博士論文

Representation Learning for Program Analysis, Testing and Repair (プログラムの解析、テスト、修復のための表現学習)

[指導教員　松尾 豊 准教授]

東京大学大学院 工学系研究科
技術経営戦略学専攻

ロヨラ　ハイフマン　パブロ　サルヴァドール

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software represents one of the most important aspects of our lives [26]. It is pervasive across all the processes and activities we perform on a daily basis and it is responsible of shaping our lifestyles and interaction with the world [8]. Moreover, software is at the core of all socio-economic processes, as an enabling framework that empower human behavior at planetary scale [81].

Software complexity has evolved exponentially over the years [12, 174]. From the early days where systems were only able to handle basic arithmetic operations to our present days, where the level of automation and autonomy has led to huge advances in space exploration, transportation, among others.

Software development is a human activity, therefore it is natural to be prone to errors [1, 41]. Moreover, they way in which software is crafted has evolved over the years, from a *lone wolf* paradigm, where a single developer worked in isolation and was entirely responsible for all the process, to a new paradigm that privileged team work and the sharing of diverse range of tasks among groups of developers from a diverse socio technical background [37,65,99]. Additionally, the emergence of web based systems and versioning systems such as Git has allowed a geographically distributed setting, where people from around the world can collaborate simultaneously to accomplish a task [52, 93]. While this new paradigm provides several advantages, mainly associated to the concept of diversity, where developers from difference cultural backgrounds and skills can collaborate to find creative solutions to challenging problems, it is also known that it adds a new layer of complexity to the development process [22].

Given this natural error proneness, continuous efforts have been directed towards the improving quality assessment tasks, until present day where software has been accepted as part of our lives because it had achieved over the years a feasible degree of reliability, ie, we have the notion that software can reach a level of maturity on which we can rely and delegate most of our activities [80,149]. Both industry and academia have spent large amount of resources over the years to design and implement methods to detect and repair failures, to improve engineering processes and to assist program understanding [18, 76].

While these efforts have generated important progress over the years, the emergence of failures at every level is still a threat hard to weaken. It is estimated that software failures cost US economy 60 billion annually. Moreover, The cost of a bug found after release is 30x higher than if it is found during test time. Additionally, indirect costs are also a matter of concern, for example the damage on the reputation of a company that has reported a security or quality incident has a considerable impact on its value, as has been reported that on average, a company loses 3.75% of shareholder value just on the day of the announcement [36, 77].

Examples of this can be seen in the spacecraft and avionics industry, where projects such as the Mariner 1, by NASA suffered from a faulty operation in the guidance system, which led to order a destructive abort 300 seconds after launch [46, 90]. This episode was so iconic that the famous author Arthur C. Clarke described the failure as "the most expensive hyphen in history." The Ariane 5 mission, from the European Space Agency (ESA), is another example. In this case, a faulty conversion from a 64-bit floating point number to a 16-bit signed integer value to overflow and cause a hardware exception. Pre-flight tests had never been performed on the inertial platform under simulated Ariane 5 flight conditions so the error was not discovered before launch [38, 74, 85]. More recently, the Hitomi mission from JAXA failed to reach the desired orbit given a defect in the inertial reference unit [13, 164].

Examples in other fields are reported cases of Knight Capital, which trading algorithm defectively cost over 440 million in just 45 minutes [143]. More critical was the case of the Therac-25 radiation therapy machine, where a high-powered electron beam struck the patients with approximately 100 times the intended dose of radiation, delivering a potentially lethal dose of beta radiation [28, 88].

Besides the economic and inherently deadly consequences that defective software could produce, it has also a dimension associated with the innovation cycles in software engineering [49, 130]. These days companies compete to transform novel ideas into products, in a natural race to capture audience and position as market leaders [44]. This is specially relevant in highly disruptive fields where usually companies have such an enormous pressure to deliver a product that quality assessment tasks, such as testing, are not considered as a priority. For example, in 2011 Sony suffered a huge data breach on its gaming network, PSNetwork, which compromised 77 million user accounts [142]. Post mortem analysis suggested that the system was not tested comprehensively before release. While asked about this case, security expert Allan Paller said: "They have to innovate rapidly. That's the business model. New software has errors in it. So they expose code with errors to large numbers of people, which is a catastrophe in the making." [11] In that sense, the rush for getting innovative products into market takes security and quality aspects to a second place, which may results in huge losses, from a business perspective.

Therefore, we consider that software quality needs to be also understood as an enabling factor for innovation. Standard testing processes are extremely expensive, both in terms of resources and time, and in most cases are seen as a burden. Existing literature reports that executing a test suite in an industrial scenario could easily take two to three weeks [14, 47]. A more general statistic says that testing account for more than 50 % of the development time [79]. Moreover, even comprehensive testing criteria, such as method or branch coverage, can not guarantee full defect detection [112].

If software quality assessments tasks could be designed to be applied in a more natural way into the development process, innovators could have the peace of mind to prioritize their efforts towards improving other aspects of the product.

Given the above, we consider it is necessary to improve software quality assessment tasks in order to come up the increasing complexity of software systems. This advance needs to take into consideration the trade off between exhaustiveness and flexibility: while it is primordial to generate methods that detect and repair failures in a comprehensive way, at the same time it is necessary that those methods do not interfere with the creative processes of software crafting.

While software complexity and size are increasing, the same has happened to the amount of data we are collecting about it [30, 139]. During the years, several logging and storing infrastructures has been implemented in order to track every single detail from the software development process. In the first place, the adoption of subversion systems such as Git [1], has allowed the recording and tracking of all the changes in a codebase, allowing developers to manage the engineering aspects in a secure way by means of branching, reversing actions and coordinating code contributions in an asynchronous way.

Moreover, the emerging role of the Web in terms of supporting collaborative work, has had an enormous impact on the visibility and verification of software , specially in the case of Open Source [150, 157]. Platforms such as Sourceforge[2] and Github[3] cannot be seen as just a rigid code repositories, but as sociotechnical ecosystems where both end-users and contributors interact in a distributed and transparent way, through the use of social enabling artifacts such as forums, bug trackers and a series of reputation based constructs, originally from social networks, such as the ability to *follow* or *like* like a project or even a developer [35]. With this, the data that can be obtained from a project is not only related to the functional properties of the system or the contribution events, but also but also the complete interaction history between developers and the code base [144].

This increasing amount of data has allowed the exploration of developer behavior in a more fine grained way. For example, in the past, researchers tried to identify the motivations of developers to join Open source project, despite the fact that they cannot received any monetary incentive for their contributions. At that time large amounts of data were not available, therefore the conclusions they reached were based on small surveys along with the use of economic models [70, 86]. In contrast, in recent years researchers have been able to undercover more specific explanatory factors related to the participation pattern by mining publicly available datasets. The depth of the analysis has also been greatly benefited. While in the past most of the studies that focused on human factors to explain software quality aspects were only able to capture high level features, such as graph metrics from co-editing developer networks [71,104], recent work has pushed the limits as to try to decode the internal cognitive state the developers, from extracting mood and emotion state of developers by mining their messages on bug trackers [59, 121], to directly using neurophysiological devices such as eye tracking and electroencephalography (EEG) to decode brain activity while writing

[1]`git-scm.com`
[2]`sourceforge.net`
[3]`github.com`

or reading code [111, 114, 145].

1.1 Motivation

Empirical software engineering is the discipline that has embraced this big data paradigm in recent years. Most of the work on this area has been focused on finding causal relationships between diverse aspects involved in the development process, commonly relying on tools from data mining and machine learning domains [51, 147]. Despite the relevant insights that these approaches have generated, little attention has given to the generation of more general models or theoretical frameworks to understand the software engineering process from a more comprehensive way [33, 75]. Nevertheless, after a exhaustive literature review from this field, we conclude that the main lines that influence the outcome of software development from a quality perspective are i) humans aspects, mainly through social network analysis, primarily used for extracting features to feed defect prediction models ii) source code understanding, as a proxy to estimate program complexity and debugging, and, iii) program behavior, in the form the study of execution traces and coverage information to define testing configurations. We consider that moving in those three directions is the correct way, in the sense that we could eventually find insights on how to improve software quality in a broader way. However, as we deepen our understanding on the state of the art, we found out a sharing element across all of them: the data representations were inherently crafted manually. While these approaches in general have been a contribution to the software development process, our concern is related to the fact that the performance of any predictive model depends heavily on the representation of the data used, and that different data representations can entangle and hide different explanatory factors. Furthermore, the decisions related to how to compute these hand crafted features may incorporate bias and, in some cases, could not be weighted effectively given their complexity [15, 45, 58] .

We consider this as a relevant problem in software engineering, given the heterogeneity of the data sources involved, their modalities and types. In that sense, we hypothesize that if we were able to design methods less dependent on feature engineering and hand crafted features, we could improve the effectiveness of quality assessment tasks. Moreover, if we are able to abstract software data to more tractable and flexible representations, we could eventually find more natural ways to combine several aspects of development, which at the end could provide a more holistic perspective on the analysis.

To this end, we rely on the representation learning paradigm, which encompasses a set methods whose objective is to automatically learn feature representations from the data. These representations comply with certain characteristics such as *smothness expresiveness* and temporal and spatial *coherence*. Moreover, they privilege the emergence of multiple explanatory factors with a *hierarchical configuration* and the ability to be *distributed*. This perspective fits our goal in the sense of providing both theoretical and technical frameworks to structure and construct our study. Therefore, in this thesis we explore methods based on representation learning and design empirical studies intended to answer the main research question :

To what extent building methods based on representation learning can help

to assess software quality ?

With this, we envision methods and tools that support developers and that are able to seamlessly aggregate diverse sources of data to provide rich explanatory factors derived from automatically learned feature representations. In that sense, in this thesis we embark in a quest to revisit several and important methods used in software engineering in the light of a representation learning paradigm. For each of them, we explore ways to extend or create representation learning based alternatives to compare against the state of the art. Therefore, we could be able to conclude on the pros and cons that the proposed methods could provide. At the core of our research is the concept of program change, as we believe it encodes the temporal and functional characteristics most related to quality.

1.2 Structure and Contributions

As we are attacking a broad problem, we decided to follow a divide-and-conquer approach. In that sense, we structure the study in several research modules, each of them focused on an specific aspect of software quality and encapsulated into a chapter. Additionally, we designed the flow of the analysis in a way it naturally follows the characteristics of the software development process in terms of temporal and complexity aspects. Moreover, it allows us seamlessly connect the ideas, allowing the reader to better comprehend the contributions in a more clear way.

We begin this work with the topic of human factors and defect prediction on Chapter 2, which means the analysis of any social aspect of the development process to estimate the likelihood of a fault. We chose this topic to be the first as it has a proactive nature: the quality issues are not explicitly present in system, but they are likely to exist and their probability resides on the people that modify the code. To this end, we propose a method that learns unified feature representation from both human and program dimensions. Therefore, we propose the concept of *structural semantic genealogies* as a construct to automatically represent the dependencies between code changes, and from them, predict the emergence of defects. Additionally, we present a novel visualization method to explore software evolution from a contributional perspective. Our approach is able to represent users and source code modules over time on a single coordinate system, which allows us to study the contribution trajectories over time.

Subsequently, we put now in the position that faults are already part of our system, and therefore we need effectively to detect them. To study such a task, we devote Chapter 3 to the topic of automated testing. In this case, we propose a method whose goal is to improve the effectiveness of the detection of faults, while explicitly taking into consideration the scalability and time overheads. Our approach deals with the concept of test case prioritization, which is the task of finding a optimal reordering of test cases in a test suite to maximize the fault detection rate. Our model learn continuous vector representations for both code changes and test cases, transforming the problem in a information retrieval task, without the need to actually execute the test cases. We comprehensively evaluated our approach on a set pf real world software artifacts comparing against state of the art methods.

In Chapter 4, our main goal is to find ways to represent programs in a way we can support developers comprehension. To this end, we designed and implemented a neural language model aimed to map relationships between source code and natural language. Our approach, CommitGen, uses an encoder decoder architecture to automatically generate natural language descriptions from source code changes. In that sense, the model tries to *explain* a source code level change by generating a small description. In that sense, we envision the use of representation learning could ease program understanding, summarizing code change activity, usually represented as code commits, into a natural language description, which could be used for both automatic documentation or debugging tasks.

Chapter 5 serves as a unification point of the work presented on the previous chapters. The core unifying element resides in the concept of automatic repair, in the sense of challenging a program to find a fix for its own defects. To this end, we propose a method based on reinforcement learning that learns to represent a state (program version) and from it select the action (a source code modification) that maximizes the expected utility, based on a defined reward signal associated to the correctness of the code (both syntactic and functional). At converge time, the program is able to learn fixing policies. With this, we envision a line of research where programs could learn from its creators and anticipate fixing tasks by both proactive detection and fixing candidate recommendation.

Chapter 6 is devoted to the discussion in terms of contrasting our main hypothesis with the actual results we obtained across the previous chapters. In that sense, we put emphasis on the advantages and disadvantages of our proposed approaches, the scale of our studies and the threats to validity. Moreover we define a set of guidelines intended to support the design of representation learning based models for software engineering.

Finally, in Chapter 7 we draw our main conclusion, after carefully reviewing the whole study. Additionally, we state the main lines of extension and future work.

Chapter 2

Human Factors

Preface

Considering the human dimension of software engineering has been one of the most important advances in software engineering research in recent years, as it has enabled a broader understanding of the development process. At the same time, this addition represents new challenges, as the data modalities increase in number and complexity. That motivated us to explore ways to support the extraction of expressive representations based on human interaction that could allows to study the emerge of defects.

2.1 Learning Socio-Technical Representations for Improving Defect Prediction

2.1.1 Introduction

Software development can be seen as a continuous stream of heterogeneous activities [113], where a community of contributors interact by committing changes to the code base, improving the overall quality or adding new functionalities. These changes differ in their nature, but commonly consist of bug fixes, new feature additions and tests suites.

In recent years, the Web and subversion systems have converged into collaborative platforms such as Github and SourceForge. These systems have boosted the participation levels in Open Source projects to the point that collaboratively generated programs are sometimes the first choice in several domains. Likewise, even private companies are opening their code bases in search for feedback, following a win-win schema [4].

The key element that these systems provide is a social layer, which acts as a catalyst for discussion and collaboration. Each proposed change is discussed transparently and the decisions can be seamlessly propagated to the community. On top of that, reputation artifacts, commonly seen in social networks, such as *followers* and *favorites*, are used as incentives for participation.

This social component can be seen as an advantage, in the sense of boosting innovation and visibility, but at the same time it increases the complexity of the development process, which could eventually make the code more prone to errors. For instance, sudden bursts in contributions which the core team, usually a very reduced number of participants with code committing privileges, cannot review properly could cause a decrease in the overall quality [113].

One way to proactively handle these threats is to rely on defect prediction models to understand the dynamics of contributions and estimate which sections of the system under development are more prone to allocate bugs. This information could represent a huge help for assigning and prioritizing human resources in maintenance and code reviewing tasks.

Defect prediction is an active field in the research community, and given the high heterogeneity of the data available, both at source code level and in the version history, a large number of approaches from different nature have been proposed during the past years [19,115,171]. Commonly, defect prediction methods consist of three defined steps. In the first place, a data source from a chosen dimension of the software development process is selected, along with its granularity. For example, literature shows defect prediction studies using data from sources such as source code, module dependencies or developer interaction history, among others. This data source will be used as the independent, explanatory variable during the prediction task. The second step consists of the construction of features from the data, a process that is usually carried out manually and that is inherently ad-hoc to the nature of the data under study. For example, if the study is focusing on source code, then code complexity metrics are usually computed. If it is focusing on developer interactions, network metrics, such as node centrality or degree, are obtained. Finally, in the last step, the set of features and the corresponding labels associated to real defects serve as training set for a classifier, and a standard prediction task is carried out.

While this setting have generated remarkable progress over the years, our concern is directly related to the second stage presented above, the feature generation, as little attention has been given to the fact that the performance of any predictive model depends heavily on the representation of the data, and different representations can entangle and hide different explanatory factors of variation behind the data [15]. Furthermore, the decisions related to how to compute hand crafted features may incorporate bias and in some cases could not be weighted effectively given their complexity. Therefore, in order to expand the scope and ease of applicability of defect prediction models, it would be convenient to design methods less dependent on feature engineering and hand crafted features.

Specifically, we are interested in analyzing the mentioned issue on defect prediction models that incorporate human activity as explanatory variable, because historically they have suffered from the sparsity issues derived from computing graph related metrics. Additionally, it allows us to visualize software development as sequence of dependent changes on the code, which eventually opens the door to understand the causal factors that could drive developers to introduce faulty code.

Given the above, we propose to study the impact of the representation of the data in defect prediction models that incorporate human factors. Then, instead of manually constructing features, we propose two methods inspired in recent advances in Representation Learning [15] which are able to automatically learn representations from the data. These new representations are subsequently compared against manually crafted features for defect prediction in real world software projects.

The first step consists of processing development activity logs and identifying the dependencies between code changes. To this end, we rely on the concept of *code change genealogies*, proposed by Brudaru and Zeller in [27], which allows us to transform code commit history into a dependency graph. Under this configuration, code changes are represented by nodes, and edges between them are dependency relationships computed based on the amount common code they are modifying. The resulting structure is a directed acyclic graph that allows us to identify the impact (both direct and indirect) of a change over the subsequent changes. See Section 2.1.3 for a detailed review on code change genealogies.

After the data has been processed and structured through code change genealogies, the second step consists of extracting features from each node, which will characterize each code change. Existing studies manually compute families of graph-based metrics, such as node centrality or degree [66, 173]. We explore a completely different approach, and instead of choosing and computing features manually, we propose two methods to automatically learn feature representations. These methods are inspired by the family of neural word embeddings, which have had a significant impact in Natural Language Processing in recent years [16, 106], and also by novel methods that discover latent representations of nodes in graphs, such as the work by Perozzi et al. [125].

The first approach, *Structural Genealogy Embeddings* (SGE) tries to learn representations directly from the structural properties of the change genealogies. This method works by firstly generating truncated random walks from each node. Then, the resulting paths, which are actually node sequences, are used to learn representations by maximizing the likelihood of a node given its context.

The second approach, *Structural and Semantic Genealogy Embeddings* (SSGE) not only considers structural properties of the change genealogy, but also the characteristics of the source code that was modified on each change. With this dual input configuration, we can explicitly relate the behavioral and semantic components behind a code change.

Subsequently, we direct our study towards answering the following research question: **To what extend the representation of the data impact on the performance of a defect prediction model?**

We performed an empirical evaluation comparing our proposed methods to several families of standard handcrafted network features. Our baseline is the work conducted by Herzig et al in [66], on which a defect prediction model based on handcrafted features extracted from code change genealogies is proposed. This setting allows us to compare directly handcrafted and automatically learned features. For our experiments, we collected version history data from a large number of projects from Github and SourceForge using the BOA repository mining framework [39] and also utilized the publicly available dataset provided by [167].

Our results show that, across several classifiers, learned representations outperforms the set of graph metrics, reaching up to a 13% of increment in prediction accuracy. Moreover, we empirically found two positive characteristics of the learned representations. In the first place, as each code change can be represented as a vector in a continuous space, we were able to avoid the sparsity issues that commonly emerge when working with graph data. This advantage was evident when we performed a deeper analysis that involved cross-project prediction. In this case, the learned representations allowed increments up to 18% in some cases.

Additionally, the flexibility of these new learned representations allowed to explicitly relate change dynamics with the fault probability, characterizing the sequences of changes in the form of trajectories, that lead to faulty behavior.

In summary, the contributions of the present study can be summarized as:

- We proposed to study the impact of the representation of the data on the performance of defect prediction models and performed a empirical study on real world software systems to provide strong evidence that it actually it is a factor that should be considered when designing defect prediction models.
- We propose two methods that are able to automatically learn meaningful representations from developer activity. This methods are highly adaptable and extensible.

These results open several opportunities for improvement by exploring other representations and by studying how to unify the diverse nature of software development data into meaningful and flexible representations to cope up with the increasing complexity of software development. We provide full access to the code and data used in this study for replication and extension purposes[1].

2.1.2 Related Work

Defect Prediction

Defect prediction is an active field in Software Engineering research. Given the richness of the data that can be extracted from the software development process, a large number of methods have been proposed. There are clearly two main categories in terms of the nature of the data used.

In the first place, we can identify source code based features, which usually take into account the complexity of the system under development. Examples of this are Halstead [61] and McCabe [102] features, which are used to map defects in a prediction setting [105]. Graph representations of software components have also been used for defect prediction, such as the work of Zimmermann et al in [173].

[1]Available after publication

In the second place, and given the relevance that human factors have gained into the software development analysis, several approaches have focused on how interactions and organizational dynamics could contribute to predict defects in the code. Examples of this paradigm are the seminal work by Weyuker et al [159] which uses developer features and the approaches by Pinzer et al [128] and Meneely et al [104], both using a graph representation to form relationships between developers and the modules they build, which are used to support defect prediction tasks. More recently, Bird et al [20] used the concept of socio-technical networks to predict bugs in both Windows Vista and Eclipse. The same author proposed in [21] the use of *code ownership*, defined as the amount of expertise a developer has over a module, as a explanatory variable for defect prediction. That work has been followed by replication studies such as [53] and [42].

