Master thesis

February 16, 2021

CINDER

Find the matching project for your next CI Study

Timothy Zemp

of Lucerne, Switzerland (13-915-632)

supervised by

Prof. Dr. Harald C. Gall

Dr.-Ing. Sebastian Proksch (TU Delft)

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master thesis

CINDER

Find the matching project for your next CI Study

Timothy Zemp

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master thesis

Author: Timothy Zemp, timothy.zemp@uzh.ch

URL: https://github.com/cinder-config

Project period: 17.08.2020 - 16.02.2021

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First and foremost, I want to thank my thesis supervisor Sebastian Proksch for suggesting this topic. Your continuous support, feedback and guidance have been of great assistance during the thesis. You gave me the opportunity to make this thesis my work. Nevertheless, you have always tried to challenge my decisions and, if necessary, guided me in the right direction. A special thanks goes to all participants of the Classification Task. Without their valuable inputs, this work would not have been feasible. I would like to express a special thanks to Jonathan Heitz. He provided valuable suggestions and guidance with the machine learning topics. Additionally, thanks go to Franziska Benz for proofreading this thesis, and all my fellow students and friends who have supported and encouraged me. Lastly, I want to thank Prof. Dr. Harald C. Gall from the Software Evolution and Architecture Lab research group for giving me the opportunity to write this thesis.

Abstract

Continuous Integration (CI) is a software development practice introduced by the Agile movement with the aim of delivering reliable software releases quickly by regularly integrating changes to the software. The spread and success of CI has lead to a spike in empirical software engineering research, examining the benefits and the impact of this new practice. Implementing Continuous Integration is relatively simple because it is only required to add a configuration file to the repository and register with a CI cloud provider. Unfortunately, due to its easy adaptability, in many software repositories the process is poorly implemented. This is a substantial risk that threatens the validity of CI-based studies unless care is taken in the selection of repositories. To overcome this risk we present CINDER, a tool that detects genuine CI configuration files. The tool works by using a random forest classifier trained on a labeled ground truth data set and various features describing the characteristics of configuration files. With CINDER we show that significant action within the pipeline and its regular adaptation is a strong indicator of the genuineness of a configuration file. By replicating a study we show that the selection of projects has a significant impact on the results of CI based studies. With CINDER we provide researchers with a tool to enhance the process of selecting applicable software repositories, consequently improving the quality and validity of their studies.

Zusammenfassung

Kontinuierliche Integration (CI) ist eine agile Softwareentwicklungspraxis, die von der Agile-Bewegung eingeführt wurde. Das Ziel besteht darin, die Veröffentlichung von Software durch häufige Integration von Änderungen an der Software zuverlässig und schnell liefern zu können. Die Verbreitung und der Erfolg von CI hat zu einem Anstieg der empirischen Softwareentwicklungsforschung geführt, welche die Vorteile und Auswirkungen dieser neuen Praxis untersucht. Das Einführen von Kontinuierliche Integration ist relativ einfach, da nur eine Konfigurationsdatei in das Repository der Software hinzugefügt und das Projekt bei einem CI Cloud Provider registriert werden muss. Leider ist der Prozess in vielen Softwareprojekten aufgrund der tiefen Einstiegshürden nur schlecht implementiert. Wenn bei der Auswahl der Softwareprojekte nicht sorgfältig vorgegangen wird, stellt dies ein erhebliches Risiko dar, welches die Gültigkeit von CI-basierten Studien bedroht. Um dieses Risiko zu überwinden implementieren wir CINDER, ein Programm, das ernsthafte CI-Konfigurationsdateien erkennt. Das Tool arbeitet mit einem Random-Forest-Klassifikator, der auf einem Wahrheitsdatensatz und verschiedenen Merkmalen, welche die Eigenschaften von Konfigurationsdateien beschreiben, trainiert wurde. Mit CINDER zeigen wir, dass viele Aktionen innerhalb der Pipeline und eine regelmässige Anpassung der Konfiguration an die Bedürfnisse der Software starke Indikatoren dafür sind, dass eine Konfigurationsdatei ernsthaft ist. Mithilfe der Replikation einer bestehenden Studie zeigen wir, dass die Auswahl der Projekte einen signifikanten Einfluss auf die Ergebnisse hat von CI-basierten Studien hat. Mit CINDER stellen wir Forschern ein Programm zur Verfügung, welches dazu dient, die Auswahl geeigneter Softwareprojekte zu verbessern und damit die Qualität und Gültigkeit ihrer Studien erhöht.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

First of all, we are going to introduce our motivation towards the topic and tear the problem of why it is important to detect genuine Continuous Integration configuration files. Then, we propose the goal of the thesis along with the research questions that arise.

1.1 Motivation

Continuous Integration (CI) is a software development practice introduced by the Agile movement with the aim of delivering reliable software releases fast [HF10]. Whenever a change is pushed to a remote software repository (*e.g.*, GitHub), a build server tool (*e.g.*, Travis-CI) detects this change and triggers the integration pipeline. The pipeline typically consists of several steps, *i.e.*, compilation, unit testing, packaging, integration testing, installation and deploy and is usually defined within a configuration file (*e.g.*, *.travis-ci.yml* in case of Travis CI).

The success of CI has lead to a spike in empirical research on this topic that investigated the impact of CI on the process of how software is developed. Studies show that, for example, CI leads to a higher code commitment frequency and reduced code chunks [ZSZ+17], different programming languages show different adoption times (*e.g.*, Ruby as being an early-adopter, whereas Java projects tend to adopt later) [VVSW+14], and proper build configurations should be maintained in order to reduce long build durations [GDCZ19].

Many free cloud CI providers exist, such as Travis-CI or Jenkins, and they are easy to adopt: once a repository is registered, it is sufficient to maintain a single configuration in the repository to use the service. Due to this low entrance barrier, many public repositories contain such configuration files. Many of them do not use CI seriously though, the developers might have played around with the provider or the repository is a toy-project altogether. Unfortunately, many studies do not differentiate between these cases and treat every project with a configuration file as a valid instance (*e.g.*, [ZSZ+17], [VVSW+14], [GDCZ19]), which might dilute the results. This presents a real threat to the validity of empirical studies on CI. Inspired by Munaiah et al. [MKCN17], who created REAPER[1], a tool that can detect seriously engineered repositories, we aim at creating a tool that can assess the *genuineness* of a pipeline configuration. To this end, we need to identify various *features* that describe the configuration (e.g., length, complexity, or number of configuration changes), build automated extraction facilities, and train a binary classifier (*e.g.*, Decision Tree, Random Forest Classifier, Support Vector Machine). To assess the success of the approach, we will establish a manually investigated *Ground Truth* data set that can be used in an automated evaluation.

