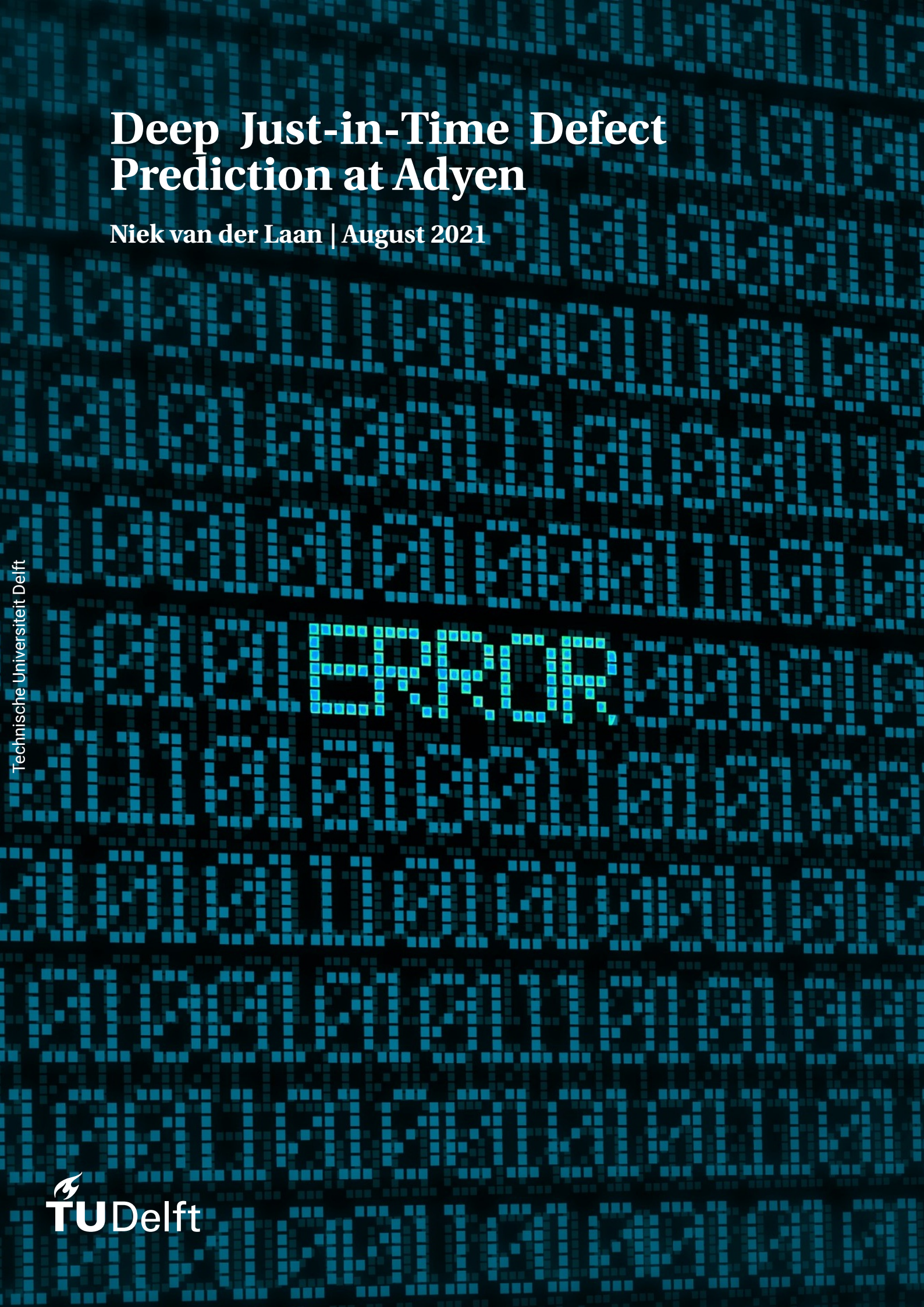
Deep Just-in-Time Defect Prediction at Adyen

Niek van der Laan | August 2021



Technische Universiteit Delft

TU Delft

Deep Just-in-Time Defect Prediction at Adyen

Niek van der Laan

to obtain the degree of Master of Science

at the *Delft University of Technology*,

to be defended publicly on Wednesday August 25, 2021 at 13:30.

Student number:	4296915	
Project duration:	December 1, 2020 – August 25, 2021	
Thesis committee:	Prof. dr. A. van Deursen,	TU Delft, chair
	Dr. M. F. Aniche,	TU Delft, supervisor
	D. Schipper MSc,	Adyen, supervisor
	Dr. ir. S. E. Verwer	TU Delft

An electronic version of this thesis is available at
http://repository.tudelft.nl/.

TU Delft adyen

Abstract

Finding defects in proposed changes is one of the biggest motivations and expected outcomes of code review, but does not result as often as expected in actually finding defects. Just-in-time (JIT) defect prediction focuses on predicting bug-introducing changes, which can help with efficient allocation of inspection time according to the defect-proneness of the changed software parts. Despite the promising results achieved by DeepJIT and CC2Vec, two deep learning-based JIT defect prediction models, industry-based JIT defect prediction studies have not opted yet to apply deep models.

In this work, the goal is to build and evaluate several JIT defect prediction models that can help Adyen developers spot defective changes during code review. To construct a new dataset with a large enough set of labels, we identify four sources of potential bug-fixing commits by analysing Adyen's way of working. We make several practical adaptations to DeepJIT and CC2Vec and compare their performances with three traditional metric-based models when making predictions at both commit-level and file-level.

Our results indicate that deep models are able to outperform the metric-based models across all three datasets. All models performed slightly worse when evaluated on Adyen data compared to an open-source setting, but both deep models still achieved respectable performances and significantly outperformed the metric-based models. When evaluated in a real-world setting on bugs manually collected by Adyen developers, DeepJIT performed consistent with earlier findings when evaluated on commit-level, but performances fall on file-level. Lastly, we find that although inclusion of each bug source generally does not lead to worse performance, whether it leads to better performance is dependent on both what type of model is used and at what granularity predictions are made.

Preface

When I started looking for a thesis project about a year ago, I had three things in mind: work at a nice company, on a challenging subject, with little room for subjectivity in the results.

"It is a bug, or it is not", is what I thought. Predicting bugs in source code seemed like a subject that definitely left little room for subjectivity. I was wrong. From the wide variety of types of bugs, to the location in the code where the bug was truly introduced, to even the answer to the simple question "is it a bug or is it not?", it is all completely subjective. On top of that, some of the work in the field of defect prediction was, ironically, filled with bugs.

I am however grateful I was wrong, as it is exactly this subjectivity that has challenged me in the last nine months. It is also this subject that has lead me to work at Adyen, that without a doubt has passed the first requirement with flying colors.

All of this could not have been possible without the help from my supervisor Maurício. Maurício, thank you for this opportunity, our weekly meetings, and your limitless enthusiasm about the research. Your messages on (Saturday) nights to suggest an interesting paper have always put a smile on my face. I wish you all the best in the future, perhaps our paths paths will cross again in the future.

I thank Daan, my other supervisor from Adyen. Despite your non-machine learning background, you have always been able to challenge my decisions, and have helped me to approach the problem from a wider angle. A great thanks also goes out to the rest of the DATT team, who have welcomed me with virtual open arms from day one, and the rest of Adyen who have had to endure my never ending begging for a GPU.

I thank my roommates, who have helped to make the best of the work-from-home situation while enduring my monologues about things they do not even fully understand. I like to think that some of it about trucks and Ferraris made sense at least, and may have even inspired some of you to dive into the world of deep learning learning yourselves.

Thanks as well to the two other members of my committee, Arie van Deursen and Sicco Verwer, for taking the time to read my report and attend the presentation.

Finally, I thank my parents for encouraging me to pursue my dreams, my brother and sister for their always joyful presence, and my girlfriend Daphne, for all her help, joy and loving support.

Niek van der Laan
Delft, August 2021

Contents

Figures

Tables

Chapter 1

Introduction

As software-based systems become an increasingly influential part of our day-to-day life, defects in such systems can substantially impact businesses and their customer's lives. The infamous Heartbleed bug, introduced into the widely used OpenSSL's source code through a flawed code change, affected billions of Internet users in 2014. Code review is a widespread practice used by software engineers to, amongst other reasons, timely catch such bugs before they reach production. In fact, finding defects in proposed changes is one of the, if not the biggest motivation and expected outcome of code review [2]. There is however a mismatch between this expectation and the actual outcome, as code review does not result as often in actually finding defects [2]. When software grows significantly in both size and complexity, finding defects and fixing them becomes increasingly difficult and costly. To mitigate this issue, efficient allocation of inspection time should be done according to the defect-proneness of the changed software parts. The field of study that attempts to model and predict such likelihood of defectiveness is called *defect prediction*.

A large part of the defect prediction research is focused on defect prediction models that identify defect-prone modules, such as methods, classes, components or files [3, 7, 30]. However, such approaches have some drawbacks: predictions are coarsely grained, meaning developers have to inspect large amounts of files. Additionally, predictions can come in late after a change was made, and as such will require some time from the original author to refresh their mind. Therefore, researchers have proposed *just-in-time* defect prediction techniques, focusing on predicting defects at change-level [16].

Just-in-time defect prediction has attracted an increasingly wide research interest, with machine learning techniques being the predominant approach. The most common way of working is to craft a set of features that represent characteristics of a code change, such as change size (e.g. number of lines added), change scope, (e.g. number of subsystems modified) [15, 24]. These features are then used as defect predictors by using them as an input to a traditional classifier (e.g. Random Forest) that predicts the probability of a code change to be defective.

While these metric-based methods have proven to be reasonably effective, the crafted metrics do not represent the actual functionality of the code changes made. Two different changes may therefore be similar in terms of metrics, but can still completely differ in terms of functionality. As a result, deep learning has received some interest in applying it to just-in-time defect prediction [10, 40]. Instead of exploiting features derived from properties of code changes, deep learning methods typ-

ically convert source code into vectors to learn its latent features directly. Vectors can preserve code information thoroughly, and may also include additional context such as the type of change that was performed. Deep learning therefore allows us to represent the semantic and syntactic structure of the code changes, where vector representations of similar code changes are close to each other.

Despite the promising results achieved by deep learning, industry-based defect prediction studies have not opted yet to apply these deep learning based models. This may be due to - apart from being more complex to implement - the relatively coarse granularity of predictions made in the proposed deep learning models. Industry-based defect prediction studies have instead opted for techniques that make predictions at finer granularities such as file- or line-level, in order to better aid reviewers in localizing potential defects [31, 34, 39].

To fill this gap, we identify the opportunity to evaluate the effectiveness in our industry setting of DeepJIT [10] and CC2Vec [11], two state-of-the-art deep just-in-time defect prediction models. We also address the coarse granularity of predictions made by deep learning models, by adapting them to predict defects at file-level within a commit. We evaluate this at Adyen, a large-scale company, and use two datasets based on large-scale open-source projects as baseline for comparison. We thereby explore new sources to label bug-introducing changes in addition the issue tracking system, to collect a large amount and qualitatively good set of labels for our new dataset.

The research questions we look to answer in our study are as follows:

- **RQ1:** How well do state-of-the-art just-in-time defect prediction models perform on a new dataset, gathered from an industry project?
- **RQ2:** How effective are the evaluated models at detecting real-world bug-introducing changes?
- **RQ3:** How important are the different bug sources for the effectiveness of the evaluated models?

The experimental results indicate that deep learning based models are able to outperform the metric-based models across all three datasets. When evaluated on Adyen data, all models performed slightly worse than in an open-source setting, but both deep models still achieved respectable performances and significantly outperformed the metric-based models. At file-level predictions, all models lost a large amount of performance compared to the observed performances at commit-level, but deep learning models significantly outperformed metric-based models. When evaluated in a real-life setting against manually collected bugs, DeepJIT [10] performed consistent with earlier findings when evaluated on commit-level, but performances fall on file-level. Lastly, by performing an ablation study on our bug label sources, we find that although inclusion of every source generally does not lead to worse performance, whether the inclusion of a source leads to information gain is dependent on both what type of model is used and at what granularity predictions are made. Therefore, when selecting sources to obtain labels, taking into account the granularity of predictions is important.

The main contributions of this work are:

- An empirical study to evaluate the performances of deep state-of-the-art defect predictions models against a novel, industry-based dataset.
- The identification and evaluation of three novel sources to collect bug-fixing changes, that can be exploited when traditional sources such as an Issue Tracking System are lacking.

The remainder of this work is structured as follows. Chapter 2 provides an explanation of the techniques that are used in this study, and outlines some of the work related to ours in the field of software defect prediction. Chapter 3 describes the steps performed to gather and process our novel dataset based on Adyen code, as well as the adaptions we make to the evaluated models in order for them to fit our use case, Chapter 4 presents the experiments performed and their respective outcomes. Chapter 5 then discusses the implications of our work for just-in-time defect prediction, both at Adyen and in the research field. Finally, Chapter 6 recaps the conclusions drawn from our study and presents some recommendations for future work.

Chapter 2

Related Work

The purpose of software defect prediction is to predict the likelihood of a defect being present within software. The benefit of defect prediction models would be to assist software developers with localizing bugs and to prioritize their testing efforts. The general approach to construct a defect prediction model begins with obtaining a dataset that labels files or commits as bug-introducing or not. Then, based on the labels and various features extracted for each sample in the dataset, a supervised machine learning classifier is trained in order to make predictions on unseen instances. In this chapter, some of the work closest to ours in the field of software defect prediction are outlined, and the techniques that are used in this study are explained in more detail.

2.1. Machine Learning for Defect Prediction

A large part of the defect prediction research is focused on defect prediction models that identify defect-prone modules. Such studies typically evaluate one or more machine learning techniques as defect predictors, among which the most common techniques include Random Forest, Naive Bayes and Logistic Regression. The models are trained through a manually crafted set of features that can function as defect predictors. The granularity of the crafted metrics and thereby also the predictions of the models varies greatly, with studies proposing approaches at method-level, class-level, component-level, file-level and process-level [3, 6, 7, 30]. When trained, these models can then be run every once in a while at moments when it is crucial to remove any remaining defects, for example before a release.

