A Comparison of Commit Validation Strategies

Lukas Bach

Institute for Program Structures and Data Organization (IPD)
Advisor: Dr.-Ing. Thomas Kühn

Commit Validation is an emerging topic in the area of Software Quality Assurance which can significantly reduce costs by decreasing development effort and faults by finding and fixing them when they are introduced. While many varying tools and research publications have appeared on this topic, a clear state of the art approach cannot be defined yet, and new methods continue to emerge. The goal of this paper is to explore how Commit Validation techniques can be compared in an objective way as well as reporting on how to find a suitable Commit Validation technique for a Software Engineering project. In an effort to satisfy this goal, five relevant Commit Validation approaches have been selected and analyzed: CLEVER, Commit Guru, Unusual Commits, Deeper and an approach by Kamei et al. Contributions of this paper include the proposal of an objective evaluation schema for Commit Validation methods, the analysis of the five approaches and an evaluation.

1 Introduction

1.1 Motivation

Software quality belongs to the many relevant topics of software engineering which often directly maps to costs and expenses. While good quality leads to well-maintainable code and reduced effort, software faults can lead to very expensive results. The US National Institute of Standards and Technology has estimated that software faults and failures cost the US economy $59.5 billion a year [TG02; RGS15].

Typical approaches focus on analyzing module artifacts, such as files or packages, and attempt to detect faults in those. However, this often introduces additional problems. Kamei et al. reported on typical drawbacks of those kinds of approaches [Kam+13]. They include that just identifying a faulty package still leaves much effort for the developer to find the actual location of the fault. Additionally it is generally a hard problem to find a responsible expert for an identified faulty code location as development teams tend to become large for large-scale projects. Finally, a primary problem is that typical approaches find faults only very late during the development cycle. Fixing those faults late is much more expensive than fixing them early on, after they were introduced [Mar08].

A recent approach for increasing software quality, and thus, decreasing costs, is the concept of Commit Validation. This concept is used at a very technical level where change commits on software projects are automatically analyzed and verified in regard to the probability that they introduce a new software fault. This is much more efficient than typical approaches because code is analyzed for faults at an early phase when the responsible developer is still involved in the changes [Kam+13].

Different approaches exist on this topic, one of the early works being a study on the topic by Kamei et al. [Kam+13], which was based on Just-in-Time Bug Detection on commit data. Similar approaches were developed by Yang and Rosen by introducing approaches called *Deeper* and *Commit Guru*, respectively [Yan+15; RGS15]. Goyal introduced a concept based on the rating of the abnormality of commits, called *Unusual Commits* [Goy+17]. Finally, one of the most recent approaches was introduced by Nayrolles et al. as *CLEVER* [Nay18], which not only detects faulty commits, but also automatically suggests code fixes. This paper will focus on those five approaches in more detail.

1.2 Problem Definition

Apart from these five approaches, there exist many more tools with varying techniques for performing Commit Validation. As this is an emerging topic with many published works in the past few years, a clear state-of-the-art approach has not been defined yet, and it is hard to choose a suitable Commit Validation technique for a new project to leverage its benefits.

The goal of this paper is to explore how Commit Validation techniques can be compared in an objective and fair way, and to define a method for choosing a fitting Commit Validation technique for a software engineering project.

To satisfy this goal, a suitable evaluation scheme will be introduced. Then a selection of five recent relevant technical approaches will be compared using the proposed evaluation scheme in an effort to give guidelines for determining which approaches are suitable in which context. Those five approaches are described in Sections 3.2 and 4.

For this paper, four success criteria have been defined: (1) The proposed comparison scheme does not take any considerations into account, which are not relevant for Commit Validation approaches; (2) the proposed comparison scheme accounts for the context which the compared approaches were designed for; (3) the comparison result of the compared approaches is specified in an explanatory way that helps readers recognize for which use case the approach is suitable; and (4) the paper provides guidelines for readers to find a suitable Commit Validation technique for their use cases.

1.3 Scope of This Paper

This paper focuses on Just-in-Time Fault Detection approaches, which have the concept of code commits in mind. While the research area for code quality is large, the focus restricts the area into a limited set of research works.

The Just-in-Time aspect and the relevance of commits is important to cope with the aforementioned drawbacks of typical quality assurance approaches, which include the costs of fixing bugs much later after they were introduced into the project. There exist various approaches which focus on detecting bugs and generating fixes such as *Getafix* by Bader et al. [Bad+19] or *iFixR* by Koyuncu et al. [Koy+19], however they are out of scope for this paper, as the concept of commits is not considered.

1.4 Outline

Section 2 introduces the background of Commit Validation, Just-in-Time Fault Detection and Prevention, as well as statistical measures used during the performance comparison. Section 3 specifies how Commit Validation approaches can be compared, while Section 4 describes the results from the comparison on the five selected approaches. A discussion of the results follows in Section 5. The paper concludes with an overview of related surveys on the topic in Section 6 and finally Section 7 concludes the paper.

2 Background on Commit Validation

This chapter introduces the topic of Commit Validation. First, a description of the underlying process, its target, and how it is implemented in a developers workflow is given. Then its two major components, Just-in-Time Fault Detection and Just-in-Time Fault Prevention are specified. Finally, some statistical measures are explained, which will be used later in this paper for comparing the methods' performances.

2.1 Commit Validation Process

There are many ways to increase software quality in the field of software engineering. While the field is very broad, there are many tools and technical utilities that have been established as part of a state-of-the-art technology stack. Among others, that also includes *version-control systems* (VCS). A version-control system is used to track the evolution of source code and enables teams of software developers to concurrently work on the same code base [CS14]. Currently, the most used VCS is *Git*, as shown in stackoverflow's developer survey in 2018, where 87.2% of $74,298$ participants reported that they were using Git for version control [Sta18].

An important concept of VCS are *commits*, small sets of code changes that usually happen atomically and are annotated by the developer with the commits purpose. Commits are explicitly performed by the developer and usually mark a finished feature, bug-fix, chore work or similar artifacts. Because of that, the time at which a developer performs a commit is a suitable time for validating the change and analyzing it to find potential bugs that have been introduced while the developer still has the changes in mind, yet considers them to be final.

Git supports a concept called *Hooks*. A hook specifies a custom script, which runs programmatically in response to an event, such as commits or uploading a set of commits to a remote server (denoted as *Push* event). Git distinguishes between *Client-Side Hooks*,

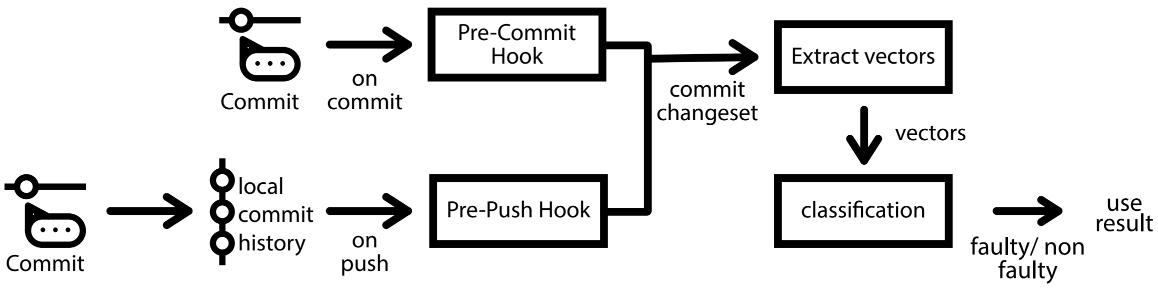


Figure 1: The typical Commit Validation process.

which run on the local device of the developer that is authoring the commit, and *Server-Side Hooks*, which run on the remote server [CS14]. Especially, hooks that run locally in response to a commit should run quickly as the developer has to actively wait for them to complete, hence it is important that a locally evaluating commit validation process executes fast. Nayrolles et al., for example, reported that his approach, called CLEVER, takes an average of 3.75 seconds to complete [Nay18]. This short frame of time does not interrupt the developers workflow and allows him to work effectively with the feedback.

