

The present work was submitted to
the RESEARCH GROUP
SOFTWARE CONSTRUCTION

of the FACULTY OF MATHEMATICS,
COMPUTER SCIENCE, AND
NATURAL SCIENCES

BACHELOR THESIS

Mining Changes of Build Processes in the Context of Continuous Integration

presented by

Benedikt Holmes

Aachen, December 10, 2017

EXAMINER

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

SUPERVISOR

Dipl.-Inform. Andreas Steffens

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Bachelor's thesis entitled
Mining Changes of Build Processes in the Context of Continuous Integration

independently and without illegitimate assistance from third parties. I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Mining Changes of Build Processes in the Context of Continuous Integration

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, December 10, 2017

(Benedikt Holmes)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Strafflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, December 10, 2017

(Benedikt Holmes)

Abstract

The use of Continuous Integration systems is a widely accepted and modern practice adopted by many open-source software (OSS) developers. As of yet there has been only little in-depth research on the understanding and usage of CI in OSS. Thus, with this thesis I present an explorative approach for analyzing this. As source I use 900+ projects from GitHub which use the Travis CI platform and mine on the change history of their CI configuration file. The goals of this research are to investigate the value of such a data set and to gather knowledge on the acceptance and robustness of the Travis CI model as well as the evolution of Travis CI build processes. In fact the configuration change history allows for detailed analysis. The findings include, that the model is well accepted and is slightly unstable towards changes. Also does the analysis of the build process evolution yield that build processes do not stabilize, in terms of changes to the configuration. Furthermore, it is possible to derive a maturity measure for the build processes. More than half of all projects' build processes are considered mature.

Contents

1. Introduction	1
1.1. Structure of this Thesis	1
2. Background	3
2.1. Domain-driven Background	3
2.2. Methodology-driven Background	7
3. Motivation	13
3.1. Topic of this Thesis	13
3.2. Why Historic Build Data is Interesting	13
3.3. The Mining Software Repositories Conference (MSR)	14
3.4. Mining in the Context of CI	16
3.5. Purpose of this Thesis	17
3.6. What to Expect	17
3.7. Methodology	18
4. Preliminary Data Analysis	19
4.1. Data Selection	19
4.2. Build Execution Logs	19
4.3. Build Configuration in Travis CI	20
4.4. Conformance to Classic CI	24
4.5. Research Focus	24
4.6. Definition of Terms	25
5. Research Goals and Questions	29
5.1. Goals	29
5.2. Feature Selection	34
6. Data Preprocessing	35
6.1. Feature Vector Modeling	35
6.2. Challenges	39
6.3. First Observations	41
7. Results and Discussion	43
7.1. Goal 1: Acceptance of the Travis CI Model	43
7.2. Goal 2: Robustness of the Travis CI Model	48
7.3. Goal 3: Build Process Evolution	52
7.4. Goal 4: Build Process Structure Build-Up	57

7.5. Goal 5: Equivalent Usage of the Travis CI Model	65
7.6. Final Discussion	68
8. Conclusion	71
8.1. Threats to Validity	72
8.2. Future Work	77
A. Clustering Results	79
B. Classification Results	87
Bibliography	91

List of Tables

4.1. Top-level keys of <i>.travis.yml</i> [Trab].	22
4.2. Phases defined by top-level keys of <i>.travis.yml</i>	26
4.3. CI-functionalities defined by phases.	28
5.1. Features selected for this research.	34
6.1. Project distribution w.r.t main programming language and CI-usage period.	41
7.1. Rankings of phase changes and usage in absolute and relative (rounded) measures.	45
7.2. Volatility of phases.	46
7.3. Build failures per phase.	49
7.4. Groups of phases that cause build failures together.	51
7.5. CI-maturity measured by CI-functionality adoption.	59
7.6. Mean time of adoption for CI-functionalities.	60
7.7. Age and CI-usage period of the projects in cluster10 (sorted by age).	64
7.8. Resulting cluster means for clustering on total configuration changes.	66
7.9. Resulting cluster means for clustering on phase change density.	67
A.1. Clustering results: phase changes prior to build failures.	80
A.2. Clustering results: CI-functionality introduction.	81
A.3. Clustering results: phase usage.	82
A.4. Clustering results: phase changes.	83
A.5. Clustering results: configuration changing commits.	84
A.6. Clustering results: phase change density.	85
B.1. Classification results: (All \Rightarrow CI-Maturity).	87
B.2. Classification results: (Meta Features \Rightarrow CI-Maturity).	87
B.3. Classification results: (Absolute Usage \Rightarrow CI-Maturity).	87
B.4. Classification results: (Relative Usage \Rightarrow CI-Maturity).	88
B.5. Classification results: (Binary Usage \Rightarrow CI-Maturity).	88
B.6. Classification results: (Absolute Changes \Rightarrow CI-Maturity).	88
B.7. Classification results: (Relative Changes \Rightarrow CI-Maturity).	89
B.8. Classification results: (Age and CI-Usage \Rightarrow CI-Maturity).	89

List of Figures

2.1. Git's Internal Structure ([Gite], Figure 151).	4
2.2. Exemplary branching scenario in Git: Directly after branching.	5
2.3. Exemplary branching scenario in Git: Some time after branching.	5
2.4. Exemplary branching scenario in Git: After merging.	6
2.5. DevOps practices compared [Awsa].	7
2.6. The GQM model [BCR94].	11
4.1. Travis CI state machine.	23
6.1. Different phase usage representations.	38
6.2. Handling invalid <i>YAML</i> files in the change history.	39
6.3. The three basic types of changes.	40
6.4. Distribution of project age and CI-usage period in the preprocessed data.	41
7.1. Amount of distinct phases used per project.	47
7.2. Plot of change frequency versus build failures of all phases.	50
7.3. Phase change frequency for all phases in groups of CI-usage periods.	53
7.4. Phase usage frequency for selected phases for all projects.	55
7.5. Examples for experimental periods in phase usage frequency graphs.	56
7.6. Project count for different thresholds T in the CI-functionality adoption measure.	58
7.7. Distribution of clusters mapped to the distribution of distinct phase usage.	62

List of Source Codes

4.1. <i>YAML</i> syntax.	20
4.2. Exemplary minimal configurations in Travis CI.	24
4.3. Different types of phase changes.	27
7.1. Default denied.	47
8.1. Ambiguity in Travis CI configurations.	76

1. Introduction

Contents

1.1. Structure of this Thesis	1
---	---

Modern collaboration platforms allow software development to become an easy task for a large group of developers. But with more frequent collaboration comes the need to confirm the software’s correctness and quality and with growing amounts of contributions to a project the need for automating these tasks arises. Automated support is given by Continuous Integration systems where build automation is used.

CI builds are no simple execution any more, but rather has complexity risen to a high scale. That is because CI systems allow for diverse configuration of their CI build processes. Modern CI systems provide a variety of tasks that are not mandatory to a build and which can be wildly composed. This increases the amount of possible configurable build processes greatly.

On a CI platform (virtual) build execution environments are provided on the organization’s servers. Also, the setup of these environments is automated via a special configuration file. Using CI configuration files has many advantages for developers. Without having to take care of the execution environment, developers have no overhead when altering the configuration of their CI builds. Also, it is ensured that two equal configurations yield the same build process as only the configuration is user-dependent.

In this thesis I perform an explorative analysis on the change history of CI configurations from open source GitHub repositories, that use the Travis CI platform. The aim of this research is to gain knowledge on how OSS (open-source software) developers understand and use CI today. Therefore this research focuses on investigating what knowledge can be extra from the change history of CI configurations (i.e., historical build data). The main drive is to learn how build processes are iteratively constructed and maintained in OSS.

1.1. Structure of this Thesis

Firstly, some background knowledge for this research is presented, which is divided into a domain-driven and a methodology-driven part (chapter 2 (page 3)). Secondly I give my motivation for this research, which also includes the review of related work (chapter 3 (page 13)). Next to the latest topics from the mining software repositories field I present related work relevant to the topic of mining in the context of CI. In section 3.7 (page 18) the research methodology of this thesis is introduced, based upon the previous

methodology background (section 2.2 (page 7)), as a KDD-like approach together with GQM. The subsequent chapters then follow the methodology structure.

Chapter 4 (page 19) gives insight into this research data's origin and context. Afterwards, in chapter 5 (page 29) the most interesting features are identified and feasible research goals and questions are defined upon those features. Chapter 6 (page 35) concerns the preprocessing, i.e., the filtering of data and the formatting of the selected features, to enable later mining activities and the challenges that are faced during. Chapter 7 (page 43) then presents the findings of my research. All results are resumed in the conclusion together with suggestions for future work and threats to validity.

2. Background

Contents

2.1. Domain-driven Background	3
2.1.1. Git, a Version Control System	3
2.1.2. GitHub, a Collaboration Platform	6
2.1.3. Continuous Integration (CI)	6
2.1.4. Travis CI, a Continuous Integration Platform	7
2.2. Methodology-driven Background	7
2.2.1. Data Mining	8
2.2.2. Knowledge Discovery in Databases (KDD)	9
2.2.3. Goal Question Metric (GQM)	10

2.1. Domain-driven Background

2.1.1. Git, a Version Control System

A version control system (VCS) [Gitc] is a tool which keeps a change history for files in a special location known as a repository. Various VCS tools differ greatly in their internal procedures. The VCS relevant to this thesis is Git [Gitc], which was developed in 2005 [Gita]. A Git repository can be used for any set of files or for any other software unrelated purpose, but it is mostly valued by software developers to allow structured collaboration upon their source code and documentation. Due to the change tracking nature of Git, the information of who changed what and when is omnipresent.

Among others Git is a distributed VCS [Gita], meaning that every developer has a complete copy of the repository on his local machine. Changes can be committed locally and then pushed back into the main repository to ensure the integrity of the files. An arbitrary number of workflow models can be used with Git, e.g., the integration manager workflow [Gitf] where developers work indepently on their local copy of a ‘blessed repository’ and push their changes to their own repository. Then they ask for their changes to be pulled back into the ‘blessed repository’ by the integrator, a developer that functions as a controlling unit.

Git’s Internal Structure

A Git repository can be simply viewed as a dedicated file-directory and its contents. Additionally the change history of the repository, i.e., the change history of the directory and its contents, is stored in each of the repositories’ local copies with the aid of Git’s

2. Background

internal data structure. This consists of (1) a set of commit objects and (2) a set of tree objects [Gite]:

- Tree objects store a current snapshot of the repositories' content, comparable to a directory tree.
- Commits are the technicality with which changes in the local working directory are contributed to the main repository. A commit is also an object of Git's internal data structure which stores additional data for the corresponding tree object that represents the current change: a unique commit id, a timestamp, the person who introduces the commit, a mandatory commit message (which has to be used by the developer to explain what changes he is introducing) and references to one or more parent commit objects (one or more previous changes).

The linked list of commits puts the tree objects in a context and yields the change history of the repository. An exemplary scenario is depicted in figure 2.1.

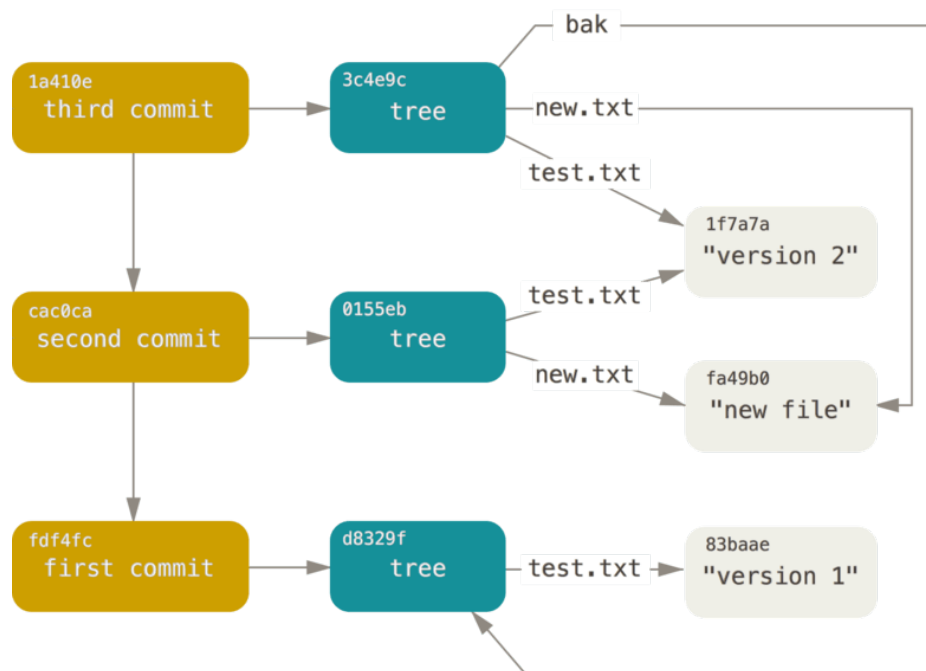


Figure 2.1.: Git's Internal Structure ([Gite], Figure 151).

Parallel Workflow in Git

Git also allows for parallel workflow with a feature called branching. Branches are created by figuratively branching of the current workflow creating a new independent one. They can then be used for experimental or independent feature development. Branches can

easily be disposed of or merged back into the original workflow or other branches and switching between branches is possible at any time.

Technically a branch is simply a pointer to a commit which represents the current version of the repository files in this branch. So creating branches does not create a new copy of the data. The main workflow of a repository, usually called the master branch, is also just a pointer to the current version indicating where the change history starts. The branch pointer can be redefined by the user himself but I omit such scenarios here. To know on which branch one is currently working Git uses another pointer called HEAD to point to the current branch pointer (cf. [Gitd]).

The change history of a branch is a list of commits that is connected by the commits' parent pointers starting at the commit that the branch pointer is pointing to. Directly after branching the original and the new branch point to the same commit (cf. figure 2.2) and after some time they only share an equal subset of commit objects in their history (cf. figure 2.3). After merging a branch back into the one it branched off from, the original branch's history also includes the one of the merged branch, and at some point in the change history a commit has two parent pointers (cf. figure 2.4). Git is well equipped with many additional features [Gitb]. More details on Git's internal working, features and best practices are omitted at this point due to relevance.

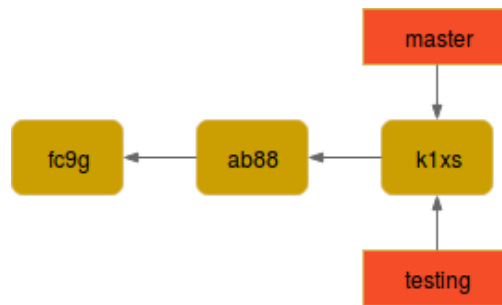


Figure 2.2.: Exemplary branching scenario in Git: Directly after branching.

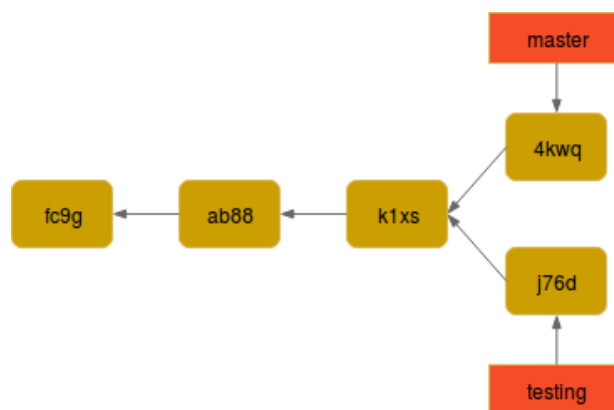


Figure 2.3.: Exemplary branching scenario in Git: Some time after branching.

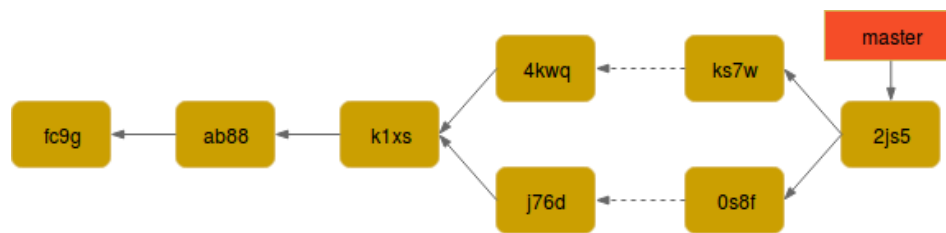


Figure 2.4.: Exemplary branching scenario in Git: After merging.

2.1.2. GitHub, a Collaboration Platform

GitHub [Ghub] is a commonly used platform for hosting Git repositories. GitHub established between late 2007 and early 2008 [Ghuc]. Since then, the community grew rapidly and the number of hosted repositories on GitHub exponentially grew to 65+ million [Ghua], with a lot of these repositories hosting open source software (19+ million open source repositories active in 2016 [Ghue]). Next to all the features of Git, GitHub by itself offers more features actively through apps in its Marketplace [Ghud] and passively through third party integration. Features include among others the customization and improvement of software development (e.g., in terms of Code Quality, Project Management, Chat Platforms or Continuous Integration).

2.1.3. Continuous Integration (CI)

Continuous Integration (CI) in its most basic understanding in the 1990's [Boo91] describes the practice in software development to integrate code changes early and often into the main repository or branch, at best multiple times a day [Awsa]. This prevents that developers, who implement features over a long period of time, are isolated from the current state of the software and makes integrating changes more easy.

In practice the understanding of CI is twofold. “Continuous integration refers to the build and unit testing stages of the software release process. Every revision that is committed triggers an automated build and test.” [Awsa]. Additionally to the ‘soft-skill’ of frequently committing to the main line, CI also has a technical side. Parts of the software release process should be run automatically for each change that is committed to the corresponding repository. The software release process can be broadly divided into five main phases: Software Compilation – Software Build – Test – Staging – Release/Deploy [Awsa]. An implementation of a software release process in the Context of CI is referred to as a Pipeline; phases in the software release process are called stages. The relevant stages for CI are the ones that occur directly after implementation: the software compilation, build and test stages. Automating stages does not only replace manual execution, which is error prone due to repetition, but also gives a direct feedback on the current software status so that no change is untested. The goal of CI is to improve productivity among developers, find bugs earlier, validate software faster, more trustworthy and release software with more confidence.

CI is a subdivision of the DevOps (Development & Operations) process which in general implies more agility through better interaction between different stakeholders of a software project (Developers, Operations, Quality Assurance) [Awsb]. In contrast to its siblings Continuous Delivery and Continuous Deployment, CI specializes in the stages that validate the software on a low-level (e.g., Unit Tests). It thereby validates the software's fitness for release or staging where high-level tests (e.g., Integration Tests, Performance Tests) are normally executed. Continuous Delivery and Continuous Deployment practices aim to automate the full software release cycle from a commit by a user to releasing the fully tested software, i.e., deploying it into production (cf. figure 2.5).

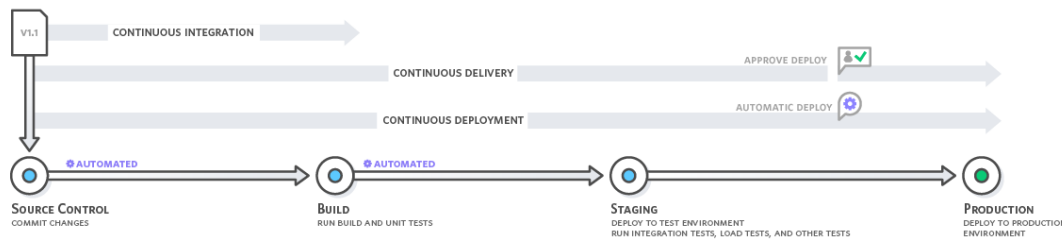


Figure 2.5.: DevOps practices compared [Awsa].

2.1.4. Travis CI, a Continuous Integration Platform

As already mentioned, a rather new feature offered towards repositories is the use of CI Systems. Travis CI (Travis) [Trac] is a distributed CI platform for projects hosted on GitHub and is used by 300.000+ open source projects [Trac]. It integrates with GitHub by cloning the repository on each commit and executing a corresponding CI build for the software's newest version. A CI build or a run is an execution of the CI build process. A CI build can be composed of many parallel or sequential tasks. Travis CI allows for high detailed configuration of these CI build processes. These mandatory configuration files lie in the root directory of a repository meaning that (1) they are collected and executed by Travis CI by cloning the repository and (2) underlay the version control of the repository itself. Remark that Travis CI is a platform and not a tool, i.e., a virtual target and build environment are provided for each CI build.

2.2. Methodology-driven Background

The practice of extracting knowledge from databases or large data sets is often referred to as Data Mining Science or Knowledge Discovery in Databases (KDD). Although these two terms are often used interchangeably they can be interpreted as two strongly-related, yet different practices as Han, Kamber, and Pei [HKP11] present in their introduction. I now want to separate these two terms by referring a definition for both.

2.2.1. Data Mining

Data mining is a misnomer [HKP11]: It is the practice of systematically analyzing, i.e., mining, a large data set to extract patterns and knowledge from it and not the mere extraction of raw data from data bases. Frawley, Piatetsky-Shapiro, and Matheus give a stronger definition: “This extraction of knowledge from large data sets is called Data Mining or Knowledge Discovery in Databases and is defined as the non-trivial extraction of implicit, previously unknown and potentially useful information from data.” [FPSM92]. Thus, data mining should comply with some basic characteristics.

Data Mining Characteristics

Knowledge characteristics. Knowledge extracted or discovered by data mining must be new knowledge, i.e., it cannot be extracted and evaluated by a DBMS query language or the like. Data mining searches for knowledge and hidden patterns that lie implicitly inside a data set, e.g., the correlation of many attributes. Knowledge extracted by data mining does not have to be defined by the database yet and knowledge extraction must not be trivial, i.e., “[...] a discovery system must possess some degree of autonomy in processing the data and evaluating its results.” [FPSM92]. Most importantly, the discoveries of a data mining application have to be potentially useful to be approved as knowledge. Furthermore, these discoveries must represent the original data to a certain degree.

Data characteristics. Data used for data mining can originate from almost any large data set. Many data mining applications work on growing data bases. In data classification applications for example, discovered knowledge is utilized to classify new data in the future. Also data bases need to represent real life data to provide valuable results. For detailed information on what data can be mined please refer to chapter 1.3 of [HKP11].

Furthermore, data mining is characterized by the use of data mining algorithms, which automate certain data mining techniques [FPSM92]. Data mining algorithms must have efficient run times. More characteristics on data mining shall not be considered at this point.

Data Mining Tasks and Technologies Used in Data Mining

The most important and basic data mining tasks are the following [ES00].