[1] https://github.com/RepoReapers/reaper

1.2 Goal

The goal of this thesis is to identify practices or configurations that qualitative Continuous Integration typically express with the intention to generalize them in order to identify software repositories adopting these practices. Such a classifier (similar to REAPER) would either accept a CI configuration file or a software repository, and output a score on the quality of the continuous integration within the software. This will allow future studies to filter out software projects that do not have serious Continuous Integration, picking only those who use Continuous Integration at a high level of quality.

RQ1: Why do repositories contain pseudo configuration files? First, we analyze the reasons why software projects contain pseudo configurations (*i.e.*, in opposite to genuine configurations). A side effect of this investigation is the creation of a labelled dataset of *genuine* and *pesudo* configuration files.

RQ2: Is it possible to automatically detect genuine CI configurations? This RQ provides initial proof that the thesis goal is achievable. We will define an infrastructure that can extract arbitrary CI-related features for existing projects, and use our ground truth data set to train a classifier and assess its accuracy in a 10-fold cross evaluation.

RQ3: What is the predictive value of different feature categories? We intend to extract features on three different levels: *configuration-features* can be extracted from the configuration itself (e.g., how many lines?), *repository-features* can be extracted from the containing repository (e.g., how often has the configuration been changed?), and *CI-features* can be extracted from the CI provider (e.g., how many builds have been performed in this project?). In this RQ, we will analyze whether simple configuration features are sufficient to yield accurate classifications, or whether advanced features should be taken into consideration.

RQ 4: Are *engineered* repositories (*reaper*) also *genuine*? Previous work has identified *engineered* software repositories. This does not automatically imply a *genuine* CI configuration, but there might be a relation to uncover. We are interested in understanding how both populations of software repositories compare and whether it is necessary to have two independent classifiers.

RQ 5: What impact does the classifier have on CI-based studies? The last RQ presents a quantification for the motivating problem of this thesis. We will replicate simple statistics taken from other empirical study and compare the results when created on *genuine* and *pseudo* configurations.

In addition to the experimental results, this thesis will produce a usable tool that allows future studies to filter out software projects that do not have genuine CI configurations so that they can focus on projects with a meaningful CI configuration.

Chapter 2

Overview

In this chapter we give an overview of the structure of the thesis. We first describe the steps necessary in order to answer the research questions, and then briefly introduce the tools we used.

Figure 2.1 shows a systematic overview of the thesis. In particular, the different tools are shown, how these tools are connected to each other and what results they produce. Recall our motivation: We want to understand how to automatically detect genuine CI configurations. To achieve this goal we have to complete several steps.

In a first step, the characteristics of genuine configuration files must be recognized. This characterization is done with the help of various features that can be derived from the configuration file. To access these features efficiently in a later step, a Feature Extractor (2) was implemented.

Next, data basis for the model is needed. We need configuration files for which we can extract the features and classify them. More precisely, software repositories containing CI configuration files are needed for which the features can be extracted and then manually classified. To identify applicable software repositories, we implemented a tool called Project Mining (1), which scrapes GitHub for projects based on the selection criteria.

To conclude the ground truth data set (4), each classifications has to be labeled. To label the configuration files, experienced developers use a classification tool (3) to evaluate whether a configuration file is genuine or not. A three-step evaluation process ensures that the quality of the labels is guaranteed.

Ultimately, with the features and the classifications the ground truth data set can be formed. A random forest classifier (5) is trained on this data set. With the help of various examinations, the performance of the model is evaluated. The focus lies on the behavior of the variables, the influence of programming languages as well as the possible limitations. In a further evaluation it will be examined whether there are similarities or dependencies to the similar tool Reaper (6). For this purpose, a larger data set (7) is compiled, which is labeled by model. Finally, the impact of the model on CI-based studies is evaluated by replicating a study performed by Widder et al. [WVHK18] about the reasons why software repositories leave Travis-CI using their replication package (8).

Finally, the discussion summarizes the knowledge gained, puts it in relation to the assumptions made, and derives corresponding findings. In doing so, we reflect critically and give indications for future work.