A variety of approaches exist for analyzing the quality of code changes at commit-time. A popular approach is the manual authoring of unit-, integration- or end-to-end-tests to verify that a project's implementation fulfills its specification [Maa18]. Such test definitions can be setup as a commit hook for them to run at commit time.

However, this paper focuses on approaches which do not require manual specifications, such as test cases, to detect a faulty commit, but instead automatically rate the probability of a commit to introduce a bug by leveraging available external data sources. While typical approaches, like manual test cases, come with extra effort and other drawbacks, a primary disadvantage over more automatized methods, like Fault Detection, becomes clear with the well known quote by Dijkstra: *"Testing shows the presence, not the absence of bugs!"* [JR69, p. 16]. A developer has to actively anticipate a fault which might be introduced and specifically test against it. Faults that the developer did not even have in mind are hard to cope with by leveraging manual test cases.

Moreover, code analysis tools are also distinguished into *executing* and *static* analysis methods. Executing methods run the code and use the evaluated responses to identify faults. Manually written test cases fall into this category. The methods considered in this paper belong to the category of static analysis methods, where the code structure is statically analyzed and checked for suspicious locations rather than actually executing the code [Wic+95].

A concluding overview of the typical commit validation process is shown in Figure 1. In detail, the process starts either in response to the pre-commit hook directly after the commit process or in response to the push hook after a set of commits was pushed to a remote server. Then, the commit changesets are usually used to extract metrical vectors, which, in turn, are used to classify the commit data as faulty or not.

2.2 Just-In-Time Fault Detection

While it may seem to be uncomplicated to hook analysis events to commit-based VCS, the actual analysis of commits to determine whether they are faulty or not, i.e., whether they introduce a bug, which was not part of the system before, is a much more complicated task [Nay18; Kam+13]. This procedure is referred to as *Just-In-Time Fault Detection*.

When trying to identify potentially faulty commits, there are many metrics and characteristics that can be taken into account. Typical metrics include the number lines of code added, removed or changed in a commit, characteristics about the developer, or the time of day during which the commit happened [Goy+17]. There are various motivations which justify these metrics, such as the fact that developers who commit during a late time are likely to implement a last-minute bug-fix that could not wait until the next day, and thus are more likely to introduce new bugs in commits made under time pressure [Goy+17].

The validation of a new commit requires a database, which it is matched against, or a model trained with a training dataset, which it can be evaluated with. Thus, an important part of Fault Detection is the data source providing this training data or matching information. Most of the evaluated approaches use the past commit history of the project as training data. Bugs need to be labeled in this data to train the bug detection model. Various ways of extracting labels for commit data exist, like using external issue systems which associate issue reports and bug-introducing commits [Nay18; RGS15].

2.3 Just-In-Time Fault Prevention

A process closely related to Fault Detection is *Just-in-Time Fault Prevention*, which describes the process of automatically generating a potential patch for a newly introduced bug [Nay18]. If the Fault Detection mechanism detects a faulty commit, as described in the previous section, the mechanism attempts to find a past commit which introduced a similar bug. Using information from issue systems, the fix-commit for a similar bug can be used to find out how it was fixed in the past, and leverage that information to suggest how the newly introduced bug can be fixed, as well.

From the identified approaches, only the one by Nayrolles et al. implemented this technique [Nay18], however other works exist in the literature where such techniques have been explored without a focus on Commit Validation. They include the paper by Pan et al. on the understanding of bug fix patterns [PKW09], and the paper by Kim et al. reporting on a novel patch generation method which learns from human-written patches [Kim+13].

In an internal expert study, Nayrolles et al. reported that 41, 6% of the fixes proposed by his method have been accepted by all participants in the study while 25% have been accepted by at least one participant [Nay18]. Obviously, the suggested fixes should be taken with a grain of salt and not naively implemented. They suggest a potential solution to the problem that is supposed to save time and help the developer understand the problem cause, rather than providing a fail-safe patch that can be applied without review.

Table 1: An example of a confusion matrix [Faw06]

		Actual Class	
		true	false
Predicted	true	true positive	false positive
Class	false	false negative	true negative

$$\text{precision} = \frac{tp}{tp + fp}$$
$$\text{recall} = \frac{tp}{tp + fn}$$
$$\text{F1 measure} = 2 \cdot \left(\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \right)$$

Figure 2: Formulars of precision, recall and F1 measure [Pow07]. The variable tp indicates true positives and fn false negatives etc.

2.4 Statistical Measures

When evaluating the performance of binary classification techniques, such as the classification into faulty or non-faulty commits, statistical measures, such as precision, recall, and F1 measure are commonly established [Pow07]. In fact, they were all used in the papers by four of the five identified approaches, i.e. [Nay18; RGS15; Yan+15; Kam+13].

The measurements are based on confusion matrices, which map an actual binary class to the predicted class by the analyzed classification mechanic [Pow07]. Its four cells describe the number of predicted positives that were correct or incorrect (true positives and false positives) and similarly the number of predicted negatives (true negatives and false negatives) [Faw06]. A typical notation for a confusion matrix is shown in Table 1. The measures precision, recall and F1 measure are then derived from the numbers in a confusion matrix [Pow07], as shown in Figure 2.

The precision measures the accuracy of true positives in comparison to all predicted positives, while recall measures the rate of true positives among actual positives. The F1 score is composed of both precision and recall, given as a harmonic mean of both [Pow07].

3 Comparing Commit Validation Approaches

A primary goal of this paper is to define a method for choosing suitable Commit Validation techniques for arbitrary projects. To achieve this goal, the following section introduces a classification scheme, which is leveraged to classify and compare the five identified commit validation approaches. Section 4 will then present the results of the comparison.

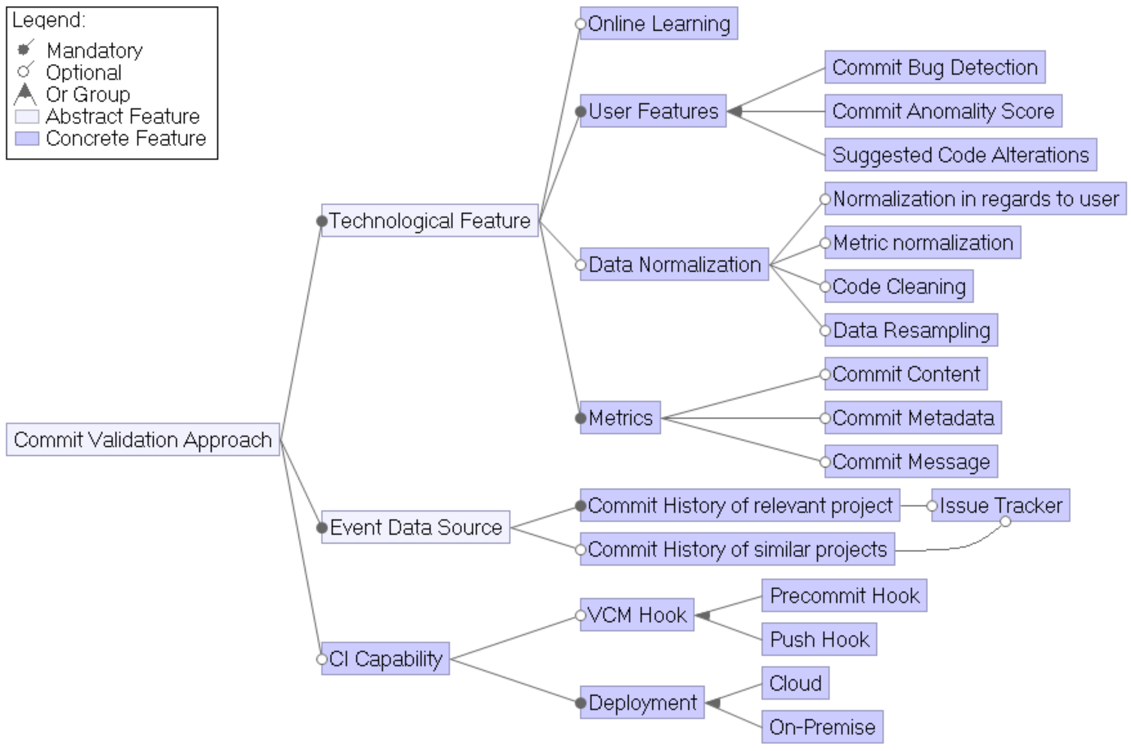


Figure 3: Feature model for Commit Validation approaches.