Although traditional defect prediction models can be useful in some contexts, they also have their drawbacks. The predictions are typically coarse grained, leaving it to developers to locate risky code snippets in potentially very large files. Predictions can also come in late after a change was made, and as such will require some time from the original author to refresh their mind. Therefore, researchers have proposed just-in-time defect prediction techniques, focusing on predicting defects at change-level.

One of the earliest works on change-level prediction was done by Kim et al. [16], who suggested to train classifiers on solely the changes made to a file, rather than the files themselves. Kamei et al. [15] were the first to introduce the notion of just-in-time defect prediction models, that predict potential defects in incoming commits. They perform an empirical study of change-level predictions on a variety of open source and commercial projects from multiple domains. To predict whether or not

a change introduces a future defect, they train a Logistic Regression model based on 14 features grouped into five dimensions, each of which had proven to perform well in traditional defect prediction. Additionally, they evaluate the effectiveness of their predictions when considering the effort required to review changes, for which they construct a customized effort-aware linear regression (EARL) model.

McIntosh and Kamei [20] note that despite the advantages of just-in-time defect prediction, like all prediction models, they assume that the properties of past events (i.e., bug-introducing changes) are similar to the properties of future ones. In their study, they show this to not be the case, as just-in-time models lose much of their discriminatory power if they are not retrained, and the importance of different feature dimensions evolves with a project.

Yang et al. [41] evaluate several machine learning classifiers for just-in-time defect prediction, where they conclude that a decision tree classifier performs best overall for this task. They then use the Random Forest model to create a two-layer ensemble learning (TLEL) model, which they show can outperform other state-of-the-art models, among which the Logistic Regression model of Kamei et al. [15].

The work up until now has focused on predicting defect-proneness of commits. However, the goal of our study is to help Adyen developers to look for defects within these commits. In fact, Bacchelli and Bird [2] find that finding defects is observed as the most difficult task for a reviewer, as it requires deeper understanding of the code. Making predictions at a finer granularity might aid the reviewers in this by steering them in the right direction. Fortunately, previous studies have also identified the need for finer grained solutions.

Pascarella et al. [22] note that defective commits may often be partially defective, i.e. such commits are composed of both buggy and clean changes to files. The goal of their work is to provide finer grained predictions for commits that are only partially defective, allowing for better prioritization during code review. They propose a fine-grained just-in-time defect prediction model which predicts which changed files in a commit are likely defective. To this end, the authors consider 24 basic features that represent a modified version of those previously proposed by [15] and [24]. Different classifiers were subsequently fed these metrics, where the Random Forest technique proved to perform the best.

Trautsch et al. [36] builds forth on the work of Pascarella et al. [22] on fine-grained just-in-time defect prediction. In their work, they observed that static analysis warnings were used in previous defect prediction studies, but not for just-in-time models. The authors therefore aim to largely follow the work of Pascarella et al. [22] and add static analysis warnings to that.

Another common method to introduce finer grained predictions, is through a two-phase approach. For example, Yan et al. [37] propose a two-phase framework where in the first phase a Logistic Regression model is used to identify potentially defective commits based on change-level metrics. In the second step, they attempt to localize the bug through an n-gram model trained on source code lines. The model aims to sort lines according to their entropy, under the assumption that the entropy of a line is correlated with the likelihood of it introducing a bug.

Similar to this, Pornprasit and Tantithamthavorn [23] leverage a Random For-

est model to predict potential bug-introducing commits. Predictions made are then normalized to account for the defect-density of a commit, which allows for consideration of inspection effort. Finally, defective lines are ranked by their level of riskiness, which is computed using LIME [26].

Yang et al. [42] note that most works follow a supervised approach, even though obtaining enough qualitative labelled samples can be an issue. To address this limitation, they propose an unsupervised model for effort-aware just-in-time defect prediction, For several change metrics originally proposed by Kamei et al. [15], they build an unsupervised model that ranks changes in descendant order according to the reciprocal of their corresponding raw metric values. In their results, they show that their unsupervised model could achieve higher recall compared to that of supervised models in a cross-project setting. However, their results have been shown not hold under a within-project setting in a follow-up study by Yan et al. [38], where unsupervised models did not perform statistically significantly better than state-of-art supervised models.

2.2. Deep Learning for Just-in-Time Defect Prediction

So far, there have only been very few studies that seek to apply deep learning to the task of just-in-time defect prediction.

In 2015, Yang et al. [40] were the first to investigate the performance of deep learning in just-in-time defect prediction. To this end, they propose *Deeper*, a model based on a Deep Belief Network [9]. The model initially automatically extracts a set of features from the basic change-based metrics proposed by Kamei et al. [15]. They then input this new set of features into a Logistic Regression model to predict potentially defective commits.

Hoang et al. [10] note that even though Deeper leverages a Deep Belief Network, the model does not leverage the true notion of deep learning as it still requires manually crafted features as input. They therefore propose an end-to-end trainable convolutional network that takes both the message and code changes as its input. To validate their approach, they adopt two datasets based on open-source projects (QT and OpenStack) originally collected by Kamei et al. [15] and include the corresponding code changes. They show that they are be able to outperform both Deeper [40] as well as the metric-based model by Kamei et al. [15] in predicting defective commits.

Hoang et al. [11] introduce CC2Vec, a deep learning-based approach to learn the distributed representation of commit. By learning the relationship between the message and the code changes of a commit, the semantics of code changes can be learned and represented through an embedding. These embeddings can be applied to a variety of software engineering tasks, but specifically to just-in-time defect prediction, they can be combined with models such as DeepJIT to improve the discriminatory power of the model. Their results show that adding CC2Vec to DeepJIT is able to improve its performance.

2.3. Just-in-Time Defect Prediction in Industry

Although scarce, some studies have preceded ours in applying just-in-time defect prediction in an industry setting. We highlight some of these studies in this section.

Shihab et al. [31] conduct a year-long study at BlackBerry to better understand risky changes. Using change-based metrics, that are largely similar to that of Kamei et al. [15], they train a Logistic Regression model to predict defective commits.

Tan et al. [34] perform just-in-time defect prediction in an online manner at Cisco. In their work, they evaluate several updateable classification algorithms that incrementally take advantage of new unseen data. This way, they appropriately deal with the time-sensitiveness of the dataset, and are able to incrementally improve the model as it runs in production.

In collaboration with Video games developer Ubisoft, Nayrolles and Hamou-Lhadj [21] developed CLEVER, which also relies on a two-phase approach to identify defective commits. In the second phase of their model, they perform a similarity analysis between the potentially defective commit and code snippets from a database of known bugs. By doing so, they are able to reduce the number of false positives generated by the model.

The most recent industry-based just-in-time defect prediction study was done by Yan et al. [39]. In their work, they evaluate several state-of-the-art models, supervised and unsupervised, when applied to a dataset based on 14 Alibaba projects. They identify CBS+ [12] as the best performing model, which is largely similar to that of Kamei et al. [15] but makes some improvements related to effort-aware sorting. They proceed by making predictions on recent changes, and ask developers at Alibaba to review a list of 33 changes that were predicted to be defective. They report that developers agreed with the model's prediction in 33% of the time.

2.4. Applying Deep Just-in-Time Defect Prediction in Industry

So far, we have learned that traditional defection prediction brings along several disadvantages. As a result, research in recent years have proposed methods that perform predictions at the change level, focusing on predicting defect-inducing changes. By using change properties that have been shown to characterize risky commits as features, models are able to achieve good performances, at different change-level granularities. However, such metric-based features do not represent the semantic and syntactic structure of the actual code changes. Two changes may therefore have similar metrics, but can very well be semantically very different, and thus may differ in their defectiveness. By leveraging deep learning, recent studies have come up with semantically sensitive defect prediction models that are able to outperform the metric based classifiers. Despite these promising results, we saw that defect prediction studies in industry have not opted yet to apply these deep learning based models. Moreover, the performances of deep learning have even been questioned by the authors of JITLine, who argue their simpler metric-based approach achieves better results [23]. We therefore identify the opportunity to evaluate the effectiveness of deep learning models in our industry setting.

In the remainder of this section, we elaborate on the three models we look to apply in our study: DeepJIT [10], CC2Vec [11] and JITLine [23].

2.4.1. DeepJIT

DeepJIT [10] is an end-to-end deep learning framework, and is regarded as the current state-of-the-art deep learning approach for just-in-time defect prediction. On a high level, the model takes the commit message and the code changes as an input, extracts features in the form of vector representations using two separate convolutional networks, and finally inputs the concatenation of these vectors into a fully-connected layer to generate the probability of the commit to be defective. To have a better understanding of what happens in each of these phases, each phase is discussed in this section in more detail. More specifically, we first discuss how commit messages are processed, then we discuss how code changes are processed and finally we discuss the final layer that combines both sources into an output. An overview of the DeepJIT model is given below in Figure 2.1.

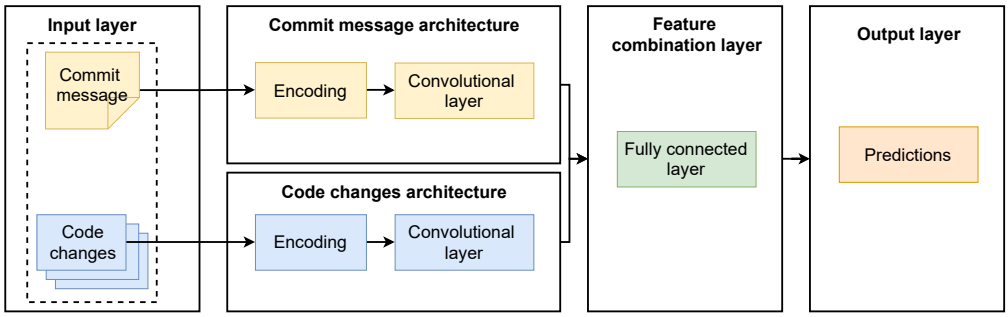


Figure 2.1: Overview of the DeepJIT model showing its architecture.

Commit message architecture Each commit message m is essentially a sequence of words, or *tokens* $[t_1, ...t_{|m|}]$. All unique tokens that occur in a commit message in the training set are added to a vocabulary V_m. Upon initialization of the network, a dense vector of size $d_m \in \mathbb{R}$ is randomly initiated for each token in the dictionary. A commit message can then be represented as an array of dense vectors that correspond to the tokens in the message, which can best thought of as its matrix representation $m \rightarrow M \in \mathbb{R}^{|m| \times d_m}$. The values in the dense vectors, to best represent the meaning of a token, are jointly learned with the rest of the network during the training process.

For the purpose of parallelization, all commit messages are padded or truncated to the same predetermined length m. To pad a message that consists of $m-n$ tokens, a special <*unk*> token is appended to the message n times.

The matrix representation of a commit message is then fed into a convolutional layer. In short, this layer performs a dot product between two matrices, where one matrix is the representation matrix of the commit message, and the other a set of learnable parameters otherwise known as a *kernel*. The kernel is spatially smaller in its height than a representation matrix but is equal in its width. During the forward pass, the kernel slides across the height of the representation matrix. This produces a one-dimensional representation known as an activation map that gives the response of the kernel at each spatial position of the input matrix.

Subsequently, the activation map is put into a pooling layer that summarizes

nearby values in the activation map, which decreases the required number of parameters and computation time. More specifically, a max pooling operation is applied, meaning nearby values are summarized according to the highest value among them.

In practice, multiple kernels of differing heights are applied to the matrix that represent the commit message, thus multiple activation maps are produced for one message. The results of the max pooling operation from each kernel are then used to form an embedding vector of the commit message. An overview of DeepJIT's commit message architecture is given below in Figure 2.2.

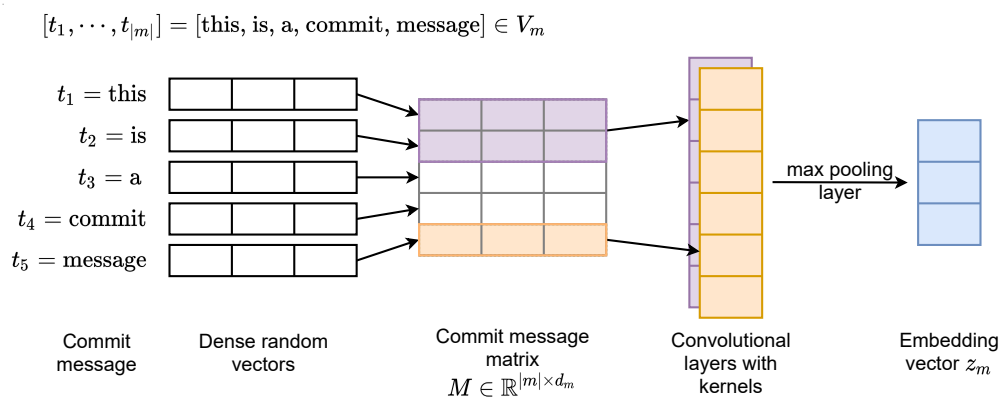


Figure 2.2: Overview of DeepJIT's commit message architecture.

Code changes architecture Code changes, similarly to messages, can also be viewed as a sequence of tokens, and therefore a large amount of steps performed in processing a commit message is also applicable to the processing of code changes. However, while the semantics of a message can be inferred from a bag of words, the code changes in a commit include a change in different files and different kinds of changes (removals or additions) for each file. Code changes are therefore processed by DeepJIT in slightly different ways compared to messages, in order to fully represent the structure of code changes. In this section, we explain these differences in more detail.

A commit that touches n files can be thought of as a collection of code changes C, which contains a list of file changes $[F_1, ...F_{|n|}]$. For every file change F_i, its salient features are computed in the same way. Therefore we first explain how a change F_i is processed, after which we explain how all results are combined.

A file change F_i is parsed into a sequence of deleted and added lines, and each line is parsed into a sequence of tokens. A special <*deleted*> token or an <*added*> token is inserted at the beginning of deleted or added lines respectively so that DeepJIT recognizes which change operation was performed. Given now that F_i is a collection of lines L, and each line is a collection of tokens T, we can in a manner similar to commit messages obtain the matrix representation of file change F_i as $F_i \rightarrow F_i \in \mathbb{R}^{|L| \times |T| \times d_c}$. Here d_c represents the size of each dense vector that is randomly initiated for each token in the dictionary V_c, similarly to the process as we observed for commit messages. The number of lines and the number of tokens in

each line in F_i are also padded or truncated for parallelization purposes using a special <*unk*> token. The matrix representation of a code line L_i is then comparable to that of a commit message, thus the same convolutional architecture as observed for commit messages is applied to each code line to extract its embedding vector. The embedding vectors obtained from each line are then stacked to form a representation of the file change F_i. The result is a 2D matrix, to which we again apply the convolutional layer and pooling layer, resulting in the embedding vector z_{F_i} that represents file change F_i.