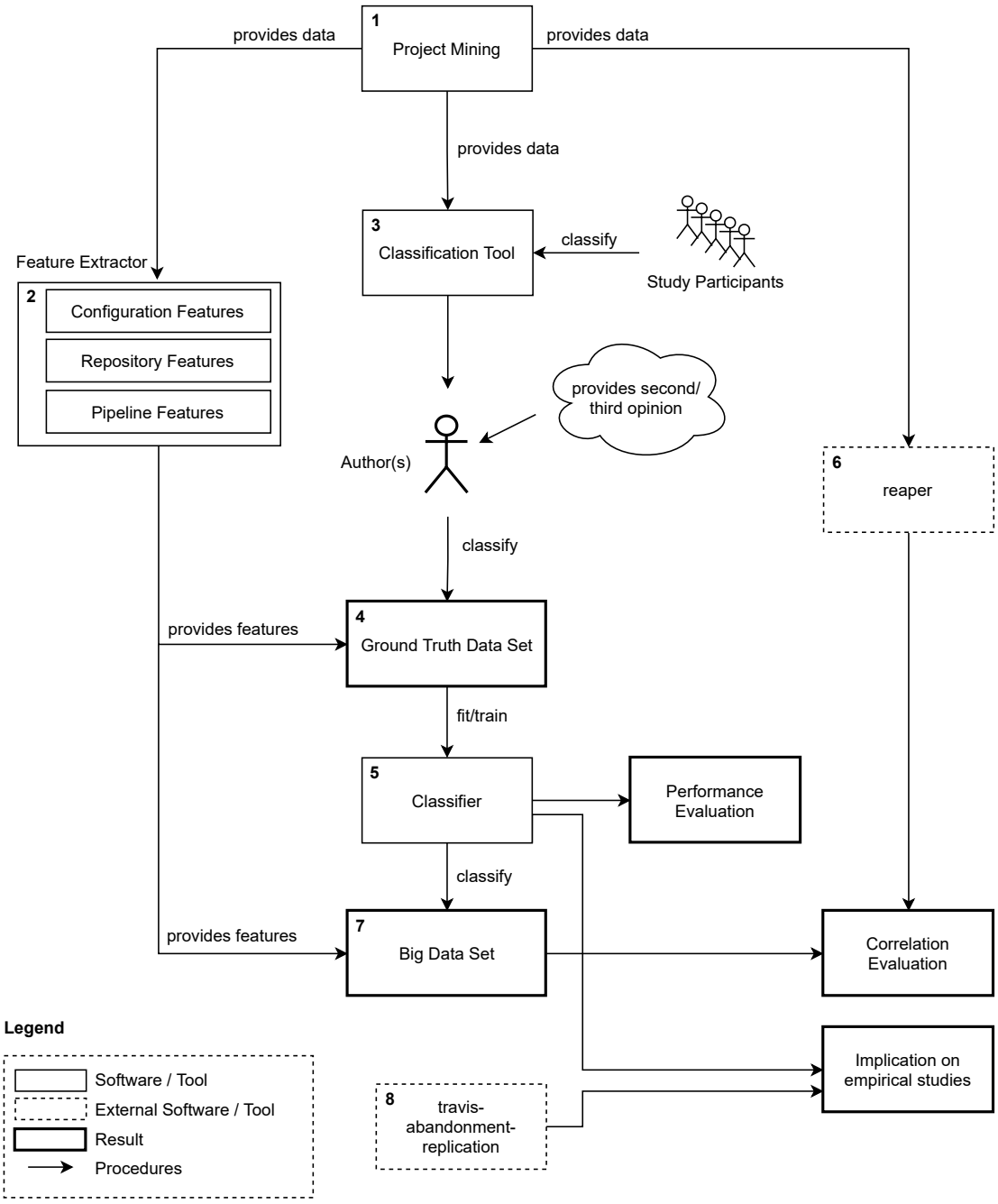

Figure 2.1: Schematic overview of the thesis

The individual tools are presented below:

Project Mining The Project Mining is responsible for identifying the appropriate projects for the data sets. Various criteria are used to search for optimal projects on GitHub. It is implemented with Python and available here: https://github.com/cinder-config/data-mining

Feature Extractor The Feature Extractor receives as an input a configuration file or a software repository that contains a configuration file. Based on the input, all possible features are extracted. The Feature Extract distinguishes between three different feature types: Configuration, Repository and Pipeline. The Feature Extractor is Java based and exposes a JSON endpoint with which the user can interact. Source-Code: https://github.com/cinder-config/extractor

Classification Tool With the help of the Classification Tool, the participants of the study receive a clear GUI with which they can decide for a configuration file whether it is genuine or not. The classification tool is a Vue.JS application with a PHP Symfony backend. The data is stored using a MySQL database. Frontend: https://github.com/cinder-config/classification-frontend, Backend: https://github.com/cinder-config/classification-backend

Classifier The classifier is the heart of the work. Based on the labeled data (ground truth data set) a random forest classifier is trained that can decide for a given configuration file whether it is genuine or not. The machine learning framework scikit-learn is used for this purpose. The model and the evaluations are available here: https://github.com/cinder-config/classifier.

Chapter 3

Related work

In this chapter, we present prior approaches to the problem of selecting appropriate software repositories in the field of empirical software engineering research and identify approaches that we can reuse. We will then discuss similar work around the CI/CD process, and finally we will dive into the topic of Continuous Integration anti-patterns and smells.

3.1 Filtering data

Mining software repositories is a branch of empirical software engineering research aiming at discovering interesting facts and information about the contents of software repositories. It is often described as a process to *"obtain lots of initial evidence"* [HZ10] by exploring software repositories and its contents. With the rise of open access to software systems - GitHub noted a growth of 60 million new repositories in 2020 [Git21] - endless possibilities have opened up for researchers. But this variety unfortunately also has its price. Through quantitative and qualitative analysis of GitHub, Kalliamvakou et al. identified the dangers posed by this great diversity. They identified that a repository is not necessarily a project, many repositories are personal and that most project contains only few commits and are inactive [KGB+14].

Therefore, researchers have often developed their own filtering criteria to reduce the large amounts of data into manageable sizes while maintaining a certain quality. For example in the code quality analysis of different programming languages, Ray et al. [RPFD14] made their selection by picking the 50 most popular repositories. Bissyandée et al. studied the popularity, interoperability and impact of different programming languages. For their quantitative analysis, they selected the first 100'000 repositories which were available through the GitHub API [BTL+13].

The need for meaningful filtering options to produce reproducible data set is highlighted in a quantitative study by Robes [Rob10] analyzing the reproducibility of 171 MSR[1] papers. He discovered that while the majority of the paper uses publicly available data sources, both data set and tools are often unavailable, although indicated otherwise by the authors. He concludes that many papers are not replication friendly. This behavior is also supported by the study of Falessi et al. [FSS17]. The analysis of 68 different studies revealed that the selection of projects was not fully reproducible. To overcome this issue, they presented STRESS, a semi-automated approach to project selection by generating a reproducible data set.

A similar approach is followed by Munaiah et al. They solve the project selection problem by proposing an innovative framework which reduces noise in the data set by efficiently identify engineered software repositories [MKCN17]. According to their definition, an *"engineered software*

[1] International Workshop on Mining Software Repositories

project is a software project that leverages sound software engineering practices in one or more of its dimensions such as documentation, testing, and project management" [MKCN17]. They identified a set of metrics which describe the seriousness of a software repository. Using a manually curated set of labeled repositories - they trained a classifier which both demonstrated high precision while maintaining high recall.

Our work is highly inspired by the work of Munaiah. Using a binary classifier, we aim at classifying Continuous Integration configuration files, and ultimately providing a filtering framework to enhance the quality of data sets for empirical research in the field of Continuous Integration.

3.2 Configuration smells

In the early days of Continuous Integration, developers used various scripts and code snippets to build, deploy and distribute software. Build notifications, error handling as well as build optimizations often had to be developed by the organization itself. With the rise of cloud-based providers such as Travis-CI, the configuration file became the unique entry point for the developers to configure the pipeline service. Because the configuration file is part of the software repository, and the build process tends to evolve [MAH12], it is important for the configuration file to be maintained as well. But unfortunately Gallaba et al. detected through empirical analysis that *"most CI configuration files, once committed, rarely change"* [GM18].

Through the configuration file, the developer can make use of the different features a pipeline provider offers, thus customizing the build process to the repositories and organizations needs. Ultimately, the software benefits from the vast majority of benefits CI/CD offers, such as increased developer productivity [VYW+15], faster and more frequent releases [HTH+16] and detecting problems earlier [SB13].