3.1 Classification Scheme for Commit Validation Approaches

To compare the Commit Validation approaches, a feature model was derived from the features described in the papers which introduce the approaches. The feature model is shown in Figure 3.

For comparing Commit Validation approaches the three categories Technological Features, Event Data Source and CI Capability have been defined.

The first category compares which *Technological Features* are implement by the different approaches. The most important facet in this category is *User Features*, which are features that are interesting to the end user of the approach. The three selected possibilities reflect features that are each implemented by at least one approach.

More relevant to the performance of a method are the facets *Metrics* and *Data Normalization*. All approaches use some metrics to compare commit data, however the respectively used metrics vary between the approaches. The latter facet shows which normalization techniques are used in the approaches to further improve their classification performance. Both facets are also derived from the five approaches, as each of the metrics or normalization techniques are also all used by at least one approach and thus define a good opportunity for comparison. Section 4.4 gives an in-depth performance comparison to provide insights about how these facets influence the performance.

Online Learning describes whether an approach is able to integrate analysis results made in production into its analysis database, otherwise only the analysis results made on initial training data are considered [Sha12]. This facet is relevant for production contexts where new commits should affect the existing classification model, however due to the scientific background of some papers, it is not implemented in all approaches, such as [Kam+13].

The *Event Data Source* describes which sources of data are considered for training the classifier. All approaches take the commit history of the repository into account, for which new commits are being analyzed. Some approaches used the commit history of similar repositories or issue trackers belonging to the repository. Especially, Nayrolles et al. describes their data source in detail and motivates how employing additional data can further improve classification performance [Nay18]. The used data source is relevant when considering using the approach in a production context, as some sources, e.g. issue systems, are not always available.

Finally, the *Capability for Continuous Integration* (or CI Capability) is considered. Continuous Integration describes the autonomous infrastructure that works alongside the developer's workflow and continuously tests, verifies and deploys code artifacts in response to new commits [Boo91]. Because this paper focuses on Commit Validation, it is important how such an approach can be integrated into the commit process. As described in Section 2.1, VCS hooks are typical integration methods. The deployment of the different approaches is interesting for practical usage, as a cloud based solution can be incompatible with company guidelines. Again, Nayrolles' recent approach implements CI capability in form of a pre-commit hook and describes the importance of such features for an improved quality assurance workflow [Nay18].

3.2 Search Process

The initial starting point of research for this work was the Commit Validation approach *CLEVER* developed by Nayrolles et al. [Nay18]. CLEVER suits the criteria, defined in Section 1.3, by being a Just-in-Time Fault Detection approach that uses software commits as underlying mechanism and, as such, can be considered to be a reference approach for the topic.

To find other suitable approaches, *Backwards Snowball Sampling* was performed on CLEVER [Goo61]. In fact, a forward search did not yield results, as CLEVER was still a very recent work at the time of this writing.

The paper by Nayrolles referenced the papers by Kamei and Rosen, the latter introduced *Commit Guru* [Nay18]. Kamei's work was identified to be one of the first approaches on this topic and is included in this comparison due to its early release [Kam+13]. Moreover, Rosen's Commit Guru fits the criteria, but differs from the other papers, because it was released as a freely usable Open-Source project and as such is a very interesting work to consider [RGS15]. Yang's *Deeper* is a similar approach, which builds upon the work of Kamei, but uses different internal mechanics and according to their statement is the first approach to use a deep neural network for classification [Yan+15]. Finally, the approach *Unusual Commits* by Goyal et al. was considered, because its scope differs from the other techniques by focusing more on the abnormality of commits rather than a binary classification into faulty or not, and, similarly to Commit Guru, it was released as Open Source [Goy+17].

3.3 Threats to Validity

A primary threat to the validity of this paper is the fact that it is written in a seminar by a single person and included less research effort than in similar papers, thus a complete survey of the scientific area was out of scope. State of the art research principles, such as Snowball Sampling and a representative sampling size, were used to still give a qualitative overview of Commit Validation techniques [Goo61].

A possible Selection Bias towards Nayrolles' approach CLEVER can be attributed, because it was selected as starting point for this research and because it is the most recent publication [Nay18]. This is due to the novelty of the research area. Approaches differing from typical Commit Validation techniques, such as Unusual Commits by Goyal et al. [Goy+17], were taken into consideration to counter this bias.

Another clear issue with the novelty of the research area is that not many papers publish openly usable and verifiable programs. Unusual Commits and Commit Guru were the only works, which made their software Open Source. Most other papers do not publish their software due to either academic or legal reasons. This not only applies to the papers mentioned in this work, but also to many others on the topic of Just-in-Time Fault Detection, such as the programs iFixR [Koy+19] and Getafix [Bad+19] mentioned in Section 1.3. This makes complete and unbiased performance reviews, even with more research effort, hard to realize. Still this work tries to give insights into the performance of Commit Validation techniques by stating the comparisons given in the papers which introduce the respective techniques. Because no performance information is available for Unusual Commits, this also introduces an Attribution Bias [Goy+17].

As this paper only attempts to give an overview on the topic of Commit Validation, it still succeeds in doing so without in-depth performance benchmarks. However, this is a relevant topic for future work, as more openly usable programs are likely to emerge and more research papers on their performance will be published.

4 Comparison of Commit Validation Approaches

With the comparison scheme described in the previous section, the presentation of the comparison results will now be presented. The five approaches have been ordered into one of three categories. First, methods that only rely on analysis of metric data are reported on in Section 4.1. Methods that additionally employ machine learning techniques are then described in Section 4.2. Finally, the usage of code matching to further improve classification performance is described in Section 4.3. The comparison results are then presented in Table 4.

4.1 Purely Metric Based Approaches

The approaches *Commit Guru* by Rosen et al. [RGS15], *Unusual Commits* by Goyal et al. [Goy+17] and the approach by Kamei et al. [Kam+13] fall into the category of purely metric-based approaches. They rely only on metrics that were directly derived from commits and use them to rate new commits. While Commit Guru works for any kind of Git repository, Unusual Commits only works for repositories hosted on GitHub. Kamei did not state any restrictions of such kind [Kam+13].

Commit Guru and Unusual Commits both support online learning, thus every new commit after the initial training is taken into account for the analysis of later commits. Moreover, Unusual Commits normalizes data in regard to user information by building a profile per project and per developer, thus user information is respected when analyzing commits, while Kamei mentioned that they were also using data resampling. Furthermore, the techniques vary in their output. Unusual Commits is the only approach to not strictly categorize a commit as either faulty or not, instead it rates the likelihood that a commit is *unusual*, i.e. it differs from previous commits. In contrast, Commit Guru and Kamei's approach output the same information as all other approaches, which is said categorization as faulty or not.