This process is applied to each file change $F_i \in C$, and the resulting embedding vectors are then concatenated to build a new embedding vector that represents the collection of code changes C:

$$z_C = z_{F_1} \oplus ... \oplus z_{F_n} \tag{2.1}$$

Feature combination layer The inputs of this layer are the two embedding vectors z_m and z_C that were produced in the previous two steps. These vectors are first concatenated to create a unified representation for the commit:

$$z = z_m \oplus ... \oplus z_C \tag{2.2}$$

This vector z is then input into a fully connected layer, where the values of z are mapped to an output vector h of a predetermined size through a matrix multiplication with a trainable weight matrix. In order to help the network learn complex patterns, the output vector h of the fully connected layer is passed through a ReLU activation function, after which h is passed to the final output layer which computes a probability score for the passed commit to be buggy or clean.

2.4.2. CC2Vec

CC2Vec [11] is an approach to learn the distributed representation of a commit. Unlike DeepJIT, which ignores the information about the hierarchical structure of code commits, CC2Vec has been designed to automatically learn the hierarchical structure of code commits using a *Hierarchical Attention Network* (HAN) architecture. In short, CC2Vec attempts to learn the relationship between the actual code changes and the first line of a commit message, which is supposed to represent the semantic meaning of the code changes in the commit. As a result, CC2Vec is able to embed code changes in a commit according to their semantic meaning. This embedding vector can be exploited by other task-specific models, or in our case, by providing it to DeepJIT as an extra input. To have a better understanding of how CC2Vec works, we briefly discuss each phase in more detail. More specifically, we first discuss how commit messages are processed, then we discuss how code changes are processed and finally we discuss the final layer that combines both sources into an output. In Figure 2.3, we provide a high level overview of the framework of CC2Vec.

Input layer Similarly to DeepJIT, a commit that touches n files is represented as a list of file changes $[F_1, ...F_{|n|}]$. A file change F_i is then split into two lists that contain the corresponding added lines and deleted code lines in F_i respectively. Both lists are subsequently represented as three-dimensional matrices B_a and B_r respectively,

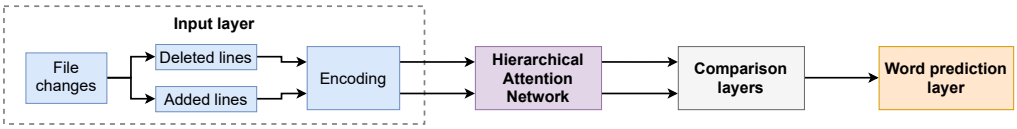


Figure 2.3: Overview of the CC2Vec model showing its architecture.

such that $B \rightarrow B \in \mathbb{R}^{H \times L \times W}$. Here, H denotes the number of groups of lines (*hunks*), L denotes the number of lines and W denotes the number of words per line. For the purpose of parallelization, all dimensions are padded or truncated to the same predetermined sizes.

Hierarchical Attention Network (HAN) The HAN is used to build vector representations from the matrices B_a and B_r. It consists of several parts, which are traversed in the following consecutive order: a word sequence encoder, a word-level attention layer, a line encoder, a line-level attention layer, a hunk sequence encoder, and a hunk attention layer.

In each sequence encoder, a bi-directional GRU is leveraged to summarize information from the context of the input sequence in both directions. For example, in the word sequence encoder, a line (i.e. a sequence of words $[w_1, ..., w_W]$ is passed to a forward GRU that reads the line from w_1 to w_W and a backward GRU that reads the line from w_W to w_1. Both passes result in a hidden state vector h, from which the annotation of each processed word can be obtained as follows:

$$\overrightarrow{h_k} = \overrightarrow{GRU(w_k)}, k \in [1, W] \tag{2.3}$$

$$\overleftarrow{h_k} = \overleftarrow{GRU(w_k)}, k \in [W, 1] \tag{2.4}$$

The vector representation of a word w_k is obtained by concatenating $\overrightarrow{h_k}$ and $\overleftarrow{h_k}$:

$$h_k = \overrightarrow{h_k} \oplus \overleftarrow{h_k} \tag{2.5}$$

In a similar manner, the line sequence encoder and the hunk sequence encoder obtain the vector representations of respectively each line and each hunk.

Each sequence encoder is followed by a corresponding attention layer, which aims to identify the most important elements in the input sequence. For example, in the word-level attention layer, the annotation of a word as represented in Equation 2.5 is passed to a fully connected layer with ReLU to obtain its hidden representation u_k. An importance weight α_k is subsequently obtained for each word through a word context vector that is randomly initialized and trained during the training process. Finally, a line can then be represented as a weighted sum of the embedding vectors of the contained words based on their importance. In a similar manner, the line-level attention layer and the hunk-level attention layer obtain the representations weighted by importance of respectively each hunk and all hunks. The final result is a set of two embedding vectors e_a and e_r that correspond to respectively the added and removed code, which are both obtained through the weighted sum of all hunk embeddings:

$$e = \sum_i \alpha_i h_i \tag{2.6}$$

Comparison layers The embedding vectors of the removed code and added code are compared through multiple comparison functions to explore their relationship. Eventually, all the embedding vectors associated with all the changed files under one commit are concatenated to construct a new embedding vector e_p representing the code change in a given patch (or commit).

Word prediction layer In order to determine the effectiveness of the embedding vector e_p of a given commit, e_p is used to predict the likelihood of each word within the vocabulary to occur in the corresponding commit message. This is done by passing the embedding vector to a fully connected layer, followed by a sigmoid function to get the probability distribution over all words. The objective function then attempts to minimize the difference between the probability scores and the actual occurrence of each word.

Combining with DeepJIT To use CC2Vec with DeepJIT, for each commit, the embedding vector e_p of the code changes extracted by CC2Vec is concatenated with the two embedding vectors extracted by DeepJIT from the commit message and codes change to form a new embedding vector. This new embedding vector is provided as input into the fully connected layer of DeepJIT to compute the likelihood that the given commit is buggy. An overview of the combined models is given below in Figure 2.4.

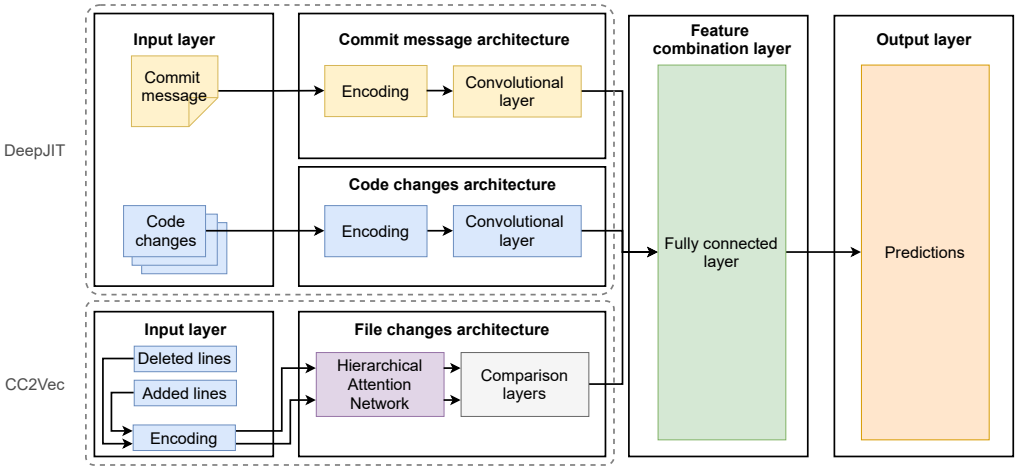


Figure 2.4: Overview of the combined model of DeepJIT and CC2Vec.

2.4.3. JITLine

JITLine [23] is a machine learning based just-in-time defect prediction approach that can both predict defect-introducing commits and identify defective lines that are associated with that commit. The goal of their approach is to make predictions based

on code changes, while also reducing the amount of training typically needed by more complex deep-learning approaches. Therefore, tokens of code changes are extracted as features, representing each change line as a bag-of-tokens.

To account for the imbalance of the datasets, JITLine makes use of the commonly used SMOTE technique for oversampling of defective samples in the training set. However, to ensure the best performance of the SMOTE algorithm, JITLine uses a Differential Evolution algorithm to determine the best value of SMOTE's k nearest neighbours hyperparameter. The fitness function of the differential evolution algorithm is to maximize the AUC obtained through using SMOTE with a candidate k.

After obtaining k and oversampling the training data with SMOTE, the commit-level metrics adopted from [20] are used to build a commit-level just-in-time defect prediction model using Random Forest as their classification technique. Predictions made are then normalized to account for the defect-density of a commit, which allows for consideration of inspection effort. Finally, defective lines are ranked by their level of riskiness, which is computed using LIME [26].

Chapter 3

Building Defect Prediction Models for Adyen

The goal of our study is to **build and evaluate several defect prediction models that can help Adyen developers spot defective changes during code review**. Such models should therefore predict the defectiveness of an incoming change based on inputs drawn from the change under inspection.

To this end, we leverage state-of-the-art just-in-time defect prediction models to classify changes made to Java files as 'defective' or 'clean' during code review. We do so by training and evaluating the models against a dataset that we collect based on the Adyen repository. To ensure a large and varied enough set of labelled samples, we identify and exploit several sources of information to label bug-introducing changes.

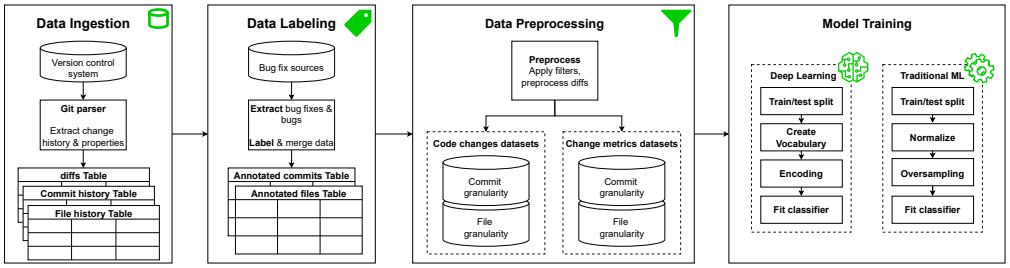


Figure 3.1: Overview of our approach towards building just-in-time defect prediction models for Adyen

This chapter consists of three sections. First, we describe the data collection steps performed to gather our new labelled dataset based on the Adyen repository. Secondly, we discuss the datasets and their characteristics that are used in our experiments. Finally, we describe the models we leverage in our study, and the adaptions we make in order for them to fit our use case. Figure 3.1 provides an overview of our approach.

3.1. Data Collection

Our data collection consists of three parts. We first ingest all raw change data and their historic properties. We then label all bug-introducing changes that are suspected to have introduced a bug. Finally, we apply a set of preprocessing steps to our data to mitigate noise and false labels in our dataset.

3.1.1. Data Ingestion

For our data ingestion, we take inspiration from the data collection tool proposed by Rosen et al. [28], that uses an extensive *git log* command to collect code metrics and historic properties for every commit. We extend their approach by also collecting the added and removed lines for each change, and collect all data also for file-level changes separately. Then, for every change, we compute a set of change properties that can represent the likelihood of introducing bugs. We calculate each metric by chronologically traversing the history of a project. We note that incorrect handling of file renames lead to large discrepancies in the calculated historic properties, thus we implement a correct handling of file renames to ensure history properties are transferred properly. Furthermore, to align with previous work, we ignore whitespace- and comment-only changes, as well as large refactorings [15, 20] in the process.

On commit-level, we extract 14 change-level features as proposed and reused by many prior studies [15, 20, 36]. We choose to use these metrics since they have proven to perform well in previous defect prediction research, capturing different properties at change-level that might explain the bug-proneness of a change. These properties of code changes measure the dispersion across the codebase (Diffusion), the change volume (Size), the modified areas of the codebase (History), and the experience of the author and involved developers (Experience). Some previous studies have also included review properties originally proposed by McIntosh and Kamei [20], representing characteristics such as the number of reviewers and the discussion length. These metrics however can only be calculated after a review has been performed, whereas the aim of this work is to make predictions before the review is conducted. Therefore, we choose to not include such review metrics in our work. Moreover, Krutauz et al. [17] find that review metrics are neither necessary nor sufficient to create a good defect prediction model, thus our decision should not be of big impact to our results.

At file-level, we collect augmented versions of the features calculated at commit-level as proposed by Pascarella et al. [22]. Some of the metrics are directly translatable, such as size properties, while some metrics that are specific to the scope of a commit are adapted to better capture the properties of the file change.

Dimension	Feature	Description
Diffusion	NS	Number of subsystems touched by the current change
	ND	Number of directories touched by the current change
	NF	Number of files touched by the current change
	ENT	Entropy across touched files
Size	LA	Lines of code added by the current change
	LD	Lines of code deleted by the current change
	LT	Lines of code in all changed files before the current change
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and current change
	NUC	Number of unique changes to the files
Experience	EXP	Developers experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

Table 3.1: Commit-level change metrics

Dimension	Feature	Description
Diffusion	ENT	Entropy of changes to the file up to the considered commit
	SCTR	Number of different directories touched by the developer in commits where the file has been modified
Size	LA	Lines of code added to the file by the current change
	LD	Lines of code deleted from the file by the current change
	LT	Lines of code in the file before the current change
History	NDEV	Number of distinct developers made changes to the file up until the current change
	AGE	Time interval between the last and current change to the file
	NUC	Number of times the file was the only file involved in a change up until the current change
Experience	EXP	Developers experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem
	OWN	Boolean value indicating whether the commit is done by the owner of the file

Table 3.2: File-level change metrics

We refer to Table 3.1 and Table 3.2 for an overview of metrics used at respectively commit-level and file-level.