But where there is usage, there is also misuse. Researchers already identified various anti-patterns, for example overzealous and slow builds [Duv10], ignoring the results of the build process or poor usage of branches [ZVP+20]. If not addressed in a timely manner, these pose a substantial threat to the maintainability of the software.

In order to identify and mitigate anti-patterns, Vassallo et al. introduced CI-ODOR [VPGDP19]. It automatically detect four anti-patterns: slow builds, skipping failed tests, broken release branches and late merging.

Similar work has focused not only on the pipeline, but on the configuration file itself. To automatically detect smells on the configuration file, Gallaba et al. introduce HANSEL, a framework to detect four common smells: redirecting scripts into interpreters, bypassing security checks, irrelevant properties and using commands in the wrong phase [GM18]. In addition, they presented GRETEL, a tool to automatically eliminate those smells in Travis-CI configuration files.

A similar approach is followed by Vassallo et al. with their linter named CD-LINTER [VPJ+20]. It can detect four important smells during the pipeline process: fuzzy versions (failing to specify the exact version), fake success (job failure does not affect build failure), manual execution (job needs to be manually executed by a user) and retry failure (rerunning a failed job until success).

Referring to our work, these approaches will significantly influence the characterization of the configuration file.

Chapter 4

Detecting Genuine Configurations

In this chapter, we describe the background information necessary for the classification task. We briefly introduce Continuous Delivery as well as the concept of genuine configuration files. Next, we show what information we want to extract from the configuration files and which insights we believe we gain from them, and finally we present different models of how we can aggregate this information into a binary decision classifier.

4.1 Continuous Delivery

Continuous Delivery is a technique which aims to deliver a new version of software in a fast and reliable way. Humble and Farley more precisely describe it as *"an automated manifestation of your process for getting software from version control into the hands of your users"* [HF10]. Continuous Delivery is tightly coupled with Continuous Integration, which aims at integration changes to the software as often as possible (*i.e.*, multiple times a day). Grady Booch more precisely defined Continuous Integration as follows: *"At regular intervals, the process of Continuous Integration yields executable releases that grows in functionality at every release"* [Boo90]. It is known to be one of the twelve Extreme Programming practices by Kent Beck [Bec00].

An important part of this process is the Pipeline. It is a sequence of steps which usually include the phases *build*, *test*, *deploy* and *release*. These steps are described in a configuration file, the subject of interest to the thesis.

4.2 Configuration file

A Continuous Integration configuration file describes the pipeline process. It contains information about the infrastructure and operating system (*e.g.*, Linux 18.04, MacOS, ...) on which the pipeline runs, which instructions and actions need to be executed and which additional services are required. It may also contain information about the deployment, about caching mechanism and much more.

The configuration file, its features and operations may also vary depending on the selected pipeline provider. Nowadays many different pipeline providers exist, for example GitHub Ac-

tions[1], Bitbucket-Pipelines[2] from Atlassian, TeamCity[3] by JetBrains or in open-source especially known Travis-CI[4]. While each of the providers would like to differentiate themselves from the competition with its functionalities, they share one common functionality. They regularly check the repository for changes and start the pipeline with the underlying configuration file once changes are detected.

```
dist: trusty
sudo: enabled
language: node_js
node_js:                                                    (1)
  - '10'
services:
  - mysql                                                   (2)
  - docker
# Skip npm install
install: true
before_install:
  - mysql -e "SET PASSWORD FOR 'root'@'localhost' =  PASSWORD('');"
  - sudo add-apt-repository ppa:mc3man/trusty-media --yes
  - sudo apt-get update                                     (3)
  - sudo apt-get install ffmpeg gstreamer0.10-ffmpeg
before_script:
  - SAFE_BRANCH_NAME=$(echo -n $TRAVIS_BRANCH | perl -pe's/[^a-zA-Z0-9_.-]/_/g' )
  - REVISION_TAG="$SAFE_BRANCH_NAME-$TRAVIS_COMMIT"
  - if [ "$TRAVIS_TAG" == "" ]; then SYMBOLIC_TAG="$SAFE_BRANCH_NAME-latest"; else SYMBOLIC_TAG="$TRAVIS_TAG";
    fi
  - echo "Branch $SAFE_BRANCH_NAME"
  - echo "Tag $TRAVIS_TAG"
  - if [ "$TRAVIS_PULL_REQUEST" = "false" ]; then echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin; fi
  - if [ "$TRAVIS_TAG" == "" ]; then docker pull "$IMAGE_NAME:$SYMBOLIC_TAG" || true
    ; fi
after_script:
  - docker images
script:
  - docker build --build-arg=TAG=$TRAVIS_TAG --build-arg=REPO=$TRAVIS_REPO_SLUG --build-arg=COMMIT=$TRAVIS_COMMIT
    --build-arg=BRANCH=$TRAVIS_BRANCH --pull --cache-from "$IMAGE_NAME" --tag "$IMAGE_NAME"
    -f docker/Dockerfile .                                                                                    (4)
  - yarn install --frozen-lockfile
  - yarn test
before_deploy:
  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
  - docker tag $IMAGE_NAME "$IMAGE_NAME:$REVISION_TAG"
  - docker tag $IMAGE_NAME "$IMAGE_NAME:$SYMBOLIC_TAG"
  - git diff
deploy:
  provider: script                                          (5)
  on:
    all_branches: true
    condition: ($TRAVIS_TAG != "" || $TRAVIS_BRANCH =~ ^(main|production|stage)$) && ! $TRAVIS_COMMIT_MESSAGE =~ ^Pontoon:*
  script: docker push "$IMAGE_NAME:$REVISION_TAG" && docker push "$IMAGE_NAME:$SYMBOLIC_TAG"
env:
  global:                                                   (6)
    - IMAGE_NAME=itsre/voice-web
    - secure: UQGXi9wTiftApnfryK7fyMliNRZ+UzG93VlehYn3K5Aw7kZpJZEX5xJyz8sN0zMErxcaQC0LZ9CJF/fDCMb/LDW1dxsMin0Y8Sqr3nikpv0qMIE9rjzpwd2VEsfT9ya6ruLtW2fWfkH3FV+6+q
    - secure: wYs0Ay42nlfJSVSgYTztgcIRQc5gl0WxrJDKlxtpEb35zjFUmXcnNfCtSt0hSE2Oi/5R2F0dFedP6/asVgE16KPL6ozx75L3FWY1JMcrJz3jrB1odeij1qt0mpjCY4R7ojUt2TbtMEhI3XGOnw
```

Figure 4.1: Example of a configuration file for Travis-CI taken from the repository *common-voice/common-voice*

