University of Zurich

BACHELOR THESIS

A Machine Learning Approach to Predicting Developers' Behaviour and Build Results in Continuous Integration

Jonas Klass
(15-703-358)

supervised by
Prof. Dr. Alberto Bacchelli
Dr. Fabio Palomba
Carmine Vassallo
at
Department of Informatics

February 7, 2019

Acknowledgements

I would like to thank Prof. Dr. Alberto Bacchelli, Head of the Empirical Software Engineering Team, for giving me the opportunity to write this thesis. Furthermore, my thanks go to Dr. Fabio Palomba from the Empirical Software Engineering Team and Carmine Vassallo from the Software Architecture and Evolution Lab at the University of Zurich for their support and guidance throughout my thesis. I am grateful for the regular meetings and the expertise they provided me.

Abstract

Continuous Integration extended by Continuous Code Quality as a software development practice is a popular approach for providing Software Quality Assurance. One of the main shortcomings of this approach is that developers only learn about insufficient code quality after their changes have been built and analyzed. Therefore, researches examined different approaches to give Just-in-Time quality predictions. As no systematic overview of the topic is available, in this paper a Systematic Literature Review on the subject is performed. The review shows that these approaches work well and are usually based on Machine Learning classifiers trained with the data of projects' change histories.

To learn more about developers' behaviour in Continuous Integration, a study utilizing the change histories of projects using Continuous Integration is conducted. For this purpose, different Machine Learning classifiers are trained with the data from the change histories. The study shows that prediction models for the behaviour of developers regarding continuous quality control and for the build status on the build server work well. Further, the results highlight the need for suggestion methods when code quality checks need to be performed.

Zusammenfassung

Just-in-time Qualitätsvorhersagen, die auf der Änderungshistorie eines Projekts basieren, sind immer weiter verbreitet. Ziel dieses Verfahrens ist, dem Entwickler einen Anhaltspunkt bezüglich der Qualität der zu implementierenden Änderungen zu geben. Eine genauere Recherche zu diesem Thema zeigt, dass mehrere Ansätze gut geeignet sind, um solche Vorhersagen zu machen, und dass maschinelles Lernen (Machine Learning) der übliche Weg ist, um solche Vorhersagen zu treffen. Das Konzept der kontinuierlichen Integration bietet Entwicklern eine Vielzahl an Qualitätsmessungen verschiedener Dimensionen. Eine Studie, welche die Änderungshistorie von Projekten mit kontinuierlicher Integration betrachtete, zeigt, dass Prognosemodelle für den Build-Status auf dem Build-Server und das Verhalten der Entwickler hinsichtlich der kontinuierlichen Qualitätskontrolle gut funktionieren.

Contents

List of Figures

List of Tables

1 Introduction

Software Quality Assurance (SQA) plays a crucial role when it comes to producing high-quality software. Therefore, a multitude of different methods exist to assure code quality. The traditional methods are usually on package or file level. This leads to problems when it comes to fixing a defect. It might not be clear which developer is responsible for the change resulting in a fault, and even if the defect can be assigned to a developer, the change may be long in the past and no longer present in their mind [Kamei et al., 2013]. Therefore a decrease in code quality should be detected as soon as possible.

A popular approach for solving this problem is Continuous Integration (CI). Continuous Integration is a software development practice with the goal of continuously integrating changes. Developers integrate their work frequently, for example daily. These changes are then built on a build server to ensure good code quality [Fowler, 2006]. CI has been shown to improve software quality, release frequency and predictability [Goodman and Elbaz, 2008], whilst also reducing risks [Duvall et al., 2007] and increases developer productivity [Miller, 2008] amongst other benefits.

Frequently, a dedicated build server is used for CI. Such a server takes commits as input and builds, tests and deploys them. Furthermore, code quality assessment tools can be used to monitor the quality of the software code continuously [Duvall et al., 2007]. In practice, a development pipeline (Figure 1) is often used to automate the Continuous Code Quality (CCQ) process. A possible workflow in the CCQ environment could look something like this: A developer commits his changes to a version control system, triggering a build on the dedicated server. This server then sends the data to the CCQ service, where a quality analysis is conducted. Thereafter, the CCQ service sends the results of the analysis back to the build server. The build server then decides whether the build passes or fails. This is only one of many possible ways to use such a pipeline. Due to various reasons, only specific commits are sent to the CCQ service in practice.

The work of Vassallo et al. [2018] shows that CCQ is not widely adopted which is why researchers have been looking into faster and easier alternatives. Imagine a scenario where a developer gets a prediction of the code quality of a to be integrated change. The developer gets the chance to review the code before integrating it, or he can get the assurance that the code is indeed of good quality. In recent times, several prediction models for such an approach have been developed [D'Ambros et al., 2012, Zimmermann and Nagappan, 2008, Turhan et al., 2009]. In this paper, the focus lies on predictions that are made just-in-time (JIT), or on a change level as some papers formulate it. The goal hereby is to predict the code quality as soon as changes are made. This gives developers the chance to improve their work while it is still fresh in their minds.

In this paper, a systematic literature review on JIT approaches is conducted, as no systematic overview of the topic exist. The review aims to find out which JIT approaches already exist and how they work. In a second step, a study is conducted which examines the predictability of developer behaviour in CI. More precisely the predictability of developers' behaviour in inspecting builds for code quality and their behaviour in refactoring builds' changes is analysed. Apart from running quality checks, the build takes a long time as such. Therefore, the predictability of the outcome of a build before actually running it is examined. This could save developers time and would prevent them from breaking the development pipeline. The study utilises data extracted from CCQ pipelines to train different Machine Learning classifiers.

The thesis follows the following structure. First, the systematic literature review is conducted under the section related work. Then the approach and methodology of the study are explained before the results are presented, followed by a discussion thereof. In the end, a summary of the thesis is given.

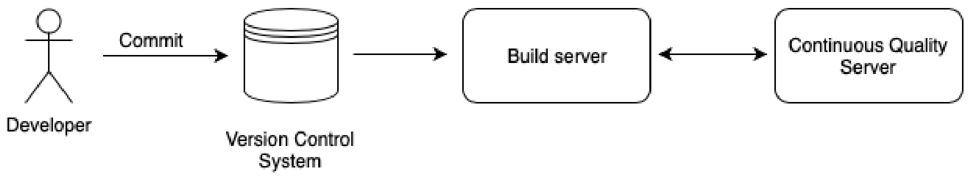

Figure 1: CCQ Pipeline

2 Related Work

For the related work section, a systematic literature review (SLR) on techniques for Just-In-Time (JIT) code quality assurance is conducted. The review follows the guidelines proposed by Kitchenham and Charters [2007]. As stated in their guide, the review will be performed in three stages: planning, conducting and reporting. Further snowballing is used to identify more relevant papers. The snowballing follows the guide by Wohlin [2014]. The goal of the SLR is to answer the research questions stated in Table 1.