3.1.2. Labelling

After collecting our raw dataset of changes, we look to annotate the changes that likely introduced a bug. Previous datasets that have been collected in order to be used for defect prediction generally rely on the SZZ algorithm originally proposed by Sliwerski et al. [32]. The SZZ algorithm was developed as an approach to identify bug-introducing commits in a software repository, and was later given its name after the initials of the three authors.

There are two main stages in the SZZ algorithm: identifying defect fixing commits, followed by identifying defect-introducing commits. In both stages, the SZZ algorithm makes extensive use of an internal *Issue Tracking System* (ITS) to track information about bugs and when they were fixed. Because a well-maintained ITS is not a given, previous studies have relied on a more ad-hoc SZZ algorithm [22, 36] that only scans the commit messages for keywords such as "fix" and "bug", which removes the restriction of needing an Issue Tracking System. However, the rate of mislabelling clean commits as buggy may drastically increase, which can have a significant impact on the performance of the model [5, 36].

At Adyen, we observe an unusually low amount of issues related to bugs. Discussions with developers internally revealed that generally a more ad-hoc style of fixing bugs is employed. For example, when a bug is noticed during a code review, or when a bug is noticed in production code where speed is of higher importance than applying best practices. Relying on the ad-hoc SZZ algorithm is not a good alternative for our study, as due to work practices at Adyen, commits that do not belong to an issue start their message with prefix "FIX". The rate of mislabelling

clean commits as buggy due to the occurrence of keyword "fix" would therefore be too big, while leaving it out as keyword will likely result in a too incomplete label set. To overcome this, we need to diversify our sources for defective samples to gain a large and accurate enough dataset of labeled defective samples, without relying on error prone keyword matching. We first discuss the identified sources for bug fixes, and subsequently discuss our approach to obtain the corresponding bug-introducing changes. An overview of our approach is shown in Figure 3.2

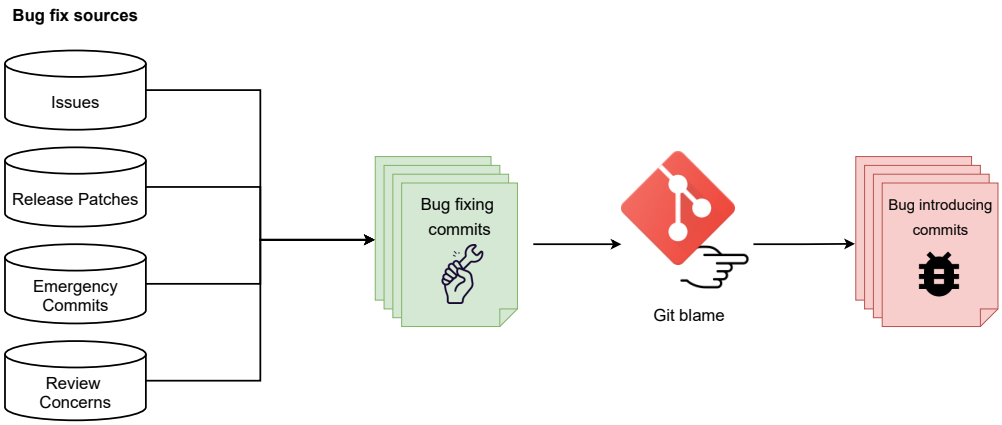


Figure 3.2: Overview of our approach to label bug-introducing changes.

1. Bug-fixing change identification
We identify four different potential sources to collect bug-fixing commits from which we can deduct the bug-inducing commits: issues, release patches, emergency commits and review concerns. In order to collect bug fixes from each of the sources, we extract data from the VCS (git), ITS (YouTrack) and the code review platform (Upsource) that Adyen uses internally. Below, we describe each of the sources individually and explain how we extract bug labels from them.

- **Issues.** A common methodology to address a bug is to create an issue in the Issue Tracking System, where the behavior of the bug is described, and labelled with type "Bug". When the developer makes a change that fixes the bug, the unique identifier of the issue (IssueID) can be added to the commit message to signal that the bug was fixed. We extract all issues in the Issue Tracking System. The issues are then filtered by type “Bug” and a resolved state, either “Fixed” or “Verified”. For each issue, we then store its IssueID and the timestamp when the bug was reported. We can then collect any bug-fixing commit by collecting commits that mention any of the collected IssueIDs.

- **Release Patches.** Commits made as patches to the release branch during beta testing or live. The priority of the commit having to go through quickly, indicates the commit is likely to be some sort of fix. We can therefore make a general assumption that these commits are bug fixes. For each release branch, we collect all commits that follow the initial commit of the release branch. From the commit message, we extract the commit hash of the corresponding com-

mit that was cherry picked from the master branch. We then extract the commit by its hash from the master branch and label it as bug fixing.

- **Emergency commits.** Identifiable by a reserved prefix in the commit message, emergency commits are very similar in their purpose to the previously discussed patch commits. However, the prefix is reserved for emergency fixes that skip a part of the normal testing procedure. Because of this, a commit is only eligible for this procedure if an important fix is required. Emergency commits are rare, but provide us with near certainty potential bug-fixing commits. We traverse all commits in the git history to collect commits that start with the reserved prefix. All these commits are labelled as bug fixing commits.

- **Review Concerns.** Every commit made to the Adyen repository will end up in a code review, generated by Upsource. Reviewers are automatically assigned to a review. A reviewer can give feedback, and choose to "accept changes" or "raise concern". A concern can be raised when a reviewer finds severe or critical problems, i.e. it is too risky that the code goes live. Therefore, we can make the assumption that a bug fix is required. To address any raised concerns, the author of the code under review tags the review's unique identifier (ReviewID) in follow-up commits, which we assume to be a bug-fixing commit. We traverse all commits in the git history to collect a set of ReviewIDs mentioned in commits. For each mentioned ReviewID, we check if at least one concern was raised in the review. If that is the case, we label all commits following the concern as bug fixing.

2. Bug-introducing change identification
After determining all defect-fixing changes, the SZZ algorithm leverages the *git blame* command to identify the lines that were changed by defect fixing changes. Each line of code changed is tracked back to the previous commit that modified the same line. Every commit that previously made a change is a potential candidate to be blamed for introducing a bug.

To identify bug-introducing changes, we make use of the implementation of the SZZ algorithm by Pydriller [33]. When we provide it the bug-fixing commits that we have already identified, we are able to retrieve the bug-introducing commits preceding these bug fixes. Commits with only whitespace and/or comment changes are automatically skipped in order to reduce noise. A file change is considered to be defective if at least one line in the change for that file is defective. A commit is then counted as defective if at least one file contained in the commit is defective.

In total, we collect 5615 bug fixes, and leverage those to annotate a total of 7532 bugs. Table 3.3 provides an overview of the number of bugs and bug fixes found per source.

3.1.3. Data Processing

After extracting data from the VCS and labelling all bug-introducing changes, we apply a set of code sanitation steps and filters to mitigate noise and false positives in our labels.

Source	# fixes	# bugs
Patches	4291	5869
Reviews	989	1528
Issues	271	472
Emergency	64	87
Total	**5615**	**7532**

Table 3.3: Number of bugs and bug fixes found per source. Note that some commits are identified as bug-introducing through bug fixes from different sources, thus the cumulative number of buggy commits is higher than the total number of (unique) buggy commits.

Code Sanitation For our code sanitation, we largely follow code sanitation steps from previous work [10] to align with our work. First, each change file is parsed into an array of added and deleted lines, with each line prepended with an <*added*> or <*deleted*> token to indicate the modification type. Each line is then parsed into a sequence of tokens, with each token being separated by whitespace. As suggested by Rahman et al. [25] and later also adopted by the authors of JITLine [23], non-alphanumeric tokens that are often part of the language specific syntax are replaced by whitespace to ensure that analyzed tokens are not unnecessarily repetitive.

In theory, the number of tokens that can occur in source code is limitless. New function and variable names are introduced on a continuous basis, leading to an ever increasing vocabulary size as the project progresses. An unlimited vocabulary in turn may lead to a dimensionality problem, and poorer generalization to unseen data [13]. To combat this, we follow the approach by the authors of DeepJIT and JITLine and replace numeric and string literals with <*num*> and <*str*> respectively. We also remove rare code tokens, i.e. tokens that occur less than three times from our vocabulary to limit our vocabulary size. Any token that is not present in the dictionary is replaced with a special <*unk*> token.

Filtering We subsequently apply a set of filters based on characteristics of the data samples. We largely follow the steps taken by previous work [10, 15, 20] .

- **Non-Java Changes.** Since we look to train models based on code changes, our model is specific to the programming language used in changed files. Moreover, what programming language used is likely to impact the change metrics. Because Java is the main programming language used at the studied system, we are interested only in changes made to Java files. We therefore filter out all changes to non-Java files in a commit. Previous work makes no mention of applying such filter, but discussions with McIntosh and Kamei [20] and Pornprasit and Tantithamthavorn [23] reveal such filter is indeed applied in their work.

- **Sanitized Changes.** We also filter out changes that after applying the sanitation described above do not contain any other changes, i.e. changes that only update comments, whitespaces, and/or perform renames.

- **Refactoring Changes.** We filter out extremely large commits, as these are likely noise caused by general refactorings or routine maintenance. We adopt boundaries that define extremely large commits as changes that include at least 10000 lines or at least 100 files.

- **Recent Date Period.** Following previous work [20], we also consider a limitation of SZZ, where we are dependent on future bug fixes to identify bug-inducing commits. As a result, bug-inducing commits made in the most recent period in our dataset may not be labeled yet as such, as the bug is yet to be fixed. More specifically, we remove recent data according to the median timedelta observed between a bug's occurrence and its fix. We determine the median value to be equal to 14 days, meaning all changes within 14 days of our data collection time are removed.

3.2. Datasets

To benchmark the performances achieved on Adyen data, we recollect two other labelled datasets based on open-source projects: QT and OpenStack. QT, developed by the Qt Company, is a cross-platform application framework and allows contributions from individual developers and organizations. On the other hand, OpenStack is an open-source software platform for cloud computing and is deployed as an infrastructure-as-a-service which allows customers to access its resources. Both datasets were originally collected by McIntosh and Kamei [20] to evaluate their proposed model based on process metrics for just-in-time defect prediction. The authors of DeepJIT [10] reused and adapted these datasets to include information on code changes, to be used for their deep-learning based defect prediction model. The authors of JITLine [23] further adapted the datasets by extracting code token features, and collected ground-truth labels at line-level in order to evaluate their predictions of defective lines. Because all three studies that we focus on in our work (i.e. CC2Vec, DeepJIT and JITLine) benchmarked their results on these two datasets, we choose to do so as well in order to guarantee the best comparability with our study. We recollect both datasets by following all data ingestion and processings steps outlined in this chapter, to ensure comparability of the obtained results between all datasets.

In the end, we obtain six different datasets, based on three different projects with labels at two different granularities. Table 3.4 presents the detailed dataset statistics.

Granularity	Project	# Changes	% Defective	Language
Commit-level	QT	24140	7.3	C++
	OpenStack	13151	12.3	Python
	Adyen	80720	9.3	Java
File-level	QT	71843	3.2	C++
	OpenStack	46354	5.1	Python
	Adyen	282634	4.2	Java

Table 3.4: Statistics for each of the collected datasets.

3.3. Models

We evaluate the performance of three different state-of-the-art approaches: DeepJIT [10], CC2Vec [11], JITLine [23]. All three models follow a supervised approach, thus inferring a function from our labeled training data to predict the probability of a change being defective. We specifically choose to evaluate the deep learning based DeepJIT and CC2Vec models, as they are regarded as the current state-of-the-art deep models and have shown to outperform the metric-based classifiers on open-source datasets. We incorporate JITLine, a metric-based classifier, as the authors have in turn shown to outperform both DeepJIT and CC2Vec with their approach [23].

By leveraging deep learning, recent studies have come up with semantically sensitive defect prediction models that are able to outperform the metric based classifiers. Despite these promising results, we saw that defect prediction studies in industry have not opted yet to apply these deep learning based models. We therefore identify the opportunity to evaluate the effectiveness of deep learning models in our industry setting.

3.3.1. DeepJIT

DeepJIT [10] is an end-to-end deep learning framework for just-in-time defect prediction. On a high level, the model takes the commit message and the code changes as an input, extracts features in the form of vector representations using two separate convolutional networks, and finally inputs the concatenation of these vectors into a fully-connected layer to generate the probability of the commit to be defective. More details on each of these steps are explained in Section 2.4.1.

In our implementation, we make some adaptations to the original implementation of the model provided by the authors of DeepJIT.

- **Generalization to multiple file changes.** The original implementation expects each commit to contain only one file code change. As commits in our dataset consist of multiple file changes, we extend their implementation to ensure the convolutional layer and pooling layer are applied to each file to build its embedding vector. These embedding vectors are then concatenated before being fed to the fully connected layer. This implementation follows the approach as described in their work, but a discussion with the authors revealed the authors had only experimented with commits that contain only one file.

- **Adaption of the loss function to deal with class imbalance.** The authors of DeepJIT propose a custom weighted loss function that imposes a higher cost on misclassification of the minority class (i.e., buggy commits). However, this loss is left out in their actual implementation, where only a *Binary Cross-Entropy Loss* is used. To deal with the imbalance, the authors randomly sample mini batches from the training data in a balanced manner, guaranteeing the same amount of positive and negative samples in each mini batch. As a result, the minority class is oversampled while the majority class is undersampled. Not only does this method lead to sampling with replacement, it likely also leads to more overfitting on the buggy samples as multiple copies of the same instance are trained on in one epoch.

In our approach, we also choose to use a Binary Cross-Entropy Loss, but add a manual rescaling weight p_c to the loss of positive examples. By tuning the value of this weight, it is possible to trade off recall and precision. In our binary classification task, the loss can be described as:

$$H_p(q) = \frac{1}{N}\left[\sum_{i=1}^{N} p_c y_i \cdot \log(p(y_i)) + (1-y_i)\cdot \log(1-p(y_i))\right] \tag{3.1}$$

To balance the performance of our model between precision and recall, the positive weights are calculated according to the distribution of the two classes in the training data. Furthermore, to include regularization and thereby limit overfitting, we shift from the proposed Adam optimizer to using the widely used AdamW optimizer originally proposed by Loshchilov and Hutter [19]. The AdamW optimizer allows us to include a weight decay component that is equivalent to adding an L2 regularization component to our loss function.