Another relevant line of research is related to *cross-project* defect prediction to alleviate the problem of the lack of training data, specially for new projects. While historically the performance has not been solid, given the complexity of the problem, several approaches has put attention on the representation of the data. For example, Nam et al proposed the use of transfer learning in [116].

The problem of the representation of the data has been present in the literature, with recent focus on the aggregation effects [170] and model validation [153]. Additional studies can be found on the issue of the noise present on the data and how it affects model performance [78]

Moreover, the use of representation learning for defect prediction has started to get traction on the software engineering community, such as the work of Wang et al [158] use *Deep Belief Networks* [69] to automatically learn semantic features from abstract syntax trees (ASTs) extracted from source code. The obtained features are used in a defect prediction setting, with emphasis in cross-project defect prediction, outperforming state of art approaches. Yang et al [166] uses a similar approach but based on the problem of just tin time defect prediction. While we took inspiration on those approaches, they work at source code level. In our case, the goal resides on trying to learn representations by unifying two or more aspects of the development process.

Graph Embeddings

The success that approaches based on Representation Learning achieved in Computer Vision and Natural Language Processing was rapidly adapted to model graph data. Perozzi et al proposed *DeepWalk* in [125], a method that provides a vector representation for each vertex on a graph by using *Skipgram* model on sequences of nodes generated through random walks. In [126], the same author extends the approach to tackle the problem of representing the membership of an individual in all the communities he participate. The idea of using random walks as sequences for encoding node context has been shared by other methods such as Grover et al [54] and Cao et al [32]. Tackling the specific problem of representing sub structures in graphs, Yanardag et al proposed *Deep Graph Kernels* in [165]. Another element of interest is the scope used in the graph to learn the representations, where approaches such as [152] and [31] have explored different configurations, specifically the definition of the loss function during training. Li et al [89], extended the concept of *Graph Neural Networks* defined by Gori et al [89] to propose a model to assists tasks in graphs that output sequences instead of a individual value, for example, paths within the graph. A recent approach is proposed by Niepert et al [119], where convolutional neural networks are adapted to discover features from connected regions from graphs.

Bourigault et al [24] presented a method for modeling diffusion cascades in social networks using representation learning. In this case, the diffusion is transferred into a continuous latent space and then the transmission probability is computed base on the relative position of the entities in this new representation.

Applications of these techniques have also been studied. One example is the work by Ni et al in [118], where they tackled the problem of measuring the similarity between documents. In this case, authors firstly leveraged the an initial linking structure from a defined knowledge base and from that learn to represent latent embeddings for each document.

Representation Learning in Software Engineering

Representation Learning and more commonly *deep learning* techniques have been introduced into the Software Engineering research community in the recent years, focusing on several aspects of the development process and providing a new set of tools to improve the analysis. From a source code level perspective, neural language models have been adapted to perform a diverse rage of analysis tasks [163], from program classification by learning features from the Abstract Syntax Tree (AST) [108] to source code summarization using attention models [3] and clone detection [162].

More recently, Gu et al [55] proposed a deep learning based approach to generate API usage sequences for a given natural language query. Similarly, Ye et al [168], propose the use of neural word embeddings to support the text retrieval task, trying to find a common ground between natural language and source code, which results increasingly useful in tasks such as bug localization, where it is necessary to locate a fault in source code based on a report generated in natural language.

2.1.3 Background

Code Change Genealogies

Our ultimate goal in this work is to build a model that learns to represent code changes and then use these learned features to predict defects in the source code. In that sense, the use of a graph approach to structure developer activity appears as a natural decision, as it has shown potential to capture structural and temporal dependencies at different levels in the software development process, such as between packages, modules or even code changes [173].

We built our approach on top of the concept of code change genealogies, introduced by Brudaru and Zeller in [27]. A *code change genealogy* is a graph representation of the code changes introduced to a software system during a defined period of time. This idea was shared by other approaches such as [2, 48, 67] in the sense of capturing the underlying factors that induce dependencies between changes in the code.

In the original formulation, the nodes in a code change genealogy represent change sets and the edges represent dependencies between them, computed based on a set of rules that consider the addition or modification of method calls between changes. To better understand this concept, let us illustrate it with an example, shown in Figure 2.1. In this case, let c_0 be the initial implementation of a portion of source code, which consists of the addition of two functions, **foo**() and **bar**(). Subsequently, a new change c_i is performed, which consisted of a modification of the function **bar**() (the addition of a new parameter). As the function **bar**() is shared among both changes, a link is generated between them. Following that, we have the change c_j, where the function **foo**() is called. Consequently, we construct an edge between c_0 and c_j as the latter is explicitly using a function defined in c_0. Finally, we have the case of c_k, a change that is calling the method **bar**(), which last modification resides in the change c_i, therefore, we generate an edge between c_i and c_k.

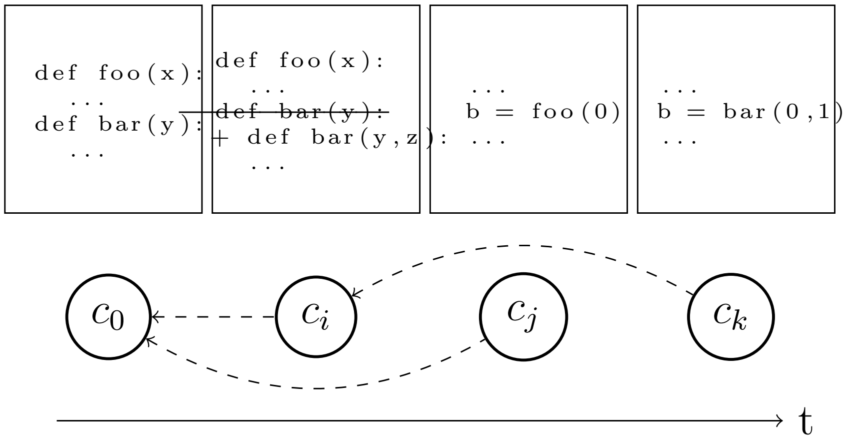


Figure 2.1: Example of a code genealogy. Successive changes that share actions over common methods are related with directed edges.

As we can see, as we traverse in time the project activity, a directed graph structure can be generated. The introduction of edges depends on the criteria used to relate changes, and while several options could be considered, we follow the set of rules proposed by Herzig et al [66], which uses a method level to compare between changes.

Code genealogies, unlike other graph structures adopted in the literature, has the advantage to encode a temporal dimension, which makes it an interesting candidate for defect prediction.

Neural Language Models

The key idea behind these models resides on casting the feature generation as a prediction problem, taking into account the order of the words in a sentence and leverage a model based on the inherent language regularities, under the assumption that closer words in sentence have a higher statistical dependency.

More formally, given a word w_t in a sentence and a defined context of length c, the Continuous Bag of Words (CBOW) model to learn the probability distribution of w_t, as seen in Figure 2.2, using the sub sequence $(w_{t-c}, \ldots, w_{t+c})$, maximizing the corresponding log likelihood:

$$L = \sum_{t=1}^{T} \log \mathbb{P}(w_t | w_{t-c} : w_{t+c}) \tag{2.1}$$

where the probability $\mathbb{P}(w_t|w_{t-c} : w_{t+c})$ is computed using the softmax function :

$$\mathbb{P}(w_t|w_{t-c} : w_{t+c}) = \frac{\exp(\bar{v}^T v'_{wt})}{\sum_{w=1}^{W} \exp(\bar{v}^T v'_w)} \tag{2.2}$$

with W the size of the vocabulary, v_w and v'_{wt} the input and output representation of the word w respectively. $\bar{v}$ is the average of the context, computed as $\bar{v} = 1/2 \sum_{-c \leq j \leq c} v_{w_{t+j}}$.

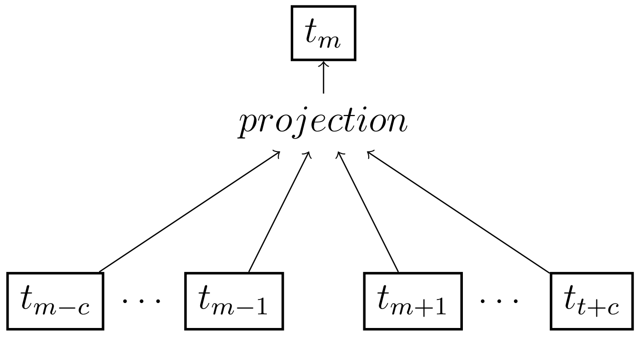


Figure 2.2: Continuous Bag of Words model.

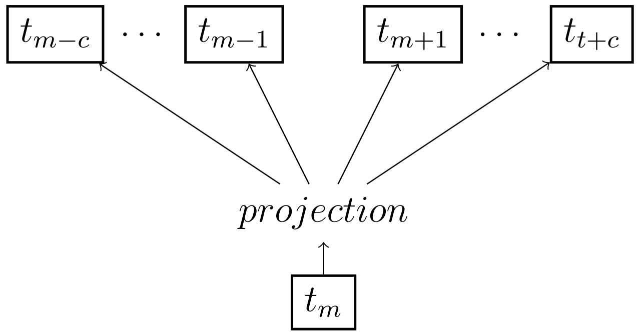


Figure 2.3: Skipgram model.

Recent advances in language models are mainly focused on the Skipgram model, which switch the definition of the problem, as seen in Figure 2.3, and then predict the context based on a specific word, transforming the objective function to:

$$L = \sum_{t=1}^{T} log\mathbb{P}(w_{t-c} : w_{t+c} | w_t) \tag{2.3}$$

The probability of the context given a specific word assumes independence between the context and the word, and is computed by:

$$\mathbb{P}(w_{t-c} : w_{t+c}|w_t) = \prod_{-c \leq j \leq c} \mathbb{P}(w_{t+j}|w_t) \tag{2.4}$$

Finally,

$$\mathbb{P}(w_{t+j}|w_t) = \frac{\exp(v_{w_t}^T v_{w_{t+j}}^{'})}{\sum_{w=1}^{W} \exp(v_{w_t}^T v_w^{'})} \tag{2.5}$$

with v_w and v_w' the input and output vector representations of the word w.

What we have just described is a summary of main mechanisms behind a neural language model. We recommend the interested reader to review the works by Goldberg and Rong [50, 136] that presents a more detailed overview.

2.1.4 Proposed Approaches

Structural Genealogy Embeddings

While structural data from the code change genealogy represents by itself a rich source of data to learn change representations, as we are dealing with source code, it is natural to try to incorporate it into the analysis.

Therefore, for this second approach, we try to explicitly relate the dependency of the human activity and the artifact been modified. To achieve that, we treat each code change as a sequential document, conforming a *change context*. This allows to learn representations for both changes and its content in a shared, low dimensional embedding space.

We assume the availability of project version history, compressing changes on a code base during a defined period of time. From that, the commit activity is structured as a directed graph using the definition of code change genealogy, following the definition presented in the previous section.

We begin from a similar scenario to the one proposed in the previous approach: A change genealogy graph G consisting of N nodes from which a set of random walks are generated from each node. We assume a set of truncated random walks extracted from a code change genealogy G, with each sequence consisting of a variable number of changes.

Additionally, let us assume that each change can be represented as a portion of source code that was modified, therefore, we can express a change c_i also as a sequence of source code tokens $t_i = \{t_{i1}, \ldots, t_{in}\}$. The changed code can be obtained by computing the difference between sequential changes, commonly by use of the *diff* command in Unix platforms.

Our idea consists of embed two embedding models that work together in a synergistic way, allowing to represent simultaneously both the change (in a structural way) and also its content. Figure 2.4 presents a diagram of the proposed architecture.

We start with an upper layer that is focused on the structural characteristics of the change sequence. This layer follows the standard configuration of the Skipgram model: given a sequence of changes coming from a random walk, we select one change, c_i and from it we try to estimate the probability of its context (the nodes surrounding it). In this case, we follow the assumption that closer nodes (code changes) have a stronger statistically dependency. This assumption is based on the nature of the graph, which was constructed setting edges between changes which explicitly share portions of code.

In the second place, we have a bottom layer, which tries to exploit the content within the changes, learning form the regularities exposed in the source code that was modified. This layer uses a CBOW configuration, which learns a representation of a token based on its context.

Finally, we propose a way to combine both layers. Initially, given a change node c_i, the upper layer generates a representation for it. On the other hand, the bottom layer does not explicitly learn a representation for c_i, but only for the tokens of the modified source code associated to it. To connect them, we pass the unique identifier of the c_i as an additional token to the vocabulary used in the bottom layer. Then, at training time, the bottom layer also generates a vector representation for c_i, which is trained jointly with the tokens belonging to the source code it modifies. This joint training can be done by simply concatenating both change and token vectors, following the same line of existing methods such as Paragraph Vector [84].

Given that, we will obtain a vector representations for c_i coming from both the upper and bottom layers, which can be then combined. In this sense, the representation we obtain for c_i is both influenced by its structural context (position in the sequence) and also by the content it contains, in the form of the source code it modifies.

More formally, given a code change genealogy G and a set of S paths extracted as random walks, our model is trained to maximize a log likelihood conformed by three components.

The first component given a sequence s and a node $c_i \in s$ is related to the probability of observing the surrounding changes based on c_i, which can be described as:

$$L_1 = \sum_{s \in S} \sum_{c_i \in s} \log P(c_{i-b} : c_{i+b} | c_i) \tag{2.6}$$

$$= \sum_{s \in S} \sum_{c_i \in s} \sum_{-b \leq j \leq b} \log P(c_{i+j} | c_i) \tag{2.7}$$

with b the size of the context. The expression for $P(c_i + j|c_i)$ follows a softmax form:

$$P(c_{i+j}|c_i) = \frac{\exp(v_{c_i}^{\top} v'_{c_{i+j}})}{\sum_{c=1}^{N} \exp(v_{c_i}^{\top} v'_c)} \tag{2.8}$$

with v_c and v'_c the input and output representation of a change c.

The second component deals with the probability of observing a source code token based on its context. Given the encapsulation present in current program systems, we consider that it is important to consider the change itself, as a source of global context. Therefore, this component of the likelihood can be expressed by:

$$L_2 = \sum_{s \in S} \sum_{c_i \in s} \sum_{t_{im} \in c_i} \log P(t_{im} | t_{m-c} : t_{m+c}, c_i) \tag{2.9}$$

As we can see, we incorporate explicitly c_i as part of the context. The expression for this probability will be:

$$P(t_{im} | t_{m-c} : t_{m+c}, c_i) = \frac{\exp(\bar{v}^\top v'_{t_{im}})}{\sum_{w=1}^{W} \exp(\bar{v}^\top v'_w)} \tag{2.10}$$

with W the total set of source code tokens, $v'_{t_{im}}$ the output representation for the token t_{im} and $\bar{v}$ represents the average of the associated context, which includes the vector representation of the change c_i.

Finally, the third component reflects the probability of observing a change c_i based only the set of source code token within it:

$$L_3 = \sum_{s \in S} \sum_{c_i \in s} \log P(c_i | t_{i1} : t_{iT}) \tag{2.11}$$

Once again, the expression for $P(c_i | t_{i1} : t_{iT})$ follows a softmax form:

$$P(c_i | t_{i1} : t_{iT}) = \frac{\exp(\bar{v}^\top v'_{c_i})}{\sum \exp(\bar{v}^\top v_c)} \tag{2.12}$$

Therefore, the likelihood we need to maximize is simply $L = L_1 + L_2 + L_3$. This optimization is carried out using standard stochastic gradient descent [23] on the complete training set for each project. One aspect that must be noticed is that computing the conditional probabilities from the expressions above could be really expensive if the number of nodes or tokens is considerable. One option available is to use *Hierarchical softmax* [107], which consists of assigning each node to the leaf of a binary tree, transforming the problem into the maximization of a specific path in the tree.

So far, we have presented two methods to represent code changes in a continuous feature space. These methods take as main input a primitive graph structure based on the definition of code change genealogies. From this graph, these methods are able to learn a continuous representation for each node, which considers its structural context (position and neighborhood) as well as the content of the code change itself. In the next section we explore how these representations perform in a defect prediction task, comparing them to hand crafted features.

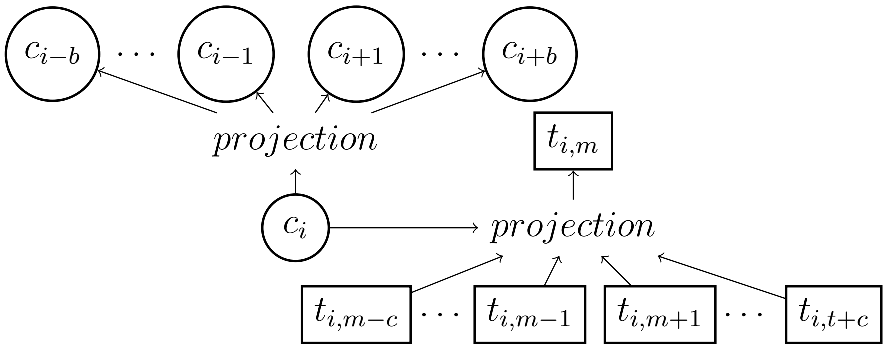


Figure 2.4: Diagram of a proposed approach. The representation for each code change is influenced by its structural properties and also by the semantics of the code it is modifying.

2.1.5 Empirical Study

We designed and implemented an empirical study in which we compare the proposed approach for feature learning to a family of handcrafted features extracted from version history data.

Data

For the first step, we collected historical data from Github[2] and Sourceforge[3], two of the most relevant platforms for collaborative software development. To this end, we rely on the BOA mining infrastructure [39] to gather the development activity data. We collected activity from a total of 470 Java projects stored in Github. In the case of Sourceforge, we collected activity from 173 Java projects. For each project we computed the code change genealogy following the approach presented in citeherzig2013predicting. This dataset only contains structural information and will serve to compare SGE to families of handcrafted graph features. We called this dataset STRUCT.

When exploring the distributional regularities of the activity, we found that Power Law distributions are present in several levels of aggregation over the STRUCT dataset, as shown on Figures 2.5 and 2.6. This finding supports our assumption of the feasibility of adapting language models, as this kind of distribution is one of the main characteristics of a natural language.

We built a second dataset for comparing SGE against SSGE. As the main goal is to obtain not only structure but also the source code associated to the changes in the genealogy, we chose to process the dataset provided by Ye et al [167], which contains activity from six Java projects and has already the bugs mapped: Tomcat, AspectJ, Birt, Eclipse, JDT and SWT. From each project we obtained the code genealogy after processing the commit history. Given the available data, for each change, we label it as bug inducing and also obtained the source code. We called this data set STRUCT-CODE.

Data Labeling: As the problem we want to solve is to predict defects in software

[2]http://github.com

[3]http://sourceforge.net

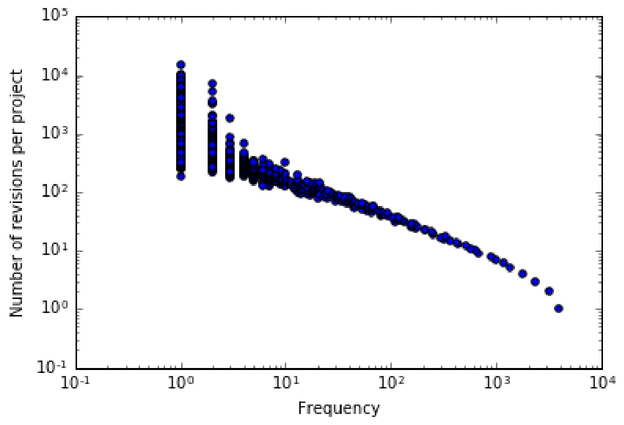


Figure 2.5: Changes per project

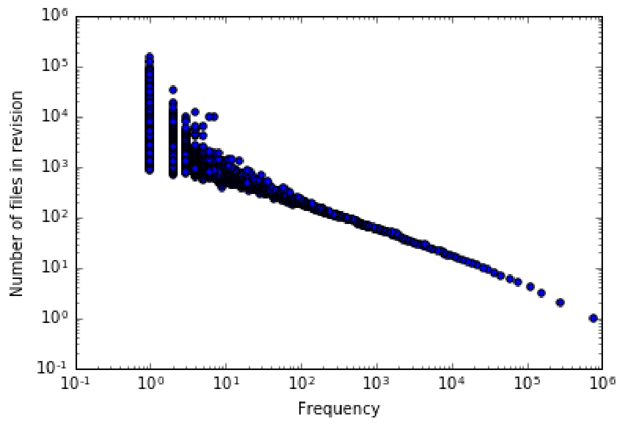


Figure 2.6: Files modified per change

components, we need to explicitly identify the changes that introduced and fixed faulty code, to label them accordingly and obtain our independent variable. To achieve that, we rely on the heuristic proposed by Zimmermann et al. This method start by examining the comments associated to each change looking for references to fixing actions[4]. Then the set of candidates are manually mapped with bug tracker information to make sure of the bug fixing. After that, we can label a change as a *fixing change* and all the files it modifies are assumed as defective.