To give a deeper understanding, Figure 4.1 illustrates an example of a configuration file for the pipeline provider Travis-CI. The first step is to define which programming language (1) should be supported in the build. Additionally, further necessary services that are needed in the course of the build, (2) are defined. The first actions in the build are performed in the *install* phase, usually responsible for installing the dependencies of the software. In this example, additional commands are executed before this phase (3). The following task is then the actual build itself. Here (4),

[1]https://github.com/features/actions
[2]https://www.atlassian.com/de/software/bitbucket/features/pipelines
[3]https://www.jetbrains.com/teamcity/
[4]https://travis-ci.org/

the software is assembled and tested. Following a successful build, the software is deployed using Docker (5). Since the deployment requires login credentials for Docker, this information is stored using encrypted environment variables (6). The figure obviously shows only a subset of all possible functions. A complete listing of all options can be taken from the Travis-CI User Manual[5]. Because arbitrary code can be executed, the pipeline offers endless possibilities.

4.3 Definition

Our motivation is to be able to automatically identify genuine configuration files. In the following we would like to specify the term *genuine* more precisely. A genuine configuration file describes a pipeline process which compiles, tests or deploys software in which the result of the build process is observed, *e.g.*, a new version is deployed, new artifacts are generated or the result is used in subsequent builds. In contrast to a pseudo configuration, a genuine configuration file demonstrates a serious engineered pipeline. A genuine configuration file is always in step with the repository, it is regularly maintained and adapted to the repositories needs. A genuine configuration file describes several phases, which follow one after the other, but are complete in themselves and does not interfere with each other. Moreover, a genuine configuration file ensures that the developers are informed in case of erroneous behaviour.

For the remainder of this thesis, whenever we refer to a good example of a configuration file, it is labeled as *genuine*. On the contrary, bad examples are labeled as *pseudo*.

4.4 Features

Previously we have introduced the concept of genuine configuration files. Based on this definition, we have identified twenty-two different metrics with which we attempt to characterize a configuration file. Table 4.1 briefly summarizes the features. These features are compiled from three different sources: (1) the configuration file itself, (2) the repository in which the configuration file lies, and (3) the pipeline-provider which parses and interprets the configuration file. Each of theses feature sources provides a different type of access, in which the higher the level, the more difficult it is to extract the features. In the following each feature is defined, described how it is extracted and indicated, if possible, the assumption about the impact of the feature on the model.

4.4.1 Configuration

On the configuration level, the features are extracted directly from the configuration file. Those features are built around the different functionalities which the pipeline-provider (in our case Travis-CI) offers, for example how many different jobs should be processed, on which operating system the job is running or where the software should be deployed to.

To make it easier for us to process the configuration, we use *travis-yml*[6]. With travis-yml configuration files can be parsed, validated and normalized. It reproduces the same steps that are performed on Travis-CI itself. The end product of the process is a job matrix in which each job is described with its specific instructions [CI21].

[5]https://docs.travis-ci.com/
[6]https://github.com/travis-ci/travis-yml

Feature	Description	Level
lines	number of lines of the configuration file	Configuration
jobs	number of pipeline jobs	Configuration
stages	number of pipeline stages in the pipeline	Configuration
env	number of environment variables used	Configuration
comments	number of comments within the file	Configuration
lint	score obtained when linting the configuration file	Configuration
notifications	number of notifications at the end of the pipeline process	Configuration
unique_instructions	number of unique instructions during the pipeline execution	Configuration
use_branches	true if conditional builds are enabled	Configuration
use_cache	true if caching mechanism is enabled	Configuration
use_deploy	true if there is a deployment at the end of the pipeline	Configuration
template_similarity	calculates a similarity score compared to a template	Configuration
config_changes	number of changes of the configuration file	Repository
config_change_contributors	number of contributors who have changed the configuration file	Repository
commits_until_added	number of commits until CI is introduced *i.e.*, configuration file has been added	Repository
days_until_added	number of days until CI is introduced *i.e.*, configuration file has been added	Repository
config_change_frequency	frequency with which the file is changed (config_changes / total_commits)	Repository
success_ratio	number of successful builds in relation to the total builds	Pipeline
build_time	average time of the building process (in sec)	Pipeline
pull_request_ratio	ratio with which builds are triggered by pull requests (opposite to push)	Pipeline
manual_interaction_ratio	ratio with which manual builds are triggered on Travis CI	Pipeline
time_to_fix	average time until a broken build is fixed (in sec)	Pipeline

Table 4.1: List of features

lines

The feature *lines* describes the total number of lines within the configuration file and is obtained by counting the number of lines of the file. This feature gives an indication of how detailed the pipeline is described. We assume that the more detailed a pipeline is, the better it is, and therefore it is consequently more genuine.

jobs

With the feature *jobs* we count the number of jobs a configuration file generates. We receive the number of jobs by expanding the configuration file to a job matrix using *travis-yml*. We assume that the more jobs, the more action is taking place, hence more genuine.

stages

In the pipeline process, a build consist of different stages, each stage grouping similar jobs. For example a build might contain the stages *compile*, *test* and *deploy*. Creating additional stages requires engineering effort, therefore we anticipate that a higher number of stages results in a more genuine configuration.

env

Environment variables are dynamic values that allow the user to customize an application on run-time behaviour. With the feature *env*, we count how many environment variables are defined in the configuration file *i.e.*, how many of them influence the build process. A higher number could indicate a better pipeline process.

comments

Comments are a central part of documenting source code, which consequently also can be applicable to the documentation of configuration files. The feature *comments* counts the number of comments in the configuration file, anticipating that the more documented the configuration file is, the more engineering took place. However, this assumption must be taken with caution, since the comment function can also be used to make parts of the configuration file inactive, *i.e.*, commenting out certain sections.

lint

The functions which can be enabled through the configuration file may change from time to time - adopting new practices and capabilities. *travis-yml* offers a linting service which gives indication about the usage of deprecated features, invalid syntax or even forbidden configuration options. To obtain the lint score, we multiply each message with a severity factor according to table 4.2. For example, a configuration file with one info, two warnings and one error will receive a lint score of 121.

notifications

Continuous Integration usually requires fast feedback. Travis-CI offers various notification-providers with which the developers of the project can be informed about the status of the build process. With this feature, we track how many notifications will be triggered at the end of each build, assuming that a genuine configuration file contains at least some sort of notification system.