- **Early stopping with validation loss to prevent overfitting.** To prevent the model from overfitting, we extend it by introducing early stopping. While the authors of DeepJIT mention applying early stopping, this is likely done after training based on the training loss of each epoch. This however does not prevent overfitting, as the decrease of training loss typically does not stop (due to overfitting). We therefore split the training set into a new training set and a validation set at a ratio of 80:20. When the validation loss does not improve for five epochs, we stop training.

- **Dynamic determination of padding values according to the underlying data.** To ensure samples are equally sized, all samples are padded to the same sizes. This happens at line-level (code line length), file-level (LOC added/removed) and commit-level (number of files per commit, message length). When determining padding values, training time is the most important factor to consider: too low padding leads to a lot of truncation, thus information loss, while too much padding leads to unnecessarily long training times. We can estimate a reasonable range of padding sizes to test by inspecting these characteristics of our training datasets. We refer to Appendix A for details on this.

3.3.2. CC2Vec

CC2Vec [11] is an approach to learn the distributed representation of commit, which effectively means the model learns to embed a code change into a vector space where semantically similar changes are close to each other. To optimize the vector representations of code changes in commits, CC2Vec predicts the words in the first line of the commit message using the actual code changes as its input. Unlike DeepJIT, which relies on its CNN's to automatically extract information about the structure of the removed code or added code, CC2Vec leverages an attention mechanism to model this structural information. To apply CC2Vec to the task of defect prediction, the DeepJIT model is augmented to make use of the information provided by CC2Vec. Specifically, the distributed representation of a commit is concatenated with the vector representations produced by the two convolutional layers of DeepJIT,

before the fully connected layers.
In our implementation, we make some minor adaptations to the original implementation of the model provided by the authors of CC2Vec:

- **Inclusion of the hunk encoder and attention layers.** When analyzing the original implementation provided by the authors, we find that CC2Vec also expects each commit to contain only one file change. However, this goes unnoticed when inputting a regular commit because the hunk encoder and hunk attention layers are simply left out. As a result, the model interprets each file as a hunk, and applies the comparison functions to compare the added and removed lines in all file changes, instead of within just one file change. To ensure CC2Vec generalizes correctly to multiple files, we adapt the model to correctly include the hunk encoder and attention layers. In doing so, we follow the descriptions in the original paper.

- **Exclusion of the test set from training.** In our implementation, we follow the critique made by the authors of JITLine [23], who note that CC2Vec incorrectly uses the whole dataset (i.e., training + testing) for model training. They do so by assuming that all unlabelled testing samples would be available beforehand. However, considering the significant amount of time needed for retraining before making predictions would render the model ineffective for our just-in-time use case. For this reason, we choose to leave out test samples in the training phase.

- **Empirical determination of file padding value.** To determine suitable padding values, we follow the same analysis as performed for DeepJIT (Appendix A). However, initial experiments showed that increasing the number of padded files greater than 2 always leads to worse performance. This leads us to believe that the implemented padding method impacts a model's classification. Therefore, we fix the number of padding values to 2.

- **Improvements similar to that of DeepJIT.** Some of the adaptions made to DeepJIT can also be applied to CC2Vec, i.e. we include early stopping based on a validation set and add an weight decay component to the loss function. Furthermore, since DeepJIT is used in combination to produce bug predictions, all adaptations to DeepJIT described above are also included here.

3.3.3. JITLine

JITLine [23] is a machine learning based just-in-time defect prediction approach that can both predict defect-introducing commits and identify defective lines that are associated with that commit. Its approach therefore consists of two phases. In the first phase, commit-level metrics are used to build a commit-level just-in-time defect prediction model using Random Forest as the classification technique. In the second phase, lines in a suspected defective commit are ranked by their level of riskiness, which is computed using LIME [26].

In our work, we focus on identifying both defective commits as well as defective lines. Because a ranking of defective lines does not provide us with clear predictions, we mainly look to leverage the first phase of the model.

To evaluate the effectiveness of JITLine at identifying defective files, we also leverage the first phase of the original model. However, we now provide it with manually crafted features at file-level that were presented earlier in Table 3.2.

3.3.4. Class Imbalance

To deal with class imbalance, we apply separate approaches for our metrics-based models and our deep learning based models. For the metrics-based models, we apply the often used SMOTE technique [4] to oversample the defective samples in our training dataset. For the deep learning models, i.e. DeepJIT and CC2Vec, such a technique cannot be applied in a straightforward manner. Instead, as described earlier, we deal with the class imbalance through a loss function that is weighted according to the distribution of the two classes.

3.3.5. Model Evaluation

Evaluation setting Our study focuses on evaluating performances at both commit-level and file-level within a change. Therefore, we perform experiments and report results for both settings. As baselines, we train a linear Logistic Regression classifier and a non-linear Random Forest model, similar to that of many existing approaches in research and industry that use only process metrics as input features [15, 22, 23, 36, 36, 37]. Both baselines apply SMOTE [4] to ensure a balanced dataset, and hyperparameter optimization is performed via grid search on a validation set. For the final hyperparameter settings, we refer to Appendix B.

Performance metrics For each optimized model, we compute the following performance metrics:

1. Precision is a good measure for a model's proneness to wrongly classifying clean changes as defect-introducing changes. The higher the precision, the better a model is able to correctly predict defect-introducing changes. The precision is calculated as $Pr = \frac{TP}{TP+FP}$.

2. Recall is used to measure the model's ability to identify defect-introducing changes as such. A recall score of 1 indicates that the model was able to predict all defect-introducing changes correctly. The recall is calculated as $Rec = \frac{TP}{TP+FN}$.

3. F1-score is the harmonic mean between precision and recall, and is therefore computed as $F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

4. AUC is the Area Under the ROC Curve, which displays the true positive rate versus the false positive rate of a model at different classification thresholds. In the field of just-in-time defect prediction, the AUC is arguably the most common metric by which models are compared. This is done as datasets in the field are naturally imbalanced, causing threshold-dependent measures (i.e., precision, recall, or F1) to be dependent on the chosen threshold. The AUC score does not need a manually set threshold and thus can be a quite objective interpretation of a model's ability to differentiate between defective or clean commits. AUC

scores range from 0 to 1, where a value of 1 indicates perfect discrimination, while a value of 0.5 indicates random guessing. A score lower than 0.5 can be resolved by flipping the binary classifications of the model.

Chapter 4

Experiments

The goal of our experiments is to evaluate the effectiveness of just-in-time defect prediction models when applied in our industry setting. We do so through a set of research questions that we look to answer by performing experiments:

- **RQ1:** How well do state-of-the-art just-in-time defect prediction models perform on a new dataset, gathered from an industry project?
- **RQ2:** How effective are the evaluated models at detecting real-world bug-introducing changes?
- **RQ3:** How important are the different bug sources for the effectiveness of the evaluated models?

4.1. Methodology

To answer our research questions, we collect a novel dataset based on a large-scale industry project. We evaluate the performance of the three proposed models (DeepJIT, CC2Vec and JITLine). As our baselines, we implement a non-linear Random Forest classifier and a linear Logistic Regression classifier, using commit-level change metrics as features.

In order to be able to compare the obtained results to that of prior work, we also train and evaluate the performance of all models on the QT and OpenStack datasets. We apply similar data extraction and preprocessing steps to each of the three datasets to ensure comparability.

To answer RQ1, we use the first 90% of the datasets as training set, and the rest as our test set. We do so as our data is time-sensitive, thus we need to make sure we only use realistically available labels during training. We train all models at two different change granularities: commit-level and file-level. We evaluate the performance of the models mainly by their F1-score, the harmonic mean between precision and recall. We also report the precision and recall scores with a decision boundary at 0.5, complemented by precision-recall curves that show the models performances at various thresholds. Finally, we report the models AUC scores as well as the corresponding ROC-curves, as these can serve as indicators of how well the model generalizes to varying classification thresholds and class distributions.

To answer RQ2, we collect a total of 11 changes that were manually labelled by developers at Adyen as buggy while performing code reviews. All bug-introducing

changes were collected over a time span of 2 months, later than the evaluated dataset in RQ1. To allow for comparison with the results achieved in RQ1, we randomly sample a number of non-buggy changes from the same period. The number of randomly sampled changes is determined by the ratio of buggy and clean changes in the original Adyen dataset. To mitigate the impact of randomness, we repeat each experiment 100 times and report the resulting mean and standard deviations.

To answer RQ3, we perform an ablation study on our four sources for bug fixing commits. Changes identified as bugs through the ablated source are labelled as clean in the training set, after which the model is retrained. We then evaluate the performance of the models on the real-world bugs collected for RQ2, following the same random sampling approach to ensure comparability.

4.2. Results

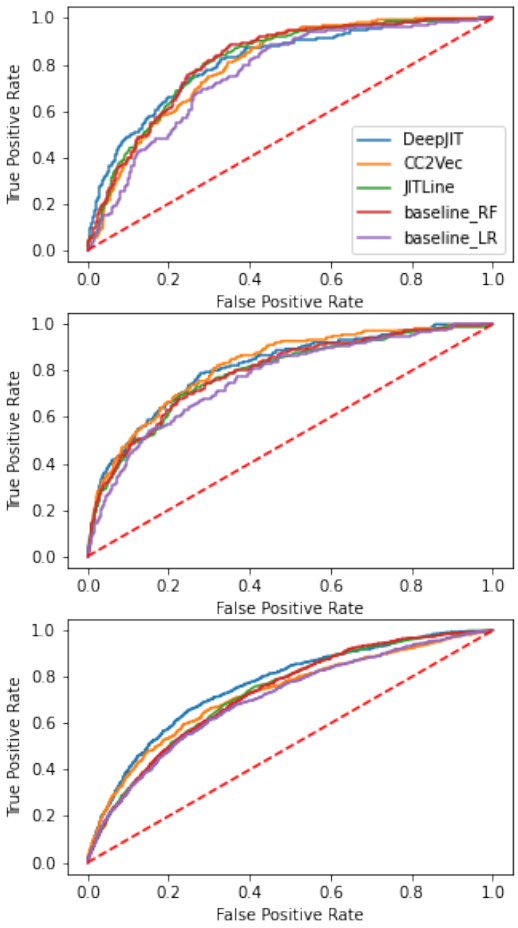
In this section, we present the results of our experiments and the observations we make based on them.

4.2.1. RQ1: How well do state-of-the-art just-in-time defect prediction models perform on a new dataset, gathered from an industry project?

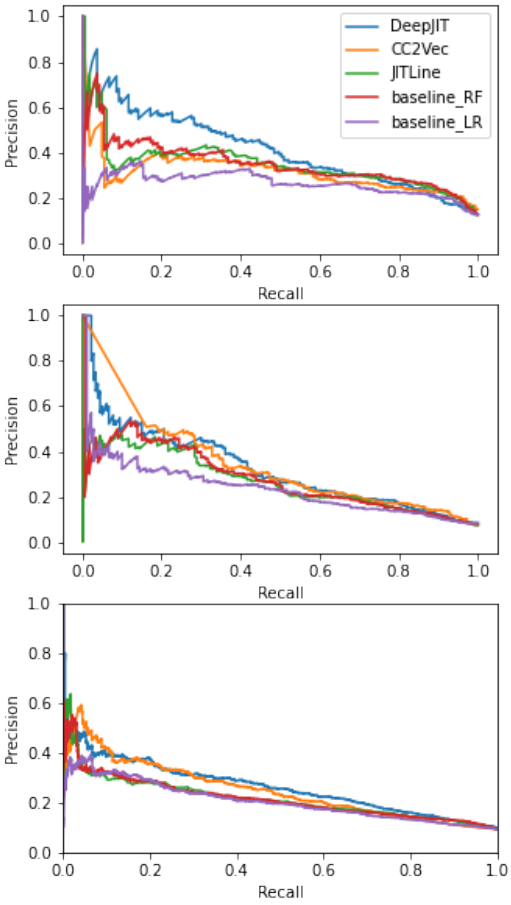
Table 4.1 shows the performance of all evaluated models at commit-level, Table 4.2 at file-level. For each model, we report the AUC score and its achieved precision, recall and F1-score at a threshold of 0.5. For completeness, we also provide the corresponding confusion matrix for each obtained result in the last 4 columns. In addition, we report the ROC curve and the precision-recall curves for all models at commit-level in Figure 4.1 and at file-level in Figure 4.2. In our observations, we first focus commit-level results before we discuss the results at file-level.

Dataset	Model	Precision	Recall	F1	AUC	TN	FP	FN	TP
Commit-level									
	DeepJIT	0.320	0.638	**0.426**	0.811	931	221	59	104
	CC2Vec	0.248	**0.767**	0.375	0.800	773	379	38	125
OpenStack	JITLine	**0.343**	0.491	0.404	0.813	999	153	83	80
	RF	0.327	0.491	0.392	**0.815**	987	165	83	80
	LR	0.259	0.681	0.375	0.762	834	318	52	111
	DeepJIT	0.262	0.533	**0.351**	0.813	1971	271	84	96
	CC2Vec	0.259	**0.544**	**0.351**	**0.821**	1962	280	82	98
QT	JITLine	**0.313**	0.350	0.331	0.787	2104	138	117	63
	RF	0.305	0.394	0.344	0.792	2080	162	109	71
	LR	0.171	0.633	0.269	0.768	1688	554	66	114
	DeepJIT	0.223	**0.593**	0.324	**0.762**	5773	1548	306	445
	CC2Vec	0.250	0.469	**0.326**	0.726	6266	1055	399	352
Adyen	JITLine	0.280	0.181	0.220	0.729	6971	350	615	136
	RF	**0.287**	0.204	0.238	0.728	6941	380	598	153
	LR	0.180	0.563	0.273	0.704	5397	1924	328	423

Table 4.1: Model performances when evaluating commit-level predictions. The best score achieved within each dataset is displayed in bold.



(a) ROC curves per dataset for every model when trained and evaluated at commit-level.



(b) Precision-recall curves per dataset for every model when trained and evaluated at commit-level.

Figure 4.1: Precision and recall trade-off at varying thresholds when making predictions at commit-level.