Research Question	Motivation
RQ1: Which just-in-time quality assurance tasks are supported?	A typical CCQ pipeline takes care of many different quality assurance tasks. Depending on the configuration of the various tools used, developers have access to a variety of different quality metrics. This includes the number of bugs, the identification of code smells, the complexity of the source code, the proneness to attacks, the need for refactoring and more. The goal behind RQ1 is to identify which of these metrics or combination thereof are supported by the existing JIT approaches.
RQ2: How are JIT techniques applied?	There are different possible approaches for building a JIT quality recommendation system. The goal of this second research question is to detect what techniques are being used. Further, the question aims to find out what data is used by the approaches. In case of a Machine Learning approach, it would be useful to examine whether the model is trained on a single project, and therefore only applies to that project, or if the model is applicable across different projects.
RQ3: How did the researchers validate their predictions?	An approach is only useful if the performance is adequate. The goal of the third research question is to find out how the performance is measured and what validation techniques are being used.

Table 1: Research Questions

2.1 Research Methodology

After stating the research questions (Table 1), the search terms (Table 2) were identified. The main search terms were directly derived from the research question and alternative spellings and synonyms were collected to complete the final search query. The alternative spellings and synonyms were connected with a logical "OR" and the search terms with a logical "AND". This query was then used to extract papers from the stated sources. After application of the inclusion/exclusion criteria, an initial set of papers was retrieved.

Search Term	Synonyms
Just-in-Time	("just-in-time" OR "JIT" OR "live" or "in time" OR "change level" OR "commit level")
Code Quality Assurance	("QA" OR "bugs" OR "smells" OR "vulnerabilities" OR "defects" OR "software quality assurance" OR "SQA")

Table 2: Search Terms for the Systematic Literature Review

(("just-in-time" OR "JIT" OR "live" or "in time" OR "change level" OR "commit level") AND ("QA" OR "bugs" OR "smells" OR "vulnerabilities" OR "defects" OR "software quality assurance" OR "SQA"))

Table 3: Search Query for the Systematic Literature Review

2.1.1 Resources to Be Searched

- IEEE Xplore Digital Library (`https://ieeexplore.ieee.org`)
- ACM Digital Library (`https://dl.acm.org`)
- ScienceDirect (`https://www.sciencedirect.com`)
- SpringerLink (`https://link.springer.com`)

2.1.2 Exclusion Criteria

The following criteria were used to select papers that need to be excluded.

- articles that were not written in English
- articles, which were not available in a full-text version
- articles which are not providing JIT quality predictions

2.1.3 Inclusion Criteria

Papers that fulfilled these criteria were included in the SLR.

- all articles written in English reporting techniques for just-in-time quality assurance
- articles suggesting improvements of other articles

2.2 Snowballing

The snowballing procedure followed the guide by Wohlin [2014]. The set of initial papers was then used as the starting set for the snowballing process, and one iteration of backward and one iteration of forward snowballing was conducted.

Resource	Number of Results
IEEE Xplore Digital Library	1,239
ACM Digital Library	146,263
ScienceDirect	637,991
SpringerLink	63,485

Table 4: Number of Results Found for Each Resource

After applying the inclusion as well as the exclusion criteria and merging the articles from the snowballing, the following final set of papers is included in the SLR.

1. A large-scale empirical study of just-in-time quality assurance [Kamei et al., 2013]
2. A Replication Study: Just-in-Time Defect Prediction with Ensemble Learning [Young et al., 2018]
3. An Empirical Study of Just-in-time Defect Prediction Using Cross-project Models [Fukushima et al., 2014]
4. Classifying Software Changes: Clean or Buggy? [Kim et al., 2008]
5. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction [Liu et al., 2017]
6. Deep Learning for Just-in-Time Defect Prediction [Yang et al., 2015]
7. Just-In-Time Bug Prediction in Mobile Applications: The Domain Matters! [Catolino, 2017]
8. Learning from Bug-introducing Changes to Prevent Fault Prone Code [Aversano et al., 2007]

9. Poster: Bridging Effort-Aware Prediction and Strong Classification - A Just-in-Time Software Defect Prediction Study [Guo et al., 2010]
10. Predicting risk of software changes [Mockus and Weiss, 2000]
11. Reducing Features to Improve Bug Prediction [Shivaji et al., 2009]
12. Revisiting common bug prediction findings using effort-aware models [Kamei et al., 2010]
13. Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction [Huang et al., 2017]
14. The prediction of faulty classes using object-oriented design metrics [El Emam et al., 2001]
15. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction [Yang et al., 2017b]
16. VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes [Yang et al., 2017a]

Dimension	Attribute Description
Type of code quality tasks	What code quality tasks are supported?
Used technique	What technique is used for the quality assurance? Machine Learning or something else?
Programming languages	What programming languages are supported?
Validation	How are the used techniques validated?
Limitations	What are the limitations of the used approach?

Table 5: Data Extraction Form

2.3 Data Extraction

The data extraction form (Table 5) was used to extract the relevant information from the papers and aggregate them in this section.

2.3.1 Type of Code Quality Tasks

The type of quality tasks supported is very general for most papers, as they only classify changes as either “free of defects” or “inducing a defect”. The specific type of defect is often not specified and some models are applicable to different kinds of defects. What can be observed is that almost all just-in-time quality assurance models are focused on predicting defects rather than vulnerabilities or smells (Figure 2). There is an explanation for this. It is much harder to identify code changes that induced a vulnerability or a smell than it is to identify a change that caused a bug. This definitely is a way to improve existing approaches, by adding more detail to the prediction and the capability to predict not only defects but also smells and vulnerabilities.

There is one significant distinction that can be made when looking at JIT quality assurance models. The models are either applicable across project or only within a particular project. This makes a difference regarding how these JIT approaches are used for quality assurance. Most of the approaches require training data. For approaches that only work within a project, JIT quality assurance would thus only be available later in the project after a sufficient amount of changes have been made. Cross-project models use data from other projects and are therefore more flexible in their application.