Severity	Description	Factor
info	deprecations, missing defaults, skips... - safe to ignore	1
warn	config contains error which were fixed during the parsing - safe to ignore	10
error	config contains error which cannot be fixed during parsing	100
alert	leaking of secrets	100

Table 4.2: travis-yml validation messages

unique_instructions

During the lifecycle of a job, custom commands can be run in several phases, *i.e.*, before, during and after the *install* and *script* phase[7]. The feature counts the number of unique instruction that will be executed during these phases. We assume that a sufficient number of actions will be carried out in a reasonable pipeline. However, it is important to see that this feature has its limitation. If external scripts are called during these phases, these scripts will not be explored. Also, the quality of the actions is not evaluated, for example, a *mvn test* (invoking maven to execute the testsuite) is equally weighted to *pwd* (displaying the current path).

use_branches

Unless otherwise specified, a build is triggered on every push/pull-request on the repository. Travis-CI allows the maintainer to specify certain repository branches to be excluded (blocklist) or disabled the build on all other branches except certain branches (safelist). Use_branches is a bit-flag tracking whether the configuration make use of this feature.

use_cache

Travis-CI gives the maintainer the option to cache content that does not often change, for example dependencies. This is especially useful to speed up the build process. The boolean feature indicates whether caching is enabled or not. We assume that caching is enabled in a genuine pipeline, as this makes the feedback loop faster.

use_deploy

A crucial part at the end of the build process is the optional deploy stage. Travis-CI offers interfaces to easily integrate different hosting providers (*e.g.*, AWS, Azure, Firebase) with minimal configuration. We track whether the configuration has enabled one or multiple deployment providers.

template_similarity

Due to the low barriers to entry offered by CI, quick registration at the pipeline provider and the addition of a simple configuration file, many repositories contain simple and template-like configuration files. With the feature *template_similarity* we want to identify these. For this purpose,

[7] https://docs.travis-ci.com/user/job-lifecycle/

we have identified a possible template for each programming language. By calculating a JSON-Patch [PB13], the difference between the template and the configuration file is measured. The higher the score, the more the configuration file deviates from the template. Example: A value of 0 means that the configuration is identical to the template, while a value of 20 means that 20 different operations (*e.g.*, adding, removing, modifying nodes) has to be performed until the configuration file is reached.

4.4.2 Repository

The repository level describes features which are accessible through the git repository in which the configuration file is placed. These features provide insight into how the configuration files is handled, for example how many times the file has been changed or how many different contributors have worked on the file.

config_changes

Within a git repository, every change to a file is recorded with a commit. Using the git history, we record how often the file has been changed and presume that the more changes, the more engineering resources have been put into the configuration file. But this hypothesis has to be taken with great care, as it may also be the case that the developers struggled with setting up the pipeline and used a lot of different tries.

config_change_contributors

Software is usually developed together. Especially in the open source context it is not uncommon to have many different contributors. With this feature we track how many different contributors have made changes to the configuration file. thus modified the CI process.

commits_until_added

The features *commits_until_added* measures how many commits were present in the repository until the configuration file is added, thus introducing Continuous Integration. Our assumption is that the earlier Travis CI was introduced, the more experience with CI was gained, hence the configuration file is more genuine.

days_until_added

This feature works similar to *commits_until_added* feature. However, it does not measure quantitatively when Travis CI was introduced, but in a temporal relation,*i.e.*, the number of days until the file was added. Our assumption is the same as in the previously referenced feature.

config_change_frequency

While *config_changes* measures the number of changes, the feature *config_change_frequency* additionally includes a temporal component. It measures how often the configuration file has been changed per commit *i.e.*, $ccf = config\_changes/commits$. Our assumption is that a high value indicates that developers place a high value on Continuous Integration, and therefore the underlying configuration file is of interest to us.

4.4.3 Pipeline

Features on the pipeline level are obtained through the API's provided by the pipeline service providers. Since the access to those API's is not always guaranteed, those features are the most difficult to obtain. Features on this level are trying to characterize the quality of the pipeline, for example the build success ratio or the time it takes for developers to resolve a broken build.

success_ratio

At the end of a pipeline build, the build is usually marked either successful, if all steps were successfully passed, or failed if any of the stages fail. The *success_ratio* measures how many builds are successful in relation to the total number of builds. We expect genuine configurations to be more stable, hence producing less build errors and consequently having a higher success ratio.

build_time

Build time is an essential component of Continuous Integration. If the build takes too long, the developer does not get quick feedback on the integration of his work and the advantage of Continuous Integration is lost. Humble et al. propose the build to take no longer than *"making a cup of tea"* [HF10]. An empirical study has shown that proper build configurations should be maintained in order to reduce long build durations [GDCZ19]. We therefore conclude that a short build time may indicate that the underlying configuration file tends to be genuine.

pull_request_ratio

A build is usually started by opening a pull request or pushing a commit to the repository. With the feature *pull_request_ratio* we measure the ratio for which builds are triggered by pull requests. Pull requests provide a great way in modern code review for developers to inspect the code before it is merged into the repository. We assume that a high pull request ratio indicates a good pipeline process.

manual_interaction_ratio

In contrast to the *pull_request_ratio*, the *manual_ interaction_ratio* measures the rate for which builds are triggered by hand. A build usually does not have to be triggered by hand unless something is quite wrong, hence we speculate that a high ratio does not imply genuineness.

time_to_fix

There are many reasons a build can fail, for example an erroneous test, a compilation error, a timeout in the virtual machine or simply a misspelling in the configuration file. An empirical study conducted by Kerzazi et al. recorded a build break ratio of 17.9% [KKA14]. Whenever the build is broken, it is of utmost importance to inspect and resolve the build failure because otherwise the software cannot be delivered anymore. We calculate the average time to fix a build by looping through the build history and whenever we encounter a faulty build, we advance until we find a successful one, and measure the time between these two builds. Our assumption is that a short remediation time indicates a good Continuous Integration adaption, and consequently a genuine configuration file.

4.5 Binary-Classification problem

Recall the initially stated goal: for a given configuration file it should be decided whether it is genuine or not. Consequently, this is a binary classification task. Since the classifier is trained on a labeled data set, a supervised learning model will be used. To solve this problem, many different machine learning algorithms and models exist, such as decision trees, random forests, support-vector machines or neural network models.

Support-vector machines A support-vector machine works by mapping the input-vectors to a very high dimensional feature space [CV95], maximising the gap between two classifications. A well-known area of application is for example handwriting recognition [MBP15]. SVMs work well when the input data is of high dimension, *i.e.*, many features, but it is prone to overfitting [skl21].