Observation 1: On open-source datasets, most models achieve good and similar results at commit-level, with deep learning models slightly outperforming the metric-based models. When comparing results obtained for OpenStack (Table 4.1), DeepJIT achieves an F1-score of 0.426, with a precision and recall of 0.320 and 0.638 respectively at the threshold of 0.5. It thereby outperformed the metric-based models and CC2Vec in terms of F1-score, where JITLine was the best metric-based performer at 0.404. AUC scores for OpenStack were largely the same for all models, with only the Logistic Regression baseline underperforming. For the QT dataset, we observe comparable performances between the deep learning and metric-based models. DeepJIT and CC2Vec achieve the same F1-score of 0.351, and thereby only slightly outperform the best performing metric-based model in JITLine with an F1-score of 0.331. AUC scores for both DeepJIT and CC2Vec are also higher than the other models at 0.813 and 0.821 respectively, with the best performer in the Random Forest baseline achieving 0.792.

Looking at the ROC-curves for all models at commit-level (Figure 4.1), we only observe minor differences between the models. The difference in terms of performance becomes more apparent when we compare the precision-recall curves (Figure 4.1). We clearly observe here that in the OpenStack dataset, DeepJIT outperforms the other models, especially as we prioritize precision more over recall.

Observation 2: Deep Learning significantly outperforms metric-based models on Adyen data at commit-level, but all models perform worse than in an open-source setting. DeepJIT and CC2Vec obtain very similar F1 scores of 0.324 and 0.326 respectively, and thereby outperform the metric-based model of JITLine and our baselines by up to 48%. F1-scores for all models are however slightly lower compared the result achieved on the OpenStack and QT datasets, with especially JITLine and the Random Forest baseline performing poor in terms of recall. The AUC scores also drop, with DeepJIT being the best performer with an AUC of 0.762. JITLine and the baseline Random Forest model achieve significantly lower recall values on Adyen data compared to their evaluation on the open source datasets: 0.181 and 0.204 respectively. Both DeepJIT and CC2Vec thereby show to generalize better to our novel industry dataset than the metric-based models.

Observation 3: At file-level predictions, All models lose a large amount of performance compared to commit-level, but deep learning models significantly outperform metric-based models. In terms of F1 score, we observe that both DeepJIT and CC2Vec outperform the metric-based models across all three datasets at file-level (Table 4.2). The metric-based models most notably give up a lot on recall when compared to the results achieved at commit-level. For example, JITLine and the Random Forest baseline achieve recall scores as low as 0.079 and 0.091 respectively when evaluated on Adyen data. Both DeepJIT and CC2Vec achieve significantly better recall scores at 0.512 and 0.434, respectively. In terms of precision, all models drop significantly in performance compared to commit-level predictions, and all achieve very comparable results. On Adyen data, CC2Vec most notably outperforms the rest with a precision of 0.141, thereby also achieving the best performance in terms of F1 score. Finally, when comparing the models by their AUC, DeepJIT and CC2Vec

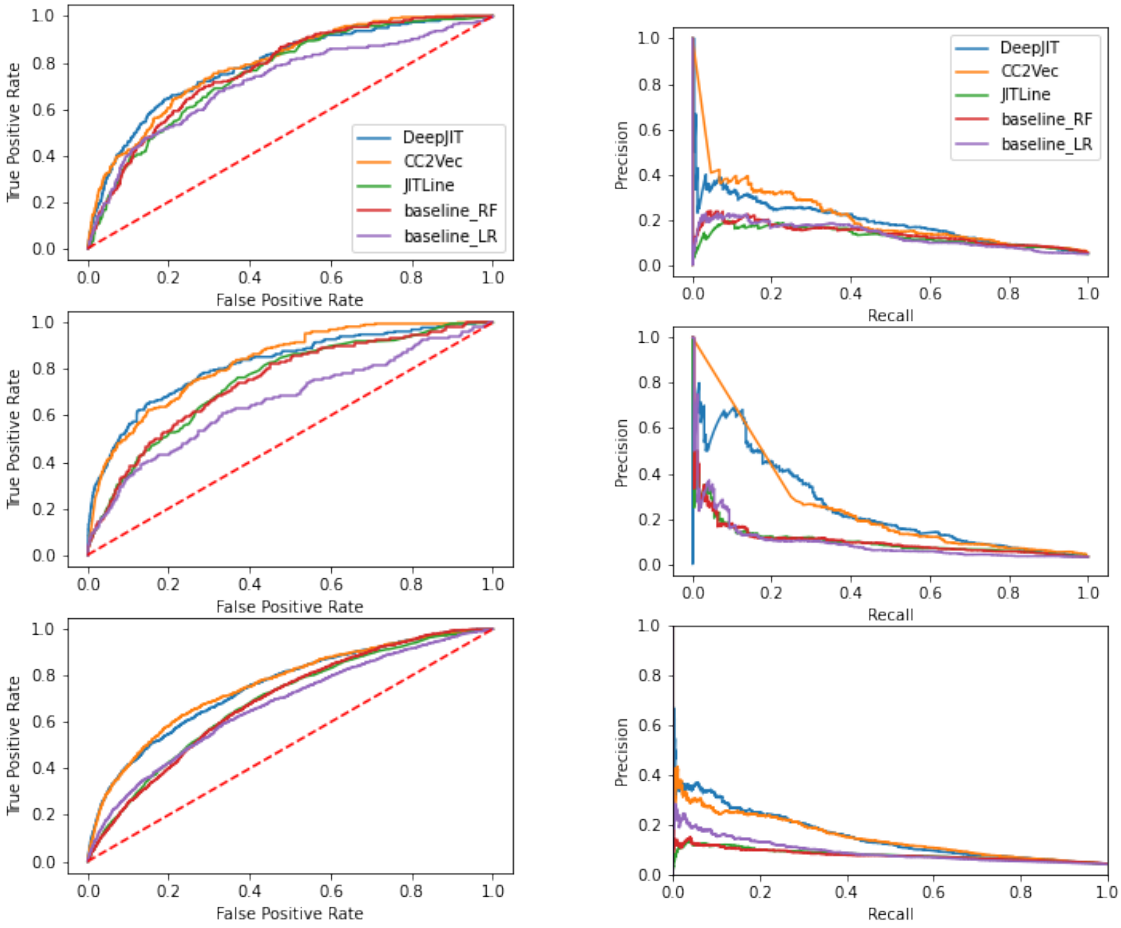
achieve comparable results and significantly outperform the rest in all three datasets.

Looking at the ROC-curves for all models at file-level (Figure 4.2), we observe that both DeepJIT and CC2Vec outperform the other models. Similar results are observed from the the precision-recall curves 4.2, where deep learning becomes the even more favorable approach as we increase the importance of precision over recall.

Dataset	Model	Precision	Recall	F1	AUC	TN	FP	FN	TP
File-level									
	DeepJIT	0.165	**0.587**	**0.258**	0.788	3810	697	97	138
	CC2Vec	0.153	0.489	0.233	**0.791**	3868	639	120	115
OpenStack	JITLine	**0.179**	0.234	0.203	0.754	4254	253	180	55
	RF	0.169	0.230	0.195	0.769	4242	265	181	54
	LR	0.115	0.523	0.189	0.718	3565	942	112	123
	DeepJIT	**0.157**	**0.527**	**0.242**	**0.821**	6582	669	112	125
	CC2Vec	0.133	**0.527**	0.213	0.829	6437	814	112	125
QT	JITLine	0.137	0.143	0.140	0.747	7037	214	203	34
	RF	0.124	0.143	0.133	0.743	7010	241	203	34
	LR	0.055	0.620	0.100	0.661	4704	2547	90	147
	DeepJIT	0.118	**0.512**	0.192	0.751	22954	4422	563	591
	CC2Vec	**0.141**	0.434	**0.213**	**0.756**	24331	3045	653	501
Adyen	JITLine	0.121	0.079	0.096	0.687	26718	658	1063	91
	RF	0.117	0.091	0.102	0.688	26581	795	1049	105
	LR	0.070	0.581	0.124	0.671	18421	8955	484	670

Table 4.2: Model performances when evaluating file-level predictions. The best score achieved within each dataset is displayed in bold.

Observation 4: CC2Vec does not clearly outperform DeepJIT in any dataset, questioning the effectiveness of the model in a just-in-time defect prediction setting. As has also been discussed in all three previous observations, DeepJIT and CC2Vec perform very similar to one another in all three datasets and at both granularities. We also observe very similar ROC-curves and precision-recall curves for both models. Since CC2Vec is an extension of DeepJIT, and only differs from DeepJIT by adding an additional vector representation of the code changes, this similarity of performance indicates that the effectiveness of these representations is limited in our study.



(a) ROC curves per dataset for every model when trained and evaluated at commit-level.

(b) Precision-recall curves per dataset for every model.

Figure 4.2: Precision and recall trade-off at varying thresholds when making predictions at file-level. The best score achieved within each dataset is displayed in bold.

RQ1 Conclusion

Deep Learning based models outperform the metric-based models across all three datasets. When evaluated on Adyen data, all models perform slightly worse than in an open-source setting, but both DeepJIT and CC2Vec still achieve respectable performances and significantly outperform the metrics-based models. At file-level predictions, all models lose a large amount of performance compared to the observed performances at commit-level, but deep learning models significantly outperform metric-based models. Deep learning therefore proves to be more effective at identifying defective changes than metric-based models, and better generalizes to our novel industry-based dataset. Considering the performances of DeepJIT and CC2Vec are comparable, the effectiveness of the embeddings produced by CC2Vec in a just-in-time defect prediction setting seems limited. As DeepJIT also requires significantly less training time, we consider DeepJIT to be the best performing model.

4.2.2. RQ2: How effective are the evaluated models at detecting real-world bug-introducing changes?

Table 4.3 shows the the performance of the evaluated models when evaluated against the manually collected bug-introducing changes mixed with randomly sampled clean changes.

Model	Precision	Recall	F1
Commit-level			
DeepJIT	**0.250 (0.033)**	**0.727**	**0.371 (0.036)**
JITLine	0.247 (0.159)	0.091	0.127 (0.016)
LR	0.234 (0.028)	0.727	0.353 (0.032)
RF	0.000 (0.000)	0.000	0.000 (0.000)
File-level			
DeepJIT	0.080 (0.005)	**0.818**	0.146 (0.008)
JITLine	0.000 (0.000)	0.000	0.000 (0.000)
LR	0.103 (0.091)	0.818	**0.183 (0.015)**
RF	**0.274 (0.091)**	0.182	0.214 (0.026)

Table 4.3: Model Performances when predicting manually collected bugs at commit-level and file-level. Reported results are the average of 100 runs, with the standard deviation reported in brackets behind it. The best score achieved within each dataset is displayed in bold.

Observation 5: DeepJIT achieves consistent Precision and Recall scores at commit-level, and significantly outperforms the metric-based models. When predicting bugs at commit-level, DeepJIT achieves a Precision and Recall of 0.250 and 0.727 respectively, and thereby outperforms all other models in all three metrics. This is very comparable to the results obtained in RQ1 (Table 4.1, where it achieved a Precision and Recall of 0.320 and 0.638 respectively. The baseline Random Forest model and JITLine proved ineffective at correctly identifying the collected bug-introducing commits, predicting respectively 0 and 1 out of the 11 buggy commits

as such. The Logistic Regression baseline was the best performing metric-based model, achieving a recall score of 0.727.

Observation 6: DeepJIT achieves better than expected recall at file-level, but is significantly outperformed by the Random Forest baseline in terms of precision. With a Recall score of 0.818, DeepJIT was able to identify most buggy file changes correctly, but does so by giving up some of its Precision observed in RQ1 (0.118, Table 4.2). The Logistic Regression baseline model achieved the same recall, but achieves a slightly better precision of 0.103. Finally, we observe a significantly higher precision achieved by the Random Forest baseline model, on average 0.274, but with a much lower recall score of 0.182. The Random Forest model thereby performs better than its commit-level based counterpart, and also becomes the overall best performing model at file-level in terms of F1-score.

Observation 7: Predictions by the same model at the different granularities do not necessarily overlap. When we inspect Table 4.4, we see some significant differences between a model's predictions at commit-level and file-level. For example, DeepJIT wrongly predicts commits #1, #2 and #4 as clean, while the same model trained and evaluated at file-level predicts them correctly as buggy. On the other hand, DeepJIT correctly identifies commits #9 and #10 as bug introducing changes, but fails to do so at file-level. For the metric-based models, we can observe the same dissimilarities between the commit-level and file-level based models.

	Commit-level				**File-level**			
bug	**DeepJIT**	**JITLine**	**LR**	**RF**	**DeepJIT**	**JITLine**	**LR**	**RF**
#1			✓		✓		✓	✓
#2					✓		✓	
#3	✓		✓		✓			
#4			✓		✓		✓	
#5	✓		✓		✓		✓	
#6	✓	✓	✓		✓		✓	✓
#7	✓		✓		✓		✓	
#8	✓		✓		✓			
#9	✓						✓	
#10	✓						✓	
#11	✓		✓		✓		✓	

Table 4.4: Correctness of prediction for each manually collected bug at commit-level and file-level. A checkmark indicates the model successfully predicted the bug as such.

RQ2 Conclusion

At commit-level, DeepJIT performs consistent with the findings in RQ1, achieving similar or even better precision and recall scores when predicting real-world buggy changes. It thereby significantly outperforms the metric-based models in both precision and recall. At file-level however, even though good recall scores are observed, some precision is lost in the real-world setting compared to our earlier findings. Comparing individual predictions at both granularities also reveals there is a dissimilarity between a model's predictions at both granularities. Overall, even though the number of samples is very limited, DeepJIT shows promising initial results when tested on a real-world bugs at commit-level, but more research is needed to effectively incorporate predictions at file-level in a real-world setting.

4.2.3. RQ3: How important are the different bug sources for the effectiveness of the evaluated models?

Table 4.5 shows the the performance of the evaluated models for each ablated source.

Observation 8: No particular source is clearly most representative of buggy changes, as it seems dependent on both model and granularity of predictions. While some models perform best when all sources are included in the labelling process, others benefit from leaving out one source. For example, ablating labels obtained through release patches leads to the largest performance loss for DeepJIT, achieving only a recall of 0.091 at commit-level. However, all other metric-based models improve either on recall, precision or both at commit-level when doing so. Ablating issues leads to a significant drop in performance for DeepJIT at both commit-level and file-level, achieving recalls of 0.273 and 0.545 respectively. For metric-based models, ablating issues does not clearly lead to a drop or gain in performance. Ablating labels obtained through emergency commits often resulted in a performance loss at both prediction granularities, with the exception of DeepJIT at file-level. Finally, ablating labels obtained through code reviews almost always lead to a decrease in recall, albeit only small.