For the STRUCT-CODE dataset the label are already given and therefore not further action was needed.

Experiment Design Overview

For each project, we collect the activity data and generate a code genealogy graph, from which random walks are obtained and used as inputs for the proposed approach. As output we obtain a representation for each node, which is associated to all the files included in that change.

[4]BOA provides a specific function for this purpose. See http://boa.cs.iastate.edu/docs/dsl-functions.php

At the same time, we computed families of hand crafted metrics shown in Herzig el al [66][5], namely, global network metrics, structural holes metrics and dependency network metrics, for each node. Most of the metrics were computed using the Networkx Python library [6] and also ad-hoc implementations.

We implemented our approach on top of the Gensim python library [7], which provides a fast implementation of the Skipgram and CBOW models. For both cases, we used a neural network to parametrize the feature learning. Hierarchical softmax was used to compute the output probabilities.

After we obtained the feature representations for with each approach, we feed each of them them to three standard machine learning classifiers: Logistic Regression (LR), Support Vector Machines (SVM) and Neural Networks (NN). In all methods we used regularization to avoid overfitting. For the NN, we use a single hidden layer of 128 units. All the classifiers were implemented on top of the sklearn python library [8].

For each project, we split the data into training and testing as 80-20 ratio. The training was done following a k-fold cross validation. We computed a series of metrics to characterize the performance of the prediction, and we are reporting micro and macro F1 scores.

Threads to Validity

Internal Validity: Our approach takes as backbone the concept of *change genealogies* to formalize the dependencies between code contributions. The rationale behind that decision is that a code genealogy not only provide a dependency measurement between code changes, but also a temporal component, based of the directness of the resulting graph, which we consider critical for inferring a degree of causality. However, it could be the case that other ways to structure the dependencies could lead to different results. For example, bipartite graphs that involves developers and modules have been also used for defect prediction in the past [128]. Nevertheless, we consider these developer-module networks static models, which can only encode atemporal dependencies.

Another element to consider as an internal threat is how the labeling of the data was performed. As stated before, we used a set of regular expressions to look for changes descriptions that contains patterns commonly used when reporting bug fixes. While one can argue that this method presents several flaws, we consider that the set of regular expressions are able to cover the majority of the common terms used. Moreover, while still not exhaustive, we performed a manual sampling of changes to verify if the labeling was correct.

External Validity: Our study is limited to Open Source projects from Github and SourceForge, which represent a particular case of software development. The voluntary and self-organized nature of the Open Source paradigm may differ from the traditional ways used in industry, therefore we cannot directly generalize our results to a setting where other

[5]See Section 3 of Herzig's paper [66] for details

[6]`https://networkx.github.io/`

[7]`https://radimrehurek.com/gensim/`

[8]`http://scikit-learn.org/`

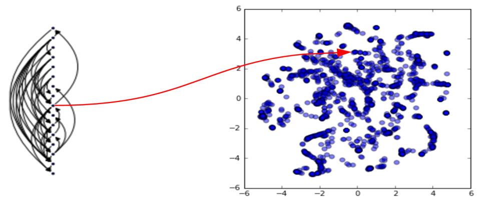


Figure 2.7: Example of the extract of the real change genealogy and the resulting vector representations visualized through t-SNE.

incentives or project management prevail. Nevertheless, we consider that the scale of the data utilized in this work, at least allows us to find robust patterns in terms of the dependencies and practices involved on code changes.

2.1.6 Results and Discussion

In this section we report the comparison of the performances of the different type of features across different classifiers. Given the space constraints, we only show aggregated results. Nevertheless, we found no considerable variance on the results (less than 5%), therefore we could that results are consistent across datasets.

To have an idea of the output of the feature learning process, in Figure 2.7 we show on the left the extract of a real code change genealogy (we cannot display the full graph as the amount of edges make the figure not understandable) and on the right a two dimensional representation of the feature vectors learned using t-SNE, which is a technique for dimensionality reduction [98]. We pointed a node to its learning representation for visualization. A deeper analysis, using clustering to perform community detection among nodes, reported that the relative distances at graph level can be preserved in the new feature space which we consider a desirable factor.

For the first experiment, on which we compare SGE against set family of handcrafted graph features proposed by [66], Table 2.1 shows the Micro (upper) and Macro (lower) F1 scores for each family of features across three classifiers, for both Github (GH) and Sourceforge (SF). From the results we can see that our representations are consistently better than hand crafted features across all the classifiers, reaching an increment up to 13% on average in the case of a Neural Network, with respect to the closest family of hand crafted features.

For the second experiment, which compares SGE and SSGE on the STRUCT-CODE dataset, Table 2.2. From these results, we can see that the addition of data related to the source code changed impacts positively on the performance, obtaining better results across projects and classifiers. Why hypothesize that each vector representation is learned from a more global context and that allows to obtain obtain more expressive feature configurations.

Interestingly, for the case SGE, there cases where varying the classifier does not affect the performance (for example in Birt), while in the case of SSGE, therefore are always increments when using Neural Networks over other model. We consider that this is because. Another interesting result can be seen in the case of Eclipse UI where, for the LR and SVM classifier

Table 2.1: Average Micro-F1 and Macro-F1 for the comparison between SGE and hand crafted features

	LR	**SVM**	**NN**	**LR**	**SVM**	**NN**
	GH	GH	GH	SF	SF	SF
Global Network	0.34	0.26	0.41	0.39	0.51	0.35
Features	0.31	0.17	0.38	0.37	0.48	0.32
Structural Holes	0.53	0.24	0.45	0.42	0.35	0.59
Features	0.44	0.22	0.41	0.30	0.28	0.54
Dependency Network	0.73	0.56	0.69	0.67	0.56	0.76
Features	0.71	0.52	0.65	0.61	0.52	0.70
SGE	0.79	0.76	0.87	0.74	0.69	0.77
	0.66	0.59	0.81	0.66	0.63	0.74

SGE perform equal or slightly better than SSGE. When we inspected the data, we found that the source code associated to the changes were minimal, and in most cases related to the code comments included in the code. Therefore, only a more complex classifier such as the Neural Network was able to produce better results.

Table 2.2: F1 scores for the prediction task comparing SGE and SSGE over the STRUCT-CODE dataset

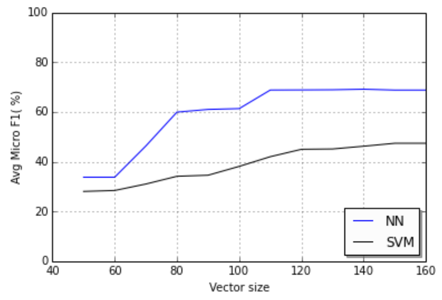
	AspectJ	**Birt**	**Eclipse UI**	**JDT**	**SWT**	**Tomcat**
SGE - LR	0.66	0.59	0.64	0.71	0.61	0.67
SSGE - LR	0.67	0.62	0.65	0.763	0.65	0.698
SGE - SVM	0.677	059	0.64	0.76	0.67	0.73
SSGE - SVM	0.71	0.64	0.63	0.751	0.81	0.78
SGE - NN	0.73	0.7	0.62	0.73	0.77	0.84
SSGE - NN	0.89	0.75	0.68	0.79	0.87	0.88

To test the flexibility and robustness of the representations, we decided to try a more difficult setting, where a classifier is trained on a subset of projects and is used to predict the defect on a completely unseen project. This is a relevant problem in software engineering research, as sometimes version data is available, but bug reports are difficult to obtain or process, therefore it is not possible to perform a labeling required for obtaining a training set. To this end we divided the STRUCT dataset in 70% and 30% proportion and trained the classifiers on the obtained representations. Then, we passed data from the unseen projects and let the models perform a prediction over the changes. In this case, the results obtained show that the learned representations show a higher level of adaptation, reaching an increment up to 18 % and average gaining over 7% over the sets of hand crafted features. Specifically, the best results are again obtained using a neural network, which provides on average a Micro-F1 of 0.66 and a Macro F1 of 0.61. We hypothesize that as the learned features are real valued vectors, they are able to abstract in a more broad way and also they are able to capture the regularities of code changes in a more effective way.

2.1.7 Effect of Dimensionality

One of the parameters we need to define when training embedding models is the length of the resulting vector representation. Therefore we studied how this value impact the performance of the prediction task on the STRUCT dataset. We experimented several combinations and our conclusion is that best results are obtained when the size is in the interval between 100 and 120. We also noted that the performance did not improve as we kept increasing the value. We tried these configurations with a neural network and a support vector machine, as seen in Figure 2.8. Interestingly, a neural network is more sensitive in the sense of the emergence of gaps in the performance, while the support vector machine experience a more linear increment.

Figure 2.8: Performance of the classifiers across different sizes of the vector representation.

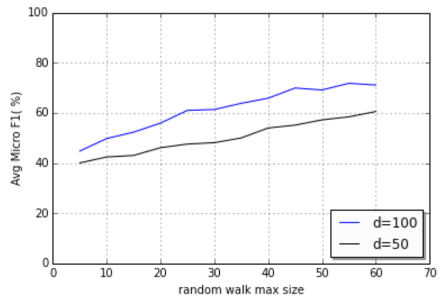


2.1.8 Effect of Sampling Frequency

Another parameter that needs to fixed before the feature learning begins is to define the maximum length of the random walks that will serve as sequences. This is a key element, as although can be parallelized, it indeed impact in the time required for running the approach in a considerable way.

We tested several combinations, and Figure 2.9 shows, the longer the random walks , the better is the performance of the model, even at different vector sizes. We hypothesize that this is because longer walks provide a richer context for the Skipgram model. But while adding more data to the model seems to contribute, it must be noted that it comes with a time overhead.

Figure 2.9: Performance of the classifiers across different random walk configurations.



2.1.9 Conclusions and Future Work

In this work, we proposed a new way to unify socio technical aspects of software development to support the feature generation in defect prediction tasks . Our method tries to learn continuous representations for each node in the code change dependency graph, which can be subsequently used in any standard machine learning method. The novelty of our approach is that during learning it combines information from the structure of the graph and

We evaluate our learned representation against an approach the uses hand crafted network metrics as features. The results shows our approach is competitive, reaching up to 16% of increment in performance, on average. We believe these results suggest that the representation of the data greatly influences the performance of defect prediction, and it is a topic the should be taken into consideration in the software engineering research community, given the diverse nature of the data modalities involved.

We propose to divide the future work into two main areas. Firstly, as network-based defect prediction are just a subset of whole defect prediction methods, we consider necessary to perform a more global comparison with the rest of defect prediction approaches to realize the real impact of the discovered representation and also try to find ways to combine features used in other approaches with the one proposed in this work. In the second place, with the aim of provide a reasonable level of interpretation to the learned feature space, we propose to study their relationship to other types of latent features in graphs, such as *spectral clustering* or *modularity*. We consider this a relevant issue as understandable features can be translated into actions to improve the software development process.

2.2 Visualizing Code Ownership Trajectories in Vector Space

2.2.1 Introduction

The crafting of a software system is usually the result of the joint effort of a group of dedicated contributors. The success of this task depends heavily on the level of coordination achieved among these members, which is not a trivial thing, specially in the case of open source software projects, where the heterogeneity of the participants, in terms of skills and incentives, is usually high. Therefore, the coordination and the distribution of work among the team directly impacts the quality of the resulting artifact. The use of developer centric metrics to support quality is a relevant field within software engineering research [101], and has attracted attention as a way to complement standard code based metrics to perform a more holistic analysis.

From a more systemic perspective, contributing to a software project can be seen as a relationship between a developer and the set of source code artifacts. This relationship evolves over time, usually in a symbiotic way, as the artifacts get benefited by the changes that the developer introduces, and at the same time, the developer can perceive a benefit in terms of her reputation or skills obtained. The key to the success in this relationship is strongly related to the ability of the developer to acquire a full comprehension of the functional characteristics of the source code artifact. Given the limited resources, both technical and cognitive, that the developer can offer, eventually she may tend to select and prioritize the artifacts she will work on. What we see in practice is that usually, even if the developer has to her disposition all the code base to work on, she selects just a subset of modules and files, with which she specialize and actively maintain.

This phenomenon has been studied in the past in the form of *code ownership*, defined as the proportion of the contributions a developer assign to a defined source code artifact. More formally, given a developer d and a software artifact a (selected based on a defined level of granularity), the code ownership that d has over a is defined by

$$O_{da} = C_d(a)/C_a \tag{2.13}$$

where $C_d(a)$ represents the number of changes that d generated on a and C_a the total number of changes on a.

This metric has served as one of the main proxy to characterize the distribution of the workload across development teams and their organizational structure. Moreover, several studies has related this metric with software quality, showing evidence that a source code artifacts which lacks proper code ownership are more prone to manifest faults, at different levels of granularity.

While code ownership has provided relevant insights and it is easy to compute, its nature is inherently rigid. Its value only provides discrete value which can only be interpreted on the specific context of the project under development . This is specially an issue when we want to generate analytic tools based on machine learning, a field on which empirical software engineering is increasingly relying on, where the degree of expressiveness and flexibility of the feature representations impact on the predictive performance of the models. Moreover, it is known that using sparse data can make it difficult to generalize in a statistical learning setting.

In this work we propose to enrich the definition of code ownership and present a method to compute it based in two main requirements: its representation should be a continuous vector, and it should encode a temporal dimension of the relationship between a developer and a code artifact.

To this end, we rely on representation learning, from which we design a method to that receives as input code commit activity and from that, learn to represent both users and source code artifacts as a continuous vector representation. As we also aim at obtaining a estimation of how the relationship between code and, we divide commit activity into defined time intervals, from which we obtain set of vector representations for each time interval. As we want an unified view, we then aligned the vector representations obtained on each time interval into one coordinate system. With this, we are able to visualize the relationship of a developer with the code base in just one feature space.

We performed an empirical study on real world open source projects in order to test our approach. We selected project with both long as also a short history in order to see if the quality of the representations and visualizations we obtain are dependent on the amount of data and the underlying distributional characteristics of the developer contributions. Our results show that it is possible to visualize the dynamics of contribution in a clear way. Moreover, while the main objective was to generate a visualization tool, the system is able to store the vector representations over time, opening the door the generate a more comprehensive metric for relating sociotechnical artifacts.

2.2.2 Related Work

Code ownership has been extensively studied over the years. It has been one the most accessible and easy to understand metrics, and it is usually used as way to estimate the chain of responsibility among users regarding the module development.

There have been several ways to operationalize the idea of characterizing the relationship between developers and modules that we can associate with ownership. For example, Pinzer et al [128] make use of developer-module networks to compute centrality metrics. While not talking explicitly about *ownership*, Weyuker et al in [160] incorporate the number developers working on a given module as impact the quality. Later, Rahman et al [131] relate the proportion of work on fix-inducing fragments of code with the quality of the resulting program version.

One of the most direct applications of code ownership is its use as explanatory variable for defect prediction. In this line we can mention the work by Bird et al, [21], where they extract the proportion of changes the developers performed on the code base of both Windows Vista and 7 and then fed them into a predictive model. The main results of that work stated that a higher the level of ownership is associated with fewer post release bugs.

While the work of Bird et al provided relevant insights about the impact of code ownership on quality, its scope was only restricted to private development ecosystem, where there could be additional constraints and rules that influence that how the teams distribute their workload. This naturally led to question what could the impact of code ownership in a more self organized ecosystem such as open source projects. To this end Foucault et al [43] conducted an empirical study on seven open source projects to test is the conclusions from the Bird et al hold on a more flexible setting. The results show that there were little correlation between weak ownership and and the post release bugs, controlling by granularity (file vs package) and size. While both studies are difficult to compare, it seems that inherent voluntary nature of open source development challenges the core assumptions of work sharing in software development .

Following that line of research, recently, Greiler et al [53] tried to replicate Bird et al study in the light of Foucault et al new evidence. In this new study, conducted on four major Microsoft products, they were able to replicate the results of the original Bird et al study.

Moreover, code ownership has been enriched in its definition by not only incorporating actual code modifications on a given artifact, but also considering other ways of developer contribution, such as code review. That is the case of the approach proposed by the work of Thongtanunam et al [155], where, where the ownership metric is explicitly extended by including the proportion of code reviews associated with a module.

More related with our work, Muller et al [110] proposed a way to quality and visualize code ownership over time. In this case, the authors develop a plugin for Visual Studio 2013 that is able to identify the responsible for each line of code on a system.

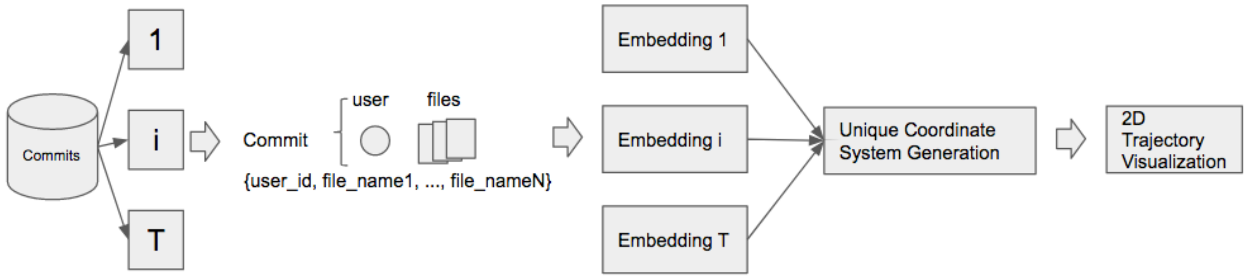
2.2.3 Proposed Approach

We propose a methodology to quantify the relationships between developers and source code artifacts in a way we can visualize them over time. Our goal is to design a method that help us to identify the shifts on developer attention among the set of available modules in a continuous way, allowing us to visually obtain trajectories. To this end, we relay on learning shared representations for both users and modules from the code commit activity. Our approach takes as inspiration recent approaches that adapt neural word embeddings, initially designed to find representation of words, to model social structures.

One of the main building blocks of word embeddings methods resides on the distributional hypothesis, which states that words that appear together along a corpus, should have also a closeness in semantic terms. We consider that such hypothesis also holds when we want to analyze commit activity in software development: If we study a dataset of historic commits, we can see that each instance is composed by the set of files and the were modified and the information about the user associated to that commit. Having a corpus of such instances, naturally makes us wonder if the co occurrence of *user-modules* can be modeled in a word embedding fashion. In that sense, what we hope to train a model in such a way that vector representations that can be learned encode that closeness. For instance, if in a given dataset that user u usually committed changes to a file f, then if we compute a similarity metric between the vector representations learned by the model of u and f, we expect to find a coherent level of closeness. Therefore, we propose to treat each instance of the commit history as a set of token which represents both files and users, and make an analogy with word embedding modeling. In that sense, in our case we will learn distributed representations of the files and users which should follow the distributional regularities of the contribution dynamics seen in the commit history.

Our approach consists of three main parts. In the first one, we generate embeddings for segments of the developer history, for example each month or year of development. Having obtained embeddings from each time period, the second step consists of aligning the embedding dimensions into one unified coordinate system, which will allow us to project the representations learned for each element at different times. Finally, the last step generates a decomposition of the vector representations to a two-dimensional space, which allows us to graph the results and easy visualize the trajectories the users follow. Figure 2.10 shows a diagram of the whole process.

Figure 2.10: Diagram of the proposed approach for visualizing code ownership trajectories.


Commits
1
i
T
Commit
user
files
{user_id, file_name1, ..., file_nameN}
Embedding 1
Embedding i
Embedding T
Unique Coordinate System Generation
2D Trajectory Visualization

Generating Developer-Artifact Embeddings

We assume the availability of the contribution activity from a software project in the form of a list of code commits. Each commit is composed of at least four elements. The id of the user the is submitting the commit, the time associated to the commit, the list of files that were added/modified /removed from the codebase by the commit and the message the user provides as way to explain the his contribution. $c_i = (u, t, F_i, m)$.

As we want to explore the trajectories of the users over time, we start by selecting a time interval from which we will divide the entire history of contribution activity. Let us assume we have T periods of developer activity.

From each period $t \in T$, we obtain the commits and treat each of this instances as a set of tokens, each one composed by the user id , along with the set of files that were modified. Therefore, we will be able to represent each commit as a *sentence*, where the tokens are in the form: (*userid*, $file_1, \ldots, file_n$). Derived from this process, we can also obtain a *vocabulary* V_t, ie, the set of all user ids and file names the appear during t. Each set is used to train a neural embedding model E_t, therefore , our goal is to maximize the probability of the elements appearing in the context of a given a defined element e_i. We make no distinction at training time regarding the nature of e_i , therefore, it could represent the name of a file or the user identification. Following the standard configuration of a word embeddings model, the probability we want to maximize is:

$$P(e_j|e_i) = \frac{exp(e_j \top e_i)}{\sum_{w_k \in V_t} exp(e_k \top e_i)} \tag{2.14}$$

Therefore, in each epoch and for each time period, what we want to do is to minimize the negative log likelihood of the elements surrounding (SU_{e_i}) a given element e_i.

$$J = \sum_{e_i \in t} \sum_{e_j in SU_{e_i}} -logP(e_j|e_i) \tag{2.15}$$

The model parameters are optimized through gradient descent,

$$E_t(e_i) = E_t(e_i) - \alpha \times \frac{\partial J}{\partial E_t(e_i)} \tag{2.16}$$

where the corresponding derivatives are computed using backpropagation and the specific values for α are obtained through a small validation set. We tested several heuristics such as hierarchical softmax and negative sampling to speed up the overall process, taking into account that we need to perform feature learning not only in one dataset but in all the periods on which the activity was divided.