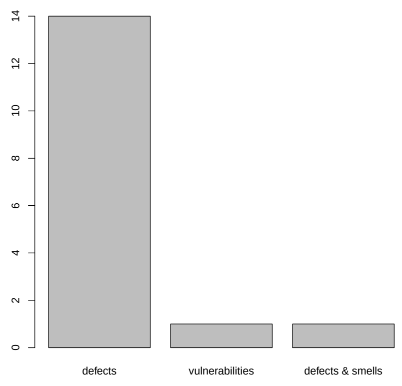

Figure 2: Different Types of Supported Quality Assurance Tasks

2.3.2 Used Techniques

All papers included in the SLR used some form of Machine Learning approach. On a very general level, the approaches used by the different papers for providing JIT quality assurance are roughly the same and can be described as follows. A dataset of changes, where each change is classified as either “free of defects” or “defect inducing”, is extracted from a project.

These classified changes are then used to train a Machine Learning classifier, which in turn can be applied to changes which are yet to be classified.

One of the main differences between the different approaches used is the way they mine a dataset of classified changes, which can be used as a training set. A few approaches used the same dataset, namely the dataset mined by Kamei et al. [2013]. They used the SZZ algorithm. The SZZ goes through the changes in a code history and identifies changes that likely induced a defect. In order to do that, the SZZ algorithm starts analysing bug-fixing changes. The algorithm then compares the bugfixing change with the previous change and identifies what part of the code caused the bug. This code is then traced back to the edit that introduced this code. This is the bug-inducing change [Śliwerski et al., 2005, Neto et al., 2018]. This procedure or some variation of it was also used by other papers. Yang et al. [2017a] used the approach to mine a dataset of vulnerability inducing changes from the change history. For this kind of approach to be useful the projects are required to have a substantial history of changes. Therefore new projects cannot profit from this kind of JIT quality assurance.

The papers also differ in the Machine Learning classifier trained on the retrieved data. Kamei et al. [2013] use logistic regression as a classifier on their dataset. Several papers suggested other classifiers for the dataset or a subset of the original dataset. For example Yang et al. [2015] suggest a deep learning approach for the dataset and found that this classifier produces statistically significantly better results than the original classifier. Some papers simply use a single Machine Learning classifier, but the more sophisticated methods are shown to produce better results. These ensemble learning approaches use a combination of Machine Learning classifiers. For example Yang et al. [2015] introduced “Deeper” and Yang et al. [2017b] introduced “TLEL”, a two-layer ensemble learner where classification trees and bagging are used to build a random forest.

2.3.3 Supported Programming Languages

Most of the applied models can be used for several programming languages, as the change measures used to make the predictions do not depend on the programming language of a project. Only the history of the changes is needed. One frequently used dataset is the one created by Kamei et al. [2013]. It includes projects written in Java and C/C++. Some approaches are only applicable to projects in a particular programming language, e.g the paper by El Emam et al. [2001], which uses object-oriented metrics from Java.

2.3.4 Validation Techniques

The validation is a critical aspect of all the approaches. Most papers use common metrics to validate their models. The most important metrics can be derived from the confusion matrix.

- accuracy
- precision
- recall
- F1-measure

The validation of the papers' various approaches shows that there is no common best fit, but rather a most suitable approach for every individual project. There is also the tradeoff between precision and recall. Depending on what is considered more important for a given project a different classifier might be chosen. For example, a classifier with high recall produces many false alarms, which is potentially annoying for a developer, so precision is more important from this point of view. But if it is absolutely crucial that all defects are detected, one has to cope with the lower precision and the higher recall.

2.3.5 Limitations

A common limitation of all approaches is that they only deliver a prediction of the existence of a defect and not the severity or the type of the defect.

Furthermore, all the approaches are limited by the number and different kinds of projects they use. If the projects considered are very diverse, a potential bias can be reduced but never completely eliminated. The problem of the potential bias is universal validity. The fact that an approach produces good classifications in terms of performance for one kind of project does not guarantee that it works well for every single project. Also, the number of features is limited, and while all approaches have good argumentation as to why they regard or disregard a particular feature, it is hard to tell which features are most suitable in general.

2.4 Results

With the extracted data and summary of the extraction, the research questions posed in the beginning of the SLR can now be answered.

2.4.1 Supported Quality Assurance Tasks

The goal of the first research question was to find out what quality assurance tasks are supported. The SLR showed that there are some promising approaches for identifying fault or defect-inducing changes. However, the techniques fail to provide the same broad spectrum of quality assurance metrics that a typical CCQ pipeline would. Most research focuses on delivering JIT predictions for a single quality assurance task. In practice, to provide a developer with good quality assurance metrics, a combination of different approaches would have to be used.

2.4.2 Application of JIT Techniques

The second research question was aimed at finding out how JIT techniques are applied. The SLR clearly shows that a Machine Learning approach is the most popular method of providing developers with JIT software quality predictions. While the initial costs of these approaches in terms of computation time are significant, predictions with the trained model can be made quickly, delivering genuine JIT quality predictions. Since the Machine Learning approaches need a training dataset, it is worth looking at the techniques used to create training datasets. For the creation of the training datasets, the approaches rely on the projects' change histories and the classification thereof.

2.4.3 Used Validation Techniques

For RQ3 the goal was to examine how the researchers validated their approaches. Since all the papers used some sort of Machine Learning classifier, the question of the validation of the technique can be regarded as the question of how the Machine Learning classifiers were validated. The SLR concluded that the most popular metrics used to validate the classifiers are the metrics calculated from the confusion matrix.

3 Study Setup and Methodology

3.1 Study Setup

The study aims to use the data extracted from CCQ pipelines to predict whether a commit will fail or pass once it gets built on the build server. JIT predictions of the build status are useful information, as each build takes time and the status information of the build reaches the developer only after some time. With a prediction model, a prediction can be made instantly. Further, the study analyzes the behaviour of developers, with regards to controlling code quality on the CCQ server and refactoring. The goal of the study is to answer the following research questions:

- **RQ1** How well can the developers' behaviour in inspecting builds for code quality be predicted?

  Vassallo et al. [2018] found that code quality inspection is performed every 18 builds and often at the end of a sprint. The motivation behind RQ1 is to find out, whether this behaviour can be predicted, and if it can be explained.

- **RQ2** How well can the developers' behaviour in refactoring builds' changes be predicted?

  The motivation behind RQ2 is to find out - in situations where a developer decided to refactor for a specific commit and to gain some indications - what might have led the developer to the decision to refactor.

- **RQ3** How well can the build outcome be predicted before running the build?

  For a developer, it is critical to know whether a build passes or fails because if the build were to fail, the developer has to improve his work until the build passes. RQ3 aims to find out if the outcome of the build can be predicted, so the developer would know more or less instantly if his work needs improvement.