Observation 9: Using emergency commits as bug source may introduce noise in labels at file-level. For DeepJIT, ablating any source of data almost always led to a deterioration in performance, except while ablating labels collected through emergency commits. Upon closer inspection of the source, it becomes evident that a large part of emergency commits are "reverts" of previous commits. The reverted commit is therefore indeed likely to be bug-introducing, but also all changed files are labelled as such, even though only one of the files is likely to be bug-introducing. This likely leads to a large amount of file changes wrongly labelled as bug-introducing, thereby introducing noise to our labelled dataset.

RQ3 Conclusion

Whether inclusion of any of the proposed bug label sources leads to information gain is dependent on both what type of model is used and at what granularity predictions are made. Bugs labelled through release patches are the most important source for our deep-learning based model, but cause the metric-based models to perform worse. Labels obtained through emergency commits are suspected to introduce some noise at file-level due to such commits often being "revert" commits. Therefore, when selecting sources to obtain labels, taking into account the granularity of predictions is important.

Commit-level

Model	Ablated Source				Precision	Recall	F1
	Iss.	Pat.	Eme.	Rev.			
DeepJIT	All sources				**0.250 (0.033)**	**0.727**	**0.371 (0.036)**
	✗				0.198 (0.042)	0.273	0.227 (0.027)
		✗			0.168 (0.069)	0.091	0.114 (0.015)
			✗		0.190 (0.025)	0.545	0.281 (0.027)
				✗	0.243 (0.043)	0.455	0.315 (0.036)
JITLine	All sources				0.247 (0.159)	**0.091**	0.127 (0.016)
	✗				0.000 (0.000)	0.000	0.000 (0.000)
		✗			**0.401 (0.225)**	**0.091**	**0.142 (0.015)**
			✗		0.000 (0.000)	0.000	0.000 (0.000)
				✗	0.000 (0.000)	0.000	0.000 (0.000)
baseline_RF	All sources				0.000 (0.000)	0.000	0.000 (0.000)
	✗				0.000 (0.000)	0.000	0.000 (0.000)
		✗			**0.250 (0.163)**	**0.091**	**0.127 (0.016)**
			✗		0.000 (0.000)	0.000	0.000 (0.000)
				✗	0.000 (0.000)	0.000	0.000 (0.000)
baseline_LR	All sources				0.234 (0.028)	0.727	0.353 (0.032)
	✗				**0.265 (0.033)**	**0.818**	**0.399 (0.037)**
		✗			0.251 (0.029)	**0.818**	0.383 (0.033)
			✗		0.099 (0.017)	0.273	0.144 (0.018)
				✗	0.216 (0.027)	0.636	0.322 (0.030)

File-level

Model	Ablated Source				Precision	Recall	F1
	I	P	E	R			
DeepJIT	All sources				0.080 (0.005)	0.818	0.146 (0.008)
	✗				0.060 (0.005)	0.545	0.107 (0.008)
		✗			0.079 (0.005)	0.818	0.143 (0.009)
			✗		**0.089 (0.006)**	**0.909**	**0.163 (0.009)**
				✗	0.079 (0.005)	0.818	0.144 (0.008)
JITLine	All sources				0.000 (0.000)	0.000	0.000 (0.000)
	✗				**0.300 (0.179)**	**0.091**	**0.134 (0.015)**
		✗			0.195 (0.110)	**0.091**	0.119 (0.015)
			✗		0.000 (0.000)	0.000	0.000 (0.000)
				✗	0.000 (0.000)	0.000	0.000 (0.000)
baseline_RF	All sources				**0.274 (0.091)**	**0.182**	**0.214 (0.026)**
	✗				0.000 (0.000)	0.000	0.000 (0.000)
		✗			0.192 (0.122)	0.091	0.118 (0.016)
			✗		0.000 (0.000)	0.000	0.000 (0.000)
				✗	0.000 (0.000)	0.000	0.000 (0.000)
baseline_LR	All sources				0.103 (0.010)	0.818	**0.183 (0.015)**
	✗				0.089 (0.008)	0.727	0.158 (0.013)
		✗			**0.104 (0.010)**	0.727	0.181 (0.016)
			✗		0.021 (0.002)	0.182	0.038 (0.003)
				✗	0.082 (0.005)	**1.000**	0.151 (0.009)

Table 4.5: Ablation study of bug sources when predicting manually collected bugs at commit-level and file-level. Reported results are the average of 100 runs, with the standard deviation reported in brackets behind it. The best score achieved within each dataset is displayed in bold. The source that is left out during training is marked with an "✗" in the *Ablated Source* column.

Chapter 5

Discussion

We summarize the results in our study, and discuss implications and threats to validity in this section.

5.1. Practical Implications

Our study reveals the following practical implications that should be considered in future studies related to just-in-time defect prediction.

Deep learning can work, but issues remain. The results from our study show that in the setting of just-in-time defect prediction, deep learning models can outperform traditional metric-based classifiers in both open-source and industry settings. Nonetheless, the issues that we encountered and the adaptations we have had to make when reusing both DeepJIT and CC2Vec raise some concern.

As already stated in Observation 4, we saw that CC2Vec, an extension on top of DeepJIT, was unable to contribute in a significant way to the ability of predicting bugs in changes. We therefore question the effectiveness of the embeddings produced by the model. While we encourage the work towards learning code representations, more research is needed to study the effectiveness of produced embeddings before such models can be effectively applied in just-in-time defect prediction. A recent study by Tian et al. [35] compares embeddings of buggy code and patched code produced by CC2Vec when combined with PatchNet, but not as a stand alone.

A study by Zeng et al. [43] that was published just before finishing this work, focuses on evaluating the current state-of-the-art deep learning models for just-in-time defect prediction (i.e. DeepJIT and CC2Vec). They draw some of the same conclusions as we do, such as the ineffectiveness of CC2Vec, but also show the results obtained in the work of CC2Vec were likely obtained with partially dummy data. As far as we know, our study and that of Zeng et al. [43] are the only studies so far to have observed this issue. We therefore conjecture that not only the results obtained in the original study, but follow-up studies such as JITLine were also impacted by this and other issues. In fact, we notified the authors of JITLine about a bug in their replication package, one that we had also observed to be present in the replication package of CC2Vec. The authors responded by pointing out they had used the code as provided in the replication package of CC2Vec, confirming our suspicion. We have proposed a fix[1] to the authors of CC2Vec, which has been approved but left unmerged

[1] github.com/soarsmu/CC2Vec/pull/2

unfortunately.

A variety of evaluation metrics is necessary to represent a model's performance in a real-world setting. In the field of just-in-time defect prediction, the AUC is arguably the most common metric by which models are compared. This is done as datasets in the field are naturally imbalanced, causing threshold-dependent measures (i.e., precision, recall, or F1) to be dependent on the chosen threshold. The AUC score does not need a manually set threshold and thus can be a quite objective interpretation of a model's ability to differentiate between defective or clean commits. We conjecture however that achieving better AUC scores does not represent a meaningful improvement without considering other performance metrics, such as precision and recall. The authors of JITLine provide us an excellent example, where they report their implementation of DeepJIT to have an AUC of 0.75, but report precision and recall scores of both 0. Although such a model might overall show decent capability of separating positive from negative classes, in practice it is of little use to help spot bugs in commits. Adding additional performance metrics such as precision and recall can give us a good indication of the expected capability of a classifier if it were to be applied in a real-world setting. Moreover, one could report the precision-recall curve, which allows for a threshold-independent evaluation of a model's performance in terms of precision and recall.

Models should be treated as risk indicators, not as oracles. Even though the evaluated models display the ability to discriminate buggy changes from clean changes, predictions are still very much prone to errors. We therefore conjecture that defect predictions should be viewed as a predicted "level of riskiness" and suggest that practitioners, instead of judging predictions to be either right or wrong, instead use such predictions as an objective source of information decision making. In our studied use case, code reviewers at Adyen could for example use the predictions on riskiness in their decision on where more attention should be payed during code reviews.

5.2. Threats to Validity

Threats to internal validity. The threats to internal validity refer to errors in the performed experiments and experimenter bias.

For each task, existing implementations of relevant data collection tools were used or adapted when available, and implementations of all models studies were obtained and reused, We have also been in contact with the authors of all models used to ensure correctness of our work, and have used the same evaluation metrics that were used in previous studies. Nonetheless, quite some adaptations had to be made to both data collection tools and reused models. Even if some if these adaptations were in order to remove bugs, errors may remain.

Another significant threat to our study regards the validity of our collected dataset, which is two-fold. Firstly, we have identified new sources to collect bug-fixing commits. All of the sources used were selected according to Adyen's way of working, and are therefore assumed to be reliable enough sources for bug-fixes. To assess the individual effectiveness of each dataset, we have also provided an abla-

tion study that mostly solidified our assumption. Nonetheless, manual validation by experts would be necessary in order to guarantee this. Secondly, a threat to the validity of our dataset is caused by the SZZ algorithm [32], through which we identify bug-introducing changes in our datasets. The SZZ algorithm is commonly used in defect prediction research, yet has known limitations [8, 27]. However, the focus of our study is to compare the performance just-in-time defect prediction models in an industry setting. Since the SZZ algorithm is the most widely used algorithm in the field, we deem it the best option available to do conduct our study.

External validity Threats to external validity concern the generalizability of our work. We emphasize that the focus of this work lies in applying just-in-time defect prediction at Adyen, in order to aid Adyen developers in performing code reviews. If the same models were to be applied in a different setting, different results may be obtained. That being said, we have mitigated some concerns regarding generalizability by evaluating all models used on publicly available datasets that have also been been used in the studies that originally proposed the models. Although these projects might not be representative of all projects out there, together they cover a variety of programming languages, application domains and sizes. We therefore believe the obtained results in this work meaningfully contribute to the validation of empirical knowledge about just-in-time defect prediction models in an industry setting.

Chapter 6

Conclusion

Finding defects in proposed changes is one of the biggest motivations and expected outcomes of code review, but does not result as often in actually finding defects. To mitigate this issue, efficient allocation of inspection time could be done according to the defect-proneness of the changed software parts. The field of study that attempts to model and predict such likelihood of defectiveness is called defect prediction. Just-in-time defect prediction models, that predict defects at change-level, can help developers spot defective changes during code review.

In this work, we have investigated the effectiveness of state-of-the-art just-in-time defect prediction approaches when applied in an industry setting. To construct a new dataset with a large enough set of labels, we identified four sources of potential bug-fixing commits by analysing Adyen's way of working. We compared three traditional metric-based models with two recent deep learning-based models that make predictions at both commit-level and file-level. We set out to answer three research questions related to the models' performances and the effectiveness of our label sources. We rephrase each of the research questions and the corresponding conclusions below.

RQ1: How well do state-of-the-art just-in-time defect prediction models perform on a new dataset, gathered from an industry project?
To answer this research question, we trained and evaluated all models on our newly collected dataset, as well as two datasets based on large-scale open source projects. We concluded that deep learning based models outperform the metric-based models across all three datasets. When evaluated on Adyen data, all models performed slightly worse than in an open-source setting, but both DeepJIT and CC2Vec still achieved respectable performances and significantly outperformed the metrics-based models. At file-level predictions, all models lost a large amount of performance compared to the observed performances at commit-level, but deep learning models significantly outperformed metric-based models. Deep learning therefore proves to be more effective at identifying defective changes than metric-based models, and generalizes better to our novel industry-based dataset. Considering the performances of DeepJIT and CC2Vec are comparable, the effectiveness of the embeddings produced by CC2Vec in a just-in-time defect prediction setting seems limited. As DeepJIT also requires significantly less training time, we consider DeepJIT to be the best performing model in both an open-source setting as well as on our large-scale industry dataset.

RQ2: How effective are the evaluated models at detecting real-world bug-introducing changes?
To answer this research question, we tested the models on a small set of manually collected bugs by developers at Adyen. At commit-level, DeepJIT performed consistent with the findings in RQ1, achieving similar or even better precision and recall scores. It thereby significantly outperformed the metric-based models in both precision and recall. At file-level however, even though good recall scores are observed, some precision is lost in the real-life setting compared to our earlier findings. Comparing individual predictions at both granularities also revealed there is a dissimilarity between a model's predictions at both granularities. Overall, even though the number of samples is very limited, DeepJIT shows promising initial results when tested in a real-world setting at commit-level, but more research is needed to effectively incorporate predictions at file-level in a real-life setting.

RQ3: How important are the different bug sources for the effectiveness of the evaluated models?
Finally, we performed an ablation study to investigate the added value of all four identified bug sources in detecting the collected real-world bugs. We found that Whether inclusion of any of the proposed bug label sources leads to information gain is dependent on both what type of model is used and at what granularity predictions are made. Bugs labelled through release patches are the most important source for our deep-learning based model, but cause the metric-based models to perform worse. Labels obtained through emergency commits are suspected to introduce some noise at file-level due to such commits often being "revert" commits. Therefore, when selecting sources to obtain labels, taking into account the granularity of predictions is important.

6.1. Future Work

Conducting this study provided us with some insights into both the models evaluated, as well as applying just-in-time defect prediction in practice. In this section, we therefore conclude by sharing some recommendations for future research.

- **Improving DeepJIT and CC2Vec.** We have made several small adaptations to both DeepJIT and CC2Vec, mostly with regards to adapting to our dataset and improving the training effectiveness. We however also propose some structural improvements for future work that we believe will likely contribute to their performances. Firstly, with regards to DeepJIT, we note that the final convolutional layer scales as the number of input files grows. Moreover, weights of the individual file embeddings are trained individually, meaning the order in which files are analyzed impacts the prediction of the model. As also discussed in our approach, the authors of DeepJIT had only experimented with commits that contain only one file, which removes the concern of file ordering. We hypothesize that by making the layer agnostic to the order of its input, DeepJIT will be able to generalize better to commits that span multiple files. We recognize that a similar methodology was applied in CC2Vec in order to concatenate

the file embeddings, thus CC2Vec may benefit from such an improvement in a similar manner.