- Classification

A data set is given in which the instances are labeled, i.e., already assigned to a class. One wants to learn classification rules from the given data to classify future data instances correctly. In this thesis three classification algorithms are used: ZeroR, OneR and J48 [FHW16]. The ZeroR classifier does not consider any features for classification. Each instance in the data set is simply classified as the label which occurs most frequently. OneR classifies data instances by one feature

only. In this case the best predictor (i.e., the feature with lowest classification error) is chosen and classification rules are constructed for each value in the best predictor's value space. The third classifier is the simple J48 tree classifier. It is an implementation of the C4.5 decision tree building algorithm in the Weka tool [FHW16]. J48 can be parameterized by setting the minimum bucket size, i.e., the minimum amount of instances to be contained in each of the resulting decision tree's leaves.

- Clustering

The goal of clustering is to partition the data set into groups of instances that are similar. For example, this can be achieved by minimizing an algebraic distance between instance in the data, e.g., the euclidean or manhattan distance. In this thesis the SimpleKMeans algorithm is used [FHW16]. It has a parameter K , which states that K clusters are output. SimpleKMeans firstly chooses K random cluster means in the instance space. Then it iteratively performs two tasks until the clusters are stable: (1) Assign each data instance to the closest cluster mean (via an algebraic distance) and (2) update for each cluster the cluster mean.

Data mining can be considered to be an inter-disciplinary subject adopting concepts from many different research fields [HKP11]. Technologies conventionally used for data mining are:

- Machine Learning, such as Neuronal Networks or Bayesian Networks
- Pattern Recognition
- Statistics
- Clustering and Classification Algorithms
- Data Base Systems and Data Warehousing
- Visualization
- ...

Statistics provide a good foundation for data analysis. Although a purely statistic approach is only data-driven, since no domain knowledge is included, statistics are a valued tool used in data mining [HKP11]. Represented here are some basic data mining characteristic and techniques to obtain a basic insight into the large data mining field. For detailed information refer to [HKP11] [ES00].

2.2.2. Knowledge Discovery in Databases (KDD)

The KDD process extends data mining to a concept of a complete discovery process in which data mining is an integral part of the tooling in data analysis [HKP11]. Data mining is defined on a lower, more technical level. KDD on the other hand is defined on a higher, more conceptual level. In addition to data mining, the discovery process defined by KDD also regards pre-analysis steps, e.g., data extraction or data pre-processing, and

post-analysis steps as the evaluation of knowledge. A classic KDD discovery process, according to Han, Kamber, and Pei [HKP11], includes the following steps:

1. Data Cleaning (to remove noise and inconsistent data)
2. Data Integration (where multiple data sources may be combined)
3. Data Selection (where data relevant to the analysis task are retrieved from the database)
4. Data Transformation (where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations, for instance)
5. Data Mining (an essential process where intelligent methods are applied in order to extract data patterns)
6. Pattern Evaluation (to identify the truly interesting patterns representing knowledge based on some interestingness measures)
7. Knowledge Presentation (where visualization and knowledge representation techniques are used to present the mined knowledge)

Steps (1) to (4) represent preprocessing tasks which include the cleaning, integration, selection and transformation of the raw data for the purpose of gathering data and making it fit for analysis. Step (5) is the main analysis in which a suitable data mining technique of choice may be applied. Step (6) to (7) review and evaluate the discoveries and present the findings in a best suitable way. For a most reasonable discovery, all steps need to be performed in the context of the analysis goal, the chosen mining activity and the data source itself [HKP11].

The use of this KDD approach or KDD-like approaches can be verified by other academic literature, e.g., by Fayyad, Piatetsky-Shapiro, and Smyth's "From Data Mining to Knowledge Discovery: An Overview" [FPSS96] or by recent papers in ACM's annual KDD conference [Kdd].

2.2.3. Goal Question Metric (GQM)

Software metrics are quantitative statements about software or software-related processes [LL13], or respectively "a quantitative measure of the degree to which a system, component, or process possesses a given attribute." [Iee]. Diversity, reproducibility and plausibility are the most important requirements towards a metric [LL13], thus metrics are a useful model to answer research questions in a reliable and empirical way. Metrics can be divided into base metrics, whose computation solely uses the existing data, and derived metrics, whose computation relies on already defined base metrics [LL13]. However, metrics must be defined top-down [BCR94]. To know which metric to use and how to interpret it, a metric must be defined in a context given by appropriate models and

research goals; otherwise metrics are useless [BCR94]. Most software-related metrics output measures which belong to an absolute scale [LL13].

The Goal Question Metric (GQM) approach [BCR94], created in the 1980's, is a measurement model widely accepted and used in computer science academia. The model sets metrics in a context of research goals and questions. The GQM model is structured by three consecutive levels: The conceptual level (GOAL), the operational level (QUESTION) and the quantitative level (METRIC) (cf. figure 2.6). The goal states the purpose, object and issue of measurement and also the viewpoint from which the measurement is done. For each goal a set of questions characterizes the measurement of the object. Metrics can then be defined in the given context of goal and questions, so that its value answers the question in a quantitative way. Goals can share common questions and questions may relate to more than one metric. To obtain a GQM model

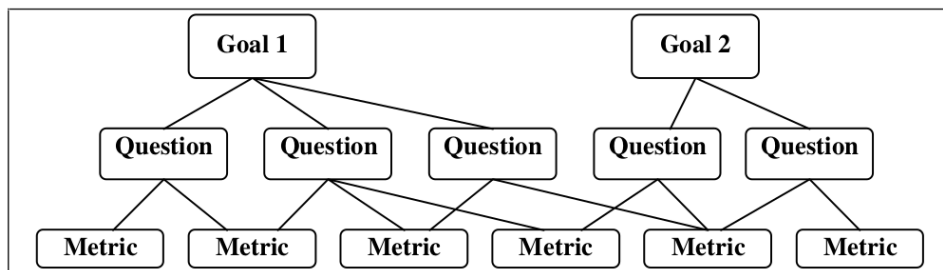


Figure 2.6.: The GQM model [BCR94].

one must first set one or more goals and identify the potentially useful data. Upon this data the goals can be characterized by a suitable set of distinct questions. Lastly provide metrics, that work upon the given data and produce useful measurement of the object, to answer the corresponding question.

3. Motivation

Contents

3.1. Topic of this Thesis	13
3.2. Why Historic Build Data is Interesting	13
3.3. The Mining Software Repositories Conference (MSR)	14
3.3.1. A Word on Mining Repository Data	15
3.4. Mining in the Context of CI	16
3.5. Purpose of this Thesis	17
3.6. What to Expect	17
3.7. Methodology	18

In this chapter I want to address the topic of this thesis again and motivate it accordingly. Then some related work on mining in the context of CI is reviewed. Lastly I state the purpose of this thesis, expectations towards this research and the methodology for my research.

3.1. Topic of this Thesis

Software development is a complex and interdisciplinary process, therefore I wish to contribute useful insights into the software development process. This research focuses on processes in the context of CI, to answer fundamental questions about the structure and evolution of such a process. Therefore I mine the historic build data, i.e., the history of CI configurations, of 900+ OSS projects that use the Travis CI platform.

3.2. Why Historic Build Data is Interesting

The motivation on mining historic build data is twofold: As CI is both a tool and a mentality the following always needs to be considered: For CI to work correctly not only is the use of collaboration and CI tools necessary but also must the correct CI mentality be incorporated by all developers. Neither does the use of a CI tool imply that CI is implemented correctly, nor is the awareness of CI Know-How without the necessary tools (for automation) always sufficient. As it is obvious that projects in this research's scope use reasonable tools, it is the mentality, or the usage of the CI tool respectively, that is to be analyzed. It is possible to verify correct usage by checking if a list of guidelines apply, for example:

- Do all developers commit to the main line on a daily basis?
- Does every commit trigger a build?
- Are build failures resolved within x minutes/hours ?
- ...

Such research soon reaches its limitations, due to its superficiality. I want to take research to a much more fine grained level by observing the complete change history of CI configurations. This allows for asking much more detailed questions about how CI is understood and implemented in OSS and maybe even why certain things are not done in the way they are prescribed in academia.

Furthermore does historic build data for CI build processes identify as a rather unexploited data set. I am therefore simply interested in what value the data presents in the first place and what knowledge is extractable. The preliminary analysis (cf. chapter 4 (page 19)) of this data set yields interesting and feasible research questions for this and possibly future research.

3.3. The Mining Software Repositories Conference (MSR)

The Mining Software Repository (MSR) conference is held annually with a wide range of topics related to mining. I here want to give a broad overview over the last three years of MSR with some examples for mining topics before I go into detail on related papers.

The side benefit of the change tracking nature of Git is that detailed insight into the software development process is also possible for research purposes. There exist VCS other than Git repositories, e.g., bug repositories, that also present themselves as an interesting (historic) data source due to an integrated change history. In the last decade the mining of repositories that are somewhat related to or have an impact on software development has gained a lot of attention. A considerable amount of contribution to this research field is given by the papers of the annual MSR conference [Msr].

The data sources used in the MSR can be any kind of software-related data, as e.g., software repositories, bug repositories, issue tracking systems, logs, communication archives, artifact repository or even software support forums. Not all research niches in the MSR conference are discussed each year and most research work cannot be associated with one niche only. In the following are some of the main niches that have emerged in the last three years of MSR. The conferences paper can be viewed in the ACM digital library [Proc][Prob][Proa].

Mobile Applications with a strong focus on Android. (Topics: Vulnerabilities in Mobile Applications, Errors in the Android Manifest, Evolution of Permission Requests, Energy Consumption in Android Applications, ...)

Social Mining and NLP, where mostly textual artifacts are mined that emerge during software development. Mining social aspects among developers and the impact of human

factors on software development goes along with this. In a majority of papers the support forum Stack Overflow is used as data source. (Topics: Sentiment Analysis in Software Development, Predicting Usefulness of Code Comments, Detecting Burnout Symptoms, Estimating Answering Times for Issues on Stack Overflow, Copy-Paste Behavior, Triage Developers for Change Requests Implementation, ...)

Testing, Bugs, Risks and Vulnerabilities (Topics: Usage of Testing Patterns, Studying Mocking Practices, Predict Issue Lifetime in GitHub Projects, Evolution of Technical Debt, Predicting Issue-Related Risks and Vulnerabilities, ...)

Source Code Mining, the classic niche which mines changes to the software's code. (Topics: Cross-Project Code Reuse, Exception Evolution in Java, Classifying Feature-Orientated Changes, Empirical Study on Architectural Changes in OSS, ...)

Dependencies and Licenses (Topics: Structure and Evolution of Package Dependency Networks, Dependencies in JEE, Detecting License Inconsistencies in OSS, ...)

Continuous Integration and Build (Topics: Empirical Analysis of GitHub's Docker Container Ecosystem, Usage of Static Analysis Tools in CI Pipelines, Empirical Analysis of Build Failures, Explorative Analysis of Travis CI with GitHub, Extracting and Classifying Build Changes from Maven Build Files...)

Meta Papers do not perform direct mining research themselves but review aspects of other papers in this conference. (Topics: Revision of MSR papers, Summaries, Methodology Reviews, Quality and Properties of Repository Data, MSR Tool Reviews, ...)

A *Data Showcase* is introduced in the year 2013. Papers present, possibly new, data sets from which the MSR community might benefit. These data sets are desired to be made available in a preprocessed and easy to use manner.

The *Mining Challenge* is held annually at MSR since 2006. This challenge motivates researchers to competitively mine a common data set. In 2017 the challenge is the TravisTorrent data set [BGZ17b], which holds information on CI builds, synthesized from both GitHub and Travis CI.

3.3.1. A Word on Mining Repository Data

Data sets used in MSR are never truly constructed for mining purposes. In contrast to data from a more business related (e.g., financial) origin, repository data is not intended to be mined for any kind of business benefit but rather used for recovery and issue tracking. However, the MSR shows that such mining is indeed possible (e.g., due to the existence of a change history).

This implies that extensive preprocessing might be needed. Also do a good deal of MSR papers perform an explorative analysis, as they simply cannot foresee what their data sources have to offer.

3.4. Mining in the Context of CI

A rather new feature offered towards projects or repositories is CI. The technical side of CI works tightly coupled with the corresponding software repository, thus CI belongs to one of the big niches in the MSR field. Mining CI related data has been done within the following areas:

1. Identifying CI Projects Characteristics [GVS17]
2. Testing Practices in CI [BGZ17a]
3. Usage, Costs and Benefit of CI [Hil+16]

Gautam, Vishwasrao, and Servant [GVS17] analyze software projects that use CI by clustering these by five main feature groups: activity, popularity, size, testing and stability. The goal of this research is the discovery of distinct characteristic for different groups of projects. The authors encourage projects to advertise their affinity to one of these groups for recruiting more suitable developers. The findings include four distinct groups: The first group of projects are highly test-driven, as they have a high test density and therefore also a large size (in KB) and fairly stable builds (73%). These project's popularity is rather low on average. Group two are the most popular and biggest (in LOC) projects. Their test case prioritization is very low, although the builds are mostly stable (74%). The authors link this to the project's low churn rate. Group three is the biggest group with projects that fit to the overall mean in all five categories. Group four are the projects that have a relative high amount of open issues and a low success rate on builds (51%). The clustering is performed with the SimpleKMeans algorithm.

Beller, Gousios, and Zaidman [BGZ17a], who also presented the TravisTorrent data set, perform an analysis on testing practices within the scope of GitHub projects that use Travis CI. The analysis focuses on (1) how commonly Travis CI is used, (2) how central testing is to CI and (3) how tests influence the build result by reviewing 2.600.000+ builds. Their results concluded that the adoption rate of Travis CI on GitHub is 30%, testing is central to a majority of builds (80%) and the test failures are the most common cause for broken builds.

Hilton et al.'s [Hil+16] work is not published within the MSR conference, but is of major importance to this thesis. The main drives of this research is to understand which CI service GitHub users prefer, how developers use CI and why certain projects to not use a CI service. The latter is answered by a survey with 422 developers. Of interest are the first two drives:

They found that about 40% of GitHub projects use a public CI service and that a majority of CI projects use the Travis CI platform (90.1%). For a more detailed analysis on how CI is used, the authors identify the 1000 most popular CI projects on GitHub, and extensively gather data on the 620 projects that use Travis CI. A part of this detailed analysis includes to ask how developers evolve their CI configuration. Hilton et al. observe

that on average the CI configuration is changed up to 12 times per project. Additionally, they analyze to which crucial areas of the configuration changes are made during project lifetime. They present a statistic for the change counts towards each of these configuration areas. In total their research concludes that CI is a growing trend, as CI projects have more frequent releases and more confident developers.

3.5. Purpose of this Thesis

All in all this thesis is motivated by (1) the unexploited nature of the historic build data and (2) comprehensively learning about CI usage in OSS through this data set.

This research serves the purpose of attempting to gain a more in-depth knowledge on how the world understands and uses CI by looking at small representative group of 900+ OSS projects that use the Travis CI tool. A deeper understanding might aid in decision making in future software development projects. The evaluation of this data set's value may not be the core goal of this research, but is a mandatory step before any mining can be done. Also shall the preliminary analysis (cf. chapter 4 (page 19)) motivate to utilize historic build data from other origin for similar research. Due to (1) I try to find answers in this data to a set of basic questions only.

3.6. What to Expect

As this is an explorative research and highly data-driven, goals and questions are constructed in an iterative process during preliminary data analysis (cf. chapter 4 (page 19)). Therefore I want to briefly anticipate that questions are asked more or less with respect to the following subgoals:

- Acceptance of the Travis CI Model
- Robustness of the Travis CI Model
- Build Process Evolution
- Build Process Structure Build-Up
- Equivalent Usage of the Travis CI Model

Firstly the acceptance and robustness of the Travis CI model are of interest: To what extend are parts of the model utilized? And how robust is the model towards changes? Next, I observe when significant parts of the model are changed or used. Then I view the most recent state of the CI processes available in the data and try to determine how functionality is incrementally adopted. This later reveals that a measure for maturity of build processes is derivable. Lastly projects are clustered to observe equivalent usage of the model.

3.7. Methodology

This research is mainly explorative and highly data-driven. So I orientate myself towards an KDD-like approach in combination with a GQM model (section 2.2 (page 7)), for defining my goals, questions and suitable metrics. My research methodology is given by the following steps.

1. Preliminary Data Analysis
2. Goal Definition
3. Data Preprocessing
4. Mining
5. Evaluation

Firstly I review the available data, especially its usefulness for this research. This results in finding two interesting sources: the change history of 900+ GitHub projects that use the Travis CI platform and the build results of these projects' CI builds. To formulate goals with GQM first feasible features are located by performing a preliminary data analysis of the data's background. This includes the structure of Travis CI configurations and the Travis CI build life-cycle.

After knowing what information the data holds, research goals, questions and suitable metrics are constructed via a GQM approach. With this the research scope is finally set. Data preprocessing handles the filtering of data and the formatting of the selected features to enable later mining activities. In the end I present and discuss my findings and interesting observations.

4. Preliminary Data Analysis

Contents

4.1. Data Selection	19
4.2. Build Execution Logs	19
4.3. Build Configuration in Travis CI	20
4.3.1. The <i>.travis.yml</i> Syntax	21
4.3.2. The <i>.travis.yml</i> Semantics	21
4.3.3. The Minimum <i>.travis.yml</i>	23
4.4. Conformance to Classic CI	24
4.5. Research Focus	24
4.6. Definition of Terms	25

This preliminary data analysis deals with the review of available data related to this research topic to form a basis for finding feasible research goals. After data selection, this includes the gathering of in-depth knowledge about the data’s origin and context, which shall be presented here. This chapter only regards the data that is finally selected for research. The extracted research questions and the integration and transformation of the data to a suitable format for later mining or analysis activities follows in chapter 5 (page 29) and chapter 6 (page 35).

4.1. Data Selection

Two interesting data sources exist for this research: Firstly the Git change history of OSS projects on GitHub and secondly the raw build logs for the CI builds of these projects available on the TravisTorrent website [Trad]. TravisTorrent is the data set used in the MSR’2017 Mining Challenge [BGZ17b]. At the time this thesis started the latest data snapshot is recorded on December 20th, 2016, which sets this analysis’ time limit. From these data sources the project’s ages and the project’s CI-usage periods are extracted easily by the dates of the first commit and the first commit that introduced the project’s CI configuration file.

4.2. Build Execution Logs

Build execution logs are the accumulated output of all tasks that are executed in a CI build. The TravisTorrent data set presents valuable data, i.e., build results and similar, that were extracted from raw Travis CI build logs. The data set is stored in a structured format to provide easily accessible data for analysis of the Travis CI platform. The

projects considered by TravisTorrent must meet the following requirements: a minimum amount of watchers on GitHub(> 10) and builds on Travis CI (> 50) [BGZ17b]. Next to the live online accessible data set, the TravisTorrent website also offers regular offline snapshots and the raw build logs for 900+ GitHub projects [Traa].

During preliminary data analysis the raw build logs are searched for relevant data. From the unstructured logs it is possible to extract the tasks that are executed in the CI build but this data is also retrievable from the project's CI configuration, which also happens to be stored in a structured format already. This concludes that only the build results are a useful feature that is uniquely retrievable from the build logs. Yet I orientate myself by the set of projects for which the raw build logs are accessible on TravisTorrent [Traa], as they present a suitable base for this analysis.

4.3. Build Configuration in Travis CI

Next to the build logs, the CI build configurations are a more valuable source for research. A CI configuration holds direct information on the parts of which a CI build is composed. In Travis CI this configuration file is present in the root directory of the project's Git repository, therefore it underlies Git's change history, which can be accessed easily.

The Travis CI build configuration is the `.travis.yml` file. This file is of type *YAML* (**Y**et **A**nother **M**arkup **L**anguage) which is a common file format, widely used for data serialization or configuration. *YAML* files are close to the *JSON* file format, due to data being stored as key value pairs in a tree structure of nested lists and maps. Yet they have a more minimal syntax compared to *JSON* due to indention being used instead of brackets and braces.

```
1 | key_simple: value
2 | key_map:
3 |     key1: value2
4 |     key2: value2
5 | key_list:
6 |     - element1
7 |     - element2
8 | # This is a comment
9 | key_complex:
10 |     - key1: value1a
11 |       key2: value1b
12 |     - key1: value2a
13 |       key2:
14 |         - element2a
15 |         - element2b
```

Source Code 4.1: *YAML* syntax.

It is important to mention that the `.travis.yml` configuration is of declarative nature, which means that these configurations only shows of which tasks a build is composed, but not necessarily determine how these tasks are executed. A *YAML* file has the syntactic structure as shown in listing 4.1. The most atomic parts of the CI configuration are the keys in the `.travis.yml`. A top-level key is a key in the root of a tree of nested lists and maps (e.g., `key_simple`, `key_map`, `key_list` and `key_complex` in listing 4.1).

4.3.1. The *.travis.yml* Syntax

The most important feature is the presence of top-level keys in the configuration file, since they express the highest semantic value. Some of these are store in a *key:value* fashion, with a predefined value space; others are composed of a list of user defined shell commands. All available top-level keys for CI configurations in Travis CI can be viewed in table 4.1.

4.3.2. The *.travis.yml* Semantics

Some top-level keys can be grouped as they express similar semantics in the build process:

- (1) - (11) States in the Travis CI state machine
- (12) - (13) Configuration of complex builds
- (14) Notifications
- (15) - (79) All kinds of environmental configuration (platform, language, cache, build,...

Some keys map to the states in the Travis CI state machine as depicted in figure 4.1 which illustrates the build life-cycle in Travis CI [Trab]. These include the main states given by the keys *install*, *script* and *deploy*. Furthermore some intermediate states for additional (before... / after...) or conditional (*after_success* / *after_failure*) configuration exist. All of these states are mapped to only one specific top-level key in the configuration. Each of these keys, except for *deploy*, hold a list of shell commands that are executed as one command block. The command blocks defined under a key are executed in the order in which the key's states appear in the state machine (cf. figure 4.1) starting at *before_install*. The use of all the keys (1) - (11) is not mandatory, most of these keys can be omitted, e.g., *before_cache* is optional (cf. section 4.3.3).

If a build fails at some point, the build result depends on the state in which a command returned an error code [Trab]. The result of the build can either be a success, a failure, an error or an abort. An abort can occur at any point of time, whenever the user manually aborts the build. The build is errored as soon as the build fails in one of the first three pre-*script* states or in one of the *before*-states. As soon as a build is errored the build stops. A build failure is due to a fail in the *script* state. After a failure the build still continues with the subsequent states. All other states have no impact on the current build result. E.g., *after_success* is executed after a successful execution in the *script* state, but a failure in this state has no impact on the overall build result.