At the end of this process, we obtain a trained model (and their learned representations) for each time period t.

Embedding Alignment

Having obtained a developer-artifact embedding for each period of time in T, the next step consists of align them into a single coordinate system. With this, the representations for each entity across time will be projected in a single vector, allowing us to directly perform similarity analysis, and more importantly, allowing us to compute the position of each entity (developer or module) over time, conforming a trajectory we can visualize.

There are several ways to normalize and merge the set of embeddings, but we must take into account the feature learning was performed in a totally independent way. Therefore we opted by setting one embedding as a point of reference, and from it, transform the vector representations of all the rest. In that sense, what we are looking is to normalize the vectors associated to a given element across all the embeddings for all time periods.

The method we selected is based on the use of a pairwise linear regression between a embedding space that we use a reference and the rest. Conveniently, we selected the embedding space associated to the last time period, E_T. Therefore, what we want in this case is to learn a linear transformation $L_{E_t \to E_T}$ from a embedding space E_t to E_T by solving the following optimization:

$$L(e_i)_{E_t \to E_T} = argmin \sum_{e_j \in S(E_t(e_i))} ||E_t(e_j)L - E_T(e_j)||_2^2 \tag{2.17}$$

where $S(E_t(e_i))$ represents the set of n elements closer to e_i in the embedding space E_t.

After this process, we are able to unify embedding into one coordinate system. It must be noted that we are explicitly making a linear assumption between embedding spaces, which, given certain data, could not hold. Nevertheless, our initial experiments show the feasibility of assuming such characteristic.

2.2.4 Results and Discussion

We implemented our approach on seven six world open source projects from Github, from which we captured the commit activity, shown in Table 2.3. We tested several granularities in terms of how to divide the commit history in order to visualize meaningful trajectories. In that sense, selecting one day as time limit resulted in too abnormal results, leading to erratic and difficult to interpret results. This is mainly because users do not contribute on a daily basis. We found the dividing the commit history on a weekly basis had the best results.

Figure **??** shows an example of the resulting visualizations obtained by our method for the Spring framework open source project. As it can be seen, we are able to visualize, for a given user, depicted by red points, the variation in contributions on the available modules over time. We can see the in the first time period, t_1 the user was focused on a program artifact that was not as close with the rest of artifacts the user focused in the following periods. This led us to think that the user had a considerable shift on his contribution, as for time periods t_2 to t_5 the artifacts he worked on appear to be considerable closed between each other.

Table 2.3: Projects selected for the analysis.

Project Name	**N selected commits**	**N selected files**	**N selected users**
core-nlp	13.000	542	65
elastic-search	26.000	544	104
guava	3.500	291	75
spring-framework	14.000	501	88
youtube-dl	12.000	107	62
tensorflow	12.000	359	201

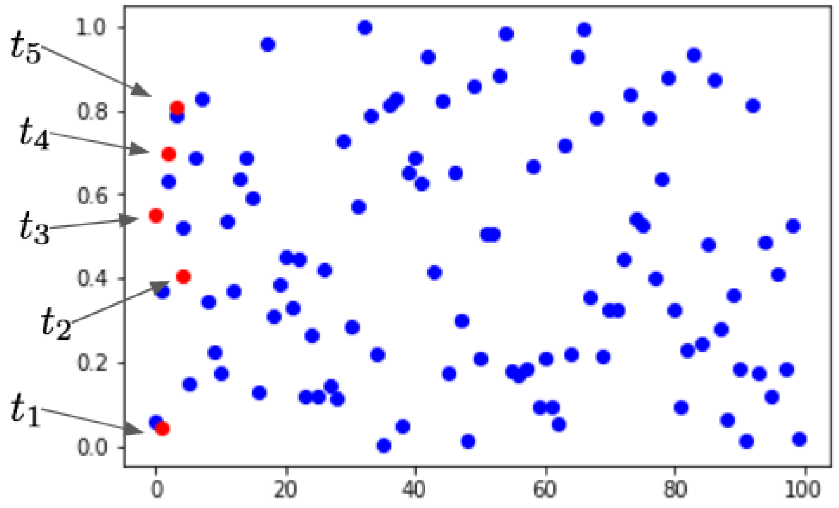


Figure 2.11: Example of a visualization for a anonymized user (red) trajectories over the space of program artifacts (blue) over time. The 2D representation of the vectors was done using t-SNE.

One element to consider deals with the relationship between standard metric, such as code ownership, and any similarity metric we can obtain from comparing the vector representations of users and software modules. We performed a validation comparing two metrics. Given a developer d and a software module a:

- Code Ownership of a with respect to d, computed as the proportion of changes that d performed over a from the total of changes performed over a, in a given period of time t.
- Cosine Similarity between the vector representations learned for a and d by the embedding model E_t.

We performed a correlation analysis for all projects, including all the feasible combination between users and software modules. The results are shown in Table 2.4 For the majority of the cases, there is a strong positive correlation between both metrics. With this, we hypothesize that the embedding models are able to capture the dynamics of contributions, showing a expressive power to model the proportion of change activity in a continuous and vector way.

Table 2.4: Correlation factor between code ownership scores computed with the original version and the proposed approach.

Project Name	R^2
core-nlp	0.89
elastic-search	0.91
guava	0.83
spring-framework	0.86
youtube-dl	0.9
tensorflow	0.83

2.3 Conclusion and Future Work

In this work, we proposed a new methodology to visualize the relationships between developers and the software artifacts they modify over time. We took inspiration on the concept of code ownership, in the sense of accounting for the proportion of changes a developer generates over a defined artifact, but we took a completely different way to compute such a metric. In our case, we make use of word embeddings as a more generic tool to obtain feature representations of entities based on their co-occurrences in a dataset.

This approach allowed us to not only obtain a more flexible metric than the standard code ownership, but also, by aligning the representations over time, we were able to visualize the trajectories that users follow when they edit elements of a code based. We consider this tool can have several applications in software project management, as it allows to have a more intuitive of the contribution dynamics, and at the same time provides a more natural way to acknowledge the chain of responsibility in terms of changes in the code.

For future work we want to extend the computation of our metric. Currently, we train the embedding models on a intra project fashion, which means we only care about the files a user interact that are part of a defined project. While this configuration is feasible, we consider is still simplistic, as there is evidence that in open source software users work in more than one project at a time. To model that we will have to come up with ways to compress commit activity from different projects, forming a small ecosystem of evolving and multi project-user relationships.

Chapter 3

Automated Testing

Preface

Testing is one of the most used ways to assess the quality of a software system. While the task appears to be simple, checking if a set of defined test cases passes its execution on the system, in reality it represents one of the most expensive task during the software engineering process. Therefore, our motivation in this chapter is to explore if representation learning can help to alleviate the its cost, both in terms of time and resources.

3.1 Test Case Prioritization Through Neural Language Modeling

3.1.1 Introduction

The current complexity of software systems has increased the need for automating and improving the quality assessment techniques. Regression testing is one of the most used techniques to evaluate changes in software systems. Commonly, this process consists of, given a test suite which is known to pass on the previous version of the system, it is now executed on the new version. Any fault introduced in this new version is expected to manifest by not passing one or more of the existing test cases in the test suite. Although effective, this method could result extremely expensive. Literature reports that at industry level, a suite of functional tests could require several weeks to execute [40, 148]. The restrictive nature of regression testing has led to the development of several prioritization techniques, where the main idea is to reorder the test cases such they that meet testing goals earlier, i.e., finding new faults executing just a subset of the entire suite [40].

Most prioritization techniques consider code coverage as main input to decide the ordering the test cases, either branch or method level, among others. That information can be obtained by instrumenting the code [148] or by adapting a dynamic analysis tool such as a bytecode analyzer [124]. Having the coverage information, test cases can be ordered following a *total-*

coverage or an *additional-block-coverage* [62].

While current dynamic prioritization techniques have produced satisfactory results in both academia and industry, there is a explicit time overhead as it is usually required to execute the test cases to obtain a measurement of their coverage [64]. While this configuration can work well for small systems, in the practice, re-running test suites is not scalable, even more in a constantly evolving scenario, where both the system under test and the test suites changes rapidly over time [95].

An alternative line of research has tried to exploit static information to reorder the test cases, transforming the prioritization problem into an information retrieval task, where the goal is to find the subset of test cases that better match a defined criteria at source code level. Examples from this area include the use of call-graphs, topic models and the computation of distance metrics between test cases and between test cases and code change [95].

Casting the prioritization as a information retrieval task is an interesting paradigm that could open the door to the incorporation of several methods and theories from natural language processing into a software engineering task. Additionally, this convergence is supported by recent evidence about the *naturalness* of code [68], which means that source code, as a human made form of communication, follows similar syntactic distributional regularities as natural language.

Currently, most of these static approaches compute similarities considering only the frequencies of the source code tokens as features, something that resembles the concept of *bag of words*. Therefore, they are ignoring the syntactic dependencies and ordering on which the tokens are arranged, a factor that encode rich information about program characteristics and functionalities. Additionally, as these features are handcrafted, the decisions related to how to compute them may incorporate bias and in some cases could not be weighted effectively given their complexity. This concern is of high relevance, as it is known that the performance of any information retrieval task depends heavily on the representation of the data used. Likewise, different representations can entangle and hide different explanatory factors of variation behind the data [15]. Therefore, in order to expand the scope and ease of applicability of static test case prioritization, it would be convenient to design methods less dependent on feature engineering and hand crafted features.

Taking the above as our main motivation, in this paper we propose a novel approach for prioritizing test cases using static information but considering the syntactic and semantic characteristics of the programming language in use. To this end, we construct a neural language model that is able to learn distributed and continuous representations of any portion of the source code. Neural language modeling is part of a more broader field usually *representation learning*, which has had enormous success in recent years in disciplines such as computer vision and natural language processing. While our primary goal is to obtain an increment in the fault detection rate with a better reordering of the test cases, we are also interested on studying the impact of the representation of the data used in test case prioritization.

Our approach begins by firstly identifying difference between release versions of the system under testing, which are computed at both file and class level. Similarly to the recent work by Saha et al [140], our goal is to find the most suitable test cases that maximize the fault detection rate based on program differences.

Both the set of program changes C and the test suite T we want to order are pieces of source code that share a context given by the system under test. Therefore, we treat each piece of source in C and T as a document that is fed to a neural language model.

In general terms, a neural language model tries to represent each word in a sentence by means of prediction prediction task that consists of estimating the given word based on its context. We provide a detailed explanation in the next section.

While generating a model for each system under testing could provide the specificity necessary to learn good representations, it could be the case that a system could not have enough data, in terms of version differences. To explore a solution to that issue, we also implemented a aggregated model, which takes as input all the data from all the systems in the empirical study simultaneously. This aggregated model then generates representations that are cross project aware, and that allow us to study to what extend we can transfer knowledge from one project to another.

We performed an empirical study considering real word open source Java systems, which we used to compare our approach against both static and dynamic test case prioritization states of the art. Our results show that the proposed approach is competitive in terms of finding orderings with high fault detection rate, and, because it does not require explicit test case execution, it is much faster than the standard dynamic test case prioritization techniques.

The results obtained represent a relevant opportunity in the sense considering static information to perform the prioritization. It must be noted that we cannot underestimate the importance of the dynamic methods for prioritization, as they encode the actual functional behavior triggered by input data. Therefore our conclusion is that we should work towards combining both static and dynamic nature of the prioritization.

3.1.2 Background

In this section, we define the relevant concepts needed to understand our approach, from the definition of test case prioritization to the the concept of neural language model.

Test Case Prioritization

Following the definition by Rothermel et al [137], given a test suite T, and a set of permutations PT of T, and f a function that allows to rank the permutations, assigning each permutation a numeric reward, the problem of test case prioritization can be modeled as finding $T^{'} \in PT$ such that $\forall T^{''} \in PT, T^{''} \neq T^{'}, f(T^{'}) \geq f(T^{''})$.

The reward function f can encode any aspects concerning testing that we consider relevant, such as execution time, code coverage, among others. Then, by ordering the test cases by a defined criteria, a tester can execute the subset of what can be considered as more *important* test cases, meeting the execution constraints without sacrificing the performance on fault detecting.

Neural Language Models

Contrary to data modalities which manifest a continuous nature, such as image and audio, when working with language, we face the problem that words are discrete, atomic symbols. This characteristic usually leads to sparsity, which makes it more difficult to reach generalization in a statistical learning setting. Therefore, it has been historically a goal to find ways to represent text in a continuous way.

A neural language model tries to learn feature representations for each word in a defined vocabulary in order to capture the distributional regularities of the associated natural language. The key idea is try to map a each word with a feature vector of continuous values. In that sense, each word could be represented as a point the feature space. The goal is that words that are functionally and semantically close should be also close in this space. For example the word *car* should be closer to *truck* than it is to *dog.* The underlying assumption is that closer words in a sentence are more likely to to have a higher statistical dependency [141].

Neural language models use as a main component a neural network to learn the feature representations from the data. This learning process is cast as an optimization, were the goal is trying maximize the likelihood of a word given its context, ie, its surrounding words. For example, in the sentence *The lion ... meat* , the probability that a verb like *eats* appear is higher than, lets say, the verb *cooks.* We would like to learn feature representations that capture such dependencies.

Formally, lets assume the existence of a set of sentences S, which will be used as training set. Given a sentence $s_i = \{w_0, \dots, w_n\}$, and w_t a word in the position t, we define a neighborhood C conformed by c words to both right and left of w_t, $(w_{t-c}, \dots, w_{t+c})$. Therefore, what we want to do is to maximize the probability of a word given its context, ie,

$$\max P(w_t | w_{t-c}, \dots, w_{t+c}) \tag{3.1}$$

and because we want to do that for all words and all the possible co-locations of target-context, the general function to maximize can be represented as :

$$L = \sum_{t=1}^{T} P(w_t | w_{t-c}, \dots, w_{t+c}) \tag{3.2}$$

The most natural way to model that scenario is known Continuous Bag of Words model (CBOW). The neural architecture can be seen in Figure 3.1. In this case, the input layer receives the C words appearing in the context of the target word w_k.

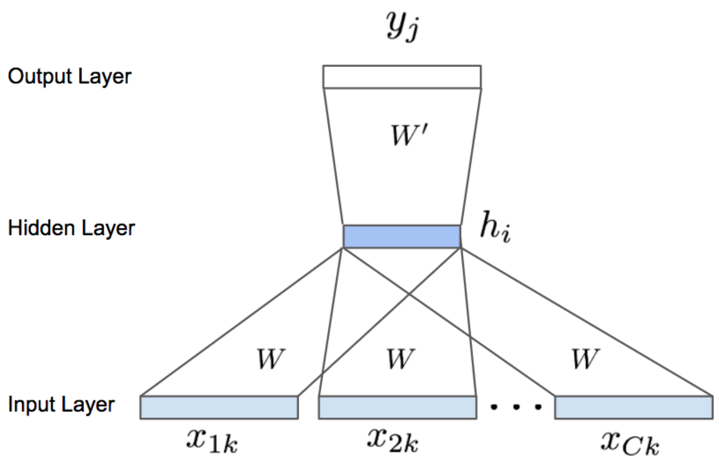


Figure 3.1: Continuous Bag of Words model (CBOW). In this case, we try to estimate a word in terms of its context.

Each word in this context is firstly transformed into a one-hot vector representation, which means a vector of size V on which only the index associated to a word has the value of 1 and the rest is 0. For example, if our vocabulary was composed of only three words, (*cat*, *dog*, *rat*), we can represent each of them as $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$, respectively.

The first learnable module of the network is W, the matrix of weights between the input and hidden layer. If we assume the hidden layer of length N, then this matrix will have dimensions $V \times N$ and each row will serve as the *input* vector representation of an associated word. For example, if row i is associated to the word w , them row i of W will be v_w^T.

The second matrix, W', of size $N \times V$ connects the hidden and the output layer. From this matrix, the j-th column, represents the *output* vector representation v'_{w_j}.

So far, we are able to obtain both an *input* and *output* representations for a given word.

In the first place, as seen in Figure, we need to aggregate the the context into one representation. CBOW model just assumes we can average their input vectors:

$$\hat{v} = \frac{1}{|C|} \sum v_{w_{t+j}} \tag{3.3}$$

Finally, the prediction of the network consists of finding, for a given context, which word as the highest probability of appear. To this end, the softmax [] function is used :

$$P(w_t | w_{t-c} : w_{t+c}) = \frac{\exp\left(\hat{v}^T v'_{w_t}\right)}{\sum_{w=1}^{W} \exp\left(\hat{v}^T v'_w\right)} \tag{3.4}$$

The Skipgram model switches the completely the formulation of the problem. Instead of predicting the a word based on its context, now the goal is to predict the context based

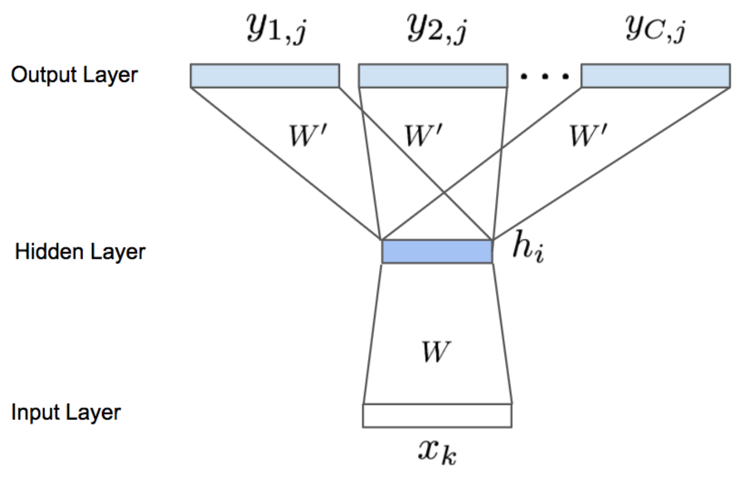


Figure 3.2: Skipgram model. In this case, we use the representation of a given word to estimate the likelihood context is predicted based on a target word.

on the a given word. The diagram for this model can be seen in Figure 3.2. Naturally, the function we want to maximize changes to:

$$L = \sum_{t=1}^{T} P(w_{t-c}, \ldots, w_{t+c}|w_t) \tag{3.5}$$

Additionally, this model assumes that words in the context are independent from each other, allowing us to formulate the prediction as follows:

$$P(w_{t-c} : w_{t+c}w_t|) = \prod P(w_{t+j}|w_t) \tag{3.6}$$

The main difference is in the output layer, where we need to output C predictions, one per each word in the context.

3.1.3 Proposed Approach

Overview

Our approach considers the availability of several consecutive versions of the system under testing (SUT), $V = \{v_0, v_1, \ldots, v_N\}$. Additionally, let us assume the existence of a test suite T, composed by m tests.

Regression testing assumes that at time t, a program version v_t passes the execution of test suite T and then, if T is executed on the next release of the system, v_{t+1}, any fault introduced in this new version is expected to manifest by not passing one or more of the existing test cases.

Following the recent work of Saha [140], we can extract differences between consecutive versions of the program, Δ_{v_i,v_i+1}. These differences can be computed in different ways and in different levels of granularity.

Once the version differences have been computed, it necessary to map them to a shared feature space along with the test cases, to allow them to be comparable using a standard metric, such as cosine similarity. This mapping can be done in several ways. Here is where we depart from the state of art, and specifically from the of Saha et al [140], which manually construct frequency-based metrics based on tf- idf for this purpose. TF-idf metrics only consider the co-occurrence between tokens, weighting accordingly. In our case, our feature learning process inherently considers the co-occurrence, but at the same time put emphasis in the order and the semantic structure of the content by learning the distributional dependencies through neural language modeling.

After we generated a continuous vector representation for each program difference and for each test case, we perform a ranking task, which consists of using the program change to query the set of test cases. This is done by computing, for each program difference, the cosine similarity between its feature representation that the representations for all the test cases, and then selecting the test case with the highest similarity. The overall process is shown in Figure 3.4.

Generating Program Version Differences

We need to explicitly characterize the changes between consecutive versions. To that end, we rely on the *diff* command to obtain an initial set of additions, deletions and modifications. From that set , we reproduce the methods proposed by Saha et al, along with our own approach:

- Low Level Diff (LDiff) : This set can be obtained just applying the *diff* command, therefore it retrieves the changes at line level.
- High Level Diff (HDiff): A variation of LDiff, where we ignore formatting differences and only consider specific atomic changes. To this end we rely on the tool Fault Tracer. The list of changes can be seen in Table 3.1.
- Compact Diff (CDiff) : Refers to LDiff or HDiff but removing duplicate changes.
- Method Encapsulation Diff (MEDiff): The previous definitions of differences tended to be too sparse in our exploratory experiments. Additionally, they do not consider the context of the token. Therefore, we hypothesize that a more local way to compare could improve the results. To this end, we proposed to first lock consecutive versions at method level and then, compute the differences.