Commit Guru and Unusual Commits both only leverage the past commit history of the project which is analyzed as their event data source, not any other data sources. Kamei's approach additionally takes information from the issue system attached to the repository into account. As Unusual Commits only measures the abnormality of commits, it does not need any data labeling specifying which past commits are faulty or not. The comparison of derived metrics of new commits with average values already shows results. Commit Guru on the other hand analyzes commit messages to detect keywords such as `fix` or `bug` to identify fix-commits. It then backtracks the latest commit before the fix that changed similar lines to find out when the matching bug was introduced. This allows Commit Guru to effectively find previous faulty commits without the need for an additional data sourc,e such as an issue system [RGS15].

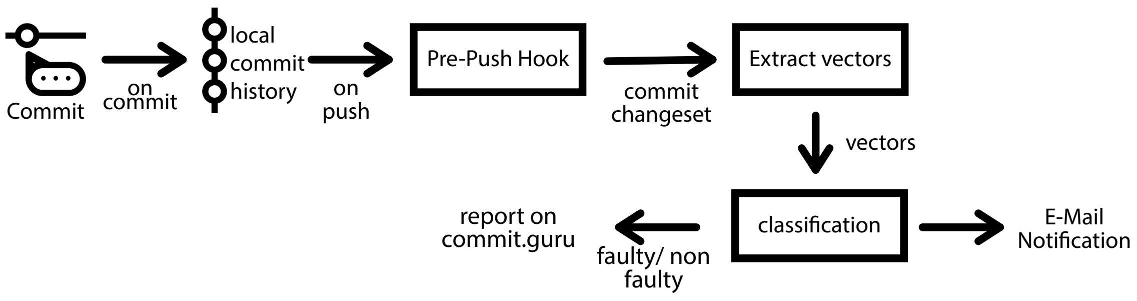
Commit Guru is implemented in Python, while Unusual Commits is realized in Java and R. It is worth mentioning that both Commit Guru and Unusual Commits implement a graphical user interface. Figure 5 their user interfaces.

Commit Guru deploys a standalone web application, which is built upon Google's Angular framework, where one can see the classification results [RGS15]. On its website[1] a Git repository URL can be entered to initialize the analysis. This repository will then be added to the ones continuously reanalyzed on changes and will be added to a list on the page. From there the analysis results can be opened, as depicted in Figure 5a. There the classification results for all commits as well as various metrics are listed. When new changes are detected on a repository, it is automatically reanalyzed and an E-Mail notification is sent as response to the event. Thus it shows CI capability by implementing a Push Hook.

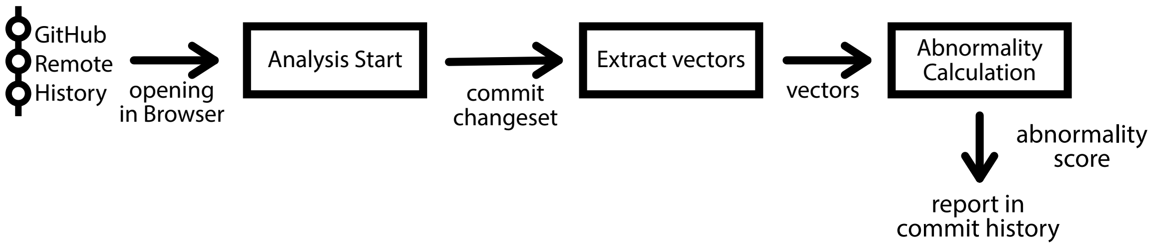
By contrast, Unusual Commits does not deploy its own webpage, but integrates into the GitHub web frontend, which causes the aforementioned restriction of only working with GitHub repositories [Goy+17]. The integration is realized by a Google Chrome plugin, which can be installed from the Chrome Extension store[2]. Figure 5b shows how the calculated abnormality scores for each commit are embedded into GitHub's commit history. Their colors indicate the abnormality, giving the user a good overview at a glance. While no kind of CI capability is realized as new commits are only analyzed by actually visiting

[1]`https://commit.guru`

[2]`https://chrome.google.com/webstore/detail/unusual-commits/hncljhkoaognphcmhdelchclkogepnae`



(a) The Commit Validation process as implemented by Rosen's Commit Guru [RGS15]



(b) The Commit Validation process as implemented by Goyal's Unusual Commits [Goy+17]

Figure 4: Commit Guru's and Unusual Commits' visualized processes.

the commit history page, it still integrates well into a typical developer's workflow, as this page is commonly visited to get an overview of changes made in the repository.

When clicking on a commit in GitHub, a detailed overview of it is opened. Moreover, Unusual Commits embeds information on this page, as depicted in Figure 5c. Here the abnormal facts are listed for this commit to justify the generated abnormality score.

Both are realized as free-to-use cloud applications. In addition, both are powered by custom backend software that runs the analysis and is deployed on proprietary servers. Unlike Commit Guru and Unusual Commits, Kamei did not publish an openly usable program, thus their CI capability or technological background could not be evaluated.

Kamei's approach and Commit Guru use the same set of metrics, which are metrics regarding the commit content, such as changed lines of codes or the number of changed files, as well as the commit history, such as the commit purpose and information about the developer. Unusual Commits additionally considers the commit content, but instead of commit history, the commits' metadata is included.

In Section 2.1, Figure 1 depicted the typical Commit Validation process. Figure 4 shows the adapted processes, as implemented by Commit Guru and Unusual Commits. As one can see, they are initialized by either a push event on the repository in the case of Commit Guru or by opening the commit history in GitHub, in the case of Unusual Commits. Both approaches then extract metrical vectors and use them to perform classification or calculations based on that, followed by reporting the process results on either Commit Guru's website or embedded into the commit history by Unusual Commits.

Classification approaches that rely on purely metric-based methods generally show high potential and are well represented in the research area. However, the availability of metrical vectors extracted from commit data offers the usage of machine learning methods to further increase classification performance. The next section will detail the use of machine learning in existing Commit Validation approaches.

4.2 Approaches Based on Machine Learning

This category describes approaches which use machine learning techniques to increase evaluation performance. While both *Deeper* by Yang [Yan+15] and *CLEVER* by Nayrolles [Nay18] implement such techniques, according to Yang et al. Deeper was the first Just-in-Time Fault Detection approach to leverage deep neural networks for improved classification performance. Nayrolles' CLEVER on the other hand additionally implemented code matching alongside a machine learning model and thus will be described in more detail in Section 4.3.

Deeper was developed primarily in a research context and builds upon the work of Kamei's approach [Kam+13], successfully attempting to increase its classification performance with the new neural network. Moreover, they mention how online learning is established by Deeper. Typical methods common in machine learning such as data normalization and resampling are used. Like most approaches Deeper only supports detecting faulty commits without special features like the abnormality score supplied by Unusual Commits [Goy+17]. It also uses the most common data sources for training its model, which are the past commit history of the project for which new commits are being analyzed alongside an external issue system. Likely inspired by Kamei's work, it also uses the same metrics for classification, taking the commits content and message into account.