All other keys relate to state independent configurations [Trab]. Keys (12) and (13) allow for definition of more complex builds by stating multiple run times versions, environments etc.; for each combination one job is issued in a run. The *notifications* key (14) provides configuration of notification mechanisms or an integration different from the default email notification. Furthermore, recipients and notification triggering events

4. Preliminary Data Analysis

ID	Top-level Key	ID	Top-level Key
1	before_install	41	ghc
2	install	42	gobuild_args
3	before_script	43	go_import_path
4	script	44	go
5	before_cache	45	haxe
6	after_failure	46	hxml
7	after_success	47	jdk
8	after_script	48	julia
9	before_deploy	49	language
10	deploy	50	lein
11	after_deploy	51	mono
12	jobs	52	neko
13	matrix	53	node_js
14	notifications	54	otp_release
15	addons	55	pandoc_version
16	cache	56	perl6
17	dist	57	perl
18	group	58	php
19	os	59	podfile
20	osx_image	60	python
21	branches	61	r_binary_packages
22	git	62	r_build_args
23	env	63	r_check_args
24	services	64	r_check_revdep
25	sudo	65	repos
26	android	66	r_github_packages
27	apt_packages	67	r
28	bioc_packages	68	r_packages
29	brew_packages	69	rust
30	bundler_args	70	rvm
31	compiler	71	sbt_args
32	cran	72	scala
33	crystal	73	smalltalk
34	dart	74	solution
35	dart_task	75	warnings_are_errors
36	disable_homebrew	76	xcode_project
37	d	77	xcode_scheme
38	dotnet	78	xcode_sdk
39	elixir	79	xcode_workspace
40	gemfile		

Table 4.1.: Top-level keys of *.travis.yml* [Trab].

may be configured. Keys (15) - (79) allow for all kinds of configuration to the platform and build environment, e.g., the OS image to be used or the programming language that is used.

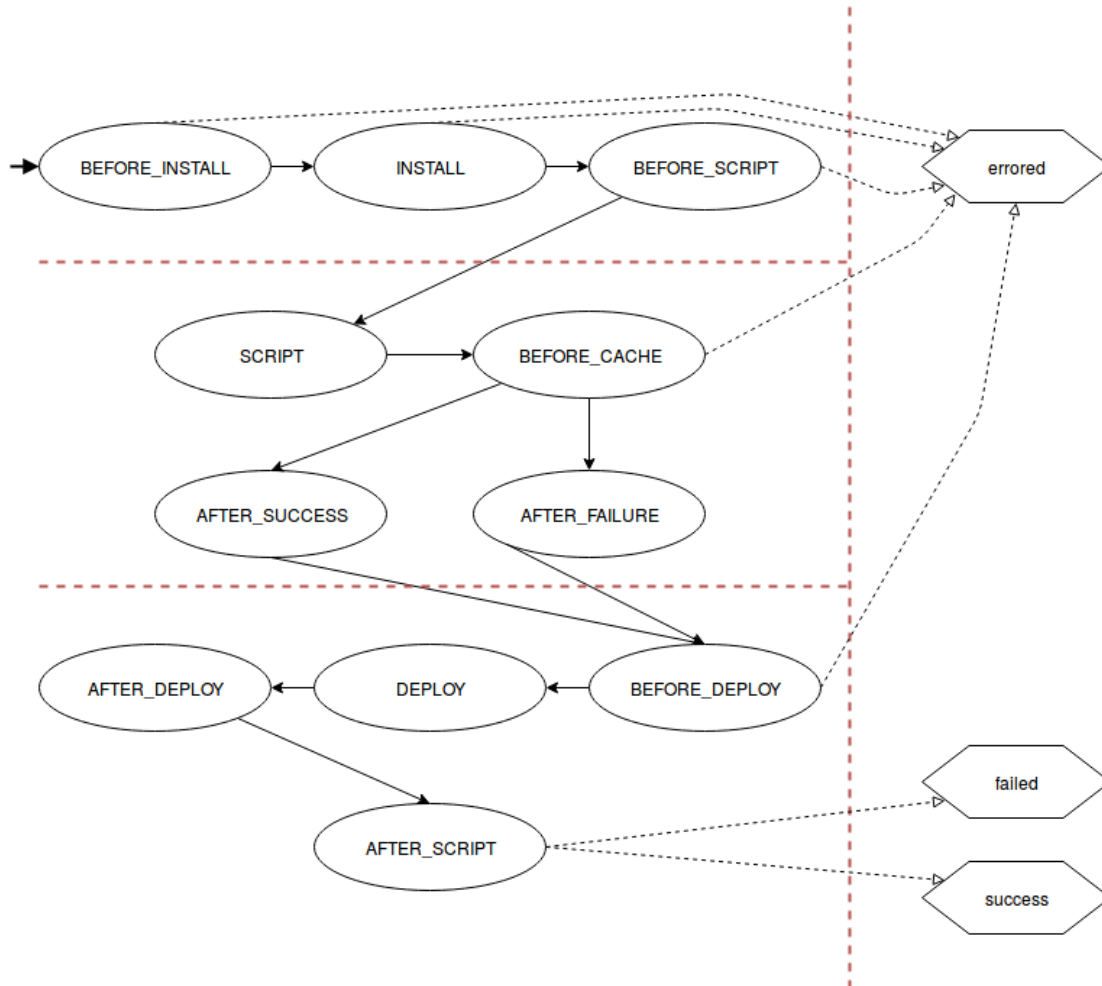


Figure 4.1.: Travis CI state machine.

4.3.3. The Minimum `.travis.yml`

Travis CI provides a set of default executions which has to be regarded in later interpretation of mining results. The Travis CI documentation [Trab] prescribes only the `addons`, `cache`-related and `deploy`-related keys to be optional, yet any key can be omitted, i.e., not used in the configuration. But not all incomplete configurations yield a build process. The minimum requirements for a semantically correct configuration in Travis CI is the use of either the `language` or `script` key (cf. listing 4.2). As `script` holds the main

execution of the build, the use of only this key suffices. Also does Travis CI perform a default language dependent build execution, if only the *language* key is set. Furthermore, default email notifications are implemented, if the *notifications* key is not used.

```
1 | script: mvn build           1 | language: java
```

Source Code: Exemplary minimal configurations in Travis CI.

4.4. Conformance to Classic CI

The classic CI pipeline consists of at least the main stages *Checkout*, *Compile* and *Unit Tests* [HF10]. Afterwards, artifacts or the like may be pushed somewhere for further tasks in the CD context. In Travis CI the repository checkout is triggered automatically on each commit per default. Git options, build environment (e.g., services) and the platform environment, on which the build is executed (e.g., distribution), are highly configurable. Such pre-execution configuration is not considered for the CI-related parts of the theoretical pipeline model [HF10], but is widely used in the analyzed Travis CI configurations. Execution of compilation and unit test tasks is not clearly separated in Travis CI, but recommended to be combined in the *script* key. Fast feedback in Travis CI is handled by default notifications via email, which are sent to all Git contributors on each finished build. Alternative notification mechanisms may also be configured. An optional deploy stage is provided by Travis CI to push artifacts to a given set of supported providers, or to a destination of choice by custom scripting. Especially this deploy functionality extends the Travis CI model to some sort of simplified version of the CD model (cf. section 2.1.3 (page 6)).

All in all the Travis CI model provides at least all the functionality of a CI pipeline. In general a correct usage of the configuration is hard to validate, since there are a lot of parts in the configuration which can be custom scripted. The custom scripting parts of the configuration primarily provide a grouping of similar tasks, yet some yield different build results if a failure occurs in one of the executed tasks (cf. figure 4.1).

4.5. Research Focus

I identified the changes to the configuration file as an interesting research focus. When analyzing changes to the configuration it is not of interest what the new values are but to which keys the changes are made. Therefore, this research is based on changes to the CI configuration file, or usage of and changes to top-level keys in the CI configuration respectively. Furthermore, the research is restricted to the Git history of the projects' master branches. The YAML file format in which the CI configuration for Travis CI is stored can be parsed and compared with other instances of this file format. Thus, I pairwise compare consecutive versions of the CI configuration file for each project to extract the change history of the project's CI build process.

4.6. Definition of Terms

Here I want to clearly state the definition and use of some terms for the rest of this thesis. When analyzing changes to the configuration it is not reasonable to analyze changes to all 79 top-level keys. I identified groups of keys with similar semantics. Hence I define a phase by a set of one or more top-level keys in the `.travis.yml` configuration file, that express unique semantics compared to all other keys. A list of all phases and top-level keys can be found in table 4.2.

Furthermore, changes are defined. To each project there belongs a set $(1, \dots, n)$ of n commits, that alter the CI configuration in some way. A *configuration change* refers to the syntactical change to the configuration file, via a Git commit in this case. A change in the configuration file may lead to a change in the CI build process. This semantic impact of the configuration change on the build process is what I refer to as a build process change, or *build change* respectively, which is the core aspect of this thesis. As already mentioned a top-level key is the most atomic part in a Travis CI configuration file with the highest semantic value. A change to a top-level key occurs if the value of the key or the value of one of the key's sub-keys is changed. A *phase change* for phase X occurs whenever at least one of the top-level keys, that belongs to X , is changed semantically. Conclusively each project's history contains a series of configuration changes. If at some point there is a semantic change (e.g., not just a comment added), this yields a build change. A build change can then include one or more phase changes.

The commit that introduces a configuration change (or build change or phase change respectively) is the *configuration changing commit* (or *build changing commit* or *phase changing commit* respectively). In case a configuration change does not lead to a build change, i.e., it is a non-semantic change, a change of whitespace or comments inside the configuration is assumed, for which the *FORMATTING* phase is used. *FORMATTING* is no phase related to Travis CI directly, but serves as a default phase. All non-semantic changes to the configuration file are accounted to this phase.

In more detail a *phase change* can be the inclusion, exclusion or maintenance of a phase's keys. Listing 4.3 depicts these three scenarios together with an example for a non-semantic change.

ID	Phase	Top-level Keys
1	BEFORE_INSTALL	before_install
2	INSTALL	install
3	BEFORE_SCRIPT	before_script
4	SCRIPT	script
5	BEFORE_CACHE	before_cache
6	AFTER_FAILURE	after_failure
7	AFTER_SUCCESS	after_success
8	AFTER_SCRIPT	after_script
9	BEFORE_DEPLOY	before_deploy
10	DEPLOY	deploy
11	AFTER_DEPLOY	after_deploy
12	BUILD_STAGES	jobs
13	BUILD_MATRIX	matrix
14	NOTIFICATIONS	notifications
15	ADDONS	addons
16	CACHE_ENV	cache
17	PLATFORM_ENV	dist, group, os, osx_image,
18	GIT	branches, git
19	BUILD_ENV	env, services, sudo
20	LANGUAGE_ENV	android, apt_packages, bioc_packages, brew_packages, bundler_args, compiler, cran, crystal, dart, dart_task, disable_homebrew, d, dotnet, elixir, gemfile, ghc, gobuild_args, go_import_path, go, haxe, hxml, jdk, julia, language, lein, mono, neko, node_js, otp_release, pandoc_version, perl6, perl, php, podfile, python, r_binary_packages, r_build_args, r_check_args, r_check_revdep, repos, r_github_packages, r, r_packages, rust, rvm, sbt_args, scala, smalltalk, solution, warnings_are_errors, xcode_project, xcode_scheme, xcode_sdk, xcode_workspace
21	FORMATTING	default change

Table 4.2.: Phases defined by top-level keys of *.travis.yml*.

	Pre-Change	Post-Change
Inclusion of the <i>SCRIPT</i> phase.	1 language: java	1 language: java 2 script: 3 - mvn compile
Exclusion of the <i>GIT</i> phase.	1 language: java 2 git: 3 depth: 3	1 language: java
Maintenance of the <i>LANGUAGE_ENV</i> phase.	1 language: java 2 jdk: 3 - oraclejdk8	1 language: java 2 jdk: 3 - oraclejdk9
A non-semantical change.	1 language: java 2 script: 3 - mvn compile	1 language: java 2 # comment added 3 script: 4 - mvn compile

Source Code: Different types of phase changes.

Furthermore can the usage of a phase be determined by the current phase change. *Phase usage* starts for a phase *X*, if a current change to phase *X* exists and it is either an inclusion or maintenance. For the case that the phase change is an exclusion, then one knows that the phase *X* is not used anymore. Conclusively the core aspects and terms for this thesis are defined. Here the distinction between inclusion, exclusion or maintenance of a phase solely serves the purpose to derive a concrete definition for phase usage. This distinction is not made when studying phase changes in this research (cf. section 6.1 (page 35)).

Further on the relation between parts of the configuration and the main stages of a CI pipeline (cf. section 4.4) is of interest. Therefore I define CI-functionalities as groups of phases (cf. table 4.3). Again the *change* and usage, or rather *adoption*, of a CI-functionality is defined by the change to or usage of at least one of its phases. A more detailed definition for both the CI-functionality adoption and consistent phase usage is derived in section 6.1 (page 37).

CI-functionality	Phase ID	Phase
PRE_EXEC	2	INSTALL
	12	BUILD_STAGES
	13	BUILD_MATRIX
	15	ADDONS
	16	CACHE_ENV
	17	PLATFORM_ENV
	18	GIT
	19	BUILD_ENV
	20	LANGUAGE_ENV
	EXEC	4
BEFORE_AFTER	1	BEFORE_INSTALL
	3	BEFORE_SCRIPT
	5	BEFORE_CACHE
	6	AFTER_FAILURE
	7	AFTER_SUCCESS
	8	AFTER_SCRIPT
	9	BEFORE_DEPLOY
NOTIFICATIONS	11	AFTER_DEPLOY
	14	NOTIFICATIONS
DEPLOY	10	DEPLOY

Table 4.3.: CI-functionalities defined by phases.

5. Research Goals and Questions

Contents

5.1. Goals	29
5.2. Feature Selection	34

This chapter presents the research goals of this thesis. Due to the explorative nature of this study, the construction of research goals is a highly data-driven process and conducted iteratively during data analysis. Different features in the data motivated certain research questions which were directly combined with a suitable metric and then later grouped by similar themes. This resulted in a GQM-like model (cf. section 2.2.3 (page 10)) where the goals represent the gathering of knowledge with respect to the stated theme. These five goals are extracted throughout data analysis:

1. Acceptance of the Travis CI Model
2. Robustness of the Travis CI Model
3. Build Process Evolution
4. Build Process Structure Build-Up
5. Equivalent Usage of the Travis CI Model

The following section presents the goals with more details on the questions and metrics.

5.1. Goals

For each goal it is briefly described by which questions they are characterized and how these are answered by metrics.

GOAL 1: Acceptance of the Travis CI Model

- a) Question: Which phases of the model are most frequently used?
 - Metric (base): The amount of projects that use phase X at least once. As this metric counts the absolute frequency it maps to an unbounded absolute scale. The higher the metric's output the more is phase X accepted. To compare the different phases, the metric's output for all phases is given in a ranking with the highest output expressing the highest rank. Thus phases that are lower ranked are used less.

- b) Question: Which phases of the model are most frequently changed?
- Metric (base): The total amount of changes to phase X . As this metric counts the absolute frequency it maps to an unbounded absolute scale. The higher the metric's output the more is phase X accepted. To compare the different phases, the metric's output for all phases is given in a ranking with the highest output expressing the highest rank. Thus phases that are lower ranked are changed less.
- c) Question: How many phases are used per project?
- Metric (base): The absolute amount of distinct phases ever used per project. As this metric counts the absolute frequency, it maps to the integer scale $[0,21]$, which is limited by the maximum amount of phases (cf. section 4.6 (page 25)). For comparison and due to the limit on this scale, a histogram is constructed with the amount of projects whose metric maps to x for every $x \in [0, 21]$.
- d) Question: What is the volatility of phases?
- Metric (derived): The ratio of absolute changes to absolute usage per phase. This metric expresses the average amount of times that a phase is changed per project. It is derived from the metrics of both questions 1.1 and 1.2. The scale of this metric is unbounded, therefore a ranking is given for comparison.

Starting with simple needs, the first goal investigates the acceptance of the Travis CI model by the projects under study to provide insights into how Travis CI is generally used. The acceptance of the model is characterized by the four questions above to which four simple statistical measures provide answers. After an analysis of phase changes both a ranking of the most used and the most changed phases in the Travis CI model give an overview of most accepted or favored phases of the model. Thirdly a histogram visualizes the distribution of distinct phase usage. From this one can learn to which extend the whole model is used per project. Lastly the volatility, i.e., the average change frequency for a phase per project, of phases is reviewed. This gives another ranking which shows if projects focus on maintaining special phases.

GOAL 2: Robustness of the Travis CI Model

- a) Question: Which phases are critical, concerning the build results of the corresponding builds after a phase change?
- Metric (base): The amount of negative build results after phase X is changed. As this metric counts the absolute frequency it maps to an unbounded absolute scale. The lower the metric's output the more the phase X is regarded robust. To compare the different phases the metric's output for all phases is given in a ranking, with the highest output expressing the highest rank. Thus for phases that are lower ranked the build fails less often after a change.

- b) Question: Does a correlation between change frequency and amount of negative build results exist?
- Metric (base): The correlation coefficient of Weka's linear regression classifier when applied to the set of phase change frequency and number of failed builds. This classifier only finds linear correlations, thus a good correlation coefficient means that there is a constant amount or percentage of the phase changes which always cause the build to fail.
- c) Question: Do phases cause build failures together?
- Metric (base): the number of final clusters with minimal SME (cf. appendix A (page 79)) when clustering upon phase changes for data instances, that have a failed build. The number of final clusters reveal whether phases cause build failures together and further do the final cluster means reveal which phases exactly. With lower SME it can be ensured that the final cluster means represent the cluster with best precision.

The second goal aims to give a measure for the robustness of the Travis CI model by identifying critical phases in the build process that often tend to cause a build failure after they are changed. Firstly, a ranking is given for the absolute amount of build failures that are caused by a change to phase X . Then the correlation between the amount of changes to a phase and the amount of unsuccessful builds that were executed upon a phase change is evaluated using linear regression. Phase change frequency is simply measured by non-negative integers. For the build results one can accumulate the amount of non-successful builds of the builds directly executed after a phase change, so the scale would also measure non-negative integers. By clustering on the phase changes of instances that have a failed build it can be revealed which combinations of phases are commonly changed together before a build fails.

GOAL 3: Build Process Evolution

- a) Question: When are phase changes made in a project's CI-usage period?
- Metric (base): To answer this question change-frequency graphs are visualized. In this case a Boolean metric could relate to the existence of one specific property of the graph. The metric that relates to the graphs convergence to a constant value is of most interest. Such property describes that the change frequency stabilizes, i.e., that the build process stabilizes.
- b) Question: When are phases used in a project's CI-usage period?
- Metric (base): To answer this question usage-frequency graphs are visualized. In this case a Boolean metric could relate to the existence of one specific property of the graph. The metric that relates to the property that the graphs curve is on average linear is chosen here. It describes if the phase usage is constant or not.

c) Question: Can foci on significant phases be found in these graphs?

- Metric (base): Again a Boolean metric for the graphs of questions 3.1 and 3.2. The existence of maxima in the frequency graphs is of interest as such property confirms that phases have a dedicated interval in which they are focused on.

This goal incorporates the time dimension and the analysis of build process evolution in terms of phase usage and changes over time is central. First it is observed when phase changes are made, secondly in which intervals phases are commonly used and lastly if there exist special foci on phases. Therefore the phase change and usage frequency graphs need to be constructed, wherefore the standardized CI-usage period and the relative time of each build change is used. Due to standardization all projects are then comparable in the same graph model that plots time versus change or usage frequency. The desired graph has the time axis $[0, 1]$ and an integer count axis to represent the frequency of phase changes to a certain point in time or the amount of projects that use a phase to a given point in time respectively. Alternatively, for a clearer comparison, all projects whose real CI-usage period lies in a similar interval, are compared to each other in an individual graph. This results in multiple graphs, one for each combination of the chosen interval and the phase for which it is constructed (local). The graph is also constructed for combinations of intervals and all phases (global).

GOAL 4: Build Process Structure Build-Up

a) Question: Are CI-functionalities used in a consistent manner?

- Metric (base): The CI-functionality adoption count for different threshold parameters T . This metric maps to an absolute scale. If the adoption count is high for a high threshold T , then this means that functionalities are indeed used in a consistent manner. Furthermore, if the metric's gain from one high threshold to a subsequently lower threshold is small, then this also implies that this thresholds is a good lower bound for measuring consistent usage.

b) Question: How mature are CI build processes in Travis CI?

- Metric (base): The amount of CI-functionalities adopted by a project. This metric maps to an absolute scale. The more functionality a project adopts, the more mature it is regarded. Furthermore, the projects are clustered upon a vector, that expresses the order in which CI-functionality is adopted in a project. The final cluster means refine the maturity measure once more through the order property.

c) Question: What is the adoption time for CI-functionalities in GitHub projects?

- Metric (base): Mean time of adoption for each CI-functionality. This shows the average order in which functionalities are adopted and thus the build-up of the build process structure.

- d) Question: What is the impact of project age, CI-usage period, phase usage and phase changes on CI-maturity?
- Metric (base): The precision of certain classifiers on different feature subsets, attempting to classify projects upon the CI-maturity levels. The metric maps to the scale $[0,1]$ as it represents the percentage of correctly classified instances. The higher this value is, the better the feature subset indicates maturity.

The fourth goal tries to answer how a CI build process is iteratively constructed, i.e., how CI-functionality is adopted over time. Therefore the previously defined CI-functionalities and the parameterizable adoption measure are used (cf. section 4.6 (page 25)). Firstly the quality of different adoption thresholds is observed, to conclude if functionalities are used consistently. Adoption sequences are represented in form of a vector. Each element of the vector is assigned to a CI-functionality and its value represents the chronological order in which the CI-functionality is adopted. Further CI-maturity levels are identified by clustering on these vectors. Each maturity level is then represented by a group of projects that use the same amount of CI-functionality which are also introduced in a likewise order. The mean time of adoption for the different CI-functionalities is also examined. The last questions investigates the relationship between project-level features (cf. chapter 6 (page 35)) and the CI-maturity levels by attempting to find good classifiers. If such good classifiers exist, then project-level features are good indicators for CI-maturity which is originally defined on commit-level features (cf. chapter 6 (page 35)).

GOAL 5: Equivalent Usage of the Travis CI Model

- a) Question: Can projects be clustered by similar usage of phases?
- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon a binary vector of phase usage.
- b) Question: Can projects be clustered by similar phase changes?
- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon a real vector of phase changes.
- c) Question: Can projects be clustered by amount of configuration changes?
- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon total amount of configuration changes.
- d) Question: Can projects be clustered by density of phase changes?
- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon density of phase changes over time.
- e) Question: Can projects be clustered by age?
- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon age.

f) Question: Can projects be clustered by CI-usage period?

- Metric (base): A Boolean metric, that answers the question with a yes or a no. The projects are clustered upon CI-usage period.