3.1.1 Datasets

Considered in this study are Java projects that use GitHub as VCS and TravisCI as the build server. The projects also had to be using SonarCloud, which is a cloud service provided by SonarQube. The histories of the CCQ data and the build data were mined by Vassallo et al. [2018]. The projects vary in size and for each project, only the commits triggering a build within the period of the project being hosted both on GitHub, TravisCI and SonarQube were considered. There were datasets constructed for 45 projects. For each project, the dataset was extended by checking the commit, which triggered the build on the build server. For these commits, the features from the VCS were extracted. To extract these features the python framework PyDriller [Davide, 2019] was used. For each build-triggering commit, the refactoring features were added as additional features. To extract the refactoring features the RefactoringMiner [Tsantalis, 2019] was used.

3.1.2 Features

The full list of features can be seen in Table 6. As illustrated in the table, the features have different dimensions.

The following **quality** features were extracted from the CCQ server. The bugs detected by SonarQube are either wrong code or code that is not fulfilling its intended purpose. Language detection is used to automatically identify the used programming language and invoke the corresponding analyzer. The feature used is the number of bugs found by this analzser [Det]. The complexity feature is the complexity found by cognitive complexity analyzis. Cognitive complexity analysis is an improved form of the cyclomatic complexity analysis.

The **build** features were mined from the build server and include the status, whether a build passed or failed, the duration it took to build the commit and the event type that triggered the build.

The features extracted from the VCS follow the approaches of Kamei et al. [2013] and Fukushima et al. [2014], with some deviations.

The **size** features were included, because prior work by Nagappan and Ball [2005] showed that larger changes are more likely to introduce defects.

Changes with the purpose of fixing a defect are more likely to introduce a new defect than other changes [Graves et al., 2000], for that reason the **purpose** feature was added to the list of features.

The **history** feature is the number of developers that previously changed the files subject to change.

The **experience** features are calculated in the same way as Kamei et al. [2013] suggested. The features were calculated according to the following formulae. The experience is the number of commits made by a developer before the current change. The recent experience was calculated similarly but the changes were weighted by their age. The following formual is used: $\frac{1}{(n+1)}$, n is time measured in years. Prior research showed that developer experience has a significant effect on the software quality [Mockus and Weiss, 2000].

Features of the **refactoring** dimension include the number of refactorings for each of 21 different types of refactoring. From these refactoring

Source	*Dimension*	*Name*	*Definition*
CCQ server	quality	BUG	bugs
		CS	code smells
		COMP	complexity
Build server	build	STA	status: passed or failed
		DUR	duration: time it took to build
		ETY	event type: e.g. “push”
VCS	diffusion	NF	number of modified files
		NU	number of methods
		CYC	cyclomatic complexety
	size	LA	lines of code added
		LD	lines of code deleted
		LT	lines of code before changes
	purpose	CT	change type, e.g. “modify”, “fix”
	history	NDEV	number of developers
	experience	EXP	general experience
		REXP	recent experience
		SEXP	experience on subsystem
	refactoring	REF	refactoring

Table 6: List of Features

features a single feature was created, which is a binary indicator if there were any refactorings performed in a specific commit.

3.2 Fitting the Machine Learning Classifier

This section explains how the Machine Learning classifiers were fitted to the data with the three different classification goals.

1. Predict the status of the build: The goal of this prediction is to give the developer an indication if a build is going to pass or fail.
2. Predict CCQ decision: This classifier gives a binary indication about whether the developer thinks it is useful to send a build to the CCQ server or not.
3. Predict refactoring decision: This classifier gives a binary indication about whether the developers think refactoring is needed for a specific commit.

For each of these three goals, three different classifiers were fitted.

3.2.1 Data Preparation

First, the data was prepared for the different classification goals. Depending on what the goal of the prediction is, the features can take different roles. For the prediction of the build status, only the feature from the VCS can be used and the build status is the target. For predicting whether refactoring is needed all the features can be used and the target is the refactoring. For the prediction, if the build will be sent from the build server to the CCQ server for more inspection all the features from the build server and the VSC features are used and the target is the existence of quality checks. The individual datasets for the specific goals are then scaled. Because the features vary in range, magnitude and units, scaling is important for the algorithms to work. To get rid of imbalances in the dataset SMOTE (synthetic minority over-sampling technique) is used. The data is imbalanced if a substantial majority of the training data falls under one classification. The predictive accuracy is not an appropriate indicator of performance if the data is imbalanced [Chawla et al., 2002]. SMOTE adds synthetic values for the class that is underrepresented. A synthetic datapoint is created by determining the distance between a feature vector of a given data point and his nearest neighbour and multiplying it by a random value between zero and one. This gives a point on the distance between the data point and its nearest neighbour. This data point then is added to the dataset as a synthetic data point. This process is repeated for each data point and its feature vector.

Feature selection is used to avoid overfitting of the model, but also for other advantages, such as easier understanding of the model, shorter training times and defying the curse of dimensionality to improve prediction performance [Guyon and Elisseeff, 2003].

3.2.2 Fitting a Random Forest Classifier

A random forest is an ensemble of multiple decision trees. Each node gets split by a random subset of the features. This ensures that the trees have low correlation between each other. For classification, the input runs through all the decision trees, and the resulting classification is the classification that appears most in the classification of the decision trees [Breiman, 2001]. For the configuration of the model parameters grid search is used. This gives the best possible configuration of the parameters.

3.2.3 Fitting a Logistic Regression

The logistic regression model uses a logistic function (sigmoid function) to assign a value between zero and one. This value can then be used to make a binary classification, so all values above 0.5 are assigned to one classification, while observations with a value below 0.5 are assigned to the other

		In Reality	
		True	False
Classified as	True	True Positive	False Positive
	False	False Negative	True Negative

Table 7: Confusion Matrix

classification [Harrell, 2015]. Also for the logistic regression grid search was used to find the best parameter configuration.

3.2.4 Fitting a Neural Network

The classifier used is a multilayer perceptron (MLP), a feed-forward artificial neural network. It consists of a minimum of three layers, namely input, hidden layer and output layer. It is possible to have multiple hidden layers [Hansen and Salamon, 1990]. To achieve the ideal configuration of the number of these layers and to find the best parameters, grid search was used. The model with the best parameters is then used to make the predictions.

3.3 Evaluating the Classifier

The following evaluation scores were used to evaluate the classifier. The data was split into training and testing data. The training data was used to train the classifier. The trained classifier was used to predict the testing data, which could then be compared to the actual values. From this comparison, the following evaluation metrics were calculated. For the classification, four possible outcomes exist: Correctly classified commits, which can be true positive and true negative. For the false classified commits, there are also two possible outcomes. The false positive classified and the false negative classified. For an illustration see the confusion matrix in Table 7.