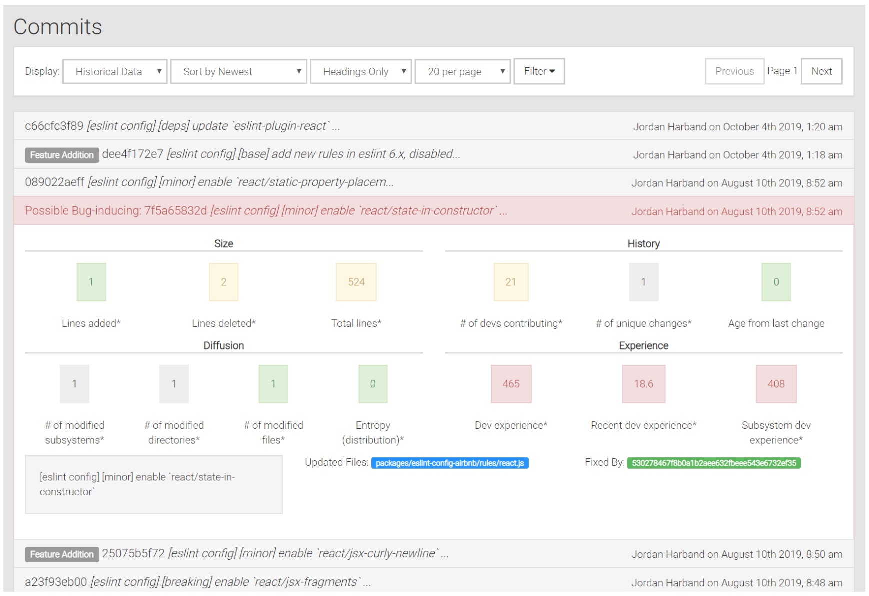
It is easy to see that Deeper is similar to Kamei's work in many aspects, likely due to their shared motivation by a scientific rather than an industrial context, as well as the fact that Yang's primary contribution was the integration of the neural network instead of Kamei's purely metric-based method.

Information on the technical platform or the CI capability was not reported in the original paper by Yang. They did, however, mention that they leveraged a deep belief network algorithm to build a set of features and performed logistic regression on that.

While research shows that machine learning can increase performance in contrast to naive metric-based methods, Nayrolles mentioned approach of code matching introduces a very different method for better classification results [Nay18]. Consequently, the next section will detail the use of code matching methods.

4.3 Approaches Based on Code Matching

Being published in 2018, Nayrolles' *CLEVER* was the most recent approach and implements features none of the previous publications did [Nay18]. To the best of my knowledge, CLEVER is the only one which not only detects faulty commits, but also automatically generates suggested alterations for fault locations based on previous fixes in the commit history. While it uses past commits from the relevant project and the affiliated issue system in a similar way as other approaches do to detect faults, only CLEVER leverages commit data from similar projects. Nayrolles reported how dependency information of projects is used to cluster them based on their similarity.



Commits

Display: Historical Data ▾ | Sort by Newest ▾ | Headings Only ▾ | 20 per page ▾ | Filter ▾ — Previous | Page 1 | Next

c66cfc3f89 *[eslint config] [deps] update `eslint-plugin-react` ...* — Jordan Harband on October 4th 2019, 1:20 am

Feature Addition dee4f172e7 *[eslint config] [base] add new rules in eslint 6.x, disabled...* — Jordan Harband on October 4th 2019, 1:18 am

089022aeff *[eslint config] [minor] enable `react/static-property-placem...* — Jordan Harband on August 10th 2019, 8:52 am

Possible Bug-inducing: 7f5a65832d *[eslint config] [minor] enable `react/state-in-constructor` ...* — Jordan Harband on August 10th 2019, 8:52 am

Size			History		
1	2	524	21	1	0
Lines added*	Lines deleted*	Total lines*	# of devs contributing*	# of unique changes*	Age from last change

Diffusion				Experience		
1	1	1	0	465	18.6	408
# of modified subsystems*	# of modified directories*	# of modified files*	Entropy (distribution)*	Dev experience*	Recent dev experience*	Subsystem dev experience*

[eslint config] [minor] enable `react/state-in-constructor`

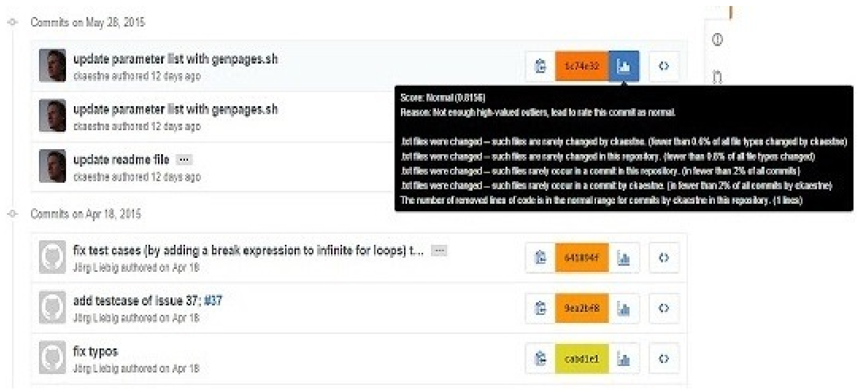
Updated Files: packages/eslint-config-airbnb/rules/react.js

Fixed By: 530278467f8b0a1b2aee632fbeee543e6732ef35

Feature Addition 25075b5f72 *[eslint config] [minor] enable `react/jsx-curly-newline` ...* — Jordan Harband on August 10th 2019, 8:50 am

a23f93eb00 *[eslint config] [breaking] enable `react/jsx-fragments` ...* — Jordan Harband on August 10th 2019, 8:48 am

(a) Web panel of Commit Guru [RGS15]. A dedicated page lists all commits of a repository and shows their classification result.



Commits on May 28, 2015

update parameter list with genpages.sh
ckaestne authored 12 days ago

update parameter list with genpages.sh
ckaestne authored 12 days ago

update readme file ···
ckaestne authored 12 days ago

Score: Normal (0.8156)
Reason: Not enough high-valued outliers, lead to rate this commit as normal.

.txt files were changed -- such files are rarely changed by ckaestne. (fewer than 0.4% of all file types changed by ckaestne)
.txt files were changed -- such files are rarely changed in this repository. (fewer than 0.8% of all file types changed)
.txt files were changed -- such files rarely occur in a commit in this repository. (in fewer than 2% of all commits)
.txt files were changed -- such files rarely occur in a commit by ckaestne. (in fewer than 2% of all commits by ckaestne)
The number of removed lines of code is in the normal range for commits by ckaestne in this repository. (1 lines)

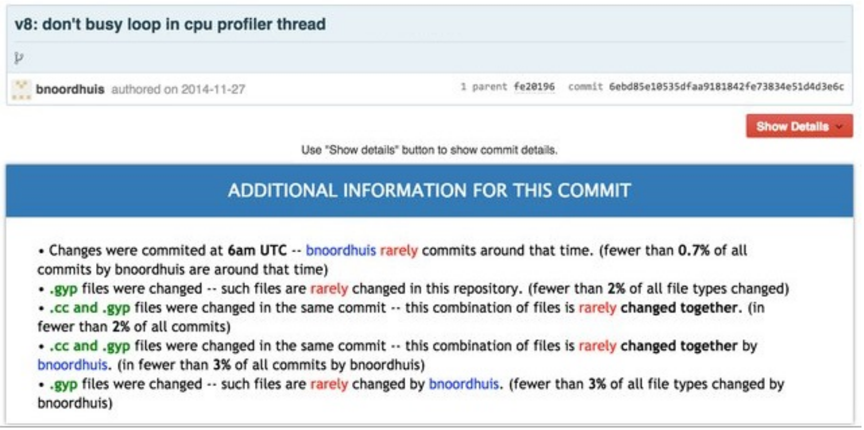
Commits on Apr 18, 2015

fix test cases (by adding a break expression to infinite for loops) t... ···
Jörg Liebig authored on Apr 18

add testcase of issue 37; #37
Jörg Liebig authored on Apr 18

fix typos
Jörg Liebig authored on Apr 18

(b) Unusual Commit embeds commit's abnormality scores in GitHub's commit history [Goy+17].



v8: don't busy loop in cpu profiler thread

bnoordhuis authored on 2014-11-27 — 1 parent fe20196 commit 6ebd85e10535dfaa9181842fe73834e51d4d3e6c

Use "Show details" button to show commit details.