The last goal aims to find clusters of projects under the perspective of equivalent usage of the model. The clustering capability of six different (groups of) features is investigated. If the feature subsets can be clustered upon, then the resulting final cluster means are evaluated which show common usage patterns for Travis CI. For the first two, time is not relevant: Phase usage per project can be expressed by a vector with one value per phase. A binary vector indicates for each phase whether it is ever used in a project. Therefore it seems suitable to cluster with a manhattan distance. Phase changes can be expressed by real vectors, this time with integer values for the amount of changes to a certain phase. In this case an euclidean distance is reasonable for clustering. The next four clustering applications cluster on one-dimensional features only: The total amount of configuration changes in a project, the project's age in seconds, the project's CI-usage period in seconds, and the phase change density of a project.

5.2. Feature Selection

During preliminary data analysis (cf. chapter 4 (page 19)) interesting features from the collected data are already identified. To answer the goals the following features are needed (cf. table 5.1).

Features:

- (i) Project age
- (ii) CI-usage period
- (iii) Knowledge of Travis CI Model
- (iv) Phase Changes for each configuration change
- (v) Phase Usage after each configuration change
- (vi) Relative point of time for each configuration change
- (vii) Build results for the associated builds

Table 5.1.: Features selected for this research.

Features (i) to (iii) have already been extracted, features (iv) to (vii) are captured by data preprocessing.

6. Data Preprocessing

Contents

6.1. Feature Vector Modeling	35
6.2. Challenges	39
6.3. First Observations	41

Preprocessing extracts the relevant data, i.e., the selected features (cf. section 5.2 (page 34)), from the data sources. To later apply algorithms the extracted features are stored in the structured *ARFF* (**A**tttribute-**R**elation **F**ile **F**ormat) file format [Arf].

6.1. Feature Vector Modeling

Features are recorded in two different granularity levels: On commit-level data instances are recorded for each configuration changing commit of each project. They hold information on the changes to each phase and usage of each phase after a configuration change.

Instances on project-level have a higher abstraction level as data instances are only recorded per project. Information on all changes and usage is summarized for each project, thus data instances on this level have a more extensive but less detailed view on the projects.

Similar features are grouped into the feature vectors given below.

Commit-Level

On commit-level the different feature vectors hold the following features with the according domains:

$$F_1 = (c_1, \dots, c_n, c_{sum})$$

- $c_i \in \{0, 1\}$ with $c_i = 1$ indicating phase i is changed, else $c_i = 0$
- $c_{sum} \in \mathbb{N}_0$ with $c_{sum} = \sum c_i$

$$F_2 = (u_1, \dots, u_m, u_{sum})$$

- $u_i \in \{0, 1\}$ with $u_i = 1$ indicating phase i is used after a change, else $u_i = 0$
- $u_{sum} \in \mathbb{N}_0$ with $u_{sum} = \sum u_i$

$$F_3 = (prj, lang, age_{abs}, age_{rel}, fail, ID)$$

- $proj$ the project's GitHub name (Organisation@Repository)
- $lang$ the project's main programming language
- $age_{abs} \in \mathbb{N}_0$ the time in seconds from start of CI-usage to this configuration changing commit
- $age_{rel} \in [0, 1]$ the relative time from start of CI-usage to this configuration changing commit w.r.t. total CI-usage period.
- $fail \in \{0, 1, 2\}$ the result of the next CI build after the configuration change is either a positive (0), negative (1) or not retrievable (2)
- $ID \in \mathbb{N}$ the number of the commit in the chronological order of the configuration changing commits of this project

Preprocessing extracts which phases are changed and which phases are used after the build change together with the commit's absolute and relative point of occurrence in the projects CI-usage period, the project's name, main programming language and the build result of the next CI build after the change for every configuration change.

The feature vectors are grouped to store data instances in the form (F_1, F_2, F_3) with F_3 holding the features that identify each data instance uniquely by the project's name and the commit number, or point of time respectively. Some values in F_3 are stored multiple times across instances of the same project.

Indices 1 to n refer to the phases in table 4.2 (page 26). The *FORMATTING* phase is used as a default phase for phase changes, but as phase it is never explicitly used, therefore it is not considered a feature in phase usage. So for phase usage $m = n - 1$ features are stored instead of n .

The build results are simplified to a Boolean scale, with a positive ($success \Rightarrow 0$) and a negative ($failure, errored, aborted \Rightarrow 1$) measure. For the rest of this thesis a negative build result is referred to as a failed build.

Project-Level

On project-level the different feature vectors hold the following features with the according domains:

$$F_4 = (c_{1,abs}, \dots, c_{n,abs}, c_{sum,abs}, c_{1,rel}, \dots, c_{n,rel})$$

- $c_{i,abs} \in \mathbb{N}_0$ the total amount of changes to phase i in this project
- $c_{sum,abs} \in \mathbb{N}_0$ the total amount of changes as $c_{sum,abs} = \sum c_{i,abs}$
- $c_{i,rel} \in [0, 1]$ the relative amount of changes to phase i w.r.t. total amount of changes

$$F_5 = (u_{1,abs}, \dots, u_{m,abs}, u_{1,rel}, \dots, u_{m,rel}, u_{1,bin}, \dots, u_{m,bin}, u_{sum,bin})$$

- $u_{i,abs} \in \mathbb{N}_0$ the number of times that phase i is used after a change
- $u_{i,rel} \in [0, 1]$ the relative usage to phase i w.r.t. total usage of all phases
- $u_{i,bin} \in \{0, 1\}$ with $u_{i,bin} = 1$ indicating phase i is at least used once in this project, else $u_{i,bin} = 0$
- $u_{sum,bin} \in \mathbb{N}_0$ the amount of distinct phases used as $u_{sum,bin} = \sum u_{i,bin}$

$$F_6 = (proj, lang, age, ci_usage, den, maxID)$$

- $proj$ the project's GitHub name (Organisation@Repository)
- $lang$ the project's main programming language
- $age \in \mathbb{N}_0$ the project's age in seconds
- $ci_usage \in \mathbb{N}_0$ the project's total CI-usage period in seconds
- $den \in [0, 1]$ the phase change density
- $maxID \in \mathbb{N}$ the number of configuration changing commits in this project

The feature vectors are grouped to store data instances in the form (F_4, F_5, F_6) with F_6 holding the project name feature which identifies each instance uniquely, due to GitHub's naming scheme.

These data instances hold the total number of changes to phases in a project. The sum of these values results in the total changes to any phase in a project ($c_{sum,abs}$). Also the relative share of changes to one phase, with respect to the total amount of changes is stored. The relative values may allow for easier comparison of projects.

For the usage of phases similar features are stored, with an additional feature expressing the binary usage of a phase, i.e., if a phase is used at least once in a project.

F_6 contains similar features as F_3 , but again for project-level. The age of the whole project and the time since CI was introduced into the project are stored here. The $maxID$ feature holds the total amount of configuration changes. Additionally there is the feature of phase change density

$$den = \frac{maxID}{ci_usage} * 3600$$

expressing the frequency at which a phase is changed. As CI configurations naturally have a low change frequency, the density measure is normalized to express the average number of phase changes per hour, rather than per second.

CI-functionality Adoption Measure

CI-functionalities are defined by groups of phases and express the same value as stages in a CI pipeline (cf. section 2.1.3 (page 6) / section 4.6 (page 25)). An approach to measure the adoption of such functionalities is given shortly. First some extra computation is needed, which is explained.

Obviously, to measure consistent phase usage the discrete usage representation alone, as it is given in binary form by the preprocessed data set, is not sufficient. To measure consistent usage I define the point of *phase introduction* over a continuous usage representation. I.e., the point in time from which on a phase is used consistently, whereby consistent usage is defined as a tolerance towards constant usage. To compute a measure for phase introduction from this data, firstly the discrete usage representation is transformed into a continuous usage representation. This is done for a project's set of configuration changing commits $(1, \dots, t, \dots, n)$, a phase i and the corresponding usage features $(U_{(i,1)}, \dots, U_{(i,t)}, \dots, U_{(i,n)}) \in \{0, 1\}^n$ by applying a function f_i to each commit t that computes:

$$f_i(t) = \frac{|\{U_{(i,k)} | t \leq k \leq n \wedge U_{(i,k)} = 1\}|}{|\{U_{(i,t)}, \dots, U_{(i,n)}\}|} \in [0, 1]$$

Thus a function value $f_i(t) = p$ expresses the relative future use of phase i , i.e., that after $(p \cdot 100)\%$ of the following configuration changes $(t + 1, \dots, n)$ the phase i is used. In this way the discrete phase usage representation is transformed to a continuous usage representation (cf. figure 6.1 for an example).

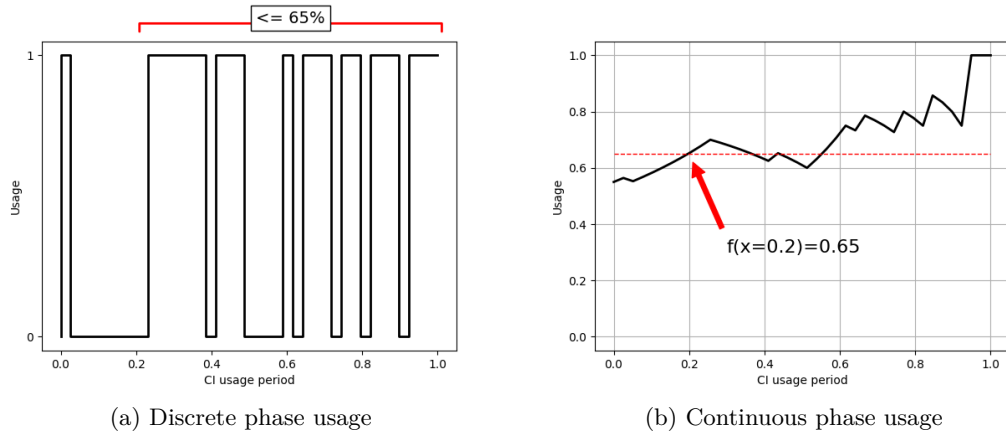


Figure 6.1.: Different phase usage representations.

When using a continuous usage representation, phase introduction can be defined as the first point in time t^* at which a certain threshold $f(t^*) = T \in [0, 1]$ is reached. A CI-functionality is then adopted, regarding a threshold T , if for one of the functionality's phases its continuous usage function f reaches the threshold T after some commit t : $f(t) \geq T$. The commit t for which $f(t) \geq T$ holds for the first time is then considered the point of adoption of the CI-functionality.

File Format

For mining I use the Weka tool [FHW16], a widely used open source software for data mining applications. Although not all questions can be answered by data mining algorithms, the recorded data instances are stored in Weka's *ARFF* file format. The *ARFF* file format is not restricted to exclusive use by the Weka tool. It is a variation of a common family of file formats, which store data instances in lines, separating values by commas (or a self-defined separator) and data instances by line breaks. Thus this structured format can be easily parsed by other tools or applications for visualization or similar.

6.2. Challenges

During preprocessing the following challenges are faced.

- (1) In the temporal delta between the extraction of the configuration files by myself (12.09.2017) and the point of time that the data set from TravisTorrent [Traa] was recorded (20.12.2016), a small amount of projects was deleted. Also for some projects there is no recorded data in the TravisTorrent set. All projects for which the data set in TravisTorrent is either empty or the repository is deleted on GitHub are removed from the preprocessing scope.
- (2) In some cases a configuration change does not alter the build process. This is the case if there is no semantic change to the file, e.g., comments added. The other case occurs if a syntactically invalid *.travis.yml* is pushed into the repository. Due to the invalidity of such a configuration file it cannot be automatically parsed by Travis CI leading to a direct termination of the build process. This leaves an uncertainty in the change history, as the real intention of the developer is not recordable. The best solution to this problem is to skip an invalid file in the pairwise comparison of CI configurations (cf. figure 6.2).

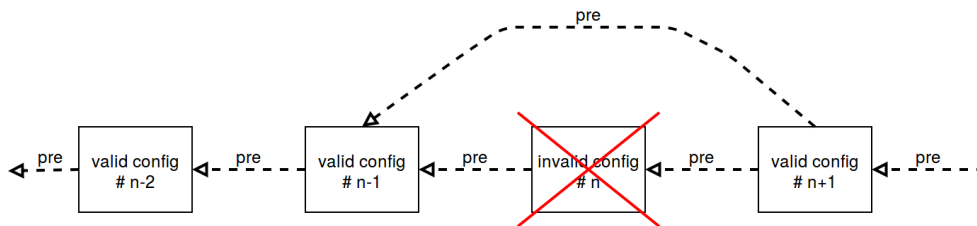


Figure 6.2.: Handling invalid *YAML* files in the change history.

For pairwise comparison two valid files are needed. Therefore in case a direct predecessor is an invalid configuration, the current file version is compared to the its first valid predecessor. If there is no valid predecessor for a configuration file, then it is treated as if it was the first version for in a project's change history.

Obviously as the file is invalid, the change is not directly extractable. Assuming the unrecordable change is the inclusion of a new key to the configuration. The developer

could then possibly fix his mistake in which case the new valid configuration reveals the intended change. This would be a best-case scenario. Alternatively the developer might deleted this key again, returning the configuration to its older state. In this case the change is not recorded by preprocessing, which is the worst-case. This uncertainty increases through higher amount of successively pushes of invalid files.

The following scheme is used to represent the three basic types of changes via the F_1 vector (cf. figure 6.3):

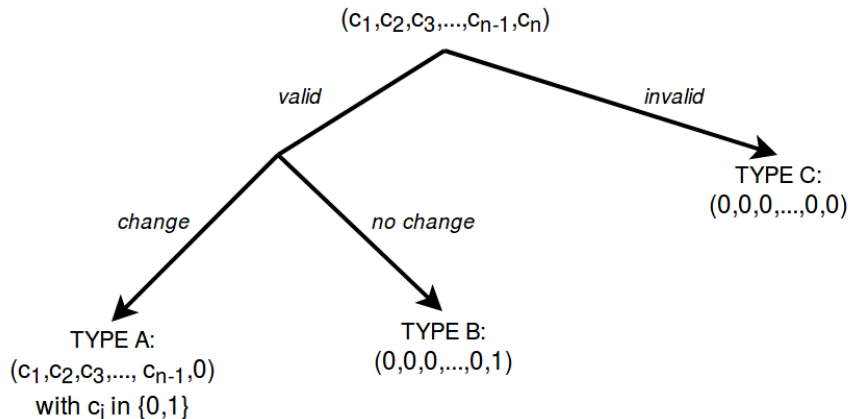


Figure 6.3.: The three basic types of changes.

For this scheme it is assumed that the previous configuration file is always a valid file. This requirement always holds due to the strategy presented in figure 6.2.

Type A represent a standard change: If the configuration file is valid and at least one phase is changed, then the related phase change variables equal 1 where the phases are changed with respect to the next valid predecessor. The variable for the *FORMATTING* phase equals 0.

A *Type B* change occurs if the file is valid but there is no semantic change. This change has no impact on the build process, therefore only the *FORMATTING* variable equals 1 and all others equal 0. Examples are changes of comments in the configuration or usage of user-defined keys, which are not recognized by Travis CI.

Type C occurs if the current file is invalid, then all variables in F_1 equal 0. The usage features however are inherited from the preceding configuration change, as there is no phase change recorded.

(3) In Git it is possible to push multiple commits at once. If multiple commits are pushed at once, Travis CI only issues a build on the latest commit. If a build changing commit is transferred in a multiple commit push, then there might be no build result link-able to a configuration change in the two different data sets (TravisTorrent and Git History) via a Git commit id. To counteract this the chronologically next build result is used.

6.3. First Observations

Preprocessing resulted in a structured data set consisting of 962 GitHub projects and 26708 configuration changing commits in total. The total number of phase changes is 49189. The average project alters its configuration about 27 times. Table 6.1a shows the language distribution: there are four main programming languages used by projects in this data set.

Language	# projects
Python	259
Ruby	245
Java	241
Go	201
Other	16
Σ	962

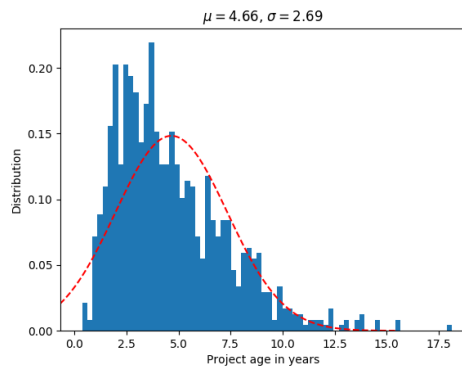
(a) Distribution of main programming language in the preprocessed data.

Group	# projects
Years [0 – 1)	68
Years [1 – 2)	203
Years [2 – 3)	256
Years [3 – 4)	197
Years [4 – 5)	153
Years [5 – 6]	85
Σ	962

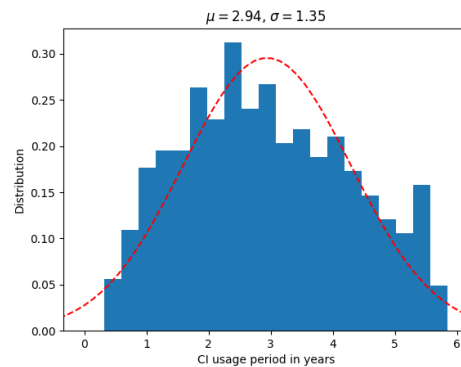
(b) Amount of projects in groups of distinct CI-usage periods.

Table 6.1.: Project distribution w.r.t main programming language and CI-usage period.

The distribution of project age and CI-usage period in years are visualized in figure 6.4a and figure 6.4b. Both distributions approximate a normal distribution (red curve) with values for μ and σ given in the figures. The average project is 4.6 years old and used Travis CI for approximately three years. For groups of distinct CI-usage period the above membership count is observed (cf. table 6.1b).



(a) Distribution of project age.



(b) Distribution of CI-usage period.

Figure 6.4.: Distribution of project age and CI-usage period in the preprocessed data.

7. Results and Discussion

Contents

7.1. Goal 1: Acceptance of the Travis CI Model	43
7.1.1. Discussion	44
7.2. Goal 2: Robustness of the Travis CI Model	48
7.2.1. Q3: Do phases cause build failures together?	50
7.2.2. Discussion	51
7.3. Goal 3: Build Process Evolution	52
7.3.1. Discussion	54
7.4. Goal 4: Build Process Structure Build-Up	57
7.4.1. Discussion	63
7.5. Goal 5: Equivalent Usage of the Travis CI Model	65
7.5.1. Discussion	67
7.6. Final Discussion	68

This chapter presents the main results of this thesis. For each question the results are shown along with some words on the execution of the mining, visualization, metric computation or the like. For each goal there is a small discussion and in the end there is final discussion about all findings.

7.1. Goal 1: Acceptance of the Travis CI Model

To gain basic knowledge on how the Travis CI model is used, I look at how frequently the different components, the phases, are used and maintained by the projects.

Q1: Which phases of the model are most frequently used?

Counting for each phase by how many projects in the data set it is used at least once results in the ranking given in table 7.1a. This is accomplished by using the binary-usage feature defined in feature vector F_5 (cf. section 6.1 (page 35)).

First of all, it is observable that there is no phase, that is used by 100% of the projects. Except for the *BUILD_STAGES* and *FORMATTING* phases there is also no phase which is not used by all project. The *BUILD_STAGES* phase relates to a functionality of the Travis CI build process, which at the point of time this data was extracted was currently introduced in a beta phase. As already stated in section 6.1 (page 35), *FORMATTING* is not a phase related to the Travis CI model. It is used as a default for non-semantic changes, therefore its usage has no meaning and is not regarded here.

The top three phases are used by a decent amount of all projects (above 75%). The

top two phases (*LANGUAGE_ENV* and *SCRIPT*) are exactly those phases which are identified as the core phases in a Travis CI build, regarding the minimum CI configuration (section 4.3.3 (page 23)). All other phases, i.e., besides the top three, are only used by circa 50% of the projects or less. These phases are also those, which are used to configure more individual and complex builds. Therefore this ranking seems quite reasonable.

Q2: Which phases of the model are most frequently changed?

Counting on commit-level how often a certain phase is changed results in the ranking given in table 7.1b. This is accomplished by using the features of F_1 (cf. section 6.1 (page 35)). There are no changes to *BUILD_STAGES*, because it is not used in any project. The most changed phase is the *LANGUAGE_ENV* phase with 8616 changes, making up circa 18% of all changes.

The majority of changes (> 64%) are made to the top five phases in this ranking, each having an individual share of 10% or more. The top five include the basic *LANGUAGE_ENV* and *SCRIPT* phases and the phases that are changed less are more likely used for advanced configuration.

Q3: How many phases are used per project?

The feature $u_{sum,bin}$ of feature vector F_5 (cf. section 6.1 (page 35)) expresses the maximum amount of distinct phases used in a project, i.e., the sum of individual phases that are included in the project's build process at least once at some point. Figure 7.1 shows the distribution of this feature among all projects.

On the one hand there exist projects that only use one phase; on the other hand the maximum lies at 16 phases. The distribution is approximated by a normal distribution (red curve) with $\mu = 6.17$ and $\sigma = 2.68$. Thus the majority of projects ($\sim 70\%$) used approximately four to nine phases in their CI-usage period.

Q4: What is the volatility of phases?

The volatility of phases is measured by the average amount of changes to a phase per project. This measure is computed by dividing the values of both rankings in table 7.1. Table 7.2 shows a ranking that deviates from the other two as projects have a clear focus on changing *BUILD_MATRIX* and *INSTALL* more often.

7.1.1. Discussion

It is observed that the more basic phases are the ones that have a higher priority in both the phase-usage and phase-change ranking, thus are highly accepted. In general the two rankings are similar in the order of the phases. Both share the top three phases *LANGUAGE_ENV*, *SCRIPT* and *BUILD_ENV* in exactly this order, which are more or less the basic parts of a Travis CI build process. Although not in the same order, the top ten phases of both ranking are also the same, indicating that there is a separation and all phases below rank 10 are less accepted. For the volatility of phases, it can be

Phase	# of projects using phase	% of projects using phase	Phase	# of changes (abs. measure)	% of changes (rel. measure)
LANGUAGE_ENV	957	99.5	LANGUAGE_ENV	8616	17.5
SCRIPT	798	83.0	SCRIPT	6845	13.9
BUILD_ENV	751	78.1	BUILD_ENV	6455	13.1
INSTALL	533	55.4	BUILD_MATRIX	5473	11.1
BEFORE_INSTALL	523	54.4	INSTALL	5364	10.9
NOTIFICATIONS	378	39.3	BEFORE_INSTALL	4508	9.2
BUILD_MATRIX	345	35.9	BEFORE_SCRIPT	2059	4.2
CACHE_ENV	343	35.7	AFTER_SUCCESS	1756	3.6
BEFORE_SCRIPT	294	30.6	CACHE_ENV	1548	3.1
AFTER_SUCCESS	277	28.8	NOTIFICATIONS	1448	2.9
ADDONS	219	22.8	ADDONS	1319	2.7
PLATFORM_ENV	156	16.2	GIT	887	1.8
GIT	156	16.2	FORMATTING	788	1.6
DEPLOY	68	7.1	PLATFORM_ENV	658	1.3
AFTER_SCRIPT	55	5.7	DEPLOY	563	1.1
BEFORE_CACHE	35	3.6	AFTER_SCRIPT	288	0.6
AFTER_FAILURE	30	3.1	AFTER_FAILURE	214	0.4
BEFORE_DEPLOY	19	2.0	BEFORE_DEPLOY	173	0.4
AFTER_DEPLOY	8	0.8	BEFORE_CACHE	169	0.3
BUILD_STAGES	0	0.0	AFTER_DEPLOY	58	0.1
FORMATTING	0	0.0	BUILD_STAGES	0	0.0
Σ			Σ	49189	100.0

(a) Ranking of phase usage.