Decision Tree learning Using a decision tree, the model is trying to predict the classification through a series of rules. Those rules were initially derived by providing the tree with training data. A big advantage of decision tree learning is that it can be visualized very easily and is therefore easy to follow. Unfortunately, they often lack predictive accuracy and tend to generalize too much [JWHT13].

Random Forest A random forest classifier is an enhanced version of decision tree learning. It works by constructing and bagging multiple decision trees to a single model. An important feature of random forest classifiers is that during the learning process whenever there is a split, a random subset of features is chosen. In contrast to Decision tree classifiers, a random forest classifier significantly reduces the variance and the risk of overfitting [JWHT13]. In addition, random forests provide a great framework in analyzing feature importance, *i.e.*, the impact of different features on the model.

In an empirical comparison of different supervised learning algorithms, Caruana and Niculescu-Mizil explored that random forest and SVMs are among the top performers, outperforming especially single trees [CNM06]. With this background knowledge and the fact that the feature space is rather small, we decided to use a random forest classifier. For the remainder of this thesis, whenever we refer to the model, a random forest classifier is meant.

Chapter 5

Methodology

The foundation of the model is a labeled data set of truth with a set of features of how this truth is gained. In the following chapter, we describe the methodology we used to compile the data set for the classifier. We describe how the data is obtained, how the labels are set and how we guarantee the quality of the labels.

5.1 Data mining

In order to build the model, we need to have a data set which acts as the source of truth for the model. Since the classifier is a novel approach and no such similar work has been curated, the following steps were necessary to collect the projects for the data set.

To accurately represent the current state of software engineering, the data set shall be as diverse as possible. Consequently, a mix of various projects, such as large-scale projects, open-source projects, research projects, plugins, as well as student/homework-projects should be represented in the data set. In addition, the data set should span across several programming languages, covering the most used ones. Also, a project should have at least some activity *i.e.*, commits. This way project that have no activity and were only created for testing purposed are eliminated. As a final and most important criteria, a project must contain a valid *.travis.yml* file and must be registered on Travis-CI.

According to the 2020 State of the Octoverse [Git21], conducted by GitHub, we selected the six most popular programming languages which are supported by Travis-CI: JavaScript (TypeScript), PHP, Ruby, C++, Java and Python.

Since the GitHub API has a lot of restrictions and provides only little possibility to discover projects, researchers introduced GHTorrent [Gou13], a scalable and queriable offline mirror of the data that is offered through the GitHub API. The most recently created mirror contains over 125 million software repositories with a compressed data volume of around 100 GB.

To ensure the previously mentioned diversity of the projects, we categorize each project based on various metrics. Table 5.1 describes the categories and their selection criteria. In order to obtain a balanced data set with respect to the classifications (genuine/pseudo), we assign a weight to each category. Our assumption is that medium, large and popular projects are more focused on adapting Continuous Integration and hence may contain more genuine configuration files. As a result, we weight these three categories equally with the small category.

Label	Selection Criteria	Desired weight
Big	commits >= 5000 and pullrequests >= 500	13.33%
Medium	commits >= 1000 and commits < 5000	13.33%
Small	commits >= 20 and commits < 1000	50.00%
Popular	watchers >= 100 and contributors >= 20 and issues >= 200	13.33%
		100.00%

Table 5.1: Project categorization

5.2 Data labeling

Once a large enough data set is compiled, for each element in the data set has to be decided whether the configuration file it contains is genuine or not. To guarantee label quality and ultimately reduce bias, this process is outsourced to experienced and talented developers. To achieve this, a classification tool was developed.

5.2.1 Classification Tool

The tool to collect the classifications is divided into two sections. In the first step, simple demographics are collected and in the second step, the classification takes place. The demographics enables in a later processing step to remove classifications that do not meet the quality standards. The following information is collected from the participants:

- Years of professional experience in a software development context that uses CI
- Self-assessment of skills and knowledge about CI (score from one to nine)
- Ever used Travis-CI (True/False)
- Consent for data processing

During the classification task, the participant has to give an indication whether the given configuration file is genuine or not. To support the decision of the participant, he is presented with some information about the repository in which the configuration file can be found. Additionally he is presented with the configuration file itself as well as external links to the repository as well as the Travis CI page.

Finally, the participant is confronted with some questions. The first four questions are aimed at the participant forming an opinion about the configuration file, while the last question finally aims at the classification itself. The questions are always answered with yes or no, the participant does not have the option to skip a question. This setting has been chosen deliberately in order to ensure that the participant is forced to form a final opinion. We believe that the above questions provide a reasonable basis to form a final picture.

Seriousness Do you think the linked repository contains a serious software project (in contrast to, for example, study projects or examples)?

Integrated Do you get the impression that TravisCI is an important part of the project's development process?

Tailored Does the TravisCI configuration look customized to the project's needs, i.e., is it more than a template or a stub?

Interesting Is this TravisCI configuration interesting for you or could it be interesting for others, for example, to learn something new?

Genuine Would you say that this TravisCI configuration is a good example of a genuine CI pipeline?

Figure 5.1 shows a screenshot from the classification tool, displaying the project under review, its configuration file and the final questions. Each participant is asked to classify 10 projects, with the option to provide more classifications if he is willing to do so.

5.2.2 Labeling review

After receiving the classification of the participants, each classification is subjected to a second opinion by the author. If the opinions agree, the classification is included in the data set. If the opinions diverge, a third opinion is obtained from the thesis supervisor, who therefore finalizes the classification. This two(-three) step approach ensures that for each classification there are at least two opinions, which may be confirmed by a third. This is necessary as only labels of high quality should be included in the data set.