3.3.1 Accuracy

The accuracy is calculated as the number of correctly classified commits out of the total number of commits. Accuracy has to be interpreted quite carefully because it can be misleading. If an outcome appears much more often than the other outcome, then classifying everything as that outcome has high accuracy, but is obviously not a good model.

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{True Positive} + \text{False Positive} + \text{True Negative} + \text{False Negative}}$$

3.3.2 Recall

The recall score is the ratio of the correctly predicted, meaning the true positives over the actual positive values.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

3.3.3 Precision

The precision is the correctly positive classified commits over all positive classified commits. As can be followed from the formulas there is a trade off between recall and precision.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

3.3.4 F1 Measure

The F1 measure is the harmonic mean of the precision and the recall.

$$\text{F1 - Measure} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

The perfect value for all these evaluation metrics is one, and only values between zero and one are possible. A value close to one is desirable.

4 Results

The following section provides an overview of the results found by the study and answers the research questions.

4.1 Predictability of Developers' Behaviour in Inspecting Builds for Code Quality

To answer RQ1, three different prediction models for predicting whether a developer will perform code quality controls were fitted. The prediction models were used on a test set and the resulting evaluation scores thereof can be seen in Table 8. As can be seen in the table, developers' behaviour can be predicted accurately. The accuracy and recall scores are high for all three prediction models. The precision scores are lower than the other evaluation scores across all models. However, this does not mean that the model is useless since the recall score is high.

When looking at the average gini importance (Figure 3) from the random forest classifier throughout the projects, we can see that the duration of the build seems to have a strong influence on the prediction. Checking the distribution of the importance of the duration feature with a boxplot shows that the data is heavily skewed and that the feature importance for the duration was much higher for a few projects only. This shows that duration still is an important feature, but not as important as it might seem at first glance. Further, it can be observed that the size features and the experience features have a higher influence on the prediction. While experience overall has a strong influence on the prediction, the recent experience seemingly has more influence than the specific experience. Less important are the diffusion features. This implies that a larger number of small changes in many different subsystems is less likely to be checked for quality, than a large change in one single subsystem. Generally, the model suggests that more experienced developers tend to perform code quality checks for larger builds. While experience overall has a strong influence on the prediction, the recent experience seemingly has more influence than the specific experience.

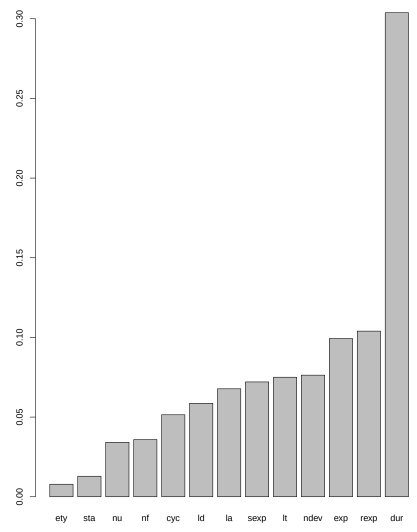

Figure 3: Average Gini Feature Importance for Quality Check Performing Prediction

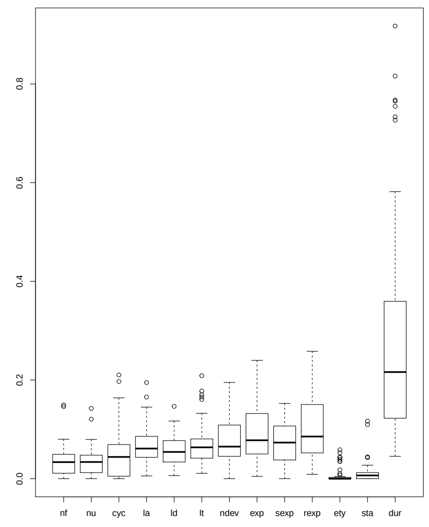

Figure 4: Boxplot of Gini Feature Importance for Quality Check Performing Prediction