Generating Source Code Representations

Our assumption is that source code, as a construct of human-machine communication, should manifest the same distributional regularities as any other natural language. This

Table 3.1: Proposed ways to compute the version changes.

Change in Method
Addition of Method
Deletion of Method
Addition of Field
Deletion of Field
Change in Instance of Field Intializer
Change in Static Field Initializer
Look-up Change due to Method Changes
Look-up Change due to Field Changes

assumption is supported by recent work exploring the concept of *naturalness* of code [68]. In that sense, our goal is to find an efficient way to adapt a standard neural language model to a testing scenario, where we need to learn to represent sequences of source code tokens instead of words.

The neural language models presented in the previous section are able to learn feature representations at word (token) level. While we could use that configuration indirectly to represent both test cases and the differences between versions, we might end up averaging the vectors of each token present on each instance. This could represent a step back, as the idea is to learn a representation directly at a higher level. By simply averaging, we could lose some of the learned compositionality derived from the token based models.

In order to alleviate that issue, we consider extending the standard model to learn not only representations for the tokens, but also a vector representation of the the structure that encapsulates them, ie, learn simultaneously a vector representation for a test case and a version difference. To this end, we associate each document, be it a test case or a computed difference between version, with an unique identifier, which works are an additional token in the overall vocabulary.

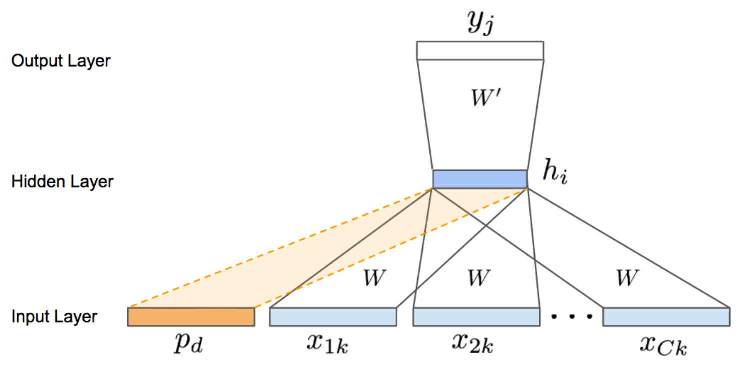


Figure 3.3: We propose to enrich the context by appending a vector representation of the associated document (a test case of a version difference) to the original set of elements

Training

Lets assume $S_{p_d} = x_1, \ldots, x_C$ a sequence of tokens corresponding to a defined portion of source code (a test case for example) p_d. From each token x_i in the sequence, we iteratively extract its context, defined through a fixed window parameter. The vector representation for each element in the context is also extracted and concatenated to form the input $i_{S_{p_d}}$ to the neural network the learns the embedding. But just before that action, we also concatenate the current vector representation of the p_d, as shown in Figure 3.3. With this, at training time, the errors are backpropagated to p_d, learning a vector representation for both the the elements conforming p_d and for p_d itself. As the co occurrence of p_d is higher, compared to the rest of the elements that conform S_{p_d}, the learned representation for p_d servers a global context for all the elements in S_{p_d}.

We repeat this process for all the test cases available and also for all the program changes computed (at a defined granularity) using the same embedding model for all the instances, with the goal of learning shared representation for both sources. At the end of the training phase, we obtain a vector representation for all the source code tokens that appear in the vocabulary resulting of merging both test case and version differences sources. Additionally, we obtained the set of vector representations for the tokens representing both the test cases and program version differences. This set will be used in the next section to perform the prioritization task.

Ranking task

Once the representations for both test cases and program version differences have been computed, it is time to use them to generate reordering on the test cases. The key idea is to cast the this task as a information retrieval problem, where we have a query, represented by a the program version difference vector and a set of possible matches, represented by the test cases. Therefore, by iteratively computing a similarity metric between the query with each match, we will obtain set of scores that represent the degree of closeness between the given program version difference and the set of test cases.

As we assume we are in a regression testing setting, when we prioritize the test cases, we want to make sure the we execute firstly the test cases that are closer the version change. Therefore, we need to return the set of test cases ordered by the similarity scores in a decreasing fashion.

As we obtained vector representations from a shared embedding model, these representations also share the same dimensionality, therefore, a simple similarity we can compute is the *cosine* similarity, defined as follows for two vectors A, B:

$$sim(A, B) = \frac{AB}{||A|||B|||} \tag{3.7}$$

3.1.4 Evaluation

We had two main goals when evaluating the proposed approach. The first one, naturally, to explore how our model behaves when compared to other prioritization methods in terms of fault detection rate. In the second place, we were also interested in study the feature learning process, in the sense of analysis the transformation from source code segments to a

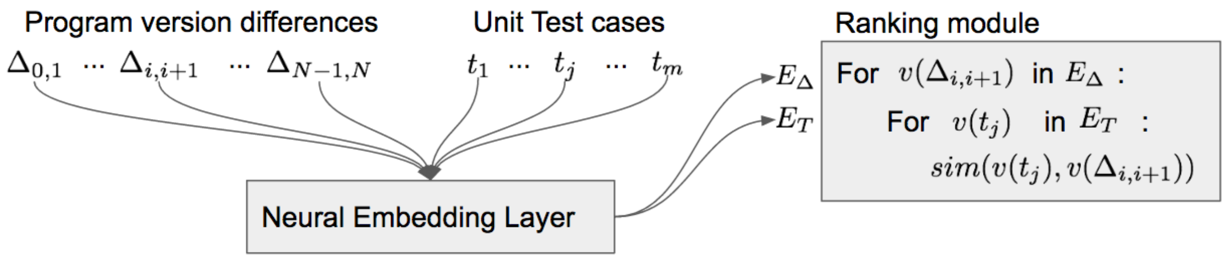

Program version differences
$\Delta_{0,1}$ ··· $\Delta_{i,i+1}$ ··· $\Delta_{N-1,N}$
Unit Test cases
t_1 ··· t_j ··· t_m
Neural Embedding Layer
E_Δ
E_T
Ranking module
For $v(\Delta_{i,i+1})$ in E_Δ :
For $v(t_j)$ in E_T :
$sim(v(t_j), v(\Delta_{i,i+1}))$

Figure 3.4: Proposed prioritization model: Both test cases and program version differences are passed through a neural embedding module that learns continuous feature vector representation for each of them. Then, a ranking module takes a the vector associated to each program difference and query the set of vectors associated to the test cases, computing the cosine similarity. Finally, the system returns the ordered list.

continuous vector representations. We designed an empirical study to explore the following research questions:

- RQ 1: Is the proposed approach more effective than a random ordering?
- RQ 2: How does the proposed approach compare to prioritization technique based on topic models?
- RQ 3: How does the proposed approach compare to the work of Saha et al [140] REPiR?
- RQ 4: How does the proposed approach compare to a dynamic prioritization technique?
- RQ 5: How do the hyper parameters of the model impact on the prioritization performance?

The first two research questions were chosen to serve as baselines. We estimate any prioritization method should be at least better than a random ordering, therefore is it critical to have it checked in that aspect. In the case of the second question, we propose to compare against a straightforward implementation of a topic model method. Topic model have been used previously and represent a more probabilistic way to obtain feature representations in terms of topic distributions.

In the case of the third and fourth questions, we directly compare against what we consider the state of the art method that considers a information retrieval to prioritize the test cases. Additionally, we wanted to replicate the results against a dynamic prioritization technique , as we consider that the emergence of methods the do not consider explicitly test case execution is a direct consequence of the cost of capturing coverage data.

Finally, the fifth question allows to explore and perform a sensitivity analysis on the parameter of the proposed model.

Table 3.2: Subject and program versions collected.

Project	Version Pair
Time and Money	3.0 - 4.0
Time and Money	4.0 - 5.0
Mime4J	0.5 - 0.6
Mime4J	0.61 - 0.68
Jaxen	1.0b7 - 1.0b9
Jaxen	1.1b6 - 1.1b7
Jaxen	1.0b9 - 1.0b11
Xml-Security	1.0 - 1.1
XStream	1.20-1.21
XStream	1.21-1.22
XStream	1.22-1.30
XStream	1.30-1.31
XStream	1.31-1.41
XStream	1.41-1.42
Commons Lang	3.02 - 3.03
Commons Lang	3.03 - 3.04
Joda Time	0.90 - 0.95
Joda Time	0.98 - 0.99
Joda Time	1.10 - 1.20
Joda Time	1.20 - 1.30
Apache Ant	0.0 - 1.0
Apache Ant	3.0 - 4.0
Apache Ant	4.0 - 5.0
Apache Ant	6.0 - 7.0

Case Examples

In the first place, we tried to replicate the same dataset presented in [140], as we consider the correct way to perform a fair comparison. Therefore, we obtained Xml-Security and Apache Ant from the SIR repository, while the rest was available from their respective websites. For each subject system, we first extract all the major releases with their test cases and consider each consecutive versions as a version-pair. For each pair, we run the old regression test suite on new version to find possible regression test failures. Then, we treat the changes causing those test failures as the regression faults. In this way, we were able to identify 24 version-pairs with regression faults, which are all used in our study and are shown on Table 3.2

Experimental Design Overview

In this study, we investigate the impact of one independent variable, the prioritization technique applied, on one dependent variable, the rate of fault detection, using randomization to control for several factors.

Independent Variable For this experimental setup, we consider four prioritization techniques. In the first place, we implemented *random* prioritization, where we randomly order the test suite. We expect this to be, on average, the most ineffective method for ordering test cases, and thus it serves as a useful baseline for understanding the effectiveness of our approach. In the second place, we implemented Repir, the approach proposed by Saha et al [140], which we consider to date the state of the art for test case prioritization that is based on information retrieval using syntactic data. In the third place, we implemented coverage based prioritization techniques, both branch and block level that we describe as *dynamic* prioritization. In the fourth place, we implemented a basic prioritization technique based on topic models. In this case, the mechanics of the approach are similar to the proposed approach, in the sense that the prioritization is cast also as a information retrieval, but differ in the sense that the feature vector representation for each entities come a probabilistic model such as Latent Dirichlet Allocation (LDA) which learns the distribution of latent topics associated to each token. Finally, we presented our approach.

Dependent Variable The most widely used metric for evaluating test case prioritization is the Average Percentage of Fault Detected (APFD), which takes into account the rate of fault detection for a given permutation of the test suite under study. Given a test suite T with n test cases that is used to test a system which contains a set F of m faults revealed by T. Let TF_i be the index of the first test case in ordering TO of T that reveals fault i. Then the score associated to the ordering TO will be given by:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n} \tag{3.8}$$

The APFD takes values from 0 to 100, where a high scores imply faster fault detection rates.

3.1.5 Threats to Validity

External: Our work is limited to the domain of Open Source source, which may not be representative of the whole software development ecosystem. Nevertheless, the subjects we used in this work has been used in a variety of scenarios by a large community.

Internal: The core of our methodology resides on representations that the neural language model is able to learn from both the program changes and the test cases. If those representations are not expressive enough or are not able to converge, the rankings obtained from the similarity computation will not be robust. To alleviate that issue, we conducted a series of preliminary experiments to test in distributional terms the resulting feature vectors. Moreover, we carried out an exhaustive hyper parameter search process, where we systematically analyzed the sensitivity of each model parameter and contrasted then with the Finally, in order to keep a safety check, we sampled groups of change -test pairs to analyze visually their similarity as a way to provide manual validation.

Additionally, one of our main concerns is related to implementation issues. In the case of our method, we built our codebase on top of existing libraries for natural language processing, such as Gensim.

Construct: The APFD metric has been systematically used for evaluating test case prioritization techniques. While it provides a quantitative measurement of the rate at which a given permutation could detect a set of faults, it does not consider other aspects, such as the time and effort necessary to compute the reordering. In that sense, when comparing test case prioritization techniques it is necessary to enrich the analysis by means of adding other dimensions, such as time and resources used.

3.1.6 Results and Analysis

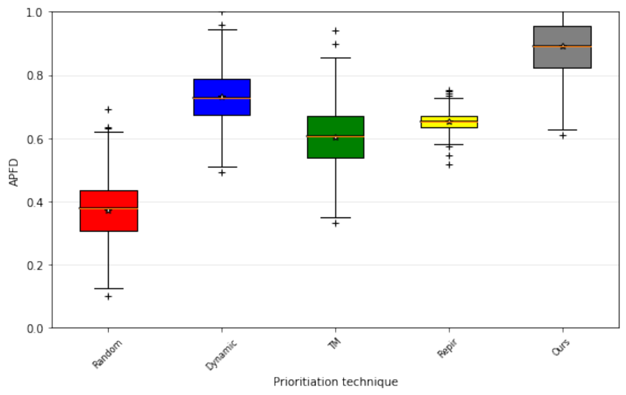


Figure 3.5: Results for Ant

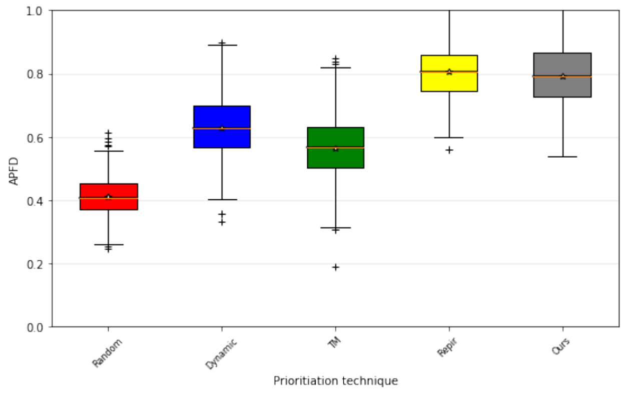


Figure 3.6: Results for Commons Lang

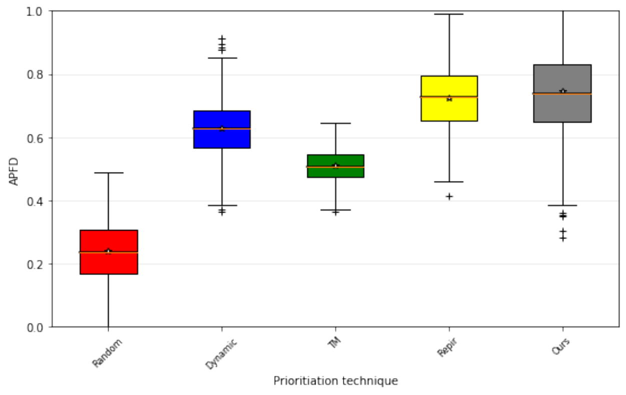


Figure 3.7: Results for Jaxen

We now discuss the results of our study in the context of our two research questions 1 to 4. We plot the computed APFD values for each subject and prioritization technique combination as a boxplot.

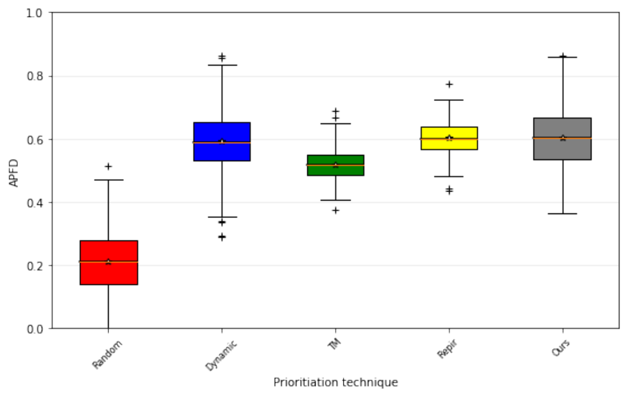


Figure 3.8: Results for Joda Time

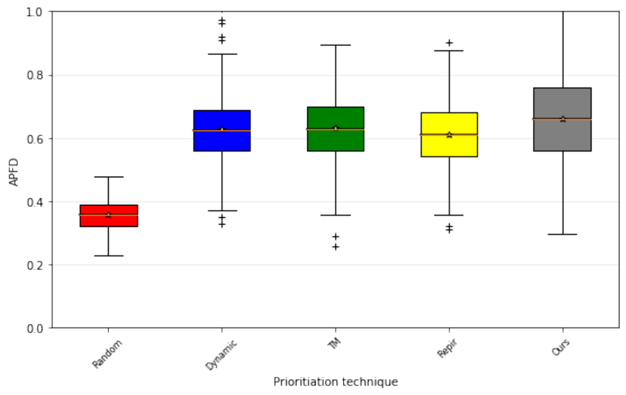


Figure 3.9: Results for Mime4J

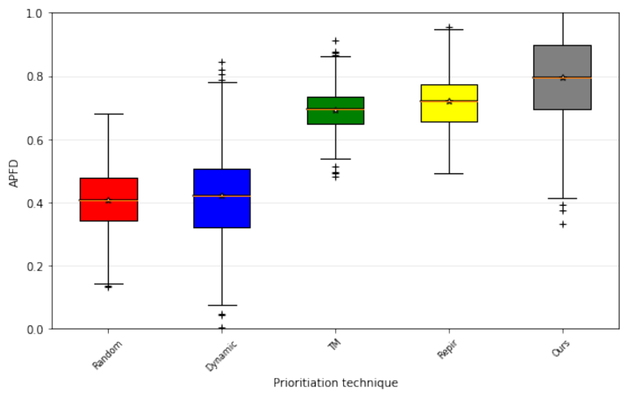


Figure 3.10: Results for Time and Money

As the results show, on average, our prioritization approach outperforms both random and topic model prioritization for each case example, with improvements ranging from 5.10% to 9.52%. When comparing against dynamic and Repir, our approach shows competitive results, taking the lead in most cases. In that sense, the results suggest that, for the systems considered, the our approach can be a generally preferable method for prioritizing test cases. However there are some cases that present a slightly fair amount of overlap exists in the APFD distribution between out prioritization method and Repir. We thus wished to determine whether our results were statistically significant, i.e., if our results were likely due to chance. To achieve that, we carried out a statistical tests by restating the research questions as pairs of null and valid hypothesis, from which we computed a hypothesis test with confidence α at

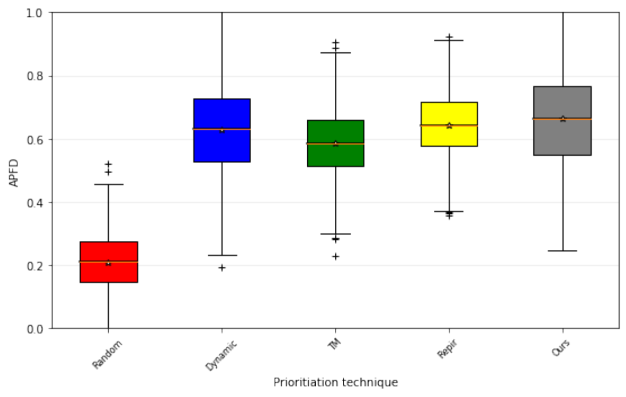


Figure 3.11: Results for XML Security

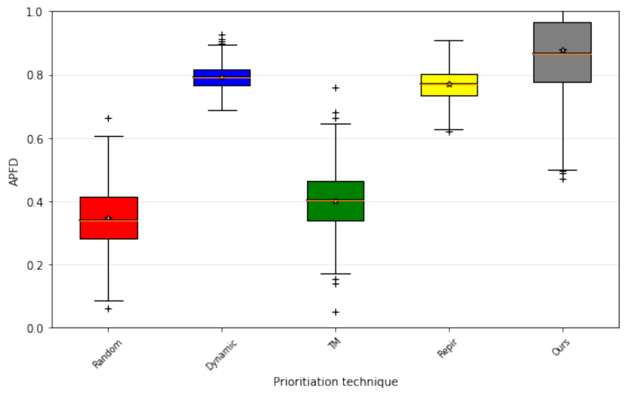


Figure 3.12: Results for XStream

0.05. The results of that tests shows that in over 95% of the configuration pairs studied, we can reject the null hypothesis with $p < 0.05$, supporting the assertion that our approach is , with respect to the APFD scores, more effective than the rest of approaches.

Another element to take into consideration is the time consumption associated to the prioritization methods. the fastest methods are random and topic model based prioritization, but the drawback associated to their performance makes us doubt about their real effectiveness. Interestingly, when we compare the time taken by the dynamic , Repir and our proposed approach, we can see that Repir is faster on average only 4% faster than dynamic prioritization, while our approach reaches improvements up to 22% in some cases.

Impact of Embedding Parameters

One the most relevant parameters that needs to be defined in our model is the length of the resulting vector representations. This value not only affect the time involved on the computation of the learning process, but also could impact on the expressiveness and robustness of the models, as the same information can be encoded into different dimensions.

Additionally, when we train our model, we define, for each token, a context, which means a set of surrounding tokens. The length of this set is also a matter of study as it is closely associated with the capacity of the embedding to encode long term relationships between the tokens, which in the case of a programming language is a relevant factor as it could allow us to obtain a notion of the dependency of the instructions.