ADDITIONAL INFORMATION FOR THIS COMMIT

- Changes were commited at **6am UTC** -- bnoordhuis rarely commits around that time. (fewer than **0.7%** of all commits by bnoordhuis are around that time)
- .gyp files were changed -- such files are rarely changed in this repository. (fewer than **2%** of all file types changed)
- .cc and .gyp files were changed in the same commit -- this combination of files is rarely **changed together.** (in fewer than **2%** of all commits)
- .cc and .gyp files were changed in the same commit -- this combination of files is rarely **changed together** by bnoordhuis. (in fewer than **3%** of all commits by bnoordhuis)
- .gyp files were changed -- such files are rarely changed by bnoordhuis. (fewer than **3%** of all file types changed by bnoordhuis)

(c) When opening a commit, Unusual Commit describes why a potentially high abnormality score was calculated [Goy+17].

Figure 5: Graphical user interfaces of Commit Guru [RGS15] and Unusual Commits [Goy+17]

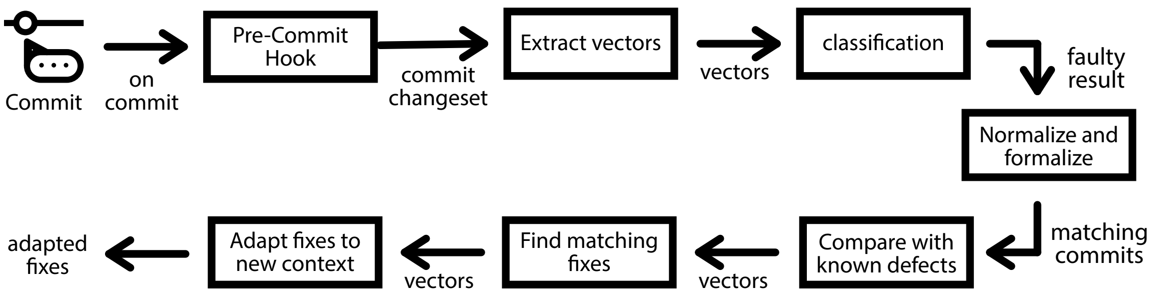


Figure 6: The Commit Validation process as implemented by Nayrolles' CLEVER [Nay18]

The metrics used by CLEVER do not vary much from those by the other approaches. CLEVER's model was built on the foundation of Commit Guru and uses the same metrics as they do. However, as Section 4.4 detailing the performance of the approaches will show, CLEVER significantly outperforms Commit Guru. According to Nayrolles' paper, this is caused by its code matching component. This attempts to reduce the number of false-positives, as unfaulty code that was classified as faulty is revalidated and then unclassified. This finally results in a better precision, which the formula in Figure 2 shows.

In CLEVER, commits are first analyzed by a machine learning model, and only after positive classification code matching is performed to revalidate the likelihood of a fault. The code matching component expands the changeset of a commit to both of its sides to extract a complete syntactically correct code block. This code block is then compared to historically known defect introducing commits. For the comparison of code blocks, text-based clone detection techniques such as the *NICARD clone detector* are leveraged [Nay18].

Nayrolles argues that just resorting to the code matching method and refraining from using the metric-based classifier entirely would not yield any noticeable performance loss. The main reason, why the metric-based model is used prior to code matching, is efficiency, as code matching is much more time consuming than a metric-based analysis.

The generation of altered code suggestions then follows from the code matching analysis results linking a similar bug introducing commit. The fix commit for the bug is determined using the issue system, and the fix changeset is adapted to the new code context by changing indentation depth and variable names.

Figure 6 depicts the Commit Validation process implemented by CLEVER. With the code matching component and the generation of altered code suggestions, this process is longer than the processes mentioned before as new steps are introduced. Notably, as CLEVER implements a pure Pre-Commit Hook which checks code changes during the commit action and blocks the action in case of a detected fault, the Commit Validation process ends after the classification step if the commit is considered non-faulty.

In contrast to other approaches, CLEVER was motivated by an industrial rather than a scientific context. It was developed for production use within Ubisoft, with a company-internal deployment planned. Consequently, its online learning and Continuous Integration capabilities are both important and thus fully supported. For Continuous Integration, Nayrolles et al. decided on pre-commit hooks, which run directly on the developers machines at the moment they commit source code. This results in fast feedback to the developer.

Table 2: Performance comparison of CLEVER and Commit Guru, as evaluated by Nayrolles et al. [Nay18]

	CLEVER	Commit Guru
Precision	79.10%	66.71%
Recall	65.61%	63.01%
F1 measure	71.72%	64.80%

Table 3: Performance comparison of Deeper and Kamei's approach as well as a standard Logistic Regression, denoted as LR. The comparison was evaluated by Yang et al. [Yan+15]

	Deeper	Kamei	LR
Precision	35.64%	35.71%	59.92%
Recall	69.03%	65.52%	18.28%
F1 measure	45.06%	43.91%	25.30%

4.4 Performance Comparison

When comparing Commit Validation approaches, the evaluation performance is an important point to consider. Due to the limited size of this work and the fact that not all approaches are publicly available, an objective performance evaluation of all five approaches with the same dataset was out of scope for this paper. However, Nayrolles et al. included an objective performance comparison between their approach and Rosen's approach on a consistent dataset [Nay18]. Similarly, Yang et al. included a performance comparison between their approach and Kamei's [Yan+15]. While both experiments use different datasets and thus cannot be compared directly, they give insights into how CLEVER compares to Commit Guru and how Deeper compares to Kamei's approach in regard to evaluation performance.

Table 2 highlights the findings from Nayrolles' paper [Nay18], while Table 3 shows those from Yang's paper [Yan+15].

Nayrolles findings indicate that his approach performs better than Commit Guru, with a precision increase of more than 13% and an increase in the F1 measure of about 7%. Similarly, Yang found that his approach performs better than the one described by Kamei. Even though the performance gap was not as significant in his case (with an increase of the F1 measure of only about 1%), there was still an improvement.

Furthermore, it is notable that Nayrolles approach primarily improved precision compared to Commit Guru, while Deeper improved recall compared to Kamei's approach. A likely cause for this is Nayrolles' implementation of code matching, which reduced the number of false positives, resulting in better precision.

4.5 Comparison Results

The previous sections have discussed the differences and highlights of the selected Commit Validation approaches. This section will bring these details into contrast and provide a qualitative comparison.

The comparison results have been collected in Table 4, where each of the described facets map to specific values per identified Commit Validation approach. The basic structure of the table maps to the feature model described in Figure 3. Interesting findings from the table will be highlighted and discussed in the Section 5.

The listed category *Technological Features* lists user features, implemented data normalization techniques and online learning as resembled in the feature model. While all approaches implement one of the features bug detection, abnormality score or altered code suggestions, the usage of data normalization and online learning varies between the five approaches.

The event data source indicates the origin of the data, which is used for training the classification model. All approaches either use the commit history of the analyzed project or the one from similar projects. Moreover, the associated issue system of the project is also commonly utilized.

The technical platform indicates which platform the approach is implemented in. This can be interesting for the end user, as it can influence the practicality of integrating the system in production environments. CI capability is reflected from the feature model to report on how approaches integrate into CI systems, a quality which was considered important for this paper. As both an approaches platform and its CI capability are very technical features and not all papers report on the deployment of their methods but rather their classification model, this information was not always available. Especially, the technical platform was only reported on by Commit Guru and Unusual Commits. While I have tried to contact the authors of Deeper and Kamei's approach, they could not be reached for further details. Thus, Cells marked with ◐ or *N/A* indicate that the respective information was not mentioned in the reporting paper.