(b) Ranking of phase changes.

Table 7.1.: Rankings of phase changes and usage in absolute and relative (rounded) measures.

Phase	# of changes (abs. measure)	#projects using using phase	Volatility
BUILD_MATRIX	5473	345	15.86
INSTALL	5364	533	10.06
BEFORE_DEPLOY	173	19	9.11
LANGUAGE	8616	957	9.00
BEFORE_INSTALL	4508	523	8.62
BUILD_ENV	6455	751	8.60
SCRIPT	6845	798	8.58
DEPLOY	563	68	8.28
AFTER_DEPLOY	58	8	7.25
AFTER_FAILURE	214	30	7.13
BEFORE_SCRIPT	2059	294	7.00
AFTER_SUCCESS	1756	277	6.34
ADDONS	1319	219	6.02
GIT	887	156	5.69
AFTER_SCRIPT	288	55	5.24
BEFORE_CACHE	169	35	4.83
CACHE_ENV	1548	343	4.51
PLATFORM_ENV	658	156	4.22
NOTIFICATIONS	1448	378	3.83
FORMATTING	788	0	-
BUILD_STAGES	0	0	-

Table 7.2.: Volatility of phases.

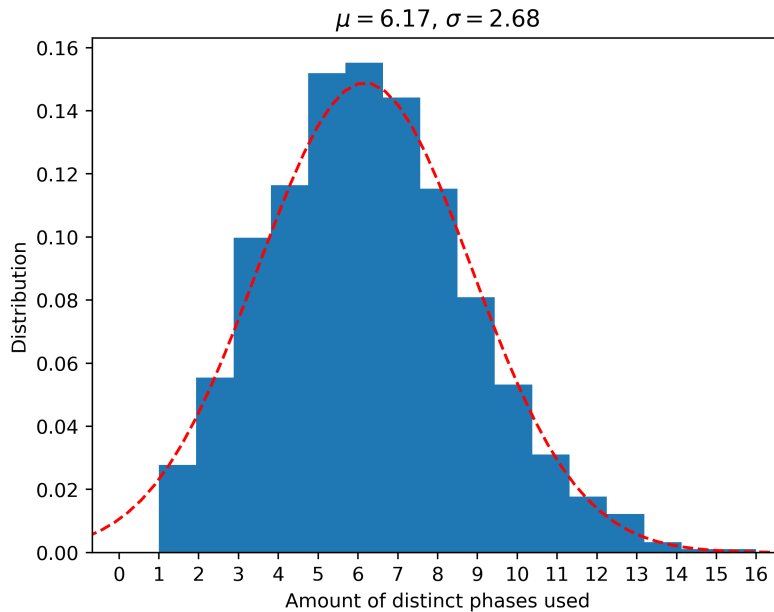


Figure 7.1.: Amount of distinct phases used per project.

seen that *LANGUAGE_ENV*, *SCRIPT* and *BUILD_ENV* are not among the top three anymore.

Phase usage implies only one phase change, namely the inclusion of the phase, thus the phase-usage ranking should not predict the ranking of phase-changes. It is a very interesting observation that nevertheless the two rankings are very similar. From this the hypothesis can be formulated that phases, which are more used are also more likely to change more frequently. This is also in contrast to the belief that basic parts of the build process should be stable, but these rankings show that this is not the case. Later when incorporating the time dimension, it is possible to confirm, if parts of the process really stabilize or not.

Next to the usage of default conventions (cf. section 4.3.3 (page 23)), the following anti-pattern of explicit denial is observed. The explicit denial of a default, e.g., by turning off all notifications (cf. listing 7.1), is recorded as usage of this phase, although the phase is not used semantically.

```
1 | notifications:
2 |   email: false
```

Source Code 7.1: Default denied.

For the acceptance of the model three conclusions can be drawn:

- (1) There is a general acceptance of the model. There is no phase which is never used, but also there is no phase, which is used by every project. As it is already known that either the *LANGUAGE_ENV* or *SCRIPT* phase is mandatory, this is legitimate. About one quarter of the phases are used by 50% of the projects or more. Usage under 50% is considered under-utilization of the phase.
- (2) There is a high acceptance of parts of the model. The top three phases are used by more than 75% of the projects and have the highest change count. Especially the *SCRIPT* phase is used by 83%, implying that at most 17% of the projects rely on the standard build execution that Travis CI offers. However, it is observed that phases, which are used by more projects, are also changed more often. The reason for this (e.g., instability of a phase) and the impact this could possibly have on the acceptance measure is unknown at this point. The robustness of the model is investigated in the next section.
- (3) There is a low acceptance of the whole model. No project uses all the offered functionality. In fact the trend lies at using four to nine phases; the mean is six. This means that on average each project uses only $\frac{6}{20} = 30\%$ of all phases, i.e., 70% of all phases are not used. If there is a reason to why only a small subset of all phases is used is investigated in cluster analysis (section 7.5). In general, all deploy related functionality is rarely accepted.

7.2. Goal 2: Robustness of the Travis CI Model

Q1: Which phases are critical, concerning the build results of the corresponding builds after a phase change?

For a build result one of the three values $\{0, 1, 2\}$ is stored in the corresponding feature in F_3 (cf. section 6.1 (page 35)). The amount of unretrievable build results (2) in this data set is rather low with only 2143 affected builds (8.02%), the amount of build failures (1) lies at 10561 (39.54%) and the amount of build success (0) at 14004 (52.43%). Besides counting the change frequency, also the build failures can be summarized per phase on commit-level. The results are listed in table 7.3.

On first sight, the amount of build failures grows linearly with the ranking of phase changes, and *BUILD_STAGES* again has value zero as it is never used. Thus the ranking is very similar to the phase-change ranking (cf. table 7.1b).

Q2: Does a correlation between change frequency and amount of negative build results exist?

In fact Weka's linear regression classifier approximates the quantitative relationship between phase changes and build failures effectively. Figure 7.2 visualizes this relation: A point represents for each phase its change frequency and amount of build failures as given in the rows of table 7.3. The line represents the linear regression function with which these points are approximated by Weka and expresses the correlation of both the change frequency and the amount of build failures. The regression's correlation coefficient is 0.99871. The gradient of the regression line implies that in circa 40% of the times a

Phase	# of changes (abs. measure)	# of build failures
LANGUAGE_ENV	8616	3438
SCRIPT	6845	2762
BUILD_ENV	6455	2633
INSTALL	5364	2231
BUILD_MATRIX	5473	2191
BEFORE_INSTALL	4508	1990
BEFORE_SCRIPT	2059	1002
CACHE_ENV	1548	633
AFTER_SUCCESS	1756	611
NOTIFICATIONS	1448	592
ADDONS	1319	556
GIT	887	380
FORMATTING	788	315
PLATFORM_ENV	658	284
DEPLOY	563	191
AFTER_FAILURE	214	95
AFTER_SCRIPT	288	85
BEFORE_DEPLOY	173	71
BEFORE_CACHE	169	67
AFTER_DEPLOY	58	20
BUILD_STAGES	0	0
Σ	<i>49189</i>	<i>20147</i>

Table 7.3.: Build failures per phase.

phase is changed the next build will fail.

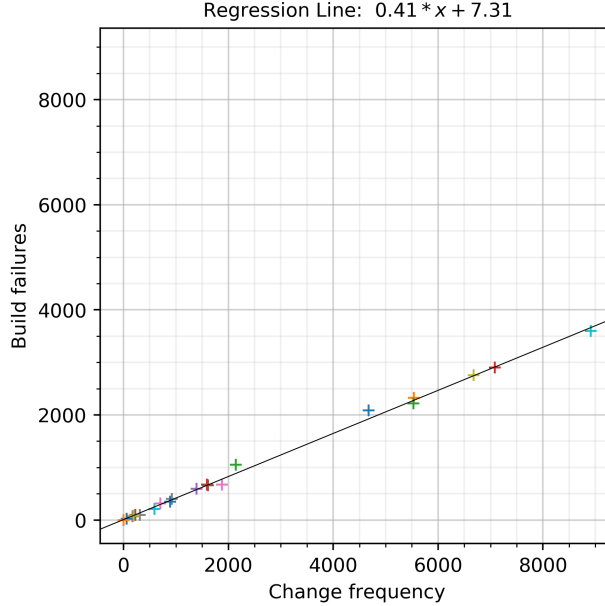


Figure 7.2.: Plot of change frequency versus build failures of all phases.

7.2.1. Q3: Do phases cause build failures together?

Previously, each phase is analyzed separately, leading to an accumulated amount of 20147 build failures (cf. last row in table 7.3), which is much higher than the real amount of build failures (10561). I.e., it is not regarded to which extend multiple phase changes in one commit cause a build failure together. From those values we see that on average two phases are responsible for a build failure:

$$\frac{\text{\#phase changes causing build failures}}{\text{\#failed builds}} = \frac{20147}{10561} \approx 2$$

To see which combination of phase changes lead to build failures, the commit-level instances are clustered upon the phase change features of F_1 (except for the *FORMATTING* phase) for instances that have a failed build result (cf. table A.1 (page 80) / table 7.4). The number of final clusters is 11, thus with $11 \leq 20$, the maximum number of phases except *FORMATTING*, one knows that there exists at least some clusters including two or phases. In fact three main groups are identified: The majority of build changes (9298 (92%)), on which the next build fails, are unique changes, i.e., only one phase is changed. Commits that induce multiple phase changes are less common (cf. table 7.4). So, a large amount of phases are changed together, but not so often as that this would create a major distortion in the observation depicted in figure 7.2. However, the phases that are

Category	Count	Phases
multiple	445	{ <i>BEFORE_INSTALL</i> , <i>INSTALL</i> , <i>SCRIPT</i> , <i>BUILD_MATRIX</i> , <i>BUILD_ENV</i> , <i>LANGUAGE_ENV</i> }
multiple	359	{ <i>BEFORE_INSTALL</i> , <i>INSTALL</i> , <i>SCRIPT</i> , <i>BUILD_ENV</i> , <i>LANGUAGE_ENV</i> }
unique	9298	{ <i>BEFORE_INSTALL</i> }, { <i>INSTALL</i> }, { <i>SCRIPT</i> }, { <i>AFTER_SUCCESS</i> }, { <i>BUILD_MATRIX</i> }, { <i>ADDONS</i> }, { <i>PLATFORM_ENV</i> }, { <i>BUILD_ENV</i> }, { <i>LANGUAGE_ENV</i> }

Table 7.4.: Groups of phases that cause build failures together.

mostly changed together before a build fails are also the ones, that are higher valued in the phase-change and phase-usage ranking (cf. table 7.1).

7.2.2. Discussion

By viewing for each phase separately its values for change frequency and build failures, it is learned that there are no phases solely responsible for an exceptional amount of build failures. In terms of robustness of the model, the observation, that, phase-independently, in 40% of the cases the build fails after a build change, implies that there is sort of constant instability in the build process, as the correlation coefficient (0.99871) from linear regression is exceptionally high.

The final clusters from clustering upon the change features of instances with failed builds strengthens the original observation of constant instability. For the cause of this phenomenon four hypotheses from contrasting perspectives are presented:

- (1) The cause lies exclusively in the Travis CI model, i.e., faults in the internal processing of the configuration and faulty build executions or grave issues in the understandability on how to use Travis CI and its configuration properly. I find this hypothesis is possible, but unlikely.
- (2) The robustness cannot be measured clearly due to the software. Builds fail due to normal causes, e.g., failing unit tests or syntax errors, as CI configuration changes often go along with big software changes. For a clearer view on the robustness the impact the software itself has on the build result must be regarded. Explicitly excluding this impact would reveal the pure robustness of a phase in the model. Obtaining this insight is not possible with this data set.
- (3) The robustness cannot be measured clearly due to the unrecorded intention of the developers. In this research the classification of the quality of build results is conducted

upon intuition (Reminder: *success* \Rightarrow 0, *failure*, *errored*, *aborted* \Rightarrow 1; cf. section 6.1 (page 35)). The real intention of a build change however is unclear. Thus a build failure might be the desired outcome of the build change, e.g., if new quality gates are introduced into the CI build. In this case the build failure has a high quality towards the user. It might be possible to extract intent from commit messages. However, obtaining such insight from this data set is also not possible.

(4) All developers build pipelines in a careless trial-and-error fashion. This is also unlikely.

7.3. Goal 3: Build Process Evolution

To gain knowledge on how Travis CI build processes evolve, the time dimension is incorporated to plot the usage and change frequency over the CI-usage period of all projects. For a clearer view, projects are grouped by their CI-usage period into the six groups of $[0 - 1)$, $[1 - 2)$, \dots $[5 - 6]$ years (cf. section 6.3 (page 41)). A graph is created for each pair of phase and group, and once for all phases per group. This resulted in 132 graphs for each the phase change frequency over time and phase usage frequency over time. In the following the most significant insights from these graphs are presented.

By normalizing the CI-usage period (i.e., plotting over $[0, 1]$) the projects become comparable, as then the relative point of time of a change or usage is known. Grouping projects by their real CI-usage period, minimizes the distortion, that is caused by the real temporal delta between any two commits of different projects in each group. Furthermore, distinct characteristics for the different groups of CI-usage periods can be evaluated.

Q1: When are phase changes made in a project's CI-usage period?

The phase change frequency graphs are plotted with the normalized time of each commit and the amount of phases that are changed by the corresponding build change (cf. section 6.1 (page 35)).

As listing all 132 graphs is not reasonable, figure 7.3a - figure 7.3f show the summarized change frequency for all phases in the six different groups of CI-usage periods. These summaries each represent the behavior of change frequency in the graphs for single phases in their CI-usage period group sufficiently. Note that the curves are plotted on a logarithmic scale for better comparison. The equidistant points on the curve each approximate the change frequency of a seven-day week. All 132 change frequency graphs show that the change frequency curves peak at the beginning of the CI-usage period and then vastly descend to oscillate at a lower value (cf. figure 7.3). No curve converges to a constant value or zero in any way.

Q2: When are phases used in a project's CI-usage period?

The phase usage frequency graphs are plotted with the normalized time of each commit and the amount of projects that are used after the corresponding build change (cf. section 6.1 (page 35)). Again from the 132 graphs only the most significant are presented here.

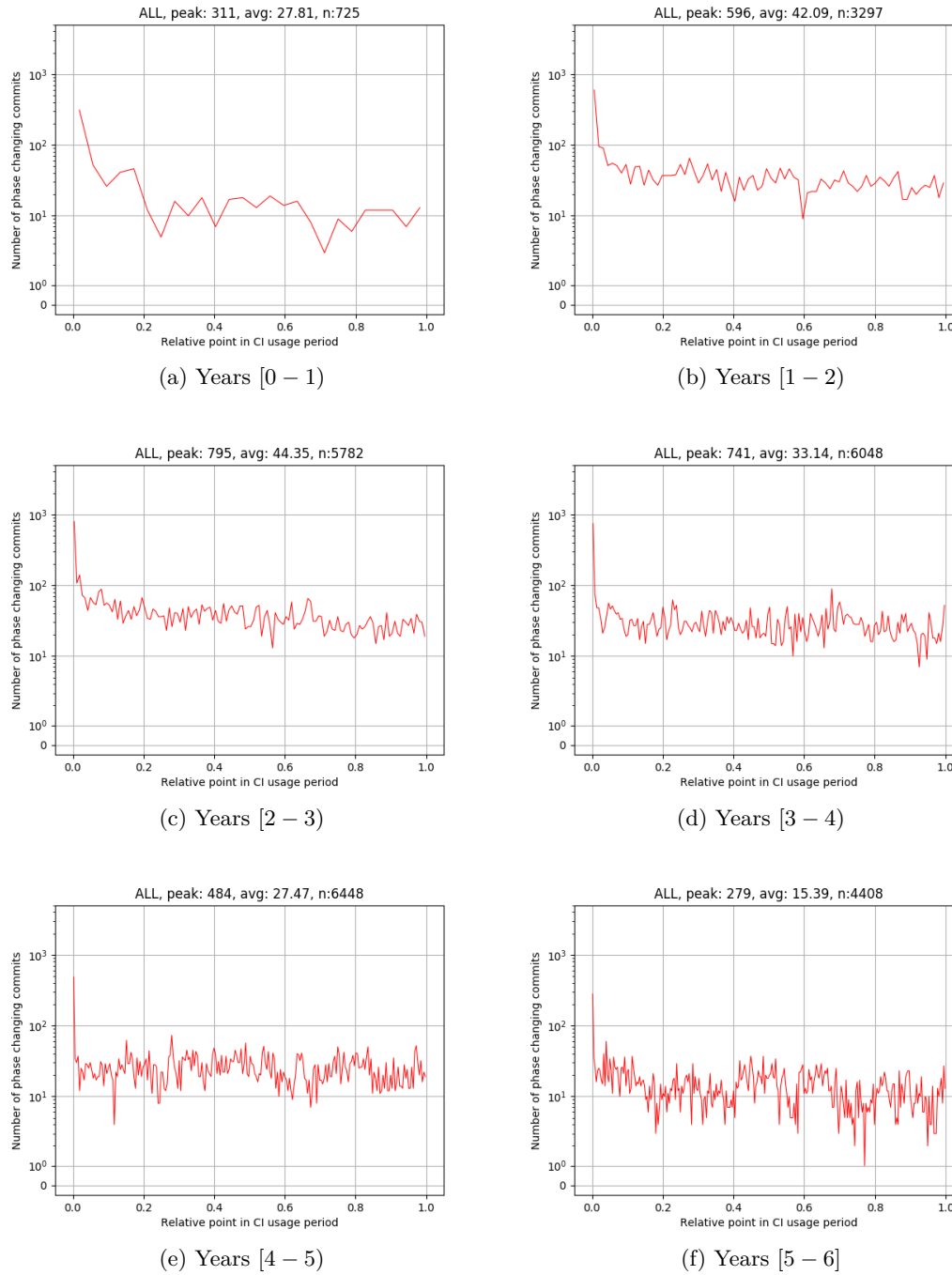


Figure 7.3.: Phase change frequency for all phases in groups of CI-usage periods.
peak: the maximum phase change frequency
avg: the average phase change frequency
n: the total amount of configuration changes to this phase in this group

On average, phase usage stays at a constant level (cf. figure 7.4). With the small exception of phases *BUILD_ENV* (cf. figure 7.4f) and *CACHE_ENV* (cf. figure 7.5c) this holds true for every phase. It is also observed that for a subset of phases the usage frequency slightly peaks at the beginning of CI-usage, i.e., in the interval $[0, 0.1]$ (cf. figure 7.5).

Because phase usage is mostly steady throughout the whole CI-usage period, the inclusion and exclusion behavior of phases at any fixed point in time is investigated. Inclusion and exclusion are shown by the green and red curves in figure 7.4 and figure 7.5. For any point in time $t \in [0, 1]$ the green curve shows the amount of projects which added the phase to their configuration, respectively the red curve shows the amount of projects that deleted the phase. Thus the delta of the total usage frequency (black curve) is expressed by the subtraction of the exclusions from the inclusions. The inclusion/exclusion rates are always very low ($< 2\%$ of total projects; cf. figure 7.4, figure 7.5) for the interval $(0, 1]$.

Q3: Can foci on significant phases be found in these graphs?

The simple answer is no. There are no significant maxima in the frequency graphs to be found, which would indicate that a distinct period of time is generally dedicated to the usage or maintenance of one or more phases. The change frequencies show that maintenance oscillates. Usage frequency is mostly constant, resulting in an absence of peaks.

7.3.1. Discussion

The fact that no phase exhibits a stabilizing behavior, indicates that the build processes are (1) constantly developed incrementally or (2) require constant maintenance. However, this also indicates that build processes themselves do not seem to stabilize over time, as they are always exposed to a lot of changes (e.g., 10-100 phase changes per week for ~ 200 projects). The current states of the build processes in this data set are apparently not the final states as desired by the developers. Does this imply that build processes never stabilize or just do so in the future, i.e., outside of the observed period of six years? Furthermore, the question remains why maintenance is always at a constant high level.

The knowledge that inclusions/exclusion rates stay very low, approves the observation that phase usage stays at a constant level and in general the following rule holds:

If a phase is not used at the beginning of CI-usage, then with high probability it will be never used in this build process and vice versa.

