEEE, 2016.

[99] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[100] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[101] Carl T Kelley. *Solving nonlinear equations with Newton's method*, volume 1. Siam, 2003.

[102] Martin Kellogg. Combining bug detection and test case generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1124–1126. ACM, 2016.

[103] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[104] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[105] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[106] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. Structured attention networks. *arXiv preprint arXiv:1702.00887*, 2017.

[107] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[108] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 262–265. ACM, 2014.

[109] Ekrem Kocaguneli, Tim Menzies, and Jacky W Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, 38(6):1403–1416, 2011.

[110] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.

[111] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[112] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE, 2015.

[113] Bruno Korbar, Andrea M Olofson, Allen P Miraflor, Catherine M Nicka, Matthew A Suriawinata, Lorenzo Torresani, Arief A Suriawinata, and Saeed Hassanpour. Deep learning for classification of colorectal polyps on whole-slide images. *Journal of pathology informatics*, 8, 2017.

[114] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[115] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[116] Matjaz Kukar, Igor Kononenko, et al. Cost-sensitive learning with neural networks. In *ECAI*, pages 445–449, 1998.

[117] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229. IEEE, 2017.

[118] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[119] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–125. IEEE, 2015.

[120] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590. ACM, 2015.

[121] Tien-Duy B Le, Shaowei Wang, and David Lo. Multi-abstraction concern localization. In *2013 IEEE International Conference on Software Maintenance*, pages 364–367. IEEE, 2013.

[122] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, pages 795–806. IEEE Press, 2019.

[123] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[124] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[125] Gwendolyn K Lee and Robert E Cole. From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development. *Organization science*, 14(6):633–649, 2003.

[126] Wee Sun Lee and Bing Liu. Learning with positive and unlabeled examples using weighted logistic regression. In *ICML*, volume 3, pages 448–455, 2003.

[127] Fabien Letouzey, François Denis, and Rémi Gilleron. Learning from positive and unlabeled examples. In *International Conference on Algorithmic Learning Theory*, pages 71–85. Springer, 2000.

[128] Sasha Levin. Building stable trees with machine learning, June 2018.

[129] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[130] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):92, 2017.

[131] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[132] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *ACM Sigplan Notices*, volume 38, pages 141–154. ACM, 2003.

[133] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *Acm Sigplan Notices*, volume 40, pages 15–26. ACM, 2005.

[134] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 709–712. IEEE, 2015.

[135] Bing Liu, Yang Dai, Xiaoli Li, Wee Sun Lee, and S Yu Philip. Building text classifiers using positive and unlabeled examples. In *ICDM*, volume 3, pages 179–188. Citeseer, 2003.

[136] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 286–295. ACM, 2005.

[137] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007.

[138] Jingzhou Liu, Wei-Cheng Chang, Yuexin Wu, and Yiming Yang. Deep learning for extreme multi-label text classification. In *Proceedings of the*

40th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 115–124. ACM, 2017.

[139] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384. ACM, 2018.

[140] David Lo, Lingxiao Jiang, Aditya Budi, et al. Comprehensive evaluation of association measures for fault localization. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[141] David Lo, Xin Xia, et al. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 127–138. ACM, 2014.

[142] Edward Loper and Steven Bird. Nltk: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[143] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[144] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[145] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[146] David MacKenzie, Paul Eggert, and Richard Stallman. Comparing and merging files with gnu diff and patch. *Network Theory Ltd*, 4, 2002.

[147] Subhransu Maji, Lubomir Bourdev, and Jitendra Malik. Action recognition from a distributed representation of pose and appearance. In *CVPR 2011*, pages 3177–3184. IEEE, 2011.

[148] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[149] Andrian Marcus and Jonathan I Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th international conference on software engineering*, pages 125–135. IEEE Computer Society, 2003.

[150] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 18. ACM, 2010.

[151] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2017.

[152] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[153] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[154] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compo-

sitionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[155] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130, 2000.

[156] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[157] Mozilla. Bug fields. `https://bugzilla.mozilla.org/page.cgi?id=fields.html`. Accessed: 2015-03-16.

[158] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[159] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[160] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[161] Annamalai Narayanan, Charlie Soh, Lihui Chen, Yang Liu, and Lipo Wang. apk2vec: Semi-supervised multi-view representation learning for profiling android applications. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 357–366. IEEE, 2018.

[162] Hwee Tou Ng and John Zelle. Corpus-based approaches to semantic interpretation in nlp. *AI magazine*, 18(4):45–45, 1997.