Vector Size In this case we performed a sensibility analysis of the parameter, ranging from $d = 50$ to $d = 150$. We chose this interval as literature has empirically found that across several natural language tasks, a vector dimension of $d = 100$ provides the best cost-effectiveness in terms of training time and model performance. It must be noted that around $d = 150$, the time needed to obtain the features vectors start to increase substantially, and that increment is not correlated with the performance we saw on APFD scores.

Context Size The context size, i.e. the window of tokens selected for prediction in the language model, was also studied. This parameter is relevant, as it controls the amount the length of the dependencies that the model needs consider. We performed a search procedure, ranging the context size from 2 to 20. The results across projects shows that on average, a window of 5 produced the best results.

3.1.7 Related Work

Test Case Prioritization

Test case prioritization has received a vast amount of attention during the years, with a wide range of of proposed techniques. Naturally, code coverage data was used in the beginning as the main criteria for reordering [40].

Dataflow information has also been studied, beginning with the work of Rummel et al. [138] which used def-use pairs to rank test cases, and also the work by Staats et al [148], which uses dataflow information to define a metric related to the test oracle.

In terms of approaches that do not consider explicitly dynamic information, we can identify firstly the use of expert information such as requirements and system knowledge to support the prioritiation. In this case, the works by Srikanth [146] and also Tonella [156]. A step further can be seen in Ma et al. [97] where expert knowledge is combined with program structure. Additionally, we can find some approaches that try to automate the extraction of insights from requirements by using natural language processing techniques. Such is the case of Arafeen et al. [5] which uses clustering of requirements and Nguyen [117] for test case prioritization of web services based on analyzing the changes in the service descriptions.

Moving to prioritization techniques that rely only on static program information, the works of Zhang et al [172] and then Mei et al [103] make use of the static call graph to

Regarding prioritization techniques based on source code changes, which are the closer to our approach, we can find the work of Thomas [154] which uses topic models. More recently, the work by Saha et al [140], REPiR, find representation of program changes and test cases based on vector space model and rank test cases based on a similarity metric.

3.1.8 Conclusion and Future Work

In this work, we presented a novel approach to prioritize test cases based on static information. We focused on automatically learning feature representations of both the source code and the test cases instead of manually computing frequency based metrics. Our results on a set of real world open source programs shows that these learned representations are expressive and allows to perform a reordering that , on average, are competitive with standard dynamic analysis based approaches and are able to outperform the state of the art in the use of information retrieval for test case prioritization, the approach proposed by Saha et al [140]. We consider that these results suggest that i) the use of static information as criteria for prioritizing test cases is a valid research line, given the competitive results and the evident saving in time and resources , and ii) the design of methods that are able to learn feature representations directly from the data is a promising.

As a future work, our goal is to explore ways to combine both dynamic and static test case data. Our hypothesis is that the flexibility of the representations obtained with the proposed method could ease its combination with dynamic data, such as execution traces.

Chapter 4

Naturalness of Code

Preface

In order to find ways to represent programs in a tractable way, we rely on the assumption that both source code and natural languages follow similar distributional characteristics, and therefore, is it possible to transfer knowledge from one modality to the other. In this chapter, we challenge that assumption attacking two relevant problems for developers. First, understanding changes in the code, for which we develop a encoder decoder method to translate code commits into natural language descriptions, and in the second place, bug localization, which is the process of understanding a report written by end user and then locate the portion of the code which is responsible from that misbehavior.

4.1 Generating Natural Descriptions from Source Code Changes

4.1.1 Introduction

Source code, while conceived as a set of structured and sequential instructions, inherently reflects human intent: it encodes the way we command a machine to perform a task. In that sense, it is expected that it follows to some extent the same distributional regularities that a proper natural language manifests [68]. Moreover, the unambiguous nature of source code, comprised in plain and human-readable format, allows an indirect way of communication between developers, a phenomenon boosted in recent years given the current software development paradigm, where billions of lines code are written in a distributed and asynchronous way [52].

The scale and complexity of software systems these days has naturally led to explore automated ways to support developers' code comprehension [87] from a linguistic perspective. One of these attempts is automatic summarization, which aims to generate a compact representation of the source code in a portion of natural language [60].

While existing code summarization methods are able to provide relevant insights about the purpose and functional features of the code, their scope is inherently static. In contrast, software development can be seen as a sequence of incremental changes, intended to either generate a new functionality or to repair an existing one. Source code changes are critical for understanding program evolution, which motivated us to explore if it is possible to extend the notion of summarization to encode code changes into natural language representations, i.e., develop a model able to *explain* a source code level modification. With this, we envision a tool for developers that is able to *i)* ease the comprehension of the dynamics of the system, which could be useful for debugging and repairing purposes and *ii)* automate the documentation of source code changes.

To this end, we rely on the concept of code commit, the standard contribution procedure implemented in modern subversion systems [52], which provides both the actual change and a short explanatory paragraph. Our model consists of an encoder-decoder architecture which is trained on a set of triples conformed by the version of a system before and after the change, along with the comment. Given the high heterogeneity of the modalities involved, we rely on an attention mechanism to efficiently learn the parts of the sequences that are more expressive and have more explanatory power.

We performed an empirical study on twelve real world software systems, from which we obtained the commit activity to evaluate our model. Our experiments explored in-project and cross-project scenarios, and our results showed that the proposed model is able to consistently generate semantically sound descriptions.

4.1.2 Related Work

The use natural language processing to support software engineering tasks has increased consistently over the years, mainly in terms of source code search, traceability and program feature location [9, 122].

The emergence of unifying paradigms that explicitly relate programming and natural languages in distributional terms [68] and the availability of large corpus mainly from open source software opened the door for the use of language modeling for several tasks [132]. Examples of this are approaches for learning program representations [109], bug localization [72], API suggestion [56] and code completion [133].

Source code summarization has received special attention, ranging from the use of information retrieval techniques to the addition of physiological features such as eye tracking [135]. In recent years several representation learning approaches have been proposed, such as [3], where the authors employ a convolutional architecture embedded inside an attention mechanism to learn an efficient mapping between source code tokens and natural language keywords.

More recently, [73] proposed a encoder-decoder model that learns to summarize from Stackoverflow data, which contains snippet of code along with descriptions. Both approaches share the use of attention mechanisms [10] to overcome the natural disparity between the modalities when finding relevant token alignments. Although we also use an attention mechanism, we differ from them in the sense we are targeting the changes in the code rather than the description of a file.

In terms of specifically working on code change summarization, [34, 91] propose a method based on a set of rules that considers the type and impact of the changes, and [29] combines summarization with symbolic execution. To the best of our knowledge, our approach represents the first attempt to generate natural language descriptions from code changes without the use of hand-crafted features, a desirable setting given the heterogeneity of the data involved.

4.1.3 Proposed Model

Our model assumes the existence of T versions of a given project $\{v_1, \dots, v_T\}$. Given a pair of consecutive versions (v_{t-1}, v_t), we define the tuple (C_t, N_t), where $C_t = \Delta_{t-1}^{t}(v)$ represents a code snippet associated to changes over v in time t and N_t represents its corresponding natural language (NL) description. Let $\mathbb{C}$ be the set of all source code snippets and $\mathbb{N}$ be the set of all descriptions in NL. We consider a training corpus with T code snippets and summary pairs (C_t, N_t), $1 \leq t \leq T$, $C_t \in \mathbb{C}$, $N_t \in \mathbb{N}$. Then, for a given code snippet $C_k \in \mathbb{C}$, the goal of our model is to produce the most likely NL description $N^\star$.

Concretely, similarly to [73], we use an attention-augmented encoder-decoder architecture. The encoder can be seen as a lookup layer, which simply reads through the source input sequence and returns the embedded tokens. The decoder is a RNN that reads this representation and generates NL words one at a time based on its current hidden state and guided by a global attention model [96]. We model the probability of a description as a product of the conditional next-word probabilities. More formally, for each NL token $n_i \in N_t$ we define,

$$h_i = f(n_{i-1}E, h_{i-1}) \tag{4.1}$$

$$p(n_i | n_1, ..., n_{i-1}) \propto W \tanh(W_1 h_i + W_2 a_i) \tag{4.2}$$

where E is the embedding matrix for NL tokens, $\propto$ denotes a softmax operation, h_i represents the hidden state and a_i is the contribution from the attention model on the source code. W, W_1 and W_2 are trainable combination matrices. The decoder repeats the recurrence until a fixed number of words or a special *END* token is generated. The attention contribution a_i is defined as $a_i = \sum_{j=1}^{k} \alpha_{i,j} \cdot c_j F$, where $c_j \in C_t$ is a source code token, F is the source code token embedding matrix and $\alpha_{i,j}$ is:

$$\alpha_{i,j} = \frac{\exp\left(h_i^\top c_j F\right)}{\sum_{c_j \in C_t} \exp\left(h_i^\top c_j F\right)} \tag{4.3}$$

We use a dropout-regularized LSTM cell for the decoder [169] and also add dropout at the NL embeddings and at the output softmax layer, to prevent over-fitting. We added special *START* and *END* tokens to our training sequences and replaced all tokens and output words occurring less than 2 and 3 times, respectively, with a special *UNK* token. We set the maximum code and NL length to be 100 tokens. For decoding, we approximate $N^{\star}$ by performing a beam search on the space of all possible summaries using the model output, with a beam size of 10 and a maximum summary length of 20 words.

To evaluate the quality of our generated descriptions we use both METEOR [83] and sentence level BLEU-4 [123]. Since the training objective does not directly optimize for these scores, we compute METEOR on our validation set after every epoch and save the intermediate model that gives the maximum score as the final model. For evaluation on our test set we used the BLEU-4 score.

4.1.4 Empirical Study

Data and pre-processing: We captured historical data from twelve open source projects hosted on Github based on their popularity and maturity, selecting 3 projects for each of the following languages: *python*, *java*, *javascript* and *c++*. For each project, we downloaded diff files and metadata of the full commit history. Diff files encode per-line differences between two files or sets of files in a standard format, allowing us to recover source code changes in each commit at the line level. On the other hand, medatada allows us to recover information such as the author and message of each commit.

The extracted commit messages were processed using the Penn Treebank tokenizer [100], which nicely deals with punctuation and other text marks. To obtain a source code representation of each commit, we parsed the diff files and used a lexer [25] to tokenize their contents in a per-line fashion allowing us to maximize the amount of source code recovered from the diff files.

Experimental Setup: Given the flat structure of the diff file, source code in contiguous lines might not necessarily correspond to originally neighboring code lines. Moreover, they might come from different files in the project. To deal with this issue, we first worked only with those commits that modify a single file in the project; we call this the *atomicity* assumption. By using only *atomic* commits we reduced our training data by an average of roughly 50%, but in exchange we made sure all the extracted code lines came from the same file. At the same time, we expect to maximize the likelihood of observing a direct relation between the commit message and the source code lines altered.

We then relaxed our *atomicity* assumption and experimented with the *full* commit history. Given our maximum sequence length constrain of 100 tokens, we only observed an average of 1,97% extra data on each project. Since source code lines may come from different files, we added a delimiting token *NEW_FILE* when corresponding.

We were also interested in studying the performance of the model in a cross-project setting. Given the additional challenges that this involves, we designed a more controlled experiment. Starting from the *atomic* dataset, we selected commits that only add or only remove code lines, conforming a derived dataset that we call *uni-action.* We chose the *python* language to maximize the available data. See Table 4.1.

Language	Project	Full	Atomic	Added	Rem.
python	Theano	24,200	65.40%	11.43%	2,83%
	keras	2,855	66.02%	11.07%	3,01%
	youtube-dl	13,968	74.49%	11.52%	2,59%
javascript	node	15,811	53.17%	11.87%	3,21%
	angular	6,204	32.90%	5.59%	1,72%
	react	7,806	53.29%	12.67%	2,72%
c++	opencv	20,480	50.08%	8.83%	1,66%
	CNTK	10,792	38.36%	6.00%	2,23%
	bitcoin	12,596	48.11%	9.84%	2,56%
java	CoreNLP	9,149	42.77%	7.84%	1,98%
	elasticsearch	25,764	43.77%	9.02%	2,61%
	guava	3,821	38.63%	8.90%	2,64%
Average		12,787	50.58%	9.55%	2,48%

Table 4.1: Summary of our collected data.

Results and Discussion: We begin by training our model on the *atomic* dataset. As baseline we used MOSES [82] which although is designed as a phrase-based machine translation system, was previously used by [73] to generate text from source code. Concretely, we treated the tokenized code snippet as the source language and the NL description as the target. We trained a 3-gram language model using KenLM [63] and used mGiza to obtain alignments. For validation, we use minimum error rate training [17, 120] in our validation set.

As Table 4.3 shows, our model trained on *atomic* data outperforms the baseline in all but one project with an average gain of 5 BLEU points. In particular, we observe bigger gains for java projects such as *CoreNLP* and *guava*. We hypothesize this is because program differences in Java tend to be longer than the rest. While this impacts on training time, at the same time it allows the model to work with a larger vocabulary space. On the other hand, our model performs similarly to MOSES for the *node* and slightly worse for the *youtube-dl*. A detailed inspection of the NL messages for *node* showed that many of them exhibit a fixed pattern in their structure. We believe this rigidity restrains the generation capabilities of the decoder, making it more prone to memorization.

Table 4.2 shows examples of generated descriptions for real changes and their references. Results suggest that our model is able to generate semantically sound descriptions for the changes. We can also visualize the summarizing power of the model, as seen in the *Theano* and *bitcoin* examples. We observe a tendency to choose more general terms over too specific ones meanwhile also avoiding irrelevant words such as numbers or names. Results also suggest the emergence of rephrasing capabilities, specifically in the second example from *Theano*. Finally, our generated descriptions are, in most cases, semantically well correlated to the reference descriptions. We also report not so successful results, such as case of *youtube-dl*, where we can see signs of memorization on the generated descriptions.

Regarding the cross-project setting experiments on *python*, we obtained BLEU scores of 14.6 and 18.9 for only-adding and only-removing instances in the *uni-action* dataset, respectively. We also obtained validation accuracies up to 43.94%, suggesting feasibility in this more challenging scenario. Moreover, as the generated descriptions from the *keras* project in Table 4.2 show, the model is still able to generate semantically sound descriptions.

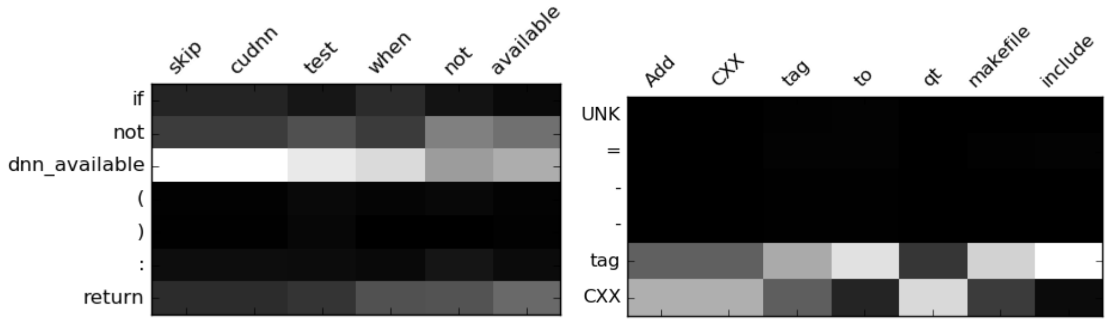


Figure 4.1: Heatmaps of attention weights $\alpha_{i,j}$.

Despite the small data increase, we also trained our model on *full* datasets as a way to confirm the generative power of our model. In particular, we wanted to test the model is able leverage on *atomic* data to also capture and compress multi-file changes. As shown in Table 4.3, results in terms of BLEU and validation accuracy manifest reasonable consistency, despite the higher disparity between source code and natural language on this dataset, which means the model was able to learn representations with more compressive power.

Soft alignments derived from Figure 4.1, which shows examples of attention heatmaps, illustrate how the model effectively associates source code tokens with meaningful words.

4.1.5 Conclusion and Future work

We proposed an encoder-decoder model for automatically generating natural descriptions from source code changes. We believe our current results suggest that the idea is feasible and, if improved, could represent a contribution for the understanding of software evolution from a linguistic perspective. As future work, we will consider improving the model by allowing feature learning from richer inputs, such as abstract syntax trees and also functional data, such as execution traces.

	Reference	**Generated**
keras	Fix image resizing in preprocessing/image	Fixed image preprocessing .
	Fix test flakes	Fix flaky test
Theano	fix crash in the new warning message .	Better warning message .
	remove var not used .	remove not used code .
	Better error msg	better error message .
bitcoin	Merge pull request 4486 45abeb2 Update Debian packaging description for new bitcoin-cli (Johnathan Corgan)	Update Debian packaging description for new bitcoin-cli
	Add two unittest-related files to .gitignore	Add : Minor files to .gitignore
CoreNLP	Add a bunch of verbs which are more likely to be xcomp than vmod	Add a bunch of verbs which are more to be xcomp than vmod
	Add a brief test for optional nodes	make this test do something
youtube-dl	[crunchyroll] Fix uploader and upload date extraction	[crunchyroll] Fix uploader extraction
	[extractor/common] Improve base url construction	[extractor/common] Improve extraction
	[mixcloud] Use unicode_literals	[common] Use unicode_literals
opencv	fixed gcc compilation	fixed compile under linux
	remove unused variables in OCL_PERF_TEST_P ( )	remove unused variable in the module

Table 4.2: Examples of generated natural language passages v/s original ones taken from the test set.

Dataset	*atomic*			*full*	
	Val. acc	**BLEU**	**Moses**	**Val. acc**	**BLEU**
Theano	36.81%	9.5	7.1	39.88%	10.9
keras	45.76%	13.7	7.8	59.30%	8.8
youtube-dl	50.84%	16.4	17.5	53.65%	17.7
node	52.46%	7.8	7.7	53.70%	7.2
angular	44.39%	13.9	11.7	45.06%	15.3
react	49.44%	11.4	10.7	48.61%	12.1
opencv	50.77%	11.2	9.0	49.00%	8.4
CNTK	48.88%	17.9	11.8	44.85%	9.3
bitcoin	50.04%	17.9	13.0	55.03%	15.1
CoreNLP	63.20%	28.5	10.1	62.25%	26.7
elasticsearch	36.53%	11.8	5.2	35.98%	6.4
guava	65.52%	29.8	19.5	67.15%	34.3

Table 4.3: Results on the *atomic* and *full* datasets.

Chapter 5

Automatic Repair

Preface

While predicting and detecting defects is a relevant part of any quality assessment pipeline in software development, the remaining part is actually taking the defects identified and fix them.

Fixing a bug requires a set of skills from the side of the developer. In the first place, the bug requires an understanding, in the sense of knowing *why* it is a bug. This information can be obtained through the output of a test case execution. In the second place, after identifying the dissonance between expected and real program behavior, along with the location of the source of the fault, there is a set of decision to be taken in order to modify the code to produce a fix. Given the flexibility that source code provides, there are several ways to fix a given defect.

In this chapter, we explore a learning approach for bug fixing. While repair can be seen as an optimization problem, in the sense of finding a code modification that maximizes that probability of passing a test suite execution, we are more interesting in finding repairing policies instead of particular solutions. In that sense, this represents a problem on which we can reuse and combine all the techniques and insights obtained in the previous chapters.

5.1 Learning Fixing Trajectory Policies

5.1.1 Introduction

Repairing a bug is essentially a search problem. At source code level, it is necessary to find the combination of structural and semantic changes that lead us obtain a version of the code on which the reported fault is not present anymore. Therefore, the code needs to be transformed, from a defective state to a healthy one. We assume this transformation can be achieved by adding, modifying or even deleting portions of the code.

Existing techniques for repair take the problem only from the source code perspective, and therefore, only consider that syntactic aspects of the program. While this configuration appears as the most natural approach, we want to explore a more top-down way, in the sense of explicitly consider how humans find their way to fix a bug.

Therefore, we propose to model the bug fixing dynamics as a sequence of progressive modifications that conform a trajectory across the space of feasible solutions followed by a human when trying to repair a program. Our goal is to understand the transitions between code level changes that lead to reliable and effective solution.

In that sense, we firstly need to let a program *observe* how real faults are fixed by humans, by means of observing editing activity. After capturing this transitions, our model could be able to extract the distributional patterns of necessary to carry out a new fix.

Therefore, instead of using supervised learning approach, where we assume the existence of a training set of the form (X, y), where X represents a set of features and y a defined label associated to the instance, here we are more interested on obtaining sequential data, in the form of chain of consecutive changes from which we can learn how the trajectory of partial changes impact the overall solution.

In terms of how to model the problem, we consider that reinforcement learning fits well our requirement: instead of learning from a fixed mapping between the dependent variable feature representations and a label, the learning process in this case is cast as sequential decision making, where the learner understand its current state and choose the action that will most likely maximize the future rewards [151].

Therefore, the first step is to obtain data that represents real transitions developers follow when they build or modify a program. To this end, we rely on data from Massive Open Online Courses (MOOCs).