This is a very coarse rule, as the usage frequency in these graphs is accumulated for all 962 projects. Yet in all other frequency graphs the results are very similar, except for a few insignificant outliers. This may also weakly imply that there is hardly any incremental adoption of phases in the evolution of build processes. However, the observed

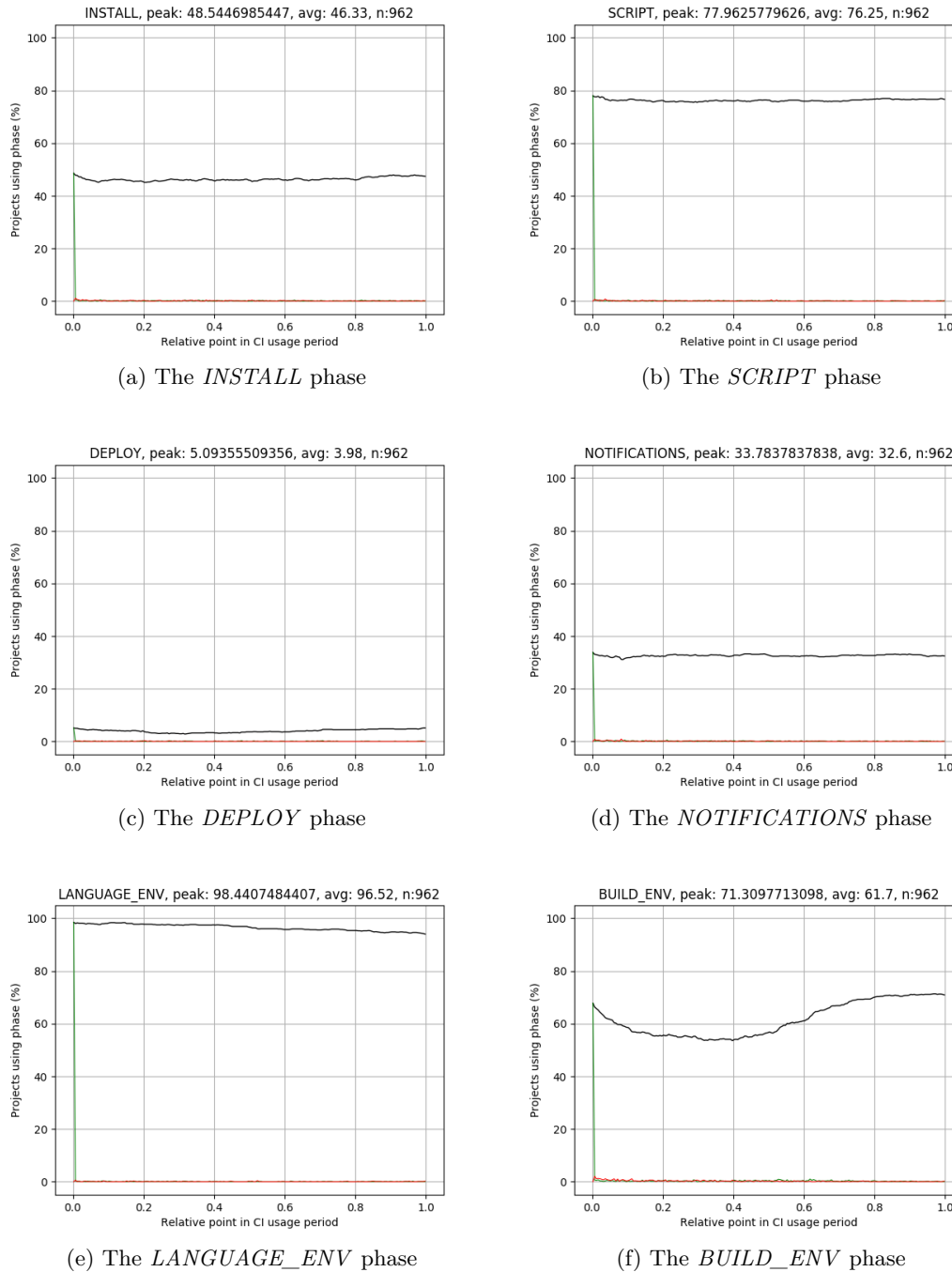


Figure 7.4.: Phase usage frequency for selected phases for all projects.

- peak*: the maximum phase usage (in %)
- avg*: the average phase usage (in %)
- n*: the total amount of projects in this group (= 100%)
- black curve*: shows the usage frequency
- green curve*: shows the current amount of inclusions
- red curve*: shows the current amount of exclusions

7. Results and Discussion

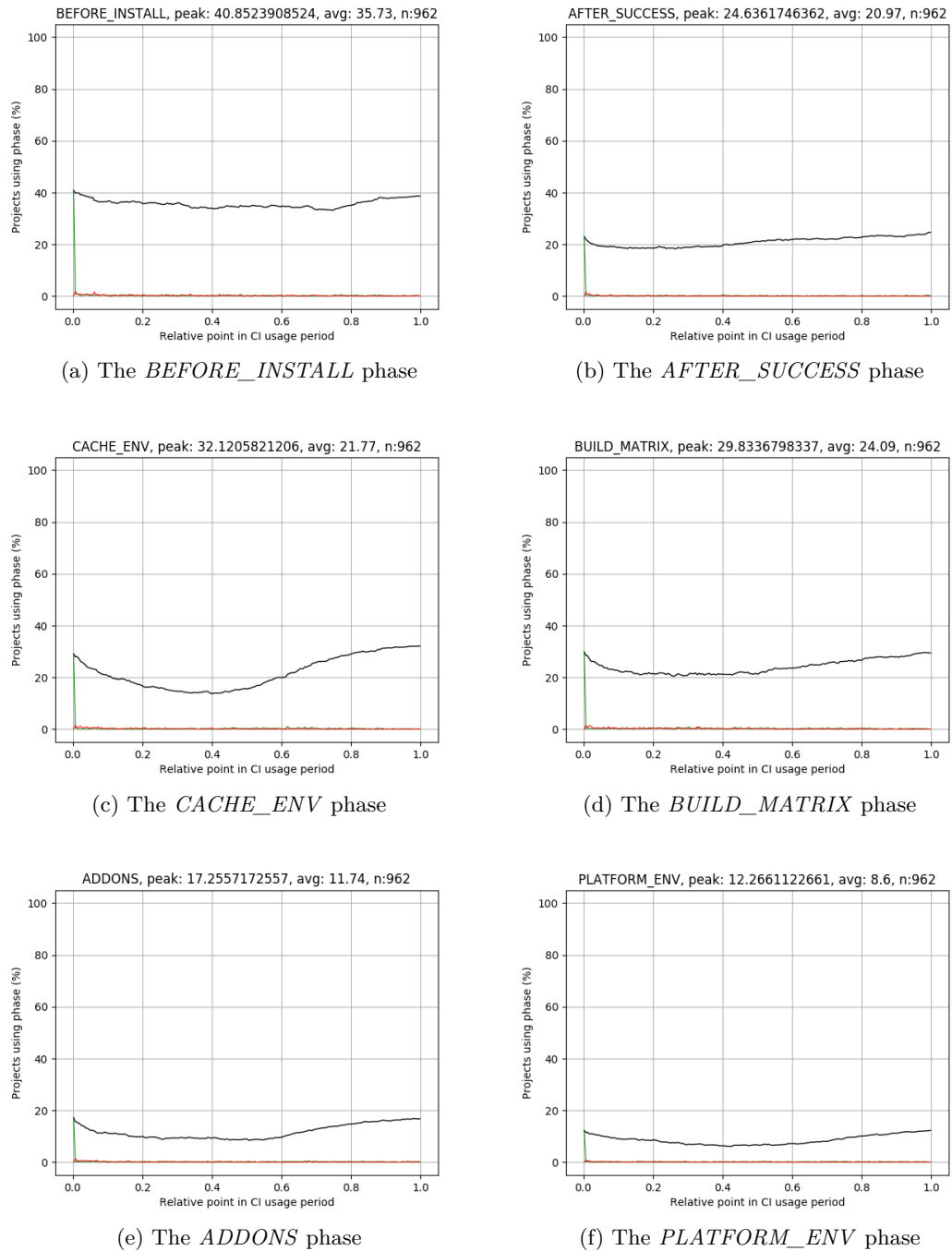


Figure 7.5.: Examples for experimental periods in phase usage frequency graphs.

peak: the maximum phase usage (in %)

avg: the average phase usage (in %)

n: the total amount of projects in this group (= 100%)

black curve: shows the usage frequency

green curve: shows the current amount of inclusions

red curve: shows the current amount of exclusions

peaks at the beginning of CI-usage (cf. figure 7.5) indicate that projects go through an experimental period for a subset of phases.

Also it could not be observed that there exist time periods in which the focus lies on maintaining special phases. The expectations were to find that for certain intervals in the CI-usage period foci on special phases exist and that at some point the build processes would stabilize. Changes are always regarded good, as the process should evolve, but the frequency was not expected to remain on such a high level. Furthermore it was expected to identify stabilizing characteristics for projects with a larger CI-usage period. None of the above expectations could be confirmed by the data.

7.4. Goal 4: Build Process Structure Build-Up

Instead of observing the build process evolution by analyzing the change and usage frequencies of phases, here the adoption of CI-functionalities is observed to investigate incremental adoption from a different point of view. CI-functionalities are defined by groups of phases (cf. section 4.6 (page 25)) and an approach is given to measure the adoption of such functionalities (cf. section 6.1 (page 37)). The results yield information on how the semantic structure of the build process is constructed iteratively and how mature a build process is.

Q1: Are CI-functionalities used in a consistent manner?

Assuming an absolute threshold ($T = 1$) as an appropriate parameter for the adoption measure is unreasonable, because consistent usage must fulfill a certain variable tolerance. For the five CI-functionalities given (cf. table 4.3 (page 28)), functionality adoption is tested for thresholds $T \in \{0.7, 0.8, 0.9, 1.0\}$. For each threshold T the amount of projects $p_{(T,C)}$, that have adopted a CI-functionality $C \in (1, \dots, 5)$ is represented as $(p_{(T,1)}, \dots, p_{(T,5)})$ (cf. figure 7.6). By lowering the threshold T one can see by how much these values increase.

A 100% threshold, which was previously assumed too strict, actually gives a high project count, especially for the *PRE_EXEC* functionality: (916, 731, 597, 310, 49). More projects are gained by lowering the threshold. The project gain from $T = 0.9$ to $T = 0.8$ is the following: (9, 7, 12, 7, 4). In four out of five cases there is a gain of less than 1% (regarding 100% as all 962 projects), thus a 90% threshold suffices the variable demand, but also remains strict enough.

With a 90% threshold the project count for the five CI-functionalities is the following: (942, 754, 620, 324, 49). Therefore it can be said that by defining a continuous usage representation consistent functionality usage is sufficiently measurable and that consistent usage is indeed observed.

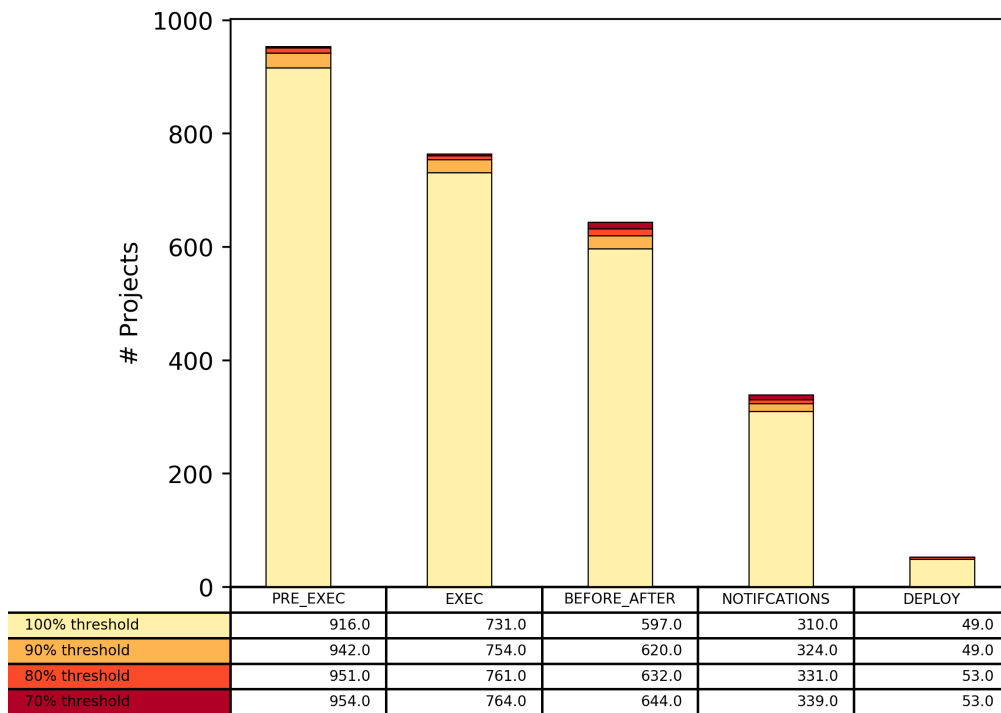


Figure 7.6.: Project count for different thresholds $T \in \{0.7, 0.8, 0.9, 1.0\}$ in the CI-functionality adoption measure.

Q2: How mature are CI build processes in Travis CI?

From the CI-functionality adoption measure it is further possible to derive maturity levels for Travis CI build processes. In order to accomplish this the temporal order, in which the CI-functionalities are adopted, is presented as a five-tuple $(F_1, F_2, F_3, F_4, F_5)$ with $F_i \in \{1, 2, 3, 4, 5\}$ uniquely representing the order of adoption, and $F_i = 0$ meaning the functionality F_i is never adopted. Here the adoption measure with the 90% threshold is used. I map the functionalities in the order as given in table 4.3 (page 28):

$$(F_1, F_2, F_3, F_4, F_5)$$

=

$$(PRE_EXEC, EXEC, BEFORE_AFTER, NOTIFICATIONS, DEPLOY)$$

By clustering on these vectors using an euclidean distance (cf. table A.2 (page 81)), 14 distinct groups are identified (cf. table 7.5). These groups are regarded as CI-maturity levels. The cluster name is simply a name, that is assigned to the cluster by the Weka tool. Of importance are the cluster codes and the cardinality of the cluster given in the count column. The clusters are sorted by the amount of adopted CI-functionality, i.e., by the amount of $F_i \neq 0$ in the cluster code. The cardinality of the clusters does not rise with increasing amount of CI-functionality. In fact most projects adopt three to four CI-functionalities, whereby *DEPLOY* is the least adopted functionality. 18 projects never use any phase consistently.

Cluster Name	Cluster Code	Cardinality	Amount of CI-functionality
cluster10	12345	15	5
cluster4	12304	27	4
cluster7	12340	134	4
cluster9	12430	49	4
cluster3	12300	300	3
cluster12	12003	5	3
cluster2	12030	69	3
cluster5	10230	15	3
cluster14	10320	23	3
cluster1	12000	161	2
cluster8	10200	60	2
cluster11	10020	17	2
cluster6	10000	69	1
cluster13	00000	18	0

Table 7.5.: CI-maturity measured by CI-functionality adoption.

Cluster3 is the cluster with the highest cardinality: $\sim 31\%$ of all projects develop their build process structure according to cluster code (12300). Clusters above cluster3 in

table 7.5 (cluster9, cluster7, cluster4 and cluster10) extend the cluster code (12300) by including *NOTIFICATION* and *DEPLOY* functionalities. The majority of projects ($\geq 54\%$) lie in cluster3 or in one of the latter four. Projects in cluster10 actually adopt all functionality in the order (12345).

Q3: What is the adoption time for CI-functionalities in GitHub projects?

The evolution of CI-functionality adoption is already given by the cluster codes in table 7.5, representing the order in which CI-functionalities are adopted. What remains is to investigate the mean adoption time for these functionalities. This is done for each functionality separately (cf. table 7.6).

CI-functionality	Time in days	Time in commits
PRE_EXEC	5.38	1.13
EXEC	52.39	2.21
BEFORE_AFTER	254.82	8.45
NOTIFICATIONS	123.41	6.72
DEPLOY	426.52	20.57

Table 7.6.: Mean time of adoption for CI-functionalities.

PRE_EXEC, the functionality which is always adopted first, is on average introduced via the first configuration change and in the first five days after the start of CI-usage. The main functionality, the *SCRIPT* functionality, is on average adopted circa seven weeks (52 days) after begin of CI-usage. This is a rather long pause, but on average this is only the second build changing commit. Next come the notifications and additional phases with mean adoption times of a third and two thirds of a year. Only after one year the adoption of *DEPLOY* takes place. In commits this is commonly reached in the 20th configuration changing commit. The average project had a total of 27 configuration changes (cf. section 6.3 (page 41)), thus *DEPLOY* is adopted quite late.

Q4: What is the impact of project age, CI-usage period, phase usage and phase changes on CI-maturity?

The relation between project-level features (cf. section 6.1 (page 35)) and the assigned CI-maturity levels can be analyzed in multiple ways. Here I decided use the Weka tool to evaluate different classifiers for CI-maturity on selected feature subsets of project-level features (cf. appendix B (page 87)). I thereby do not aim at optimizing or finding an optimal classifier, but only observe if there exists a classifier which classifies better than a given threshold classification.

The OneR, ZeroR and J48 classifiers are used (cf. section 2.2.1 (page 8)). The two simple ZeroR and OneR classifiers are considered a minimum threshold for good classification,

as a good classifier should exceed the percentage of correct classified instances, i.e., the precision, of ZeroR and OneR significantly [FHW16].

Classifiers are evaluated upon the following feature subsets:

- (i) All features (F_4, F_5, F_6)
- (ii) meta features: $(c_{sum,abs}, u_{sum,bin}, lang, den, maxID) \subseteq F_4 \times F_5 \times F_6$
- (iii) features of absolute usage: $(u_{1,abs}, \dots, u_{m,abs}) \subseteq F_5$
- (iv) features of relative usage: $(u_{1,rel}, \dots, u_{m,rel}) \subseteq F_5$
- (v) features of binary usage: $(u_{1,bin}, \dots, u_{m,bin}) \subseteq F_5$
- (vi) features of absolute changes: $(c_{1,abs}, \dots, c_{n,abs}) \subseteq F_4$
- (vii) features of relative changes: $(c_{1,rel}, \dots, c_{n,rel}) \subseteq F_4$
- (viii) age and CI-usage period in F_6

As the ZeroR classifier does not rely on the features, it obviously gives the same threshold precision (31.19% correctly classified instances) for each of the following classifications:

- (i) OneR classifies with precision 41.37% and uses the *NOTIFICATIONS_changed_relative* feature as the best predictor. For J48 and a bucket size of two the precision rises to 78.27%, which is significantly higher than both the ZeroR and OneR precisions; even more than twice as precise as ZeroR (cf. table B.1 (page 87)).
- (ii) For classification upon meta features OneR has precision 36.28% whereas J48 with bucket size two only has a precision (30.77%) even worse than both ZeroR and OneR. For bucket size 30 the precision of J48 rises to 38.88%, for greater bucket size it decreases again (cf. table B.2 (page 87)).
- (iii) Classifying upon absolute usage features yields a precision of 42.83% for OneR and the maximum precision of J48 (71.93%) is again achieved with a low bucket size of two (cf. table B.3 (page 87)). Again J48 exceeds ZeroR's precision significantly. OneR once more uses the notification related feature as best predictor.
- (iv) With the relative usage features the highest precision of all J48 classifications is achieved (79.42%) with bucket size two (cf. table B.4 (page 88)). OneR (with best predictor being the notifications) has a low precision of 41.99%.
- (v) - (vii) For these three, again OneR has a precision of circa 41% and J48 exceeds both ZeroR and OneR with an optimal bucket size of five (cf. table B.5 (page 88)) or 20 (cf. table B.6 (page 88) / table B.7 (page 89)). For all three subsets OneR again classifies with the notification related feature.

(viii) The time related features classify with very low precision. OneR (25.88%) classifies even worse than ZeroR and with J48 and bucket size 40 the precision just matches the one of ZeroR (cf. table B.8 (page 89)).

Additionally I manually review the classification capability of distinct phase usage per project upon the visual distribution of the latter (cf. figure 7.1). In figure 7.7 the distribution of CI-maturity levels (colors) is plotted onto the distribution of distinct phase usage (histogram).

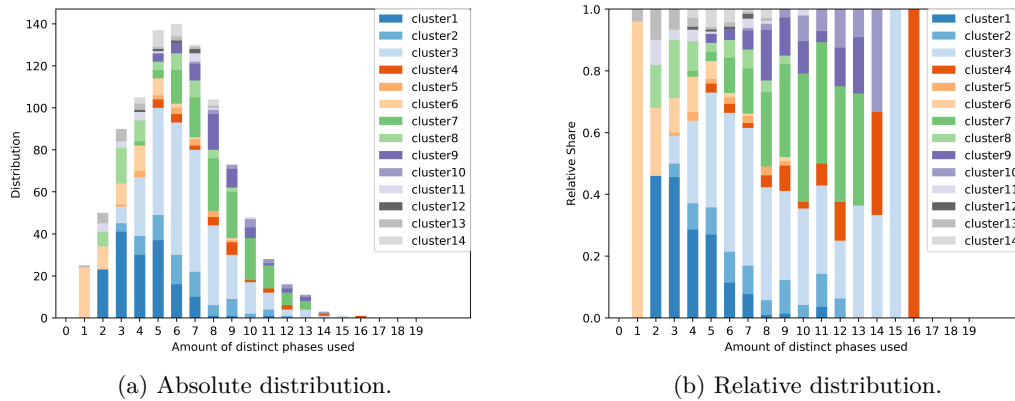


Figure 7.7.: Distribution of clusters mapped to the distribution of distinct phase usage.

Figure 7.7a shows that CI-maturity levels are widely scattered in this distribution. CI-maturity levels cluster3 (12300), cluster1 (12000) and cluster7 (12340) make up most of the absolute shares, which is already known from table 7.5. Viewing the relative distribution of CI-maturity levels upon distinct phase usage (cf. figure 7.7b) gives a better picture for classification. The marginals of the relative distribution are quite exclusively used. Projects using only one phase belong to CI-maturity level cluster6 (10000). 16 distinct phases are used by projects of maturity cluster4 (12304), 15 distinct phases by projects of maturity cluster3 (12300).

Between these marginals there is the following situation: cluster1 (12000) holds the majority of projects for two distinct phases and its relative share then slowly vanishes towards the value eight or nine. Similarly cluster6 vanishes around value seven. or the highly mature and large levels cluster3 (12300), cluster7 (12340) and cluster10 (12345) a growth of relative share with increasing distinct phase count is observable. Cluster3 first appears at value two and quickly grows and holds a circa 30% relative share. Similar is true for cluster7, only that its share grows slower. Cluster10 appears rather late and maintains a low share due to its small size (15 projects only cf. table 7.5). The immature cluster13 has the most of its appearance for values one to four. All other clusters are neglected at this point due to an insignificant absolute share.

7.4.1. Discussion

Consistent usage is analyzed upon CI-functionalities and not for phases. Measuring adoption for single phases only might not reflect the adoption of functionalities as well as when the phases are grouped together as in table 4.3 (page 28), especially as Travis CI allows for ambiguous configurations of builds (cf. section 8.1 (page 72)). It is observed that consistent usage is measurable and a 90% threshold suffices greatly for this, as a lot of projects adopt functionality when this threshold is used.

Cluster3 (12300) is what I determine as the basic maturity level, since the core parts of the Travis CI model are adopted: obviously the *SCRIPT* phase is used, along with some pre-build execution steps, e.g., configuration of the platform environment, and some basic conditional phases (*BEFORE_AFTER*). Notifications are already given by default and the *DEPLOY* is not a necessary part of a CI model, as it can be seen as an extension to CD (cf. section 4.4 (page 24)). Thus the absence of these functionalities is no violation of CI principles. Together with cluster4 (12304), cluster7 (12340) and cluster9 (12430), the projects of these four clusters make up the mature CI processes. Cluster10 holds the projects, which are overly mature, as they fully use the Travis CI model and its CD extensions. More than half of the clusters below cluster3 do strangely not adopt the main *EXEC* functionality, yet they make use of the conditional phases of the Travis CI model or use notifications. Cluster1 (12000), cluster2 (12030), cluster5 (10230), cluster12 (12003) and cluster14 (10320) can be considered some kind of premature build processes, all other clusters below cluster3 are with reservation assessed as immature.