	Random Forest				Neural Network				Logistic Regression			
Project	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Accuracy*	*Precision*	*Recall*	*F1 Score*
org.codehaus.sonar-plugins_sonar-branding-plugin	0.8	0.37	1	0.54	0.6	0.17	0.78	0.31	0.4	0.16	1	0.28
drewboswell_converge	0.92	0.87	0.96	0.92	0.9	0.84	0.96	0.91	0.94	0.91	0.96	0.94
com.impossibl.pgjdbc-ng_pgjdbc-ng	0.75	0.63	1	0.77	0.72	0.61	1	0.76	0.71	0.6	1	0.75
io.trane_future	0.91	0.44	0.98	0.62	0.55	0.14	1	0.25	0.3	0.09	0.94	0.17
underscore	0.86	0.26	0.33	0.4	0.81	0.21	0.33	0.33	0.71	0.14	0	0
de.rinderle.softvis3d_softvis3d	0.87	0.57	0.76	0.72	0.89	0.64	0.84	0.77	0.83	0.51	0.81	0.67
org.jmxtrans_jmxtrans-parent	0.81	0.5	0.92	0.67	0.81	0.42	0.67	0.59	0.75	0.37	0.75	0.55
org.leandreck.endpoints_parent	0.76	0.48	0.71	0.63	0.84	0.6	0.79	0.73	0.76	0.48	0.71	0.63
io.github.mbarre_schemacrawler-additional-lints	0.83	0.76	0.82	0.82	0.74	0.66	0.73	0.73	0.52	0.49	0.45	0.48
swellaby_generator-swell_old	0.73	0.28	0.67	0.44	0.78	0.27	0.5	0.43	0.55	0.19	0.58	0.3
org.pitest_pitest-parent	0.72	0.23	0.5	0.38	0.86	0.44	0.67	0.62	0.64	0.23	0.67	0.38
org.sonarsource.xml_xml	0.99	0.85	1	0.92	0.89	0.23	1	0.37	0.83	0.15	0.98	0.27
org.codehaus.sonar-plugins_sonar-build-stability-plugin	0.86	0.26	0.99	0.42	0.68	0.12	0.92	0.22	0.68	0.12	0.92	0.22
org.sonarsource.update-center_sonar-update-center	0.98	0.46	0.98	0.63	0.98	0.47	0.98	0.64	0.81	0.09	0.98	0.17
com.redhat.lightblue.hook_lightblue-audit-hook	0.96	0.74	0.7	0.82	0.96	0.74	0.7	0.82	0.98	0.82	0.8	0.89
com.thoughtworks.xstream_xstream-parent	0.79	0.58	0.82	0.72	0.74	0.52	0.82	0.67	0.68	0.47	0.82	0.62
wirecard-paymentSDK-php	0.87	0.32	0.61	0.52	0.95	0.64	0.87	0.78	0.77	0.22	0.65	0.39
socket.io	0.72	0.13	0.67	0.25	0.86	0.21	0.67	0.4	0.84	0.19	0.67	0.36
camflow_dev	0.98	0.72	0.95	0.84	0.89	0.34	0.98	0.51	0.75	0.09	0.54	0.2
org.apache.cayenne_cayenne-parent	0.85	0.74	0.86	0.83	0.91	0.85	0.86	0.89	0.79	0.66	0.86	0.77
org.almrangers.auth.aad_sonar-auth-aad-plugin	0.77	0.24	0.85	0.4	0.82	0.33	1	0.5	0.74	0.21	0.81	0.36
com.github.gantsign.errorprone_error-prone-checks	0.81	0.14	0.5	0.29	0.73	0.11	0.5	0.22	0.69	0.1	0.5	0.2
net.sourceforge.pmd_pmd	0.65	0.11	0.69	0.21	0.88	0.13	0.38	0.29	0.69	0.07	0.38	0.14
org.sonarqubecommunity.buildbreaker_sonar-build-breaker-plugin	0.84	0.45	1	0.62	0.81	0.35	0.81	0.53	0.74	0.27	0.78	0.45
xtl	0.64	0.13	0.33	0.19	0.85	0.22	0.33	0.36	0.53	0.12	0.33	0.15
net.masterthought_cucumber-reporting	0.86	0.28	0.77	0.48	0.92	0.39	0.69	0.6	0.61	0.11	0.62	0.21
es.usc.citius.hipster_hipster-pom	0.84	0.43	0.71	0.61	0.93	0.68	0.93	0.81	0.89	0.59	0.93	0.74
org.sonarsource.clover_sonar-clover-plugin	1	0.88	1	0.94	0.91	0.25	1	0.41	0.78	0.13	0.99	0.22
numdifftools	0.82	0.92	0.74	0.85	0.84	0.93	0.77	0.87	0.84	0.93	0.77	0.87
dolphin-platform-js_master	0.78	0.32	0.9	0.5	0.89	0.49	0.9	0.67	0.61	0.18	0.7	0.31
com.mpatric_mp3agic	0.97	0.18	0.5	0.4	0.97	0.18	0.5	0.4	0.95	0.11	0.5	0.29
com.github.katari_k2	0.79	0.9	0.76	0.84	0.93	0.94	0.95	0.95	0.89	0.9	0.95	0.93
org.codehaus.sonar-plugins_sonar-crowd-plugin	0.97	0.64	1	0.78	0.82	0.2	0.95	0.34	0.82	0.21	0.95	0.35
org.demoiselle.jee_demoiselle-build	0.79	0.15	0.33	0.27	0.79	0.15	0.33	0.27	0.81	0.16	0.33	0.29
org.sonarsource.clirr_sonar-clirr-plugin	0.97	0.58	1	0.73	0.87	0.21	0.94	0.36	0.72	0.11	0.94	0.21
org.codehaus.sonar-plugins_sonar-doxygen-plugin	0.83	0.31	1	0.48	0.75	0.22	0.94	0.37	0.49	0.12	0.95	0.22
com.twitter.ambrose_ambrose	0.97	0.44	1	0.62	0.96	0.4	1	0.57	0.95	0.33	1	0.5
org.sejda_sejda-parent	0.83	0.44	0.83	0.62	0.73	0.3	0.72	0.47	0.67	0.22	0.56	0.36
com.squareup.okio_okio-parent	0.91	0.42	0.73	0.62	0.94	0.51	0.73	0.7	0.8	0.27	0.82	0.45
socializa	0.92	0.2	0.6	0.4	0.98	0.62	0.6	0.75	0.93	0.22	0.6	0.43
com.github.checkstyle_checkstyle-sonar-plugin-parent	0.84	0.77	0.83	0.83	0.8	0.72	0.83	0.8	0.51	0.48	0.66	0.56
Average	0.85	0.47	0.79	0.60	0.84	0.43	0.78	0.56	0.73	0.31	0.74	0.42

Table 8: Quality Prediction Results

4.2 Predictability of Developers' Behaviour in Refactoring Builds' Changes

To answer RQ2, three different prediction models for predicting whether a developer will perform refactoring were fitted.

This research question could not be answered as there were not enough datasets containing data points with refactoring. Almost all projects were completely imbalanced, meaning that none of the built commits contained any form of refactoring. For the two projects for which predictions could be made (see Table 9) the accuracy score was very high. This is not surprising as the datasets are strongly imbalanced. When looking at the other evaluation score it can be seen that the models do not perform well. Also from only testing the prediction on two projects no conclusions can be drawn.

	Random Forest				**Neural Network**				**Logistic Regression**			
Project	*Accuracy*	*Precision*	*Recall*	*F1 Fcore*	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Accuracy*	*Precision*	*Recall*	*F1 Score*
com.restfb_restfb	0.93	0.29	0.5	0.5	0.96	0.54	0.5	0.67	0.93	0.07	0	0
org.jmxtrans_jmxtrans-parent	0.94	0.06	0	0	0.94	0.5	1	0.67	0.94	0.06	0	0
Average	0.94	0.18	0.25	0.25	0.95	0.52	0.75	0.67	0.94	0.07	0	0

Table 9: Refactoring Prediction Results

4.3 Predictability of the Build Status

To answer RQ3 three different prediction models for predicting the build status were fitted. The trained prediction models are then used on a test set and the evaluation scores are calculated (see Table 10). The table shows the four evaluation scores for each classification model and project, as well as the average evaluation scores over all the projects. For the prediction whether a build is going to fail, the random forest classifier and the neural network have similar performances according to the evaluation scores derived from the confusion matrix. The logistic regression classifier performs well, but especially the recall is worse compared to the other two models. When looking at the average evaluation scores for all the projects, it can be seen that on average all models work well.

The model would be used to save developers time. It is not worth waiting for a build to finish only to learn that it failed anyway. Equally bad would be not to build a commit based on a prediction that it is going to fail when in reality it would not have failed. Therefore, it is crucial for the model to have a good recall score. Since the recall score is high for all three models, the prediction models could be useful in practice.