MOOCs represent one of the most interesting initiatives aimed at spreading high quality education content, with the goal of democratizing education. As the impact of that goal could be huge, the amount of complexity involved is also considerable, in both social and technical dimensions.

One the key elements that a massive online system needs to address is related to the diversity of students. Contrary to a standard teaching setting, where we can expect certain level of homogeneity of skills and cultural background, the openness and accessibility of massive online classes is characterized by the diversity of the participants. Given the low entry barriers, we can expect users coming from the most diverse cultural backgrounds, geographical locations, etc. While diversity represents a positive aspect, as it is in fact one of the desired elements these systems seek, at the same time it represents challenge, in terms of the volume and the variability of the content that students generate.

In terms of understanding participants, literature has focused on several aspects, such as motivations, incentives, among others, which are key to optimize the user experience and retention rate, similarly to the research in other collaborative systems such as Open Source software development [92].

One specific element that has attracted attention is how can we track and understand users learning ability. This is of high relevance , as it touches the core of the platform and impacts directly on the the success of the initiative: If we cannot understand users learning ability and design or adapt tools to optimize their performance, the degree of engagement will drop naturally. One way to explore this problem is to analyze the students solutions to the assignments. Recently, a online learning initiative called One Hour Code[1], which tries to teach programming skills released a public dataset consisting on the sequences of partial solutions users submit in order to complete a programming assignment. The interesting part of this dataset is that it contains additionally a *hint* for a given partial state contributed by experts. This hints are aimed at improving the task solving process when the user get stuck. Moreover, as we can see the full set of trajectories that each user follow on each assignment, we can capture the transitions (code changes) that were most effective, which means, that led to reach a correct solution in the shortest way.

Therefore, the above dataset can be used for our purpose of learning bug fixing policies: We are in the presence of an agent whose goal is to write a correct solution for a given programming task. This agent start in a initial state and perform a *move*, in the sense of writing an initial partial solution. From that point, he immerses in a sequential decision making process that consists of, given a current status on his assignment, select one of the feasible code modifications in order to maximize a expected reward, which can be computing as a function of how far the partial solution is from the *ideal* and correct one. In that sense, for us the *fixing* action is actually a source code modification that takes a partial solution to a correct solution.

One of the key elements in any reinforcement learning scenario is the representation of the states. As in the current configuration we are in the presence of source code, we need to find a way to obtain a flexible representation. If we work at token level, we may encounter sparsity issues, and given the usually small vocabulary of source code tokens , compared to a natural language setting, it could be difficult to differentiate partial states. To overcome that issue, we defined a method to learn feature learning representations from the abstract syntax trees of each partial solution.

We performed an empirical study comparing our reinforcement learning approach against two methods proposed in the literature, namely *Desirable Path* , *Independent Probable Path* and *Rivers Policy*, showing that the proposed approach is competitive, outperforming in terms of final accuracy.

5.1.2 Related Work

Automatic repair has been one of the most desired goals in software engineering research. Historically, the problem has been attacked as a search task, where , given a defined defect, the system need to generate strategies to find candidate patches. This process is iterative and several heuristics have been used, such as genetic algorithms [161], where the goal is to

[1]`https://code.org/research`

construct better candidates y maximizing a fitness function based on a set of test cases [6,7].

Another line of research deals with using human generated patches , available in open source software to train a discriminative model over a set of features computed from the patch, such as the work of Long et al [94].

The use of representation learning has recently attacked attention in the automatic repair community. The flexibility of these tools may allow to generate end-to end solution, which do not need to rely heavily on feature engineering. An example of this new familiy of methods is DeepFix, by Gupta et el [57], where they implement an encoder decoder architecture to generate patches based on a training set obtained from student assignments. A similar approach is followed by Pu et al [129].

5.1.3 Proposed Approach

Our approach consists of modeling the sequential process of fixing , or more generally, the trajectory of source code changes from an initial (partial or incomplete) state to a final (complete) state with an reinforcement learning agent. Figure 5.1 shows a diagram of the proposed approach.

Contrary to Supervised Learning, Reinforcement learning is a goal directed learning paradigm that focuses on the interaction between a learning agent and its environment. In that sense, there no explicit class or label upon which we can compute a loss function in order perform learning. The key element in reinforcement learning lies on the existence of a reward function, which acknowledge that actions taken given a specific state [?] .

Therefore, we begin by assuming an agent capable of modifying a program, which we associate as the environment ξ. At each step t, the agent selects an action a_t, in our a source code level modification, from a valid set of modifications $A = 1, \dots, K$. The modification is applied to the program, which reach a state x_t, which is a new version of the system. The agent also receives a reward r_t which is associated to the quality of the modification. For example, we could construct an evaluation function that associates positive scores if the resulting program passes a set of units test, or is syntactically valid. In this case, the feedback that the user receives will immediate. But that could be not the case, for example, we could select a configuration on which the reward is a signal that the agent receives only at the end of a sequence of steps, for instance, of the program reaches an optimal state.

Based on this configuration, we are in the presence of a Markov decision process, where we consider the sequences of actions and observations, $s_t = x_1, a_1, x_2, a_2, \dots, a_{t-1}, x_t$ the agent follows as the state representation at time t.

As in any reinforcement learning scenario, the goal of the agent is to continuously interacts with the environment, the program in our case, the choosing the actions in a way that future rewards are maximized. We follow the standard assumption that future rewards are discounted, defining the return at time t as:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{r'} \tag{5.1}$$

where γ is the discount factor per time step and T the time step at which the task ends (assuming a finite number of steps).

To relate both states and actions, an action-value function $Q^*(s, a)$ is defined as the maximum expected return the agent can obtain, following any feasible strategy, after seeing the sequence s and, because of that state, having chose to perform the action a:

$$Q^* = max_\pi E(R_t | s_t = s, a_t = a, \pi) \tag{5.2}$$

where π is the *policy* the agent learns in order to map states (sequence of actions) to actions. The optimal action-state function must follow the *Bellman equation*:

$$Q^*(s, a) = E_{s' \sim \xi}[r + \gamma max_{a'} Q^*(s', a') | s, a] \tag{5.3}$$

Then, the main idea is trying to estimate the action-value function, using the recursiveness of the Bellman equation:

$$Q_{i+1}(s, a) = E[r + \gamma max_{a'} Q_i^*(s', a') | s, a] \tag{5.4}$$

This configuration reaches convergence, $Q_i \to Q^*$, when $i \to \infty$. One common way to avoid this is to use a function approximator to estimate the action-value function, in a way we can assume $Q(s, a, \theta) = Q(s, a)$. In our case, we parametrize it with a neural network, which in literature is usually refereed as the Q-network. This network can be trained by minimizing the sequence of loss functions $L_i(\theta_i)$ at each iteration i :

$$L_i(\theta_i) = E_{s,a\sim\rho}[(y_i - Q(s, a, \theta_i))^2] \tag{5.5}$$

where

$$y_i = E_{s' \sim \xi}[r + \gamma max_{a'} Q^*(s', a', \theta_{i-1}) | s, a)] \tag{5.6}$$

At training time, the gradients we need to compute with respect to the weights can be computed as:

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,\rho(\dot{}),s' \sim \xi}[(r + \gamma max_{a'} Q^*(s', a', \theta_{i-1}) | s, a) - Q(s, a, \theta_i))\nabla_{\theta_i} Q(s, a, \theta_i)] \tag{5.7}$$

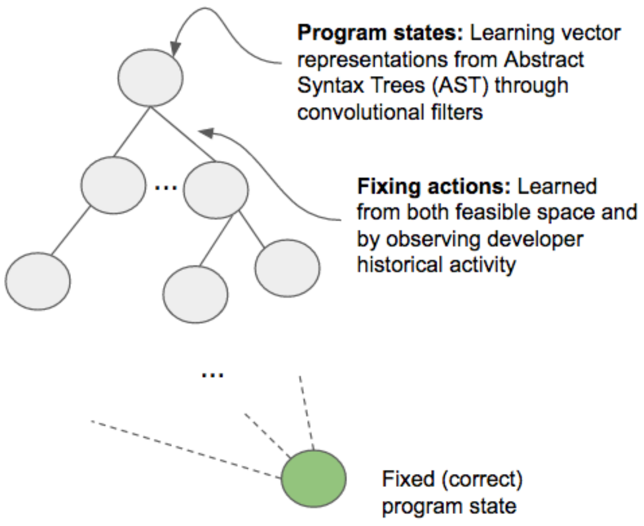

Program states: Learning vector representations from Abstract Syntax Trees (AST) through convolutional filters
Fixing actions: Learned from both feasible space and by observing developer historical activity
Fixed (correct) program state

Figure 5.1: Proposed approach. The agents iteratively generate candidate solutions in terms of source code modifications, which lead to further states. The goal of the agent is to reach an optimal program state.

State Representation

As we are dealing with consecutive source code versions form the program, we should not represent the each version just a as a sequence of atomic tokens, as it inherently has sparsity problems. Therefore, we need to find a way to obtain richer representations that could more expressive for the network.

State Representation through Embeddings : One initial and simple way is to, for a given token in the current version of the program, map it to a pre-trained continuous vector representation. Then, a final representation can be obtained by appending or concatenating the associated vectors. This method is easy to compute, but the program structure could not be retained efficiently. Therefore, depending on the change performed on the program, several states at different times could be represented by very similar vectors, when in reality there is considerable difference between them. We abbreviate this approach as *token-based state*. Another option is to take the abstract syntax tree associated to the version at time t, and from it learn feature representations using a graph embedding method, such as the ones we proposed in Chapter 2. From this, each node from the abstract syntax tree will have a continuous vector representation , based on the iterative generation of random walks starting at each node, which serve as sequences for a Skipgram-like model. Then, a representation for the full abstract syntax tree, and therefore for the program at a given state, can be computed by aggregating the vectors, for examples, through average of concatenation. This method requires more computational resources than the previous one, but it is able to capture the structure of the resulting version of the program. We abbreviate this approach as *node-based state*.

While *node-based state* representation can encode program structure, it lacks a description of the type of the node, which is associated to the token. Therefore, we try a third approach that consist of combining , for each token in the program version under study, its *token-based state* and *node-based state* representation.

State representation through Tree-based Convolutional Filters : In this case, we inspired on recent work in natural language processing to represent sentences through analyzing the parse tree and from them, build a method to obtain continuous representation for the whole abstract syntax tree of a program version. We begin by capturing information from each node, in the form of a vector representation. For each node p and its directed children nodes $c_1, \ldots, c_n$ we state the following methodology:

$$vec(p) \sim tanh(\sum l_{W_{code,i}} * vec(c_i) + b_code) \tag{5.8}$$

where $W_{code,i}$ is a weight matrix associated to node c_i, b_i is the bias and l_i is the coefficient of the weight. Then, the distance between $vec(p)$ and the trained vector can be defined as:

$$d = ||vec(p) - tanh(\sum l_{W_{code,i}} * vec(c_i) + b_code)|| \tag{5.9}$$

with this distance, we can set a training objective that consists of minimizing the distance between the instance and a set of negative examples (for simplicity could be any random token), which we denote as d_c.

$$min_{W_{code}, vec()} max 0, \Delta + d - d_c \tag{5.10}$$

The reward is a key element in any reinforcement learning task, as it provides the signal for the boosting the actual learning of the system. We selected two ways to encode such feedback, in order to explore how the system could learn better.

The first reward configuration follows a standard setting, where the value of the reward is given to the agent at the end of the path. In this case, the value is related to how far the last state, in the sense of the last program version generated is from an ideal or correct state.

The second reward configuration we follow make use of a suite of test cases we automatically generated for each problem. On each instance, based on the program difference observed, we first prioritize the test suite, following the approach proposed on Chapter 3. Then, the test suite is executed in order to find ant fault in the resulting program. The value of the reward depend on the number of test cases passed successfully.

5.1.4 Empirical Study

We designed an empirical study to test the feasibility of the proposed approach, comparing it against other relevant techniques on the data from Hour of Code.

Data

The richness of the data from Hour of Code resides in that, for each users that attempts to solve an assignment, a version of the current work is saved automatically on the system every time the user perform a change. Therefore, we are able to capture the sequences of steps that user conducted from the beginning until he reaches to a final solution.

The dataset comprises user activity from December 2013 to March 2014, conformed by more than 137 million partial solutions for two defined coding problems, P_A and P_B. P_A requires the users to string together a series of moves and turns, which is a problem of medium difficulty. P_B required users to efficiently make use of a if-else condition inside a loop, which we can consider a more challenging problem. Table 5.1 shows statistics about the users and submissions. A unique submission means that its has a unique abstract syntax tree.

In addition to the partial solution transitions, this dataset contains a goal standard for both P_A and P_B that consists of a manually labeled, for each partial state, which next state the user *should* follow, based on the expert knowledge of a group of experts.

Table 5.1: Statistics about the dataset from Hour of Code

Statistic	P_A	P_B
users	509.405	263.569
submissions	1.138.506	1.263.360
unique submissions	10.293	79.553

Alternative Methods

In order to test the performance of the proposed approach, we implemented a series of alternative methods present in the literature to generate viable comparison.

Poisson Path : Proposed by Piech et al [127], it assumes that a path from a partial solution s to a perfect solution that takes the smallest amount of time follows a Poisson process, whose rate parameter can be estimated based on the aggregated user data. Therefore, the Poisson Path for a partial solution s is defined as :

$$\gamma(s) = argmin_{p \in Z(s)} \sum_{x \in p} \frac{1}{\lambda_x} \tag{5.11}$$

where $Z(s)$ are all the paths to solution from s and λ_x is the frequency the solution x is seen in successful paths.

To compute this Poisson Path, it is necessary to generate a reduction to a Dijkstra shortest path algorithm, by generating a graph from all legal transition, where the edges have a weight $\frac{1}{\lambda_b}$ for any transition between solutions $a \to b$.

Independent Probable Path: Also proposed in [127]. In this case, the idea is to find the path to the solution state from a given partial solution s that would have been that most probable in on average for all the population of users. Therefore it is firstly assumed that the probability of reaching a partial solution s for an average user is proportional to the frequency that s appear in the complete set of all registered paths. Therefore, the Independent Probable Path can be computed as:

$$\gamma(s) = argmax_{p \in Z(s)} \prod_{x \in p} p(\psi_t = x) \tag{5.12}$$

where $Z(s)$ are the set of all paths from s to the solution state, λ_x is the frequency of users that submitted the solution x.

$$\gamma(s) = argmax_{p \in Z(s)} \sum_{x \in p} log(\frac{\lambda_x}{k}) \tag{5.13}$$

$$\gamma(s) = argmin_{p \in Z(s)} \sum_{x \in p} -log(\frac{\lambda_x}{k}) \tag{5.14}$$

where k is the number of available submissions. This configuration can also be transformed and reduced to a instance of Dijkstra short path problem.

Rivers Policy: Proposed in [134], this method takes a partial solution and computes a score of all potential *next* partial solutions, and then select the one with a highest score. For computing the score, the model capture a set of features, which in our case will associated to the set of all legal transitions between partial solutions. We follow the re implementation by Piech et al [127], where the policy function is expressed as:

$$\pi(x) = argmax_{n \in N(x)} \theta_0 \lambda_n + \theta_1 (1 - \gamma(n, g)) + \theta_2 v(n) \tag{5.15}$$

where $N(x)$ is the set of direct neighbors of x in the graph of legal moves, λ_n is the popularity of the partial solution, $\gamma(n, g)$ is the minimum abstract syntax tree edit distance between a neighbor n and correct solution g, and $v(n)$ is the unit test score of the neighbor. The values for the θ_i are found through hyper parameter search.

Results

We implemented the alternative methods, and then we run the experiments using the complete dataset from Hour of Code. The main results can be seen on Table 5.2 where w can see that our method outperforms the baselines, specially the variation the uses a convolutional filter to learn a state representation.

One additional aspect to consider is related to the execution time associated with each approach. In this case, the approach based on reinforcement learning has a clear overhead as it needs to obtain a representation of the state. We observed that our approach, while more effective in terms of accuracy and overall performance, takes up to 24% more time in the case of P_A and 25% more in the case of P_B. The differences in time with the proposed variations of state representations are not relevant. Just the method that uses a convolutional filter , on average, is 10 % slower. Therefore, we can see that there is indeed a trade off in terms of time vs accuracy.

Table 5.2: Accuracy reported on the Hour of Code dataset for problems P_A and P_B

Method	P_A	P_B
Possion Path	0.930	0.791
Independent Probable Path	0.912	0.805
Rivers Policy	0.632	0.669
Ours (token-based)	0.811	0.78
Ours (node-based)	0.958	0.812
Ours (node-token-based)	0.959	0.889
Ours (tree-conv-based)	0.971	0.93

5.1.5 Conclusion and Future Work

In this work, we have presented the first attempt to model program repair as reinforcement learning task. To do that we took inspiration of the current research conducted on MOOCs, specifically the methods related to automatically generating hints for students. In that sense, we combined a Q-learning agent, whose ultimate goal is to achieve a complete solution for a specific programming task, which means, traverse the space of possible *partial* solutions until reach the state the represents a fully compilable and functionally correct code. We made an explicit analogy between that search task and the problem of automatic repair, in the sense that

As for future work, we will explore the feasibility of the proposed model in a more realistic setting. The experiments presented in this work are limited as they come from programming assignments, whose goal is to serve a pedagogic material for leaning to program, and they cannot be considered a fully realistic piece of code.

Additionally, we consider necessary to advance in topic of the representation of the states. In a more realistic scenario, the set of possible states that can be reached could be too large for any standard computational method, therefore it should be necessary to generate methods that learn a more compressed or hierarchical feature representation of both the bug that need to be fixed, as well as the states on which the program enters as it is being repaired.

The use reinforcement learning represents a new alternative, which has the inherent advantage of not need a strict mapping between explanatory feature and dependent variable. But on the other hand, this apparent sense of freedom usually comes with cost, which is the large amount of time required for achieving convergence. Other recent alternatives, mainly based on recurrent models, do require a labeled dataset, which somehow restricts the amount generalization the can be obtain, but at least are able to come up with a feasible set of fix candidates in a reasonable amount of time. Therefore, the main vision for future work could be the design of a model that could combine both paradigms in an efficient way. From one side, the use of a supervised dataset that could serve a ground truth and from which a generative model could be trained in order to obtain fixing candidates that follow the distributional regularities from real data. And then, from the set of candidates, a reinforcement learning module that could generate the fixes, in the of learning effective policies of code change.

Chapter 6

Discussion

6.1 General Discussion

During all the course of this thesis work, our main theme has been the feature representation of the diverse elements that conform the software engineering process. The pattern we followed on each chapter was simple: propose a method or model which is capable to automatically generate feature representations for a given software artifact, and then, based on a specific task, estimate if these learned features are competitive with traditional metrics or handcrafted features.

Behind this approach relies the assumption that software engineering research, to some extend, can be transformed into a data problem. From a pure software engineering perspective, that assumption can be seen as reductionist, as software by itself keeps several aspects that cannot be modeled in a black-box fashion.

Therefore, we should not seek for a total replacement of the standard program analysis disciplines with a data driven approach. Indeed, our results suggest that while representation learning appears as a promising tool in the software engineering research space, it is far from perfect. Designing and deploying the methods proposed in this thesis required to overcome a series of technical and conceptual obstacles.

In this chapter we analyze in depth those concerns and evaluate then in terms of how they impact in the overall goal.

6.1.1 Hyperparameter Optimization

While representation learning approaches tend to pose the feature learning process from raw data as a seamlessly task, in reality for these methods to work require a set of parameter tuning steps.

As the revival of representation learning is relatively new, there is no underlying theory that can be used as backbone when we design the feature learning architecture and their and select its respective parameters. Indeed, this process is carried out using heuristic methods combined with researcher intuition and expert knowledge. Moreover, the decisions taken when designing and implementing these methods are conditioned to the relative success they obtain on a validation set. Therefore, currently, if we change our architecture from one to another configuration, the rationale and justification of that change is based purely on the empirical results and not on any theoretical background.

Additionally, as there is little invariance of these models with respect to the data, a given architecture that could be performing well on defined data set, could dramatically decrease its performance if the the data suffer a sudden shift in distributional terms. Of course, this could be an indicator of poor generalization power from the perspective of the model, but could also be the result of a poor parameter setting.

Searching for hyperparameters is usually carried out by grid search, which means to perform an exhaustive search over a defined search space of combinations of defined parameters, or, more recently, by means of Bayesian optimization, where the search is cast as a sequential strategy. We tested both methods, and while we were able to obtain parameter configurations that greatly improved the model, the time associated to the search was considerable. Therefore, there is trade off that needs to be considered.

Moreover, out take on this issue is more conceptual: While representation learning promise ways to learn representations from the data directly, avoiding hand crafted feature engineering, we are still forced to make rigid decisions the architectures that learn those features.

In that sense, we consider that problem has only moved, passing from handcraft features to handcraft architectures.

In concrete terms, we had to perform hyper parameter search across all the tasks involved in this thesis. This process was specially complex for the case of defect prediction, in which case initial experiments presented considerable instability. We consider this process to be unavoidable and it must be considered as part of the development and deployment pipeline associated to our approaches. While we were able to converge in all cases to a set of effective hyper parameters, we consider there is still room for improvement and work to be done.