Two extreme cluster groups have been found: The groups which are assessed as immature (cluster6, cluster8, cluster11, cluster13) and the projects that fully adopt all of the model's CI-functionalities (cluster10). Of all projects only 15 (1,5%) adopt every functionality. The possible reasons are manifold: Is it hard to reach full maturity? Is it unnecessary? Or is reaching full maturity depending on the project's age or CI-usage period?

Assuming full maturity as hard to reach is not reasonable: maturity is purely computed upon phase usage, i.e., the usage of keywords in the configuration, thus there is no complexity in reaching a maturity level. The 15 projects from cluster10 can be typed as two platforms, two frameworks, four python libraries, three data management / processing systems, one mobile app, one database plug-in and one hoax project. Especially for the hoax project (a project which enables the user to include webcam selfies in Git commit messages) it is surprising that it was developed with a mature CI configuration. However, it can not be said that the project's type implies a high maturity level. The same can be said for the ages and CI-usage periods of these 15 projects, displayed in table 7.7. The values are widely spread and do not allow for identification of distinct characteristics of highly mature CI builds.

For the non-mature clusters the following has to be noted: Having a low quantitative adoption of CI-functionality does not necessarily imply that the build process is weak or ill suited. I picked a random sample of projects from the non-mature clusters and reviewed their repositories on GitHub. It turned out that for some projects the *SCRIPT*

ID	Age	CI-usage period
1	1.41	0.90
2	1.61	1.61
3	2.35	2.27
4	2.59	1.98
5	3.00	2.76
6	3.33	3.34
7	3.57	2.92
8	3.71	3.54
9	5.08	4.70
10	5.22	4.67
11	6.31	5.37
12	6.60	4.68
13	6.67	2.63
14	7.01	4.59
15	11.47	2.35

Table 7.7.: Age and CI-usage period of the projects in cluster10 (sorted by age).

phase is used but just not recognized as adopted, because the 90% threshold is never crossed. In other cases projects rely on the conventional default build execution in the *SCRIPT* phase (cf. section 4.3.3 (page 23)). Maturity levels are constructed with the implicit assumption that using conventions is lower valued than ownership of functionalities, improving portability and readability of the configuration without much a priori knowledge. Yet using conventions may ensure builds in different projects of the same owner run homogeneously. The benefits of using these conventions are debatable, but beyond the scope of this thesis. Maybe for Travis CI the usage of effective configurations (after pushing a configuration default steps are explicitly added to the configuration file automatically) is more convenient. However the immaturely classified clusters are small in size (summarized: $\sim 20\%$ of all projects). Hence a negative impact on the quality assignment to the clusters is rather neglectable.

For classification of projects the J48 classifier is used and it is evaluated for which feature subsets the latter provides a significantly better classification precision than the ZeroR and OneR classifiers. Meta data, i.e., all data that does not directly relate to a certain phase, and the combination of the projects age and CI-usage period both did not suffice for simple and good classification (precision $\sim 25 - 38\%$). The phase related data (phase usage, phase changes and all features) however presented a good basis for classification as precision ($\sim 70 - 80\%$) is approximately doubled compared to ZeroR and OneR. An interesting observation is that notification related features are always chosen as the best predictor in every case such a feature is present in one of the subsets (i)-(viii).

For the classification capability of distinct phase usage the following is observed. Except for the marginals, for all values of distinct phase usage no CI-maturity level holds a

50% or higher relative share (cf. 7.7b). Thus a classifier, that would perform a ZeroR classification for each value of distinct phase usage, would have an overall precision of lower than 50%.

Hence distinct phase usage by itself does also not provide a better classification capability, but it confirmed the plausibility of the CI-maturity level assignment (cf. section 7.4) I do not want to strictly assign higher quality to build processes only because they use more distinct phases, but through figure 7.7a it becomes clear that projects, which use more distinct phases, tend to have a higher CI-maturity level.

7.5. Goal 5: Equivalent Usage of the Travis CI Model

In this section I aim to find characteristics of equivalent usage of the Travis CI model by clustering projects upon selected feature subsets on project-level (Feature Vectors F_4, F_5, F_6 ; cf. section 6.1 (page 35)). Clustering all features at once yields a too distorted view on the projects, therefore this option is not further pursued.

Q1: Can projects be clustered by similar usage of phases?

Yes, as clustering upon the binary phase usage features on project-level (cf. section 6.1) yields in which combination phases are used together in a project (cf. table A.3 (page 82)). Clustering revealed that there are nine main groups. All clusters, as seen several times already, use the basic *SCRIPT* and *LANGUAGE_ENV* phase. *BEFORE_INSTALL* and *INSTALL*. are also often used, and if then they are commonly used together. Strangely projects of cluster two (127 projects) utilize *BEFORE_INSTALL* without the *INSTALL* phase. Fewer used phases (cf. table 7.1a) are used to such small extend, that they are not recognized in any cluster's mean. The 84 projects of clusters five and the 58 projects of cluster seven show usage of exceedingly many phases. Of all remaining phases (phases 6,8-12 cf. table 4.2 (page 26)), cluster five uses nearly all phases and moreover three of them exclusively. It is the only cluster which configures the *BEFORE_SCRIPT*, *PLATFORM_ENV* and *GIT* phases. Cluster seven is the same as cluster five, except for the usage of the latter three phases. No other significant correlation between projects upon phase usage can be observed.

Q2: Can projects be clustered by similar phase changes?

Yes, as clustering upon the phase change sum features on project-level (cf. section 6.1 (page 35)) yields groups of projects with characteristic focus on phases that they change more frequently. In table A.4 (page 83) the cluster means for the six clusters are listed. I compare each cluster against the full data means.

Cluster zero (64 projects) shows a significantly larger amount of phase changes for the phases *BEFORE_INSTALL*, *SCRIPT*, *BUILD_ENV* and *LANGUAGE_ENV*. Projects in this cluster often change the notifications configuration.

For cluster one, the cluster with the highest cluster size (544 projects), the characteristic is, that for none of the phases the clusters mean exceeds the means of the full data. Most

of the projects, because this cluster holds the majority ($\sim 56\%$) of all projects, change most phases not more than once. The phase undergoing the most changes in cluster one is the *LANGUAGE_ENV* with an average of four changes.

Projects of cluster two (58 projects) have a strong focus on changing the *BUILD_MATRIX*, *BUILD_ENV* and *LANGUAGE_ENV* phases. This cluster contains those projects which among others strongly focus on modifying the build environment and maintaining multiple jobs in the build process.

Cluster three (212 projects) is similar to cluster zero except for a focus on the *INSTALL* phase instead of *BEFORE_INSTALL*, and no special focus on the *NOTIFICATIONS*. As in cluster zero the focus also lies on the *SCRIPT* phase.

Cluster four (74 projects) is a mixture of clusters two and three. Additionally the *PLATFORM_ENV* is special to these projects. Projects in this cluster focus both on the build environment / multiple jobs and the main *SCRIPT* phase.

The 10 projects of cluster five exceed the full data means for nearly every phase, most of the times by the factor ten. In contrast to all other clusters, here the focus on deploy related phases clearly emerges. The same holds for the conditional phases *AFTER_FAILURE*, *AFTER_SUCCESS* and environmental phases as *GIT*, *ADDONS* and *CACHE_ENV*.

Q3: Can projects be clustered by amount of configuration changes?

Yes, clustering upon the total amount of configuration changes per project resulted in five clusters (cf. table A.5 (page 84) / table 7.8).

Cluster#	Mean # configuration changes	Share (rel.)
0	189.08	12 (1.25%)
1	7.38	452 (46.98%)
2	99.15	55 (5.71%)
3	53.64	146 (15.17%)
4	26.34	297 (30.87%)

Table 7.8.: Resulting cluster means for clustering on total configuration changes.

The overall mean of configuration changes per project is 27. Circa 30% of all projects gather around this value. Even more ($\sim 47\%$) projects change their configuration even less. Fewer projects lie above the mean, with cluster size decreasing as the cluster mean increases. This shows that the average amount of configuration changes lies in the range [7, 54]; applying significantly more changes to the configuration is rather unusual. This could possibly be due to a lower amount of phase changes per build change in these projects. So projects that have a lower amount of build changes might change more phases at once.

Q4: Can projects be clustered by density of phase changes?

Yes, clustering on phase change density per project resulted in six clusters (cf. table A.5 (page 84) / table 7.9). In contrast to viewing the absolute amount of configuration changes, this clustering considers the density of changes regarding time. Furthermore is the *den* feature (cf. section 6.1 (page 35)), on which the clustering is executed, a measure for mean phase changes, not configuration changes.

Cluster#	A phase change every ... days	Share (rel.)
0	6.94	62 (6.44%)
1	69.44	422 (43.87%)
2	4.21	24 (2.49%)
3	2.42	6 (0.62%)
4	23.15	298 (30.98%)
5	11.90	150 (15.60%)

Table 7.9.: Resulting cluster means for clustering on phase change density.

Phase change density is given as the ratio of phase changes per hour. In the middle column of table 7.9 the results are simplified to express the mean time between two phase changes, as if the phase changes were equidistantly mapped onto the CI-usage period. Thus a lower value expresses a higher density. The total mean of phase change density lies at 19.84 days (\sim three weeks) between each phase change. Again it is apparent, that most projects are gathered around this mean or above (clusters one, four and five), implying that the density is quite low. Few projects (accum. \sim 9.56%) exhibit a high density, up to a phase change every two to three days, or two to three times a week respectively.

Q5: Can projects be clustered by age?

No, not in any beneficial way. It was already seen in figure 6.4a (page 41), project age is approximated by a normal distribution (with mean 4.66, variance 2.69). Thus clustering solely on the project age feature yields no valuable information, as there is only one focus point in the feature space. For increasing K in the SimpleKMeans, K clusters of equally large span are outputted.

Q6: Can projects be clustered by CI-usage period?

For clustering upon the CI-usage period holds the same as for clustering upon project age. Thus the answer is also no.

7.5.1. Discussion

Except for the age and CI-usage period features, clustering is possible and revealed that on average 4-6 distinct clusters can be identified. Furthermore it is learned that extremes

with notably high values are always grouped in clusters of small size, whereas cluster sizes for extremes with significantly lower values than the overall mean are very large. The final clusters in the second clustering revealed that there exist different types of projects, e.g., projects that focus on maintaining the main *SCRIPT* phase or maintaining complex build processes.

As the final clusters in the third and fourth clustering both include small outlier clusters with extraordinary high values, it can be assumed that the projects that change their configuration often also have higher count of phase changes, or higher phase change density respectively. The previous assumption that projects with more build changing commits maybe have a lower phase change rate per build change could not be confirmed. Concrete links between these extreme clusters for the different feature subsets are not further investigated in this thesis.

7.6. Final Discussion

Acceptance of phases in the Travis CI model (cf. section 7.1) is measured by the rank of the phase in the overall usage and change rankings (cf. table 7.1a / table 7.1b). These rankings are recognizable in many other findings, as there are always two sets of phases observable. One of which are the phases that are regarded rather basic and are therefore often included in the configuration (thus higher ranked), the other phases are regarded as advanced configuration of the build process (thus lower ranked). It can be seen, especially with the CI-maturity levels, that not all phases need to be used for a good build process.

The analysis of phase usage frequency yields that phase usage is mostly at a constant level. These levels actually align with the values of the phase-usage ranking in section 7.1, which shows by how many projects a phase is used in total (cf. table 7.1a). This measure expresses the maximum amount of projects by which a phase is used. Furthermore, this measure neither regards any temporal aspects, nor does it imply that these projects used this phase in the same period of time or that they used this phase consistently. Yet the frequency graphs presented in section 7.3 show that the usage levels from table 7.1a do not differ greatly from the real usage at any point throughout the whole CI-usage period.

Also for the CI-functionalities a comparison to section 7.1 can be drawn: The CI-functionality adoption rates are higher for functionalities, which include phases that have higher acceptance given by the rankings in table 7.1a and table 7.1b.

Furthermore 14 CI-maturity levels have been identified (cf. section 7.4), yet the number of final clusters resulting from the clustering in section 7.5 is significantly lower (e.g., six final clusters for phase changes). Obviously there is no 1 : 1 mapping possible, when assigning maturity levels to projects, due to comparatively few distinct characteristics. Thus the classification precisions in section 7.4 are particularly good.

In section 7.5 projects are clustered upon project-level feature subsets. The final clusters for phase usage and phase changes once again relate to the results of section 7.1, in

particular to the rankings (cf. table 7.1a / table 7.1b). The higher ranked phases form a basis that is used in nearly every project. The lower ranked phases then are used in different combination with these basis phases. Similar holds for the phase changes, as lower ranked phases are less likely to be changed often. This is observable in the small size of the clusters, for which such phase change values are significantly high.

The majority of projects perform fewer configuration changes and have a lower average phase change density (cf. section 7.5) than the mean of the full data. The visualization of phase changes and phase usage (cf. figure 7.3 / figure 7.4) within the CI-usage period show that phase change rates drop sharply after first setup and there exists a small experimental usage period at beginning of CI-usage (cf. section 7.3). However this does not have any impact on the maturity levels assigned. It can be seen that only few projects raise the effort to maintain their CI configuration consistently, yet a majority of projects share a moderate to high CI-maturity level (cf. section 7.4). Also in contrast to the steady usage of phases, the change rates for phases oscillate at a moderate level. This implies that projects develop their build processes incrementally, but only upon the phases they used since the beginning. In sum for all 962 projects a stabilization, i.e., progressive reduction of phase change rates, could not be observed (cf. section 7.3).

8. Conclusion

Contents

8.1. Threats to Validity	72
8.1.1. Construct Validity	72
8.1.2. Inner Validity	73
8.1.3. Outer Validity	75
8.2. Future Work	77

The first rough insights that are presented in this thesis already give a good view on modern CI practices in OSS projects. It is possible to make statements on the acceptance and robustness of the Travis CI platform, and also on the evolution and maturity of build processes. With the general assumption that developers use Travis CI appropriately, the following conclusions are drawn.

Rankings of phase changes and usage show that phases that express more basic CI-functionality are preferred to be used in configurations more often than others. The deploy functionality and other more advanced functionalities are less favored.

The Travis CI model is regarded stable towards changes. Although the probability of the build result being a failure after a build change is quite high (40%), it is observed that this probability is constant and independent of the phases that are changed (cf. section 7.2 (page 48)). Without further investigation of the true cause of the build failure, it cannot be clearly said that this anomaly is solely due to the Travis CI model. Ultimately however, it is not observed that a special set of phases cause more build failures upon change as others do, therefore the Travis CI model is considered stable with slight constant uncertainty.

It is further possible to derive a maturity measure for projects from this data set by clustering projects upon the order in which they adopted CI-functionality in their configuration. A great share of projects adopt the necessary CI related functionality that Travis CI has to offer.

Most projects pursue a fast setup and minimal involvement strategy, whereby minimal involvement in this case only relates to the low adoption of new phases (thus somewhat confirming the findings of Hilton et al. [Hil+16]). It follows that projects develop their build processes incrementally but only upon the phases they used since the beginning. In sum for all 962 projects a stabilization, i.e., progressive reduction of phase change rates, could not be observed (cf. section 7.3 (page 52)).

Furthermore there exists a fairly good classification capability for most of the recorded project-level features, with regard to the CI-maturity levels. Age turned out to be no measure for CI-maturity at all. The concrete relationship between CI-maturity levels and

the feature's values is not investigated, yet a strong relation exists (classification precision up to 80%). This enables the prediction of maturity for future CI build processes of OSS projects, using project-level features as indicators.

The data set which is explored by this thesis holds the change history of CI configuration files from 962 GitHub OSS projects, that use Travis CI. In retrospective this data set provides useful insights into the usage of Continuous Integration on the Travis CI platform. The results give satisfying answers towards the research questions, however the potential of this data set is not yet exhausted. Thus such historic build data can indeed be successfully utilized for future research.

8.1. Threats to Validity

In the following I want to address some of the main threats towards my work. For the structure of this chapter I orientate myself by the work of Beller, Gousios, and Zaidman [BGZ17a] and Hilton et al. [Hil+16]. Threats are grouped by the construct, inner and outer validity.

8.1.1. Construct Validity

Construct validity affects the setup of my research and the collection of the data.

Research Setup

Did I ask the right questions? This research is conducted on a previously rather unexploited data set. The research focus (cf. section 4.5 (page 24)) is constructed upon viewing related work and an own analysis of the historic build data available. Thus the research is set up to revolve around changes to CI configuration files (cf. section 4.5 (page 24)) and to exploratively gather knowledge on how CI is used in OSS. From there on it is only logical to ask basic questions first while also following the premise to keep preprocessing (including feature selection) and mining tasks as simple as possible for first insights with this new data set.

Why did I not pursue other questions? The research questions in this thesis view the data broadly, but from many perspectives. In retrospective research could be improved by incorporating knowledge from other research questions (e.g., perform visualization of changes and usage not upon groups of CI-usage period (cf. section 7.3 (page 52)), but rather on the clusters of CI-maturity (table 7.5 (page 59))). Such iterative research or the inclusion of more suitable research questions does simply not conform to the scope of this thesis.

Data Collection

The collection of CI configurations is performed by custom extraction scripts, which simply automated the cloning of a repository and the extraction of each version of the `.travis.yml`

file visible on the repository's Git history. For this only a simple sequence of Git shell commands are used, thus threats to the correct collection of CI configuration belong solely in the outer validity (cf. section 8.1.3). For the build results the TravisTorrent [Traa] data set is used, inheriting the threats of those who supplied it [BGZ17b]).

8.1.2. Inner Validity

The inner validity affects the internal quality of my results, i.e., it concerns how performing this research and the decisions I made during affect the quality of my results.

Research Performance

Travis CI leaves a lot of freedom in its model, especially through the phases that can call custom scripts. Research is performed under the assumption that developers use the Travis CI model appropriately. This means that if beyond simple key value configuration custom scripts are used, they are assumed to be called in the phase, which they are semantically destined for. Inappropriate usage can significantly affect the data, e.g., would the result of a build process, whose unit tests fail in the *INSTALL* phase differ from the models desired result, where the tests should be executed in the *SCRIPT* phase. A developers benefit from inappropriate model usage is questionable, yet inappropriate usage is possible.

Although it is not further investigated to which extend such anomalies occur, for future research it is only correct that preprocessing is performed under the above assumption, i.e., preprocessing being conservative. It minimizes distortion from the original data and enables that such anomalies can be investigate in the future.

Preprocessing tasks are also automated by the use of custom scripts, which handled the comparison of two consecutive configurations to extract differences. By testing preprocessing on self-written exemplary configuration files, I am confident that the preprocessing fully meets the correctness requirement.

In preprocessing the feature of CI-usage period is measured as the time passed since the start of CI-usage until December 20th, 2016. As I research the usage of CI in OSS, especially along with the use of the specific CI platform Travis CI, the start of CI-usage is defined as the start of CI platform usage. Although CI mentality (cf. section 3.2 (page 13)) could have been adopted earlier, it is hard to measure this from the available data and would also produce a lot of overhead for this research. The usage of Git could be viewed as the usage of an CI tool but I chose to disregard this. Thus, the start of CI-usage is the date of the commit that introduced the first version of the *.travis.yml*. Analogously the project age is determined by the time that passed since the very first commit in a project's Git history.

Keeping the research simple is the guideline presented in research setup (cf. section 8.1.1), thus simplification in preprocessing took place multiple times introducing small biases. The grouping of the configuration's top-level keys to phases and the grouping of phases to

CI-functionalities (cf. section 4.6 (page 25)) serves the purpose to simplify research performance and extract simple results. However, the information loss through simplification is rather low.

Lastly a word on the used mining algorithms: I used the algorithms versions as implemented by the Weka tool [FHW16], which is commonly used for mining purposes. For clustering the SimpleKMeans algorithm is used, for classification the OneR, ZeroR and J48 algorithms. The following threat is also justified by the simplicity guideline in the research setup (cf. section 8.1.1):

Why are exactly these algorithms used? It is not the goal of this research to optimize the clustering and classification applications, but rather to observe if such clustering and classification is possible and if so, then the evaluation of adequate resulting clusters (cf. appendix A (page 79) for explanation of clustering approach) and classifiers takes place. The SimpleKMeans algorithm may be limited to finding cluster of certain geometrical shape in the feature's values spaces only (convex volumes), but unusually shaped clusters were not expected. Thus the simplicity of the SimpleKMeans suffices greatly for this research. The same simplicity argument holds for the classifiers, as finding an over-fitting classifier would not be of worth. As seen in the results (cf. section 7.5 (page 65)), the J48 classification algorithm supplies many good classifiers (precision of 75 – 80%), thus no further classifiers are tested.

Handling Blurriness in Data

A configuration change without a semantic build change is possible by pushing a syntactically incorrect configuration, as it cannot be parsed by the Travis CI platform. This creates unrecordable change intent. The best way to handle this is by skipping the syntactically incorrect configuration in the extraction of the change history (cf. section 6.2 (page 39)). In all other cases in which the configuration change does not yield a build change (e.g., by changing a comment) the configuration is not excluded from preprocessing. However, the existence of non-semantic configuration changes are stored (cf. section 6.2 (page 39)), so that the data set remains compliant to other meta data (e.g., total amount of build changes).

Due to changes often being pushed in a multi commit push build results are not always directly link-able by a Git commit id, as Travis CI issues only one build on behalf of the latest commit in the push. As a heuristic the chronological next build result is used.

Responsibility for Blurriness in Results

Many of the threats against how I perform the preprocessing and mining are justified by the conservative and simple nature of my preprocessing and the simple nature of the mining. Therefore I just wish to point out some of the decision that are made during this thesis, which may be the cause for some distortion in the results.

Default phases in Travis CI introduce two problems: Firstly for projects that do not explicitly use them and rely on the Travis CI default conventions changes to and usage of these phases is not recorded. Secondly I observed an anti-pattern listing 7.1 (page 47) in which phases are only used to explicitly deny their default functionality. Apparently some developers decide against using defaults in the model. If so, changes and usage are erroneously recorded. In general only syntactical changes are recorded but yet they slightly differ from the semantic changes I analyzed. The occurrence of invalid files together with the such anti-patterns, e.g., denying defaults, pose a threat towards user intent.

For visualizing change and usage frequency, the projects need to be comparable, thus the CI-usage period is normalized to $[0, 1]$ and the configuration changing commits are linked to a relative point in time in this interval. To counteract the distortion caused by the real age of the commits projects are grouped by age (cf. section 7.3 (page 52)).