Finally, the last category of considered metrics describes which measures were used by the classification model for training and evaluation. Metrics regarding the commit content contain added, removed or changed lines of code, as well as the number of changed directories, files or similar. Commit metadata contains information on the length of the describing message, the time of commit and the types of files changed in that commit. Metrics regarding the commit message contain the purpose of the commit, such as a fix or a new feature, as well as information on the developer.

5 Discussion of Comparison Results

When considering the classification results in Table 4, various interesting observations can be made. In the following section, notable correlations will be emphasized and their possible causes discussed.

All approaches show some results to the user, either Commit Bug Detection, where each commit is marked as either faulty or not; a score, which rates the abnormality of

Table 4: Classification results of the five identified approaches: CLEVER by Nayrolles [Nay18], Commit Guru by Rosen [RGS15], Unusual Commits by Goyal [Goy+17], Deeper by Yang [Yan+15], Kamei's approach [Kam+13].

	CLEVER [Nay18]	Guru [RGS15]	Unusual [Goy+17]	Deeper [Yan+15]	Kamei [Kam+13]
Technological Features					
Online learning	●	●	●	●	○
Normalization to user info	○	○	●	○	○
Metric normalization	◐	◐	●	●	●
Code cleaning and matching	●	○	○	○	○
Data resampling	◐	◐	○	●	●
Commit bug detection	●	●	○	●	●
Commit abnormality score	○	○	●	○	○
Code alteration suggestions	●	○	○	○	○
Event Data Source					
History relevant project	●	●	●	●	●
History similar projects	●	○	○	○	○
External issue-system	●	○	○	●	●
Technical Background					
Technical internal platform	N/A	Python	Java, R	N/A	N/A
CI Capability					
Pre-commit hook	●	○	○	◐	◐
Push hook	○	●	○	◐	◐
E-Mail notification	◐	●	○	◐	◐
Cloud or on-Premise	On-Pr.	Cloud	Cloud	N/A	N/A
Considered Metrics					
Commit content	●	●	●	●	●
Commit metadata	○	○	●	○	○
Commit message	●	●	○	●	●

commits in contrast to other commits; or automatically generated suggested alterations for possibly faulty code. Since a primary topic of Commit Validation lies in Fault Detection, the binary detection of bugs in commits is implemented by most approaches. Only Unusual Commits [Goy+17] generates an abnormality score, which leaves the decision whether a commit is actually faulty up to the user, only *strange* commits are highlighted to guide the user's focus.

CI Capability is a feature, which was considered to be very important for this paper, however not many approaches actually implemented it. Especially, among papers focusing on research this was usually either not realized or not reported on, while performance was more emphasized. With more emerging papers on the topic of Commit Validation and increasing usage in industry, this capability is likely to be more broadly implemented in future methods.

The feature online learning was implemented by all approaches except Kamei's, which is likely due to its academic goal of verifying the concept rather than defining an industrially usable program [Kam+13]. Most other approaches like CLEVER and Commit Guru realized methods that were meant to be deployed in industrial context, thus online learning becomes relevant as the retraining on recent faults is desired [Nay18; RGS15].

One interesting finding comes from the numbers of the aforementioned performance comparisons, as both CLEVER and Deeper perform better than their reference approaches. They also implement similar methods as both rely on machine learning. This correlation suggests that establishing machine learning methods brings noticable performance improvements compared to relying solely on metric-based techniques. Nayrolles also addressed that incorporating code matching significantly increases precision in CLEVER, which explains the major advantage it has in numbers compared to Commit Guru [Nay18].

The event data source is also closely related to the classification performance. Here CLEVER also stands out as it is the only approach to consider projects that are similar to the project for which commits are analyzed. The usage of previous commit data of the analyzed project are typical event sources which are used by all approaches. Incorporating data from issue systems correlates with increases in performance and usually makes sense to do, because issue systems are used in most industrial software engineering projects, thus their data is available anyway.

Another feature that likely affects classification performance is data normalization regarding user information and behavior, a feature which is only implemented by Unusual Commits [Goy+17]. Because no performance comparison of Unusual Commits was found, an implication of the positive or negative effect of this method cannot be concluded here.

A topic closely related to the classification performance are the metrics which are were considered during classification. Between most approaches, the used metrics do not differ much: CLEVER, Commit Guru, Deeper and Kamei all leverage metrics regarding the commit content and the commit message. Only Unusual Commits uses commit metadata instead of the commit message. This suggests that a typical set of relevant metrics for defect detection has been established in the research area. The metrics were initially introduced by Kamei's original paper and referenced by the other papers, thus Kamei et al. initially defined those relevant metrics and they were used mostly unchanged since [Kam+13].

In conclusion, there are many differences in existing Commit Validation approaches and more variations are likely to appear in the future. Due to the scientific nature of many approaches and their lack of open availability, choosing the right method for a production project difficult. CI capability is a relevant feature that is especially implemented by industrial and more recent approaches and is important to consider for production deployments. While some kind of VCM hooks are often used for continuous integration, the kind of hook depends on the application context: Nayrolles et al. noted that *pre-commit hooks* have the advantage of giving feedback to the user during the moment when they are most concerned with the changes [Nay18]. By contrast, push hooks are easier to set up as they do not have to be installed on every developers machine. The event data source is also important to consider, as not all possible data sources are always available in specific production contexts. External issue systems are not always used in a development environment. Contrarily, using an approach, which does not use a data source which is available anyways, can unnecessarily hurt classification performance. Similar to CI capability, online learning is also generally implemented by approaches that focus on an industrial context. This feature is relevant for production use as not supporting online learning only makes sense for approaches whose scientific analysis of classification performance is of primary importance. There are more facets which primarily affect classification performance, yet existing approaches are hard to compare due to the aforementioned complications and lacking performance reports. However, noticeable trends can be seen in data, where novel features such as neural networks or code matching can significantly increase performance, and thus should be taken into account when choosing a commit validation method for a production project. One undoubtable takeaway of the study is that new technologies on the topic continue to emerge and positively impact the performance of future approaches.

6 Related Surveys on Commit Validation

Various studies and surveys exist on the topic of either Commit Validation or Just-in-Time Defect Prediction. While, for instance, Catolino et al. [CDF19] and Syed [Sye19] took practical implementations into account, most focused on comparing the performance of underlying components, such as the prediction model, resampling and ensemble learning methods or used metrics. For Catolino's and Syed's papers it is notable that both included and discussed CLEVER and Commit Guru.

Kiehn et al. reported on existing publications and developed a new classification model, which extended the used metrics from previous approaches by new information based on, among others, code ownership, showing that this information can be successfully applied in change risk classification [KPC19].

A similar focus on evaluating the underlying components of Commit Validation approaches was set by the study of Zhu et al. [Zhu+18]. They explored *imbalance data handling methods*, such as resampling and ensemble learning, to evaluate their performance effect in change classification. They found that a combination of ensemble learning and resampling shows better results than other methods.

Similarly, Malhotra et al. [Mal+17] explored various machine learning methods to determine their overall predictive capability. According to them, their work confirms the overall predictive capability of all machine learning methods, that they have inspected, for defect prediction. This matches our suggestion made in Section 5, that approaches which rely on machine learning yield better results than the ones that do not. They also highlighted that a single layered perceptron shows the most effective result as a machine learning method.

Finally, in a very early study, almost five years before all other mentioned studies, Punitha and Chitra developed a novel approach for defect prediction, which was based on three different classification methods, namely *Support Vector Machine*, *Fuzzy Inference System* and *Genetic Algorithm* [PC13]. They verified that the combination of those learners leads to a better prediction model.