In Figure 5, the average gini importance extracted from the random forest classifier over all projects can be seen. The size features and the experience features have the highest impact on the prediction. Specific experience does not seem to be as important as overall experience and recent experience for predicting the build status of a build. The less important features are the diffusion features, for example the number of modified files.

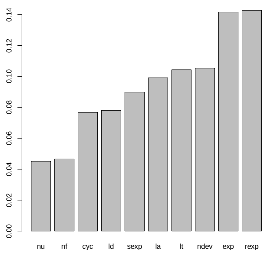

Figure 5: Average Gini Feature Importance for the Build Prediction

	Random Forest				Neural Network				Logistic Regression			
Project	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Accuracy*	*Precision*	*Recall*	*F1 Score*
drewboswell_converge	0.96	0.99	0.95	0.97	0.94	0.98	0.92	0.96	0.86	0.93	0.84	0.9
com.impossibl.pgjdbc-ng_pgjdbc-ng	0.94	0.99	0.93	0.96	0.87	0.96	0.89	0.93	0.84	0.95	0.84	0.9
io.trane_future	0.13	0.97	0.09	0.17	0.13	0.97	0.09	0.17	0.1	0.96	0.06	0.12
underscore	0.9	0.6	1	0.75	0.95	0.75	1	0.86	0.9	0.49	0.67	0.67
de.rinderle.softvis3d_softvis3d	0.99	0.99	1	1	0.99	0.99	1	1	0.99	0.99	1	1
org.jmxtrans_jmxtrans-parent	0.71	0.92	0.74	0.82	0.88	0.97	0.89	0.93	0.66	0.93	0.66	0.78
org.leandreck.endpoints_parent	0.9	0.98	0.91	0.95	0.94	0.98	0.96	0.97	0.9	0.98	0.91	0.95
io.github.mbarre_schemacrawler-additional-lints	0.91	0.95	0.95	0.95	0.96	1	0.95	0.98	0.78	0.99	0.77	0.87
swellaby_generator-swell_old	0.93	0.97	0.96	0.96	0.9	0.97	0.93	0.95	0.89	0.97	0.91	0.94
org.pitest_pitest-parent	0.69	0.87	0.78	0.82	0.86	0.89	0.97	0.93	0.58	0.88	0.62	0.73
org.sonarsource.xml_xml	0.79	0.96	0.8	0.88	0.79	0.96	0.8	0.88	0.78	0.96	0.8	0.87
org.sonarsource.update-center_sonar-update-center	0.62	0.98	0.6	0.75	0.62	0.98	0.6	0.75	0.61	0.98	0.59	0.74
com.redhat.lightblue.hook_lightblue-audit-hook	0.98	0.99	0.99	0.99	0.99	0.99	1	0.99	0.99	0.99	1	0.99
com.thoughtworks.xstream_xstream-parent	0.85	0.97	0.88	0.92	0.97	0.97	1	0.99	0.44	0.96	0.45	0.61
wirecard-paymentSDK-php	0.77	0.92	0.81	0.87	0.84	0.92	0.89	0.91	0.75	0.93	0.79	0.85
org.apache.cayenne_cayenne-parent	0.94	0.5	1	0.67	0.97	0.67	1	0.8	0.97	0.67	1	0.8
com.github.gantsign.errorprone_error-prone-checks	0.88	0.95	0.91	0.93	0.88	0.95	0.91	0.93	0.85	0.94	0.87	0.91
ch.mibex.bitbucket_sonar-bitbucket-plugin	0.96	1	0.95	0.98	0.98	1	0.98	0.99	0.96	1	0.95	0.98
net.sourceforge.pmd_pmd	0.83	0.94	0.87	0.9	0.82	0.94	0.86	0.9	0.74	0.93	0.77	0.84
xtl	0.68	0.55	0.7	0.65	0.68	0.55	0.55	0.59	0.68	0.56	0.75	0.67
net.masterthought_cucumber-reporting	0.8	0.98	0.8	0.88	0.92	0.96	0.96	0.96	0.55	0.95	0.56	0.71
org.sonarsource.clover_sonar-clover-plugin	0.93	0.87	0.97	0.92	0.93	0.87	0.97	0.92	0.93	0.87	0.97	0.92
numdifftools	0.91	0.98	0.89	0.94	0.86	0.92	0.89	0.91	0.73	0.85	0.77	0.82
dolphin-platform-js_master	0.71	0.75	0.88	0.81	0.75	0.78	0.88	0.83	0.69	0.8	0.7	0.76
org.demoiselle.jee_demoiselle-build	0.79	0.93	0.83	0.88	0.83	0.91	0.9	0.91	0.47	0.9	0.48	0.62
com.sparkjava_spark-core	0.72	0.94	0.75	0.83	0.92	0.95	0.96	0.96	0.78	0.94	0.82	0.88
org.sonarsource.clirr_sonar-clirr-plugin	0.84	0.86	0.77	0.85	0.84	0.86	0.77	0.85	0.84	0.86	0.77	0.85
org.codehaus.sonar-plugins_sonar-doxygen-plugin	0.63	1	0.63	0.77	0.63	1	0.63	0.77	0.63	1	0.63	0.77
org.sejda_sejda-parent	0.89	0.98	0.9	0.94	0.79	0.97	0.8	0.88	0.67	0.98	0.67	0.8
socializa	0.85	0.96	0.85	0.91	0.86	0.95	0.88	0.92	0.59	0.94	0.55	0.71
com.github.checkstyle_checkstyle-sonar-plugin-parent	0.9	0.98	0.91	0.95	0.93	0.98	0.95	0.96	0.89	0.99	0.88	0.94
Average	0.82	0.91	0.84	0.86	0.85	0.92	0.86	0.88	0.74	0.91	0.74	0.80

Table 10: Build Prediction Results

5 Discussion and Future Work

5.1 Implications

The following points highlighted by the results of the study are discussed further.

- **Performing quality checks**. The results show that developers perform quality checks for large changes in terms of size rather than diffusion. Vassallo et al. [2018] found that code quality checks are performed in intervals and at critical points. To this result can be added, that one of these critical points corresponds to the introduction of a change with large size features. Further, the results show that developers consider large changes regarding diffusion to be less critical and decide to skip quality checks for these changes. Prior work showed, that changes with high diffusion features are more likely to induce defects [Kamei et al., 2013]. Therefore, developers should probably change their regime for inspecting the quality of builds.
- **Time savings through build prediction**. The results show that the status of the build can be accurately predicted before it is actually run. Developers can reduce the number of times the development pipeline breaks and less time is spent on running builds which will fail. Prediction models can be a valuable assistance for developers working with CI.