Secondly, we hypothesize the performance of CC2Vec is hampered by its padding method. In the original implementation, lines, hunks and files are padded with a generic <unk> token. The padding tokens are however weighted in the prediction, meaning the model attempts to make sense of useless information. We hypothesize that as the size of the padding increases, the amount of noise that contributes to the prediction grows. This concern can likely be fixed by incorporating *masking*, a common technique used to help the model identify and ignore all padded values.

- **Improving the interpretability of deep predictions.** In order to better aid reviewers during code review, we have in this work focused on making finer grained predictions within a commit. However, this still does not add to the interpretability of the model, i.e. "why does the model think this change is defective?". One approach to add interpretability to deep models is through an interpretation algorithm such as GradCAM [29], that utilizes the weights in a CNN to visually show the important regions that affect model decisions. Another approach is through incorporating attention layers within the model, which are usually easy to interpret as their value represents the importance of each connection towards the output. However, improved interpretability of a model often comes with more simplicity of the model, thus it is believed that there is a trade-off between interpretability and performance of a model [18],

- **Automated patch repair models could serve as recommendations.** As an alternative to explaining why a model suspects a bug, we could also aim to provide the user with a recommendation on how to fix it. The research field that revolves around solving this task is called *Automated Program Repair* (APR). Recent studies on APR typically employ a data-driven approach, where by leveraging a large database of existing patches, source code snippets or both, relevant fixes can be recommended [1, 14].

- **Empirical assessment of models in code reviews.** As a next step towards introducing just-in-time defect prediction at Adyen, an in depth case study that applies the studied just-in-time defect prediction models in code reviews should be conducted. Such a study should focus on understanding the information needs of a reviewer that a prediction should provide, in order to effectively guide a reviewer in spotting a bug. Additionally, it should be investigated how reviewers perceive the usefulness and trustworthiness of a model at varying classification thresholds. This will allow for determination of a suitable trade-off between precision and recall. Finally, bringing a model to production should allow for collecting more ground-truth labels on bug-introducing changes, which will help tremendously in improving the task further.

Bibliography

[1] Tbar: revisiting template-based automated program repair. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 31–42. ACM, 2019. doi: 10.1145/3293882.3330577. URL https://doi.org/10.1145/3293882.3330577.

[2] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 712–721. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606617. URL https://doi.org/10.1109/ICSE.2013.6606617.

[3] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Syst. Appl.*, 36(4):7346–7354, 2009. doi: 10.1016/j.eswa.2008.10.027. URL https://doi.org/10.1016/j.eswa.2008.10.027.

[4] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16: 321–357, 2002. doi: 10.1613/jair.953. URL https://doi.org/10.1613/jair.953.

[5] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Trans. Software Eng.*, 43(7):641–657, 2017. doi: 10.1109/TSE.2016.2616306. URL https://doi.org/10.1109/TSE.2016.2616306.

[6] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empir. Softw. Eng.*, 17(4-5):531–577, 2012. doi: 10.1007/s10664-011-9173-9. URL https://doi.org/10.1007/s10664-011-9173-9.

[7] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Software Eng.*, 38(6):1276–1304, 2012. doi: 10.1109/TSE.2011.103. URL https://doi.org/10.1109/TSE.2011.103.

[8] Steffen Herbold, Alexander Trautsch, and Fabian Trautsch. Issues with SZZ: an empirical assessment of the state of practice of defect prediction data collection. *CoRR*, abs/1911.08938, 2019. URL http://arxiv.org/abs/1911.08938.

[9] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science (New York, N.Y.)*, 313:504–7, 08 2006. doi: 10.1126/science.1127647.

[10] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 34–45. IEEE / ACM, 2019. doi: 10.1109/MSR.2019.00016. URL https://doi.org/10.1109/MSR.2019.00016.

[11] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: distributed representations of code changes. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 518–529. ACM, 2020. doi: 10.1145/3377811.3380361. URL https://doi.org/10.1145/3377811.3380361.

[12] Qiao Huang, Xin Xia, and David Lo. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empir. Softw. Eng.*, 24 (5):2823–2862, 2019. doi: 10.1007/s10664-018-9661-2. URL https://doi.org/10.1007/s10664-018-9661-2.

[13] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: a machine translation approach. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 20–30. ACM, 2018. doi: 10.1145/3196321.3196330. URL https://doi.org/10.1145/3196321.3196330.

[14] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309. ACM, 2018. doi: 10.1145/3213846.3213871. URL https://doi.org/10.1145/3213846.3213871.

[15] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70. URL https://doi.org/10.1109/TSE.2012.70.

[16] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008. doi: 10.1109/TSE.2007.70773. URL https://doi.org/10.1109/TSE.2007.70773.

[17] Andrey Krutauz, Tapajit Dey, Peter C. Rigby, and Audris Mockus. Do code review measures explain the incidence of post-release defects? *Empir. Softw. Eng.*, 25 (5):3323–3356, 2020. doi: 10.1007/s10664-020-09837-4. URL https://doi.org/10.1007/s10664-020-09837-4.

[18] Xuhong Li, Haoyi Xiong, Xingjian Li, Xuanyu Wu, Xiao Zhang, Ji Liu, Jiang Bian, and Dejing Dou. Interpretable deep learning: Interpretations, interpretability, trustworthiness, and beyond. *CoRR*, abs/2103.10689, 2021. URL https://arxiv.org/abs/2103.10689.

[19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

[20] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. *IEEE Trans. Software Eng.*, 44(5):412–428, 2018. doi: 10.1109/TSE.2017.2693980. URL https://doi.org/10.1109/TSE.2017.2693980.

[21] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 153–164. ACM, 2018. doi: 10.1145/3196398.3196438. URL https://doi.org/10.1145/3196398.3196438.

[22] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *J. Syst. Softw.*, 150:22–36, 2019. doi: 10.1016/j.jss.2018.12.001. URL https://doi.org/10.1016/j.jss.2018.12.001.

[23] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction, 2021. URL https://doi.org/10.1109/MSR52588.2021.00049.

[24] Foyzur Rahman and Premkumar T. Devanbu. How, and why, process metrics are better. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 432–441. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606589. URL https://doi.org/10.1109/ICSE.2013.6606589.

[25] Musfiqur Rahman, Dharani Palani, and Peter C. Rigby. Natural software revisited. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 37–48. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00022. URL https://doi.org/10.1109/ICSE.2019.00022.

[26] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144. ACM, 2016. doi: 10.1145/2939672.2939778. URL https://doi.org/10.1145/2939672.2939778.

[27] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Inf. Softw. Technol.*, 99:164–176, 2018. doi: 10.1016/j.infsof.2018.03.009. URL https://doi.org/10.1016/j.infsof.2018.03.009.

[28] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 966–969. ACM, 2015. doi: 10.1145/2786805.2803183. URL https://doi.org/10.1145/2786805.2803183.

[29] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *Int. J. Comput. Vis.*, 128(2): 336–359, 2020. doi: 10.1007/s11263-019-01228-7. URL https://doi.org/10.1007/s11263-019-01228-7.

[30] Martin J. Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. Software Eng.*, 40 (6):603–616, 2014. doi: 10.1109/TSE.2014.2322358. URL https://doi.org/10.1109/TSE.2014.2322358.

[31] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 62. ACM, 2012. doi: 10.1145/2393596.2393670. URL https://doi.org/10.1145/2393596.2393670.

[32] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005. doi: 10.1145/1082983.1083147. URL https://doi.org/10.1145/1082983.1083147.

[33] Davide Spadini, Maurício Finavaro Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 908–911. ACM, 2018. doi: 10.1145/3236024.3264598. URL https://doi.org/10.1145/3236024.3264598.

[34] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 99–108. IEEE Computer Society, 2015. doi: 10.1109/ICSE.2015.139. URL https://doi.org/10.1109/ICSE.2015.139.

[35] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 981–992. IEEE, 2020. doi: 10.1145/3324884.3416532. URL https://doi.org/10.1145/3324884.3416532.

[36] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 127–138. IEEE, 2020. doi: 10.1109/ICSME46990.2020.00022. URL https://doi.org/10.1109/ICSME46990.2020.00022.

[37] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.2978819.

[38] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. File-level defect prediction: Unsupervised vs. supervised models. In Ayse Bener, Burak Turhan, and Stefan Biffl, editors, *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 344–353. IEEE Computer Society, 2017. doi: 10.1109/ESEM.2017.48. URL https://doi.org/10.1109/ESEM.2017.48.

[39] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E. Hassan, and Xindong Zhang. Effort-aware just-in-time defect identification in practice: a case study at alibaba. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1308–1319. ACM, 2020. doi: 10.1145/3368089.3417048. URL https://doi.org/10.1145/3368089.3417048.

[40] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, pages 17–26. IEEE, 2015. doi: 10.1109/QRS.2015.14. URL https://doi.org/10.1109/QRS.2015.14.

[41] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Inf. Softw. Technol.*, 87: 206–220, 2017. doi: 10.1016/j.infsof.2017.03.007. URL https://doi.org/10.1016/j.infsof.2017.03.007.

[42] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pages 157–168. ACM, 2016. doi: 10.1145/2950290.2950353. URL https://doi.org/10.1145/2950290.2950353.

[43] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: how far are we? In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 427–438. ACM, 2021. doi: 10.1145/3460319.3464819. URL https://doi.org/10.1145/3460319.3464819.

Appendix A

Padding hyperparameter determination

To ensure parallellization, all samples for DeepJIT and CC2Vec are padded to the same sizes. This happens at line-level (code line length), file-level (LOC added/removed) and commit-level (#files, message length). We can estimate the right padding values by inspecting the datasets. We inspect all three levels in Figures A.1, A.2 and A.3 respectively. The determined hyperparameters for each model are provided in Table A.1

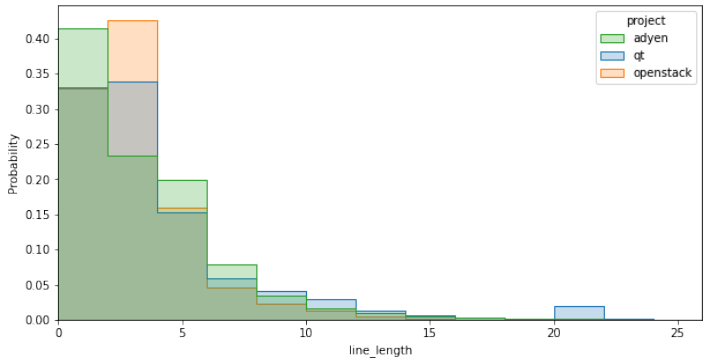


Figure A.1: The above graphs display the values for the code line lengths for each dataset. Bin width is set to 2

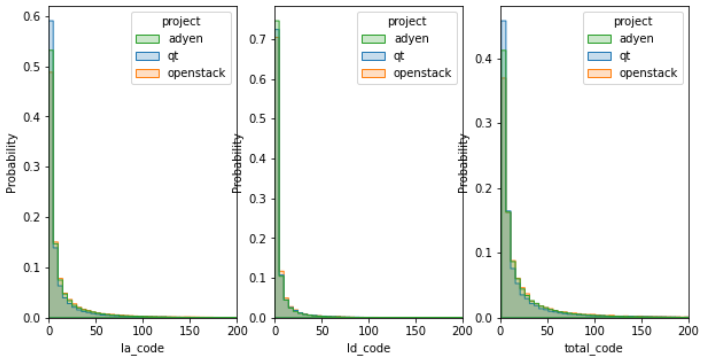


Figure A.2: The above graphs display the values for each dataset at file-level. From left to right: number of lines added, number of lines removed, total number lines (added + removed). Bin widths are set to 5, 5 and 5 respectively.

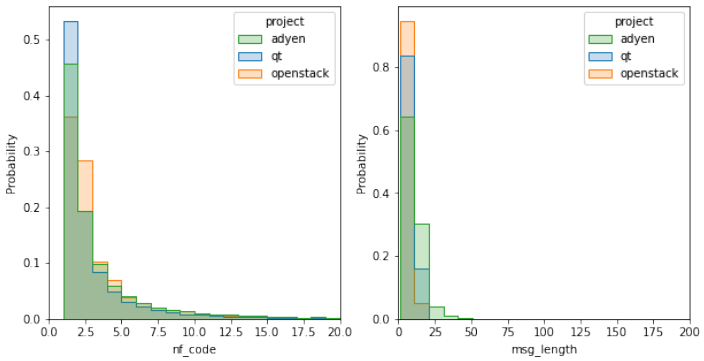


Figure A.3: The above graphs display the values for each dataset at commit-level. From left to right: number of files changed, message length. Bin widths are set to 1 and 10 respectively.

Dataset	Hyperparameter	DeepJIT	CC2Vec
QT	MESSAGE_LENGTH	20	20
	CODE_LINE_LENGTH	16	16
	NUM_CODE_LINES	100	15
	NUM_FILES	10	2
OpenStack	MESSAGE_LENGTH	20	20
	CODE_LINE_LENGTH	16	16
	NUM_CODE_LINES	100	15
	NUM_FILES	10	2
Adyen	MESSAGE_LENGTH	**30**	**30**
	CODE_LINE_LENGTH	16	16
	NUM_CODE_LINES	100	15
	NUM_FILES	10	2

Table A.1: Determined hyperparameters for both CC2Vec and DeepJIT for every dataset

Appendix B

Baselines hyperparameter optimization

Hyperparameter	OpenStack commit	OpenStack file	QT commit	QT file	Adyen commit	Adyen file
N_ESTIMATORS	1400	1800	200	1000	1600	2000
MAX_FEATURES	auto	sqrt	auto	sqrt	auto	auto
MAX_DEPTH	80	10	110	10	80	70
MIN_SAMPLES_SPLIT	2	10	5	5	5	4
MIN_SAMPLES_LEAF	4	1	2	2	4	2
BOOTSTRAP	true	true	true	true	true	true

Table B.1: Best set of hyperparameters for the Random Forest baseline model for every dataset at both granularities.

Hyperparameter	OpenStack commit	OpenStack file	QT commit	QT file	Adyen commit	Adyen file
C	1.0	1.0	1.0	1.0	1.0	0.9

Table B.2: Best value for the hyperparameter C for the logistic regression baseline model for every dataset, at both granularities.