For the computation of a CI-functionality adoption measure (cf. section 7.4 (page 57)) there are three aspects worth knowing: Firstly, for the computation of a continuous usage representation configuration changes are viewed as equidistant events, i.e., the highs and lows of the discrete usage representation (cf. figure 6.1a (page 38)) are not weighted with the real temporal distance between two changes. Secondly continuous usage is computed for each phase separately. Later, when continuous usage for CI-functionality, i.e., a group of phases, is computed the following is done. A CI-functionality is considered adopted only if at least of its phases is continuously used, with respect to a certain threshold. The possibility that the discrete usage of two phases of a CI-functionality complement each other (thus accumulated the CI-functionality could be seen as continuously used by combination of two phases), is not considered. Thirdly when having the point of time from which on a phase is continuously used, the corresponding value in the discrete usage representation must not be a 1, i.e., the continuous usage can reach the threshold before the phase is ever really used. Due to the high threshold (90%) such distortion is hardly decisive.

8.1.3. Outer Validity

The outer validity affects the generalizability and external quality of my findings.

Scope of this Thesis

The generalizability of the findings is mainly limited by the scope and depth of my research. I only view a subset of 962 (arbitrary) OSS projects, whereby I restrict myself to the GitHub and Travis CI platforms. Arbitrary because I use a ready-made list of projects meeting certain basic constraints (cf. section 4.2 (page 19)) given by the TravisTorrent raw build logs [Traa]. The projects are not further reviewed, so everything from small custom to big enterprise software projects are included. In retrospective only four main programming languages are included in this project list (*Java, Ruby, Python*

and *Go*). Furthermore, only the Git master branches is consulted for extracting a CI configuration history. Unfortunately no non-OSS projects are included in this research. As CI is a rather modern practice (tool-wise) and Travis CI only emerged after 2011, the longest CI-usage period recorded is five years.

There is a limit to the depth of my research, as I only regard the direct changes to the CI configuration. Going deeper and analyzing changes to called scripts and everything that these call is not part of this research. In contrast to the Travis CI configuration, custom scripts have no underlying model on which changes can be classified and interpreted as easily. However, this also evokes an unavoidable threat towards the minimal involvement theory. It is indeed possible to change the main parts of the build process by only changing called scripts and letting the configuration as is for a long period.

Data Quality

Lastly some notes on data quality which also affects the generalizability of my findings. This research is conducted on the naive assumption that the data is complete and untampered with. Due to the temporal delta between data analysis and preprocessing, it cannot be ensured that the Git histories were not tampered with by the developers. Moreover would a tampered Git history obviously not match with the TravisTorrent data set which holds data on the build results. To which extend this could have happened in those few months I cannot say but I believe it to be small.

In this thesis only the configuration files of the project's master branches are of interest. But this also means that if some branches might have been merged back into the master branch, their history is integrated into the history of the according master branch. How vastly merging of branches influences the correct linearity of the CI-configurations cannot be said at this point either.

Travis CI's interpretation of the *.travis.yml* configuration allows for ambiguous definitions of semantically equivalent build processes. In the following example both configurations describe a build process which expands to 2 jobs, although the first file does not use the *matrix* key (cf. listing 8.1 (page 76)). This implies that build processes are in most cases, but not in every case, uniquely describable by the usage of the phases.

1	language: ruby	1	language: ruby
2	rvm:	2	matrix:
3	- 1.9.3	3	- rvm: 1.9.3
4	- 2.1.0	4	env: DB=mongodb
5	env:	5	- rvm: 2.1.0
6	- DB=mongodb	6	env: DB=mongodb

Source Code: Ambiguity in Travis CI configurations.

Conventions like relying on default phases pose a threat against the CI-maturity measure. Although such phases do not have to be exclusively used, i.e., indirectly these phases are used in every build process, it is higher valued if the developers owned their own *SCRIPT* and *NOTIFICATIONS* phase. Also

It is quite possible that this thesis is at an unfortunate point in time to examine the cross-sections of Travis CI build processes. As seen in figure 6.4b (page 41) most projects use CI since around half or less of the life time of Travis CI showing a growing trend in usage of this platform. Results may be blurred by extremely young projects and hard to generalize for projects that have used CI for a significantly longer period.

8.2. Future Work

This data set provided many useful options for researching usage of Continuous Integration in OSS projects. This thesis only presented some simple preprocessing of the data and thus only basic insights. It therefore subjects a lot of research possibilities to deal with in future work.

Firstly the findings of this thesis should be compared to other CI tools and platforms, processing their CI configuration in a similar fashion, provided that these tools possess such configuration files. Secondly research on the Travis CI platform should be continued, e.g., by

- performing deeper analysis of this data and possibly including new features, e.g., for analyzing how build changes go along with software changes (thus analyzing the true cause for build failures after build changes) or for analyzing changes to scripts which are defined in the CI configuration but whose internal changes are not recognized in the configuration's change history.
- performing analysis on a new or larger set of projects or to a later point in time, thereby reviewing the generalization of these findings.
- performing dedicated analysis with focus on specific programming languages or specific project age for instance.
- analyzing how the clusters found under the goal of equivalent usage and the clusters of CI-maturity relate to specific project typologies.
- performing a more detailed preprocessing of the data, e.g., distinguishing phase changes by inclusion, exclusion and maintenance to a present phase in the configuration.
- analyzing the usage of anti-patterns in CI configurations, e.g., the explicit deny of default email notifications or execution of commands via custom scripts in phases that are not semantically destined for such usage.
- performing an analysis on and comparison to non-OSS projects on the GitHub and Travis CI platforms.

A. Clustering Results

Clustering is performed with the SimpleKMeans clustering algorithm of the Weka data mining tool [FHW16].

For each clustering task the amount K of clusters to be created is increased progressively until at least one of the following two criteria matches: Either if the clusters reflect a sufficiently clear separation of the data, i.e., K is increased until the SME (squared mean error) does not significantly decrease for larger K . Secondly for each clustering task there is a expectation of the maximum amount of clusters to find, so that the interpretation of the resulting clusters remains reasonable. From 2-5 random initial seeds the final clusters with the lowest total SME are chosen as most suitable result.

Clustering is mostly performed on numerical features. For clustering on binary values, a hamming distance is used. For clustering on real numbers or mixed value spaces an euclidean distance is used.

Attribute	Full Data (10102.0)	Cluster# 0 (3539.0)	1 (306.0)	2 (1161.0)	3 (1254.0)	4 (50.0)	5 (359.0)	6 (445.0)	7 (923.0)	8 (1152.0)	9 (785.0)	10 (128.0)
BEFORE_INSTALL_changed	0	0	0	1	0	0	1	1	0	0	0	0
INSTALL_changed	0	0	0	0	1	0	1	1	0	0	0	0
BEFORE_SCRIPT_changed	0	0	0	0	0	0	0	0	0	0	0	0
SCRIPT_changed	0	0	0	0	0	0	1	1	0	1	0	0
BEFORE_CACHE_changed	0	0	0	0	0	0	0	0	0	0	0	0
AFTER_FAILURE_changed	0	0	0	0	0	0	0	0	0	0	0	0
AFTER_SUCCESS_changed	0	0	0	0	0	0	0	0	0	0	0	1
AFTER_SCRIPT_changed	0	0	0	0	0	0	0	0	0	0	0	0
BEFORE_DEPLOY_changed	0	0	0	0	0	0	0	0	0	0	0	0
DEPLOY_changed	0	0	0	0	0	0	0	0	0	0	0	0
AFTER_DEPLOY_changed	0	0	0	0	0	0	0	0	0	0	0	0
BUILD_STAGES_changed	0	0	0	0	0	0	0	0	0	0	0	0
BUILD_MATRIX_changed	0	0	0	0	0	0	0	1	1	0	0	0
NOTIFICATIONS_changed	0	0	0	0	0	0	0	0	0	0	0	0
ADDONS_changed	0	0	1	0	0	0	0	0	0	0	0	0
CACHE_ENV_changed	0	0	0	0	0	0	0	0	0	0	0	0
PLATFORM_ENV_changed	0	0	0	0	0	1	0	0	0	0	0	0
GIT_changed	0	0	0	0	0	0	0	0	0	0	0	0
BUILD_ENV_changed	0	0	0	0	0	0	1	1	0	0	1	0
LANGUAGE_ENV_changed	0	1	0	0	0	0	1	1	0	0	0	0

Table A.1.: Clustering results: phase changes prior to build failures.

Attribute	Cluster#													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Full Data	(161.0)	(69.0)	(300.0)	(27.0)	(15.0)	(69.0)	(134.0)	(60.0)	(49.0)	(15.0)	(17.0)	(5.0)	(18.0)	(23.0)
PRE_EXEC	0.9875	1.0062	1	1.0033	1.037	1	1	1	1	1	1	1	0	1.1304
EXEC	1.6227	1.9938	2.0558	2.0533	1.8519	0	2.1791	0	2.1429	1.6	0	2	0	0.0435
BEFORE_AFTER	1.8877	0	0	2.9433	3.0741	2	2.9104	2	3.9592	3.1333	0	0	0	3
NOTI	1.0956	0	2.942	0	0.0741	3	3.9104	0	2.898	3.6667	2	0.6	0	2
DEPLOY	0.2006	0	0	0	3.8519	0	0.058	0	0	4.6	0	3.2	0	0

Table A.2.: Clustering results: CI-functionality introduction.

Attribute	Cluster-#								
	0 (237.0)	1 (193.0)	2 (127.0)	3 (117.0)	4 (81.0)	5 (84.0)	6 (53.0)	7 (58.0)	8 (12.0)
BEFORE_INSTALL_used_binary	1	1	0	1	1	1	0	1	0
INSTALL_used_binary	1	1	0	1	1	1	1	1	1
BEFORE_SCRIPT_used_binary	0	0	0	0	0	1	0	0	0
SCRIPT_used_binary	1	1	1	1	1	1	1	1	1
BEFORE_CACHE_used_binary	0	0	0	0	0	0	0	0	1
AFTER_FAILURE_used_binary	0	0	0	0	0	0	0	0	0
AFTER_SUCCESS_used_binary	0	0	0	0	0	1	0	1	0
AFTER_SCRIPT_used_binary	0	0	0	0	0	0	0	0	0
BEFORE_DEPLOY_used_binary	0	0	0	0	0	0	0	0	0
DEPLOY_used_binary	0	0	0	0	0	0	0	0	0
AFTER_DEPLOY_used_binary	0	0	0	0	0	0	0	0	0
BUILD_STAGES_used_binary	0	0	0	0	0	0	0	0	0
BUILD_MATRIX_used_binary	0	0	1	0	0	1	0	1	0
NOTIFICATIONS_used_binary	0	0	0	0	1	1	0	0	0
ADDONS_used_binary	0	0	0	0	0	1	0	1	0
CACHE_ENV_used_binary	0	0	1	0	1	0	0	1	1
PLATFORM_ENV_used_binary	0	0	0	0	0	1	0	0	0
GIT_used_binary	0	0	0	0	0	1	0	0	0
BUILD_ENV_used_binary	1	1	1	1	1	1	1	1	1
LANGUAGE_ENV_used_binary	1	1	1	1	1	1	1	1	1

Table A.3.: Clustering results: phase usage.

Attribute	Full Data (962.0)	Cluster#					
		0 (64.0)	1 (544.0)	2 (58.0)	3 (212.0)	4 (74.0)	5 (10.0)
BEFORE_INSTALL_changed_absolute	4.6861	9.0938	1.5717	7.3966	7.4151	8.0405	47.5
INSTALL_changed_absolute	5.5759	7.9375	2.2132	1.8621	10.8208	10.1351	50
BEFORE_SCRIPT_changed_absolute	2.1403	4.8438	0.4081	5.8276	4.434	1.9054	10.8
SCRIPT_changed_absolute	7.1154	12.375	2.9816	7.0345	12.0189	12.8243	52.6
BEFORE_CACHE_changed_absolute	0.1757	1.0313	0.0846	0.1724	0.1415	0.2297	0
AFTER_FAILURE_changed_absolute	0.2225	0.125	0.0239	0.2069	0.2736	0.2027	10.8
AFTER_SUCCESS_changed_absolute	1.8254	3.4219	0.8382	0.6379	2.3396	3.8243	26.5
AFTER_SCRIPT_changed_absolute	0.2994	1.25	0.0588	0.3103	0.5849	0.2568	1.5
BEFORE_DEPLOY_changed_absolute	0.1798	0.0938	0.0202	0	0.283	0.5541	5.5
DEPLOY_changed_absolute	0.5852	0.7188	0.2371	0.2759	1.283	0.6351	5.3
AFTER_DEPLOY_changed_absolute	0.0603	0	0.0129	0	0.1038	0.3919	0
BUILD_STAGES_changed_absolute	0	0	0	0	0	0	0
BUILD_MATRIX_changed_absolute	5.6892	5.0156	1.1397	28.6379	6.9764	10.2027	63.7
NOTIFICATIONS_changed_absolute	1.5052	8.6875	0.4688	2.2931	1.0377	2.4054	10.6
ADDONS_changed_absolute	1.3711	2.875	0.2849	1	2.3679	3.8243	13.7
CACHE_ENV_changed_absolute	1.6091	3.7344	0.5055	3.9655	2.316	2.3649	13.8
PLATFORM_ENV_changed_absolute	0.684	0.6875	0.1857	0.3448	0.0755	5.9054	4
GIT_changed_absolute	0.922	2.4063	0.3382	3.2069	0.8019	1.3378	9.4
BUILD_ENV_changed_absolute	6.71	11.5313	2.2757	13.4828	10.5377	10.6351	67.6
LANGUAGE_ENV_changed_absolute	8.9563	10.8125	4.3897	34.2586	11.9481	9.9595	27.9

Table A.4.: Clustering results: phase changes.

	Cluster#					
Attribute	Full Data	0	1	2	3	4
	(962.0)	(12.0)	(452.0)	(55.0)	(146.0)	(297.0)
#changes	27.763	189.0833	7.3673	99.1455	53.6438	26.3434

Table A.5.: Clustering results: configuration changing commits.

Attribute	Cluster#					
	0	1	2	3	4	5
Full Data	(962.0)	(422.0)	(24.0)	(6.0)	(298.0)	(150.0)
phase_change_density	0.0021	0.0006	0.0099	0.0172	0.0018	0.0035
AVG (in days): A phase change every	19.84	6.94	4.21	2.42	23,15	11.90

Table A.6.: Clustering results: phase change density.

B. Classification Results

(Remark: Classification with J48 was conducted as a 10-fold cross validation.)

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	41.37%	NOTIFICATIONS_changed_relative
J48 (2)	78.27%	-
J48 (5)	75.36%	-
J48 (10)	74.64%	-

Table B.1.: Classification results: (All \Rightarrow CI-Maturity).

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	36.28%	phase_change_density
J48 (2)	30.77%	-
J48 (10)	36.59%	-
J48 (20)	37.84%	-
J48 (30)	38.88%	-
J48 (40)	36.69%	-

Table B.2.: Classification results: (Meta Features \Rightarrow CI-Maturity).

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	42.83%	NOTIFICATIONS_used_absolute
J48 (2)	71.93%	-
J48 (10)	70.16%	-
J48 (20)	69.65%	-

Table B.3.: Classification results: (Absolute Usage \Rightarrow CI-Maturity).

B. Classification Results

(Remark: Classification with J48 was conducted as a 10-fold cross validation.)

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	41.99%	NOTIFICATIONS_used_relative
J48 (2)	79.42%	-
J48 (5)	78.07%	-
J48 (10)	75.88%	-

Table B.4.: Classification results: (Relative Usage \Rightarrow CI-Maturity).

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	41.99%	NOTIFICATIONS_used_binary
J48 (2)	70.48%	-
J48 (5)	70.79%	-
J48 (10)	69.96%	-

Table B.5.: Classification results: (Binary Usage \Rightarrow CI-Maturity).

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	41.48%	NOTIFICATIONS_changed_absolute
J48 (2)	67.77%	-
J48 (10)	68.61%	-
J48 (20)	69.33%	-
J48 (30)	69.33%	-
J48 (40)	64.97%	-

Table B.6.: Classification results: (Absolute Changes \Rightarrow CI-Maturity).

(Remark: Classification with J48 was conducted as a 10-fold cross validation.)

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	41.41%	NOTIFICATIONS_changed_absolute
J48 (2)	67.88%	-
J48 (10)	70.78%	-
J48 (20)	70.79%	-
J48 (30)	69.65%	-

Table B.7.: Classification results: (Relative Changes \Rightarrow CI-Maturity).

Classifier	Correctly Classified Instances	Classification Attribute
ZeroR	31.19%	-
OneR	25.88%	ci_usage
J48 (2)	22.25%	-
J48 (10)	25.88%	-
J48 (40)	31.19%	-

Table B.8.: Classification results: (Age and CI-Usage \Rightarrow CI-Maturity).

Bibliography

- [Arf] *Attribute-Relation File Format (ARFF)*. <https://www.cs.waikato.ac.nz/ml/weka/arff.html>. Accessed: 2017-08-10 (cited on page 35).
- [Awsa] *What is Continuous Integration?* <https://aws.amazon.com/devops/continuous-integration/>. Accessed: 2017-08-24 (cited on pages 6, 7).
- [Awsb] *What is DevOps?* <https://aws.amazon.com/devops/what-is-devops/>. Accessed: 2017-08-24 (cited on page 7).
- [BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of Software Engineering*. Wiley, 1994 (cited on pages 10, 11).
- [BGZ17a] M. Beller, G. Gousios, and A. Zaidman. “Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub”. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 356–367. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.62. URL: <https://doi.org/10.1109/MSR.2017.62> (cited on pages 16, 72).
- [BGZ17b] M. Beller, G. Gousios, and A. Zaidman. “TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration”. In: *Proceedings of the 14th working conference on mining software repositories*. 2017 (cited on pages 15, 19, 20, 73).
- [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> (cited on page 6).
- [ES00] M. Ester and J. Sander. *Knowledge Discovery in Databases: Techniken und Anwendungen*. Springer Berlin Heidelberg, 2000. ISBN: 9783540673286. URL: <https://books.google.de/books?id=QNat6WM73Q8C> (cited on pages 8, 9).
- [FHW16] E. Frank, M. A. Hall, and I. H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 2016 (cited on pages 8, 9, 39, 61, 74, 79).
- [FPSM92] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. “Knowledge Discovery in Databases: An Overview”. In: *AI Mag.* 13.3 (Sept. 1992), pp. 57–70. ISSN: 0738-4602. URL: <http://dl.acm.org/citation.cfm?id=140629.140633> (cited on page 8).

- [FPSS96] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. “Advances in Knowledge Discovery and Data Mining”. In: ed. by U. M. Fayyad et al. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1996. Chap. From Data Mining to Knowledge Discovery: An Overview, pp. 1–34. ISBN: 0-262-56097-6. URL: <http://dl.acm.org/citation.cfm?id=257938.257942> (cited on page 10).
- [Ghua] *About GitHub*. <https://github.com/about>. Accessed: 2017-08-23 (cited on page 6).
- [Ghub] *GitHub*. <https://github.com>. Accessed: 2017-08-23 (cited on page 6).
- [Ghuc] *GitHub Blog - One Thousand Strong*. <https://github.com/blog/5-one-thousand-strong>. Accessed: 2017-08-23 (cited on page 6).
- [Ghud] *GitHub Marketplace*. <https://github.com/marketplace>. Accessed: 2017-08-24 (cited on page 6).
- [Ghue] *The state of the Octoverse 2016*. <https://octoverse.github.com/>. Accessed: 2017-08-23 (cited on page 6).
- [Gita] *A Short History of Git*. <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git>. Accessed: 2017-08-23 (cited on page 3).
- [Gitb] *About Git*. <https://git-scm.com/about>. Accessed: 2017-08-23 (cited on page 5).
- [Gitc] *Git - everything is local*. <https://git-scm.com>. Accessed: 2017-08-23 (cited on page 3).
- [Gidt] *Git Branching - Branches in a Nutshell*. <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>. Accessed: 2017-09-11 (cited on page 5).
- [Gite] *Git Internals - Git Objects*. <https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>. Accessed: 2017-08-23 (cited on page 4).
- [Gitf] *Integration Manager Workflow*. <https://git-scm.com/about/distributed>. Accessed: 2017-08-23 (cited on page 3).
- [GVS17] A. Gautam, S. Vishwasrao, and F. Servant. “An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration”. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 495–498. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.38. URL: <https://doi.org/10.1109/MSR.2017.38> (cited on page 16).
- [HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 9780321670229. URL: <https://books.google.de/books?id=6ADDuzere-YC> (cited on page 24).

- [Hil+16] M. Hilton et al. “Usage, Costs, and Benefits of Continuous Integration in Open-source Projects”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 426–437. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970358. URL: <http://doi.acm.org/10.1145/2970276.2970358> (cited on pages 16, 71, 72).
- [HKP11] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123814790, 9780123814791 (cited on pages 7–10).
- [Iee] *INTERNATIONAL STANDARD ISO/IEC/IEEE 24765-2010 - Systems and software engineering – Vocabulary*. <http://ieeexplore.ieee.org/document/5733835/>. Accessed: 2017-12-08 (cited on page 10).
- [Kdd] *KDD2017*. <http://www.kdd.org/kdd2017/>. Accessed: 2017-08-28 (cited on page 10).
- [LL13] J. Ludewig and H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken. 3. Aufl.* dpunkt.verlag Heidelberg, 2013. ISBN: 978-3-86490-092-1 (cited on pages 10, 11).
- [Msr] *MSR 2017 - The 14th International Conference on Mining Software Repositories*. <http://2017.msrconf.org/#/home>. Accessed: 2017-10-16 (cited on page 14).
- [Proa] *Proceedings of the 12th Working Conference on Mining Software Repositories*. <https://dl.acm.org/citation.cfm?id=2820518&picked=prox>. Accessed: 2017-10-28 (cited on page 14).
- [Prob] *Proceedings of the 13th International Conference on Mining Software Repositories*. <https://dl.acm.org/citation.cfm?id=2901739&picked=prox>. Accessed: 2017-10-28 (cited on page 14).
- [Proc] *Proceedings of the 14th International Conference on Mining Software Repositories*. <https://dl.acm.org/citation.cfm?id=3104188&picked=prox>. Accessed: 2017-10-28 (cited on page 14).
- [Traa] *Index of /buildlogs/, howpublished = "https://travistorrent.testroots.org/buildlogs/", note = Accessed: 2017-10-20* (cited on pages 20, 39, 73, 75).
- [Trab] *Travis CI - Customizing the Build*. <https://docs.travis-ci.com/user/customizing-the-build/>. Accessed: 2017-08-24 (cited on pages 21–23).
- [Trac] *TravisCI - Test and Deploy with Confidence*. <https://travis-ci.org>. Accessed: 2017-08-24 (cited on page 7).
- [Trad] *TravisTorrent - Free and Open Travis Analytics for Everyone*. <https://travistorrent.testroots.org/>. Accessed: 2017-10-20 (cited on page 19).