6.1.2 Interpretability and Accountability of the Learned Representations

One of the main drawbacks that we envision for the application of representation learning in software engineering is the current difficulty that emerges when we try to understand the learned representations from the data.

To visualize this problem, let us set an standard scenario for standard defect prediction using hand crafted features: In the first step, the engineer select and compute a set of hand crafted features, which could come from any data source. This features are fed into a standard classifier, along with the class label associated to each instance. A standard classifier will then be able to map each instance to a label by associating a specific weight for each feature. After a successful training, we can easily explore the weight distribution over the set of features and conclude , based on their magnitude, which is the set that influence then most the predictive task, ie, which feature has more explanatory power. This process is inherently actionable, as the developer can make sense of the features he engineered and, based on that, take decisions to modify the software task under study. For example, let us imagine a defect prediction task, on which the most explanatory feature was the number of the modules a developer work simultaneously, a concept that is usually called *ownership*. If lower levels of ownership are associated to a higher probability of defects, then this information can be used by the managing team in order to optimize and re distribute the work load among the developer team.

Something different happens when we let the model learn the feature representation by itself. If we inspect them afterwards, the chance that we can associate them to a real characteristic of the software related task we are studying is low. Usually, we will be in the presence of a vector of real values which encodes the entity we are studying, but a priori, we are not able to tell anything about each component of this vector.

We had the chance to experience from first hand this, when we showed our initial results to real software engineers. We performed an informal interview, trying to capture their impressions in terms of the results obtained. While they praised the increase in performance and the replacement of tedious feature learning engineering, they criticized that lack of control and understanding on the resulting learned features.

There are ways to narrow the dimensionality of the feature representations in order to be visualized, such as PCA or T-SNE, which area able to provide a two dimensional representation of the vectors, allowing us to compare and relate them in terms of a distance in a standard coordinate system. But still no way to obtain reliable insights about the meaning of each component.

This topic is currently of high relevance in the research community, specially in the natural language processing field, where representations are expected to encode a semantic meaning, a goal that could enable natural language understanding.

6.1.3 Real Applicability in a Software Development Scenario

Software engineering research is characterized by a rigorous evaluation process. Research papers coming from that area usually show strong empirical studies, most of the times following a standard structure that explicitly requires to identify controlled factors, threats to validity. This approach is understandable, as the a big factors that drive the progress in the field is the reliability of the results.

Machine Learning, specially the connectionist paradigm, is inherently stochastic, which lead to solutions the are usually approximated and with an unavoidable margin of error. Moreover, recent evidence shows the representation learning methods can be easily fooled by adversarial attacks, which undermines their applicability in a real world context.

Therefore, it is important to weight the need for deterministic or stochastic solutions in software engineering research when using representation learning. For example, one could say these machine learning techniques are more suitable for analysis that can tolerate a higher error rate, or when analyzing systems that are also stochastic in nature. On the other hand, if we are working with critical systems, were the error tolerance should be minimized, machine learning could be used only as a supporting tool rather than that at the core of the decision making.

6.1.4 Transfer Learning Capabilities

In tasks such as defect prediction or source code summarization, we assume the existence of a reliable training set, whose size is large enough to provide a good level of generalization to the model.

In practice, that configuration is realistic in real world projects. For example, if we want to provide estimations of quality aspects on newly software projects, we will find that the amount of data available is not enough to provide results with a reasonable level of significance. This issue is widespread along machine learning, for example, one of the most studied issues comes from the field of recommender Systems, where the goal is to provide customers a product or service recommendation that could maximize the likelihood of purchase. In this setting, when there is not enough data to characterize the user, we are in the presence of a *cold start* problem, a topic which has been attacked over the years through a large number of approaches.

Similarly, if we want to study software development processes in a more realistic way, we should not assume the existence of a large training set for each software artifact under study. One alternative is to take an holistic approach, for example, the idea of including developers and projects as part of an ecosystem and from that generate a network framework that allow us to relate them. Unfortunately, those approaches, while comprehensive, are most of the time impractical to deploy, given their scale and complexity.

Another alternative is to rely on transfer leaning techniques, which learn to construct representation *bridges* between datasets in order to transfer the knowledge that a model learned on a large system to make prediction on a small, unknown one.

The main assumption behind this approach is that different datasets from the same field should share distributional similarities. Evidence from literature shows that machine learning models are able to achieve a reasonable level of transfer learning, given specific conditions.

We explored that setting across our experiments and we could not obtain conclusive results. In the first place, for defect prediction, we found that when implementing transfer learning between projects that are developed using the same programming language, there is a visible success. One can argue that this is expected as the vocabularies are limited and structural and syntactic aspects are shared, despite the that functional aspects could be different. When we tried to transfer from knowledge from projects that did not share the programming language, we encountered several technical issues, primary related to the asymmetry between the vocabularies from each side. Along with that, we reported that the increments in terms of accuracy were minor, and in some cases not statistically significant.

In the case of our experiments on test case prioritization, we observed considerable increments in all configurations. This could be explained by the fact the task in different and it is based on a matching given a specific query, rather than a generation process.

Chapter 7

Conclusion

7.1 General Conclusions

In this work, we conducted a comprehensive study on the role of the representation of the data when building analytic models to understand software engineering processes. Specifically, we focused on the quality aspects, as we consider an essential factor that condition the role of software as enabler of human endeavour.

Based on the empirical evidence collected along the several experiments, ranging from defect prediction to software repair, and considering both the technical and human aspects, our main concluding remarks can be summarized as follows:

1. **It is possible to unify features from different modalities:** One of the main issues we faced when trying to develop models to understand software processes in a more holistic was the large amount of data modalities present. Each of then with data from a specific nature and in a particular format. Literature showed attempts to combine then, but the concerns related to sparsity and rigidness represented a real threat. Along this thesis work, we empirically demonstrated the feasibility of finding efficient mapping functions to i) transfer from one modality to another , and ii) unify different modalities into a shared feature space. While the degree of success depends on the specific task, in general we believe the results suggests a positive contribution.

2. **Expressive power of learned representations:** Depending of the problem we faced , and the specific technique or model we use to learn representations from the data, we found that in general the obtained representation are expressive and flexible enough to improve task performance, when compared to standard metrics. This was specially evident in prediction of defects and also in bug localization tasks.

3. **Support rather than replacement:** While the contribution of representation Learning cannot be denied, we consider that this methods should be used along with standard techniques from program analysis and testing. In that sense, we do not argue that taking solely a machine learning approach could be a replacement for standard program

analysis methods. We showed evidence that the representation learning techniques we designed required several parameter tunning and should not be seen as fully automated tools.

In our opinion, the introduction of these techniques needs to be done in a sequential and incremental way, taking into consideration the particularities of the problem we need to attack, the nature of the available data and the requirements of the solution. This will serve as a way to evaluate the real contribution.

Of course, our dream is that in the future we could design software capable of full autonomy, in these of advancing artificial intelligence to naturally merge with software development, and ultimately removing the need of human intervention.

7.2 Future Work

As for future work, our goal is to naturally move to an unification scenario. In this work we have explored several ways to represent software engineering data, but a more challenging task is to combine them into a single pipeline that could allow to represent the development process in a holistic and more comprehensive way.

We structured this work from the point of view of a developer and his concerns about the quality aspects of the software he is currently building. Therefore, we can a natural transition between the actions taken against the defects:

prediction $\rightarrow$ detection $\rightarrow$ understanding $\rightarrow$ repair

The, we envision future approaches that are based on the combination of two or more of these actions:

- detection $\leftrightarrow$ understanding : In this case, the goal is to unify testing and code comprehension tasks to improve program understanding. For one side, testing provides a functional description of the behavior of the program, as it is encoded in the execution traces that can be obtained from the test cases. On the other hand, the neural models we proposed can be re used to obtain a characterization of the program in syntactic ways. Finally, both functional and syntactic representations can be combined.

- detection $\leftrightarrow$ repair: This combination has as main goal to explore an scenario where a system is required to have a higher level of autonomy, for example, space exploration , where human intervention is not feasible. In that sense, a program needs to be able to both detects its own bugs and also generate strategies for fixing them. While we could see both tasks as non overlapping and sequential, we consider relevant to explore ways on which both activities can share representations and also be trained jointly.

- prediction $\leftrightarrow$ detection: One of the main issues in when designing testing strategies is that the resulting test cases need to cover a reasonable portion of the code. Moreover, it is not only relevant that the only achieve high coverage, but also they should be efficient in that task, as testing is expensive. Therefore, we propose to unify defect prediction and test case generation: Instead of focusing on general coverage, we could direct the test case generation to cover parts of the code that are more likely to contain a defect, based on the estimation provided by a defect prediction model.

Besides those direct ideas, we also envision other extensions, which consider a more global approach:

- Recommender Systems for Human-Program Interaction: If we are able to map both user activity and software artifacts in a shared feature space, naturally we could be able to find the most suitable developers in that should accomplish a certain task. For example, we could be able to answer questions such as: Which developer should fix a specific bug?

- Test Oracle Generation: A test case is composed by two main elements: The test input, and the test oracle. The former allows us to explore program states with following a defined execution path, therefore it is critical to trigger any abnormal behavior that could exist. The oracle, on the other hand, is responsible for comparing the observed behavior with the expected value. In that sense, we think that the expresiveness of the learned representations could be very useful for assisting in the construction of test oracles. Concretely, if we are able to generate program systhesis through a neural architecture, we could, for a given input, estimate the corresponding output, which can be used as oracle.

Bibliography

[1] S. T. Acuna, N. Juristo, and A. M. Moreno. Emphasizing human capabilities in software development. *IEEE software*, 23(2):94–101, 2006.

[2] O. Alam, B. Adams, and A. E. Hassan. A study of the time dependence of code changes. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 21–30. IEEE, 2009.

[3] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

[4] M. Andersen-Gott, G. Ghinea, and B. Bygstad. Why do commercial companies contribute to open source software? *International Journal of Information Management*, 32(2):106–117, 2012.

[5] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 312–321. IEEE, 2013.

[6] A. Arcuri. On the automation of fixing software bugs. In *Companion of the 30th international conference on Software engineering*, pages 1003–1006. ACM, 2008.

[7] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 162–168. IEEE, 2008.

[8] R. S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[9] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 95–104, New York, NY, USA, 2010. ACM.

[10] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[11] L. B. Baker and J. Finkle. Sony playstation suffers massive data breach. *Reuters, April*, 26, 2011.

[12] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[13] M. Banks. Exomars delayed by two years. *Physics World*, 29(6):12, 2016.

[14] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987.

[15] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[16] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[17] N. Bertoldi, H. Barry, and J.-B. Fouet. Improved minimum error rate training in moses. *The Prague Bulletin of Mathematical Linguistics*, pages 1–11, 2009.

[18] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[19] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 60–69. IEEE Press, 2012.

[20] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

[21] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[22] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.

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

[24] S. Bourigault, S. Lamprier, and P. Gallinari. Representation learning for information diffusion through social networks: An embedded cascade model. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, WSDM '16, pages 573–582, New York, NY, USA, 2016. ACM.

[25] G. Brandl. Pygments: Python syntax highlighter. `http://pygments.org`, 2016.

[26] P. Brereton, D. Budgen, K. Bennnett, M. Munro, P. Layzell, L. MaCaulay, D. Griffiths, and C. Stannett. The future of software. *Communications of the ACM*, 42(12):78–84, 1999.

[27] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 30–32. ACM, 2008.

[28] A. Burns, M. E. Johnson, and P. Honeyman. A brief chronology of medical device security. *Communications of the ACM*, 59(10):66–72, 2016.

[29] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[30] R. P. Buse and T. Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering*, pages 987–996. IEEE Press, 2012.

[31] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 891–900, New York, NY, USA, 2015. ACM.

[32] S. Cao, W. Lu, and Q. Xu. Deep neural networks for learning graph representations, 2016.

[33] D. Clark, R. Feldt, S. Poulding, and S. Yoo. Information transformation: an underpinning theory for software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 599–602. IEEE, 2015.

[34] L. F. Cortés-Coy, M. L. Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *SCAM*, volume 14, pages 275–284, 2014.

[35] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[36] S. Das, A. Mukhopadhyay, and M. Anand. Stock market response to information security breach: A study using firm and attack characteristics. *Journal of Information Privacy and Security*, 8(4):27–55, 2012.

[37] C. R. de Souza, D. Redmiles, and P. Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*, pages 105–114. ACM, 2003.

[38] M. Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[39] R. Dyer, H. A. Nguyen, H. Rajan, and T. 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.

[40] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[41] G. Fischer. Desert island: software engineering—a human activity. *Automated Software Engineering*, 10(2):233–237, 2003.

[42] M. Foucault, J.-R. Falleri, and X. Blanc. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 39. ACM, 2014.

[43] M. Foucault, C. Teyton, D. Lo, X. Blanc, and J.-R. Falleri. On the usefulness of ownership metrics in open-source software projects. *Information and Software Technology*, 64:102–112, 2015.

[44] J. Freeman and J. S. Engel. Models of innovation: Startups and mature corporations. *California Management Review*, 50(1):94–119, 2007.

[45] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[46] S. Garfinkel. History's worst software bugs. *Wired News, Nov*, 2005.

[47] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[48] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[49] R. A. Ghosh. Economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ict) sector in the eu. 2007.

[50] Y. Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[51] I. Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008.

[52] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[53] M. Greiler, K. Herzig, and J. Czerwonka. Code ownership and software quality: a replication study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 2–12. IEEE Press, 2015.

[54] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. *arXiv preprint arXiv:1605.05273*, 2016.

[55] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. *arXiv preprint arXiv:1605.08535*, 2016.

[56] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.

[57] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. 2017.

[58] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[59] E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in github: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 352–355. ACM, 2014.

[60] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

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

[62] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):10, 2014.

[63] K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013.

[64] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering*, pages 523–534. ACM, 2016.

[65] J. D. Herbsleb and D. Moitra. Global software development. *IEEE software*, 18(2):16–20, 2001.

[66] K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 118–127. IEEE, 2013.

[67] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 60–69. IEEE, 2011.

[68] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[69] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[70] E. v. Hippel and G. v. Krogh. Open source software and the "private-collective" innovation model: Issues for organization science. *Organization science*, 14(2):209–223, 2003.

[71] Q. Hong, S. Kim, S. C. Cheung, and C. Bird. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 323–332. IEEE, 2011.

[72] X. Huo, M. Li, and Z.-H. Zhou. Learning unified features from natural and programming languages for locating buggy source code.

[73] S. Iyer, I. Konstas, A. Cheung, and L. 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, Berlin, Germany, August 2016. Association for Computational Linguistics.

[74] J.-M. Jazequel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[75] P. Johnson, M. Ekstedt, and I. Jacobson. Where's the theory for software engineering? *IEEE software*, 29(5):96–96, 2012.

[76] P. C. Jorgensen. *Software testing: a craftsman's approach.* CRC press, 2016.

[77] K. Kannan, J. Rees, and S. Sridhar. Market reactions to information security breach announcements: An empirical analysis. *International Journal of Electronic Commerce*, 12(1):69–91, 2007.

[78] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490. IEEE, 2011.

[79] E. Kit. *Software testing in the real world.* Addison-wesley, 1995.

[80] B. Kitchenham and S. L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE software*, 13(1):12–21, 1996.

[81] R. Kitchin and M. Dodge. *Code/space: Software and everyday life.* Mit Press, 2011.

[82] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[83] A. Lavie and A. Agarwal. Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, StatMT '07, pages 228–231, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

[84] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 14, pages 1188–1196, 2014.

[85] G. Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 339–346. IEEE, 1997.

[86] J. Lerner and J. Tirole. Some simple economics of open source. *The journal of industrial economics*, 50(2):197–234, 2002.

[87] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[88] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[89] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *Proceedings of ICLR'16*. arXiv, May 2016.

[90] H. Lieberman and C. Fry. Will software ever work? *Communications of the ACM*, 44(3):122–124, 2001.

[91] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 709–712. IEEE Press, 2015.

[92] T. R. Liyanagunawardena, A. A. Adams, and S. A. Williams. Moocs: A systematic study of the published literature 2008-2012. *The International Review of Research in Open and Distributed Learning*, 14(3):202–227, 2013.

[93] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[94] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM, 2016.

[95] Q. Luo, K. Moran, and D. Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 559–570. ACM, 2016.

[96] T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[97] Z. Ma and J. Zhao. Test case prioritization based on analysis of program structure. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 471–478. IEEE, 2008.

[98] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[99] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 33–41. ACM, 1993.

[100] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[101] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. 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.

[102] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[103] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[104] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[105] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.

[106] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[107] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[108] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*, 2014.

[109] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proc. AAAI*, pages 1287–1293. AAAI Press, 2016.

[110] C. Müller, G. Reina, and T. Ertl. In-situ visualisation of fractional code ownership over time. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, pages 13–20. ACM, 2015.

[111] S. C. Müller and T. Fritz. Using (bio) metrics to predict code quality online. In *Proceedings of the 38th International Conference on Software Engineering*, pages 452–463. ACM, 2016.

[112] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[113] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, 2010.

[114] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 448–451. ACM, 2014.

[115] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 508–519, New York, NY, USA, 2015. ACM.

[116] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[117] C. D. Nguyen, A. Marchetto, and P. Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 636–643. IEEE, 2011.

[118] Y. Ni, Q. K. Xu, F. Cao, Y. Mass, D. Sheinwald, H. J. Zhu, and S. S. Cao. Semantic documents relatedness using concept graph representation. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 635–644. ACM, 2016.

[119] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. *arXiv preprint arXiv:1605.05273*, 2016.

[120] F. J. Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 160–167, Sapporo, Japan, July 2003. Association for Computational Linguistics.

[121] M. Ortu, A. Murgia, G. Destefanis, P. Tourani, R. Tonelli, M. Marchesi, and B. Adams. The emotional side of software developers in jira. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 480–483. ACM, 2016.

[122] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 522–531, Piscataway, NJ, USA, 2013. IEEE Press.

[123] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[124] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[125] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[126] B. Perozzi, V. Kulkarni, and S. Skiena. Walklets: Multiscale graph embeddings for interpretable network classification. *arXiv preprint arXiv:1605.02115*, 2016.

[127] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 195–204. ACM, 2015.

[128] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[129] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40. ACM, 2016.

[130] J. B. Quinn, J. J. Baruch, and K. A. Zien. Software-based innovation. *Sloan Management Review*, 37(4):11, 1996.

[131] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[132] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[133] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[134] K. Rivers and K. R. Koedinger. Automating hint generation with solution space path construction. In *International Conference on Intelligent Tutoring Systems*, pages 329–339. Springer, 2014.

[135] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401. ACM, 2014.

[136] X. Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[137] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[138] M. J. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1499–1504. ACM, 2005.

[139] I. Rus and M. Lindvall. Knowledge management in software engineering. *IEEE software*, 19(3):26, 2002.

[140] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 268–279. IEEE, 2015.

[141] M. Sahlgren. The distributional hypothesis. *Italian Journal of Linguistics*, 20(1):33–54, 2008.

[142] K. Sangani. Sony security laid bare. *Engineering & Technology*, 6(8):74–77, 2011.

[143] R. Seyfert. Bugs, predations or manipulations? incompatible epistemic regimes of high-frequency trading. *Economy and Society*, 45(2):251–277, 2016.

[144] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell. Understanding watchers on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 336–339. ACM, 2014.

[145] J. Siegmund. Program comprehension: Past, present, and future. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 13–20. IEEE, 2016.

[146] H. Srikanth and L. Williams. On the economics of requirements-based test case prioritization. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–3. ACM, 2005.

[147] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, 21(2):126–137, 1995.

[148] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 311–320. IEEE, 2012.

[149] D. Stavrinoudis, M. Xenos, P. Peppas, and D. Christodoulakis. Early estimation of users' perception of software quality. *Software Quality Journal*, 13(2):155–175, 2005.

[150] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 359–364. ACM, 2010.

[151] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[152] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1067–1077, New York, NY, USA, 2015. ACM.

[153] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 2016.

[154] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.

[155] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1039–1050. ACM, 2016.

[156] P. Tonella, A. Susi, and F. Palma. Using interactive ga for requirements prioritization. In *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pages 57–66. IEEE, 2010.

[157] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social computing (SocialCom), 2013 international conference on*, pages 188–195. IEEE, 2013.

[158] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[159] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 8. IEEE Computer Society, 2007.

[160] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[161] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[162] M. White, M. Tufano, C. Vendome, and D. 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.

[163] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[164] A. Witze et al. Software error doomed japanese hitomi spacecraft. *Nature*, 533(7601):18–19, 2016.

[165] P. Yanardag and S. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015.

[166] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.

[167] X. Ye, R. Bunescu, and C. 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.

[168] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 404–415. ACM, 2016.

[169] W. Zaremba, I. Sutskever, and V. Oriol. Recurrent neural network regularization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[170] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 2016.

[171] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM, 2014.

[172] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing junit test cases in absence of coverage information. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 19–28. IEEE, 2009.

[173] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.

[174] H. Zuse. Software complexity. *NY, USA: Walter de Cruyter*, 1991.