In conclusion, all identified studies differed from the scope of this paper by focusing on the performance of defect classification. As mentioned, studies concentrating on user features and the practical use of Commit Validation methods are rare, especially because the research topic focuses on Fault Detection and less on the aspect of commits. Especially CI capability is a feature which, was considered important for this paper, but was usually not taken into account by any of the related studies.

7 Conclusions

The aim of this paper was to give an overview on the topic of Commit Validation as well as highlighting existing works that have emerged in this area. In an effort to objectively compare and discuss differences between Commit Validation approaches, a feature model for such approaches was defined in Section 3.1. Using this feature model, five relevant yet diverse methods were classified and compared. The evaluation focused not only on CI Capability and user features, which both were considered important for the practical applicability of Commit Validation, but also on the precision of these methods.

The contributions of this paper include the basic introduction of Commit Validation, the definition of a novel evaluation scheme to highlight differences between Commit Validation approaches and the qualitative comparison of the five mentioned approaches.

Our findings indicate that many differences in the classification approaches exist and affect their precision. Moreover, they highlight that many new publications and methods continue to emerge on this novel topic, and new concepts and features are introduced. With the topic becoming more relevant in industrial contexts, production features, such as CI capability and online learning are likely to be implemented more often, as well. Especially, the deployment of Nayrolles' CLEVER in Ubisoft is an important step towards bringing the topic closer to being an industry standard [Nay18].

As discussed earlier in Section 3.3, comparing the precision of Commit Validation methods continues to be a difficult objective without comparable measurements, with many approaches staying closed source and not being openly published, and with few reports on the performance of programs available. This marks an important area for future research, with more emerging methods and papers. While the precision of high-level programs seem to be not well covered yet, the performance of their underlying components such as relevant machine learning models will continue to be the topic of further investigations. This will lead to more research and more interesting findings that involve those performance information. Also, approaches continue to implement new features and ways of further optimizing the productivity in a developers workflow, as the differences in many recent publications highlight, and I expect this trend to continue in the future and assume it will be taken into account by future surveys.

References

[Bad+19] Johannes Bader et al. *Getafix: Learning to Fix Bugs Automatically.* 2019. arXiv: 1902.06111. URL: http://arxiv.org/abs/1902.06111.

[Boo91] G Booch. *Object Oriented Design: With Applications.* The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991. ISBN: 9780805300918. URL: https://books.google.de/books?id=w5VQAAAAMAAJ.

[CDF19] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. "Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment". In: *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2019), pp. 99–110. DOI: 10.1109/mobilesoft.2019.00023.

[CS14] Scott Chacon and Ben Straub. *Pro Git.* 2nd. Berkely, CA, USA: Apress, 2014. ISBN: 1484200772, 9781484200773.

[Faw06] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874. ISSN: 01678655. DOI: 10.1016/j.patrec.2005.10.010.

[Goo61] Leo A Goodman. "Snowball Sampling". In: *The Annals of Mathematical Statistics* 32.1 (1961), pp. 148–170. ISSN: 00034851. URL: http://www.jstor.org/stable/2237615.

[Goy+17] Raman Goyal et al. "Identifying unusual commits on GitHub". In: *Journal of Software: Evolution and Process* August 2017 (2017), pp. 1–16. ISSN: 20477481. DOI: 10.1002/smr.1893.

[JR69] Buxton John N and Brian Randell. *Software Engineering Techniques.* Tech. rep. Report on a Conference Sponsored by the NATO Science Committee, 1969, p. 16. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[Kam+13] Yasutaka Kamei et al. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 757–773. ISSN: 00985589. DOI: 10.1109/TSE.2012.70.

[Kim+13] Dongsun Kim et al. "Automatic patch generation learned from human-written patches". In: *Proceedings - International Conference on Software Engineering* 1.c (2013), pp. 802–811. ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606626.

[Koy+19] Anil Koyuncu et al. "iFixR: bug report driven program repair". In: *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 314–325. DOI: 10.1145/3338906.3338935. arXiv: 1907.05620.

[KPC19] Max Kiehn, Xiangyi Pan, and Fatih Camci. "Empirical study in using version histories for change risk classification". In: *IEEE International Working Conference on Mining Software Repositories* 2019-May (2019), pp. 58–62. ISSN: 21601860. DOI: 10.1109/MSR.2019.00018.

[Maa18] Dor D. Ma'ayan. "The quality of junit tests". In: *IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE)* (2018), pp. 33–36. DOI: 10.1145/3194095.3194102.

[Mal+17] Ruchikan Malhotra et al. "Empirical comparison of machine learning algorithms for bug prediction in open source software". In: *Proceedings of the 2017 International Conference On Big Data Analytics and Computational Intelligence, ICBDACI 2017.* 2017, pp. 40–45. ISBN: 9781509063994. DOI: 10.1109/ICBDACI.2017.8070806.

[Mar08] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Robert C. Martin Series. Upper Saddle River, NJ: Prentice Hall, 2008. ISBN: 978-0-13235-088-4. URL: https://www.safaribooksonline.com/library/view/clean-code/9780136083238/.

[Nay18] Mathieu Nayrolles. "Software Maintenance At Commit-Time". In: August (2018).

[PC13] K. Punitha and S. Chitra. "Software defect prediction using software metrics - A survey". In: *2013 International Conference on Information Communication and Embedded Systems, ICICES 2013* (2013), pp. 555–558. DOI: 10.1109/ICICES.2013.6508369.

[PKW09] Kai Pan, Sunghun Kim, and E. James Whitehead. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14.3 (2009), pp. 286–315. ISSN: 13823256. DOI: 10.1007/s10664-008-9077-5.

[Pow07] David M W Powers. "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation". In: *Human Communication Science SummerFest* December (2007), p. 24.

[RGS15] Christoffer Rosen, Ben Grawi, and Emad Shihab. "Commit guru: Analytics and risk prediction of software commits". In: *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* (2015), pp. 966–969. DOI: 10.1145/2786805.2803183.

[Sha12] Shai Shalev-Shwartz. "Online Learning and Online Convex Optimization". In: *Foundations and Trends® in Machine Learning* 4.2 (2012), pp. 107–194. ISSN: 1935-8237. DOI: 10.1561/2200000018. URL: http://dx.doi.org/10.1561/2200000018.

[Sta18] Stackoverflow. *Stackoverflow Developer Survey Results 2018.* 2018. URL: https://insights.stackoverflow.com/survey/2018 (visited on 01/24/2020).

[Sye19] Arsalan Syed. *Investigating the Practicality of Just-in-time Defect Prediction with Semi-supervised Learning on Industrial Commit Data.* 2019. URL: `http://www.diva-portal.org/smash/record.jsf?pid=diva2%7B%5C%%7D3A1336751%7B%5C&%7Ddswid=28`.

[TG02] Tassey and Gregory. "The Economic Impacts of Inadequate Infrastructure for Software Testing". In: *Program Office Strategic Planning and Economic Group, 309* (2002). URL: `https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf`.

[Wic+95] B. A. Wichmann et al. "Industrial perspective on static analysis". In: *Software engineering journal* 10.2 (1995), pp. 69–75. ISSN: 02686961. DOI: `10.1049/sej.1995.0010`. URL: `https://web.archive.org/web/20110927010304/http://www.ida.liu.se/%7B~%7DTDDC90/papers/industrial95.pdf`.

[Yan+15] Xinli Yang et al. "Deep Learning for Just-in-Time Defect Prediction". In: *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015* 1 (2015), pp. 17–26. DOI: `10.1109/QRS.2015.14`.

[Zhu+18] Xiaoyan Zhu et al. "An empirical study of software change classification with imbalance data-handling methods". In: *Software - Practice and Experience* 48.11 (2018), pp. 1968–1999. ISSN: 1097024X. DOI: `10.1002/spe.2606`.