5.2 Future Work

While the prediction of the build status works well and could potentially offer developers some useful assistance when working with CI, it would be possible to train further prediction models based on these types of datasets.

For example, the decision whether or not to perform a code quality analysis for a specific build could be supported by a prediction model. The practice of just checking the quality in regular time intervals or at some expected critical point is not ideal. Real world application showed that monitoring the quality of each build [Vassallo et al., 2018] is not very popular and therefore also not perfect. Future work could examine under which circumstances performing quality checks should be encouraged.

6 Threats to Validity

This section considers threats to the validity of the conducted study.

6.1 Replicability

This threat is addressed by providing all the used data and scripts. The material can be found on the attached CD (Appendix A).

6.2 External Validity

Although a large number of different projects are used, there are only datasets extracted from Java projects. These projects might not be representative of all projects. However, the limitation to one programming language should not affect the results much, as previous work has shown that these types of prediction models perform similarly across different programming languages [Kamei et al., 2013]. Other features not included in this study might have a significant effect on the prediction. Using more features could improve the results of the predictions.

6.3 Internal Validity

The classification for the purpose feature is based on the commit messages. Depending on how the commit messages of the individual projects are written, a faulty classification could be extracted.

The experience features are based on the number of commits a developer made, rather than the amount of time spent on a project. Both approaches have their advantages and disadvantages. For this paper, it was decided that the number of commits made is a better indicator of experience.

For the extraction of the refactoring features the RefactoringMiner library was used, which doesn't guarantee that all cases of refactoring are detected. What is more, the amount and types of refactorings performed were not considered.

7 Summary

Continuous Integration and Continuous Code Quality checks have increased in popularity over the recent years. The way developers use these methods in practice shows the need for faster just-in-time quality predictions. The systematic literature review revealed that these methods exist and are typically constructed by training a Machine Learning classifier on the change history of a project.

In a study using data extracted from projects that implement a CCQ pipeline, it was shown that the result of the builds on the build server can be well predicted. Further, a prediction model showed that more experienced developers decide to perform quality checks on larger builds. The results conclude that build prediction could potentially be a useful tool when working with CI. Finally, this paper highlighted the need to take the guesswork out of code quality checks.

A Attached CD

The attached CD contains all the necessary data and scripts to replicate the results of the study. The instructions to follow can be found in the *readme.md* file in the replication package.

References

Detect Software Bugs — SonarQube. https://www.sonarqube.org/features/issues-tracking/.

L. Aversano, L. Cerulo, and C. Del Grosso. Learning from Bug-introducing Changes to Prevent Fault Prone Code. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 19–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-722-3. doi: 10.1145/1294948.1294954.

L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct. 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324.

G. Catolino. Just-In-Time Bug Prediction in Mobile Applications: The Domain Matters! In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 201–202, May 2017. doi: 10.1109/MOBILESoft.2017.58.

N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002. ISSN 1076-9757. doi: 10.1613/jair.953.

M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-011-9173-9.

S. Davide. Python Framework to analyse Git repositories. Contribute to ishepard/pydriller development by creating an account on GitHub, Jan. 2019.

P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, June 2007. ISBN 978-0-321-63014-8.

K. El Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, Feb. 2001. ISSN 0164-1212. doi: 10.1016/S0164-1212(00)00086-8.

M. Fowler. Continuous Integration. page 14, May 2006.

T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An Empirical Study of Just-in-time Defect Prediction Using Cross-project

Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597075.

D. Goodman and M. Elbaz. "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation What Worked, How We Did it, and What Still Puzzles Us. In *Agile 2008 Conference*, pages 112–115, Aug. 2008. doi: 10.1109/Agile.2008.87.

T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000. ISSN 0098-5589. doi: 10.1109/32.859533.

P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, page 495, Cape Town, South Africa, 2010. ACM Press. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806871.

I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. page 26, 2003.

L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, Oct. 1990. ISSN 0162-8828. doi: 10.1109/34.58871.

F. E. Harrell. Ordinal Logistic Regression. In J. Harrell, Frank E., editor, *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*, Springer Series in Statistics, pages 311–325. Springer International Publishing, Cham, 2015. ISBN 978-3-319-19425-7. doi: 10.1007/978-3-319-19425-7_13.

Q. Huang, X. Xia, and D. Lo. Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170, Sept. 2017. doi: 10.1109/ICSME.2017.51.

Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept. 2010. doi: 10.1109/ICSM.2010.5609530.

Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, June 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.70.

S. Kim, J. E. J. Whitehead, and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2): 181–196, Mar. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70773.

B. Kitchenham and S. Charters. *Guidelines for Performing Systematic Literature Reviews in Software Engineering.* 2007.

J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19, Nov. 2017. doi: 10.1109/ESEM.2017.8.

A. Miller. A Hundred Days of Continuous Integration. In *Agile 2008 Conference*, pages 289–293, Aug. 2008. doi: 10.1109/Agile.2008.8.

A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, June 2000. ISSN 1538-7305. doi: 10.1002/bltj.2229.

N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, May 2005. doi: 10.1109/ICSE.2005.1553571.

E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390, Mar. 2018. doi: 10.1109/SANER.2018.8330225.

S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing Features to Improve Bug Prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 600–604, Auckland, New Zealand, Nov. 2009. IEEE. ISBN 978-1-4244-5259-0. doi: 10.1109/ASE.2009.76.

J. Śliwerski, T. Zimmermann, and A. Zeller. When Do Changes Induce Fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-123-8. doi: 10.1145/1082983.1083147.

N. Tsantalis. Contribute to tsantalis/RefactoringMiner development by creating an account on GitHub, Jan. 2019.

B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, Oct. 2009. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-008-9103-7.

C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall. Continuous Code Quality: Are We (Really) Doing That? page 6, 2018.

C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, pages 1–10, London, England, United Kingdom, 2014. ACM Press. ISBN 978-1-4503-2476-2. doi: 10.1145/2601248.2601268.

L. Yang, X. Li, and Y. Yu. VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–7, Dec. 2017a. doi: 10.1109/GLOCOM.2017.8254428.

X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, Aug. 2015. doi: 10.1109/QRS.2015.14.

X. Yang, D. Lo, X. Xia, and J. Sun. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, July 2017b. ISSN 0950-5849. doi: 10.1016/j.infsof.2017.03.007.

S. Young, R. University, T. Abdou, R. University, A. Bener, and R. University. A Replication Study: Just-in-Time Defect Prediction with Ensemble Learning. page 6, 2018.

T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 13th International Conference on Software Engineering - ICSE '08*, page 531, Leipzig, Germany, 2008. ACM Press. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368161.