[163] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 858–868. IEEE, 2015.

[164] Giang Hoang Nguyen, Abdesselam Bouzerdoum, and Son Lam Phung. Learning pattern classification tasks with imbalanced data sets. In *Pattern recognition*. IntechOpen, 2009.

[165] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping api elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758. IEEE, 2016.

[166] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017.

[167] Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction*, pages 109–125, 2009.

[168] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[169] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[170] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[171] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and

improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[172] Francisco Pereira, Tom Mitchell, and Matthew Botvinick. Machine learning classifiers and fmri: a tutorial overview. *Neuroimage*, 45(1):S199–S209, 2009.

[173] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, pages 654–664. IEEE Press, 2017.

[174] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[175] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[176] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[177] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[178] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 147–157. ACM, 2013.

[179] Mohammad Masudur Rahman and Chanchal K Roy. Textrank based search term identification for software change tasks. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 540–544. IEEE, 2015.

[180] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 222–231. IEEE, 2016.

[181] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

[182] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[183] E Raymond. The cathedral and the bazaar: Musings on linuxand open source by an accidental revolutionary, 2002.

[184] James Renegar. *A mathematical view of interior-point methods in convex optimization*, volume 3. Siam, 2001.

[185] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[186] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[187] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[188] Aliaksei Severyn and Alessandro Moschitti. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 373–382. ACM, 2015.

[189] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[190] Amit Singh. *Mac OS X internals: a systems approach*. Addison-Wesley Professional, 2006.

[191] Bunyamin Sisman and Avinash C Kak. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 50–59. IEEE, 2012.

[192] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[193] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[194] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT*

International Symposium on Software Testing and Analysis, pages 273–283. ACM, 2017.

[195] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[196] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[197] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

[198] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 2018.

[199] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823. IEEE, 2015.

[200] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011):429–489, 2002.

[201] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec learning embeddings for software libraries. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 18–28. IEEE Press, 2019.

[202] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 168–179. IEEE Press, 2015.

[203] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012.

[204] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*, pages 386–396. IEEE Press, 2012.

[205] Yuan Tian, Chengnian Sun, and David Lo. Improved duplicate bug report identification. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 385–390. IEEE, 2012.

[206] Giorgos Tolias, Ronan Sicre, and Hervé Jégou. Particular object retrieval with integral max-pooling of cnn activations. *arXiv preprint arXiv:1511.05879*, 2015.

[207] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

[208] S Vijayarani, Ms J Ilamathi, and Ms Nithya. Preprocessing techniques for text mining-an overview. *International Journal of Computer Science & Communication Networks*, 5(1):7–16, 2015.

[209] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[210] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.

[211] Shaowei Wang, David Lo, and Julia Lawall. Compositional vector space models for improved bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 171–180. IEEE, 2014.

[212] Shaowei Wang, David Lo, Bogdan Vasilescu, and Alexander Serebrenik. Entagrec++: An enhanced tag recommendation system for software information sites. *Empirical Software Engineering*, 23(2):800–832, 2018.

[213] Shuohang Wang and Jing Jiang. A compare-aggregate model for matching text sequences. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[214] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[215] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[216] Yang Wen, Jicheng Cao, and Shengyu Cheng. Ptracer a linux kernel patch trace bot. *arXiv preprint arXiv:1903.03610*, 2019.

[217] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[218] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE Press, 2015.

[219] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[220] Peter Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[221] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[222] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[223] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Tag recommendation in software information sites. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 287–296. IEEE, 2013.

[224] Xin Xia, Xiaozhen Zhou, David Lo, Xiaoqiong Zhao, and Ye Wang. An empirical study of bugs in software build system. *IEICE TRANSACTIONS on Information and Systems*, 97(7):1769–1780, 2014.

[225] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[226] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in sbse for

spectrum based fault localisation. In *International Symposium on Search Based Software Engineering*, pages 224–238. Springer, 2013.

[227] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200. IEEE, 2014.

[228] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[229] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[230] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699. ACM, 2014.

[231] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE, 2018.

[232] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *International Symposium on Search Based Software Engineering*, pages 244–258. Springer, 2012.

[233] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.

[234] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[235] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[236] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[237] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for image restoration with neural networks. *IEEE Transactions on Computational Imaging*, 3(1):47–57, 2016.

[238] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.

[239] Zhi-Hua Zhou and Xu-Ying Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge & Data Engineering*, (1):63–77, 2006.