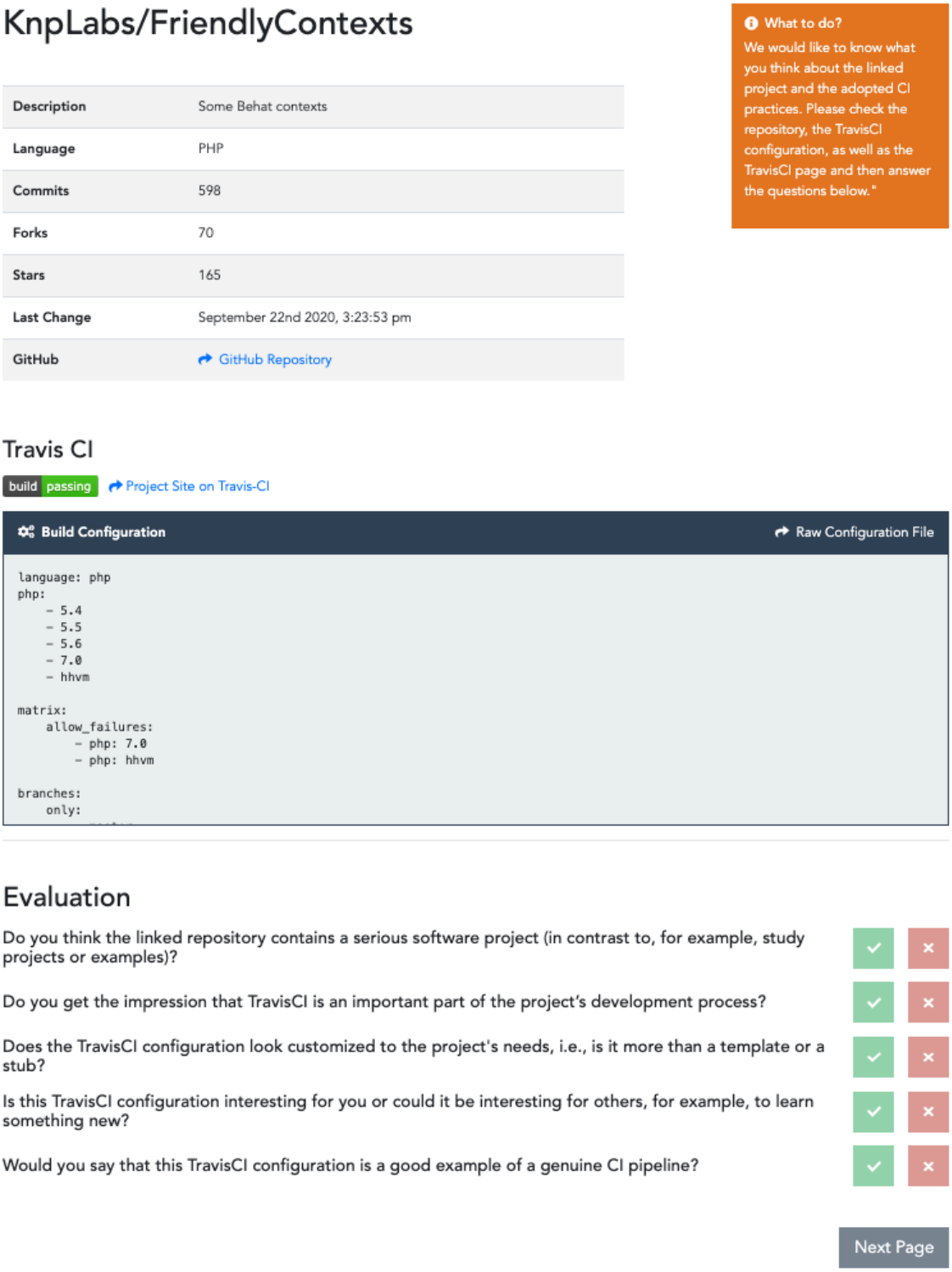

Figure 5.1: An example from the classification tool

Chapter 6

Results

This chapter presents the results of this thesis. First, the results of the classification task and the resulting ground truth data set is presented. Subsequently, the performance of the classifier, trained on the data set, is evaluated. We also investigate the influence of the variables, the categories as well as the programming languages. After making an additional comparison with engineered repositories, the impact on CI-based studies is investigated.

6.1 Ground truth data set

6.1.1 Classification task

The call-to-action to build the ground truth data set has been followed by 68 participants, submitting a total of 338 classifications. Since the goal is to curate a data set of high quality, all classifications from participants who did not submit at least 5 classifications, effectively 40 participants, were eliminated. Furthermore, only classifications from participants who have used Travis-CI in a professional context or demonstrated a sufficient knowledge of the domain (Criteria: $years\_experience >= 4$ and $relative\_experience\_score >= 7$) were included. This reduced the total amount of classifications to 151, respectively 12 participants. A participant contributed on average 12 classifications, while taking around 61 seconds for each classification task. Additional demographics from the classification task are visible in Table 6.1.

avg experience	4.583 years
avg relative experience score	6.583
used travis	83.33%
classifications per user	12.583
avg time spent per classification	60.85s
dropout rate	58.82%
discard ratio	44.67%

Table 6.1: Demographics participants

Based on the proposed methodology (Chapter 5.2.2), we subjected each classification to a second and optionally a third opinion, furthermore ensuring the quality of the labeled data set. Finally, the resulting data set consists of 151 projects. Table 6.2 provides detailed information about the distribution of the labels within the programming languages. From this it is observed that genuine as well as pseudo classifications are evenly distributed.

	Ruby	JavaScript	Python	Java	C++	PHP	Total
Big	3/1	1/0	5/1	5/1	4/0	3/0	21/3
Medium	2/1	4/3	2/1	0/4	3/1	4/0	15/10
Small	6/12	1/10	9/8	1/7	8/6	6/7	31/50
Popular	1/4	1/5	0/1	2/2	1/0	3/1	8/13
Total	12/18	7/18	16/11	8/14	16/7	16/8	75/76

Table 6.2: Classification matrix per language and bucket. genuine/pseudo classification. Example: For medium JavaScript projects there are 4 genuine and 3 pseudo classifications

6.1.2 Characteristics

For each configuration file in the data set, the 22 features (Chapter 4.4) were extracted using the Feature Extractor. Together with the classifications, they form the ground truth data set.

Figure 6.1 shows the distribution of each variable within the data set, split by genuine and pseudo. From this visualization, certain trends can be identified. For example, genuine configuration files have more environment variables defined. They also have more jobs on average and use more unique_instructions.

On the other hand, both genuine and pseudo configuration files sparsely use the notifications feature. Also, very few configurations include a deployment section.

Also not surprising is that the majority of configuration files classified as genuine are on average much larger (more commits) and much better known (more stars) than projects with non-genuine configuration files. This is also in line with the assumption we made when collecting the projects.

To better understand the dependencies of the variables, Figure 6.2 visualizes the correlation of the different features. The graph shows how the values for the features relate to each other, the darker (*i.e.*, blue = positively correlated, red = negatively correlated) the more strongly the variables are linked to each other.

At first glance, it is noticeable that most features within the same category correlate with themselves. We attribute this to the fact that they are all extracted from the same source.

It is also noticeable that the feature *lines* correlates with many features. This is to no surprise, as one can assume that the larger a configuration file is, the more it will specify, so for example more jobs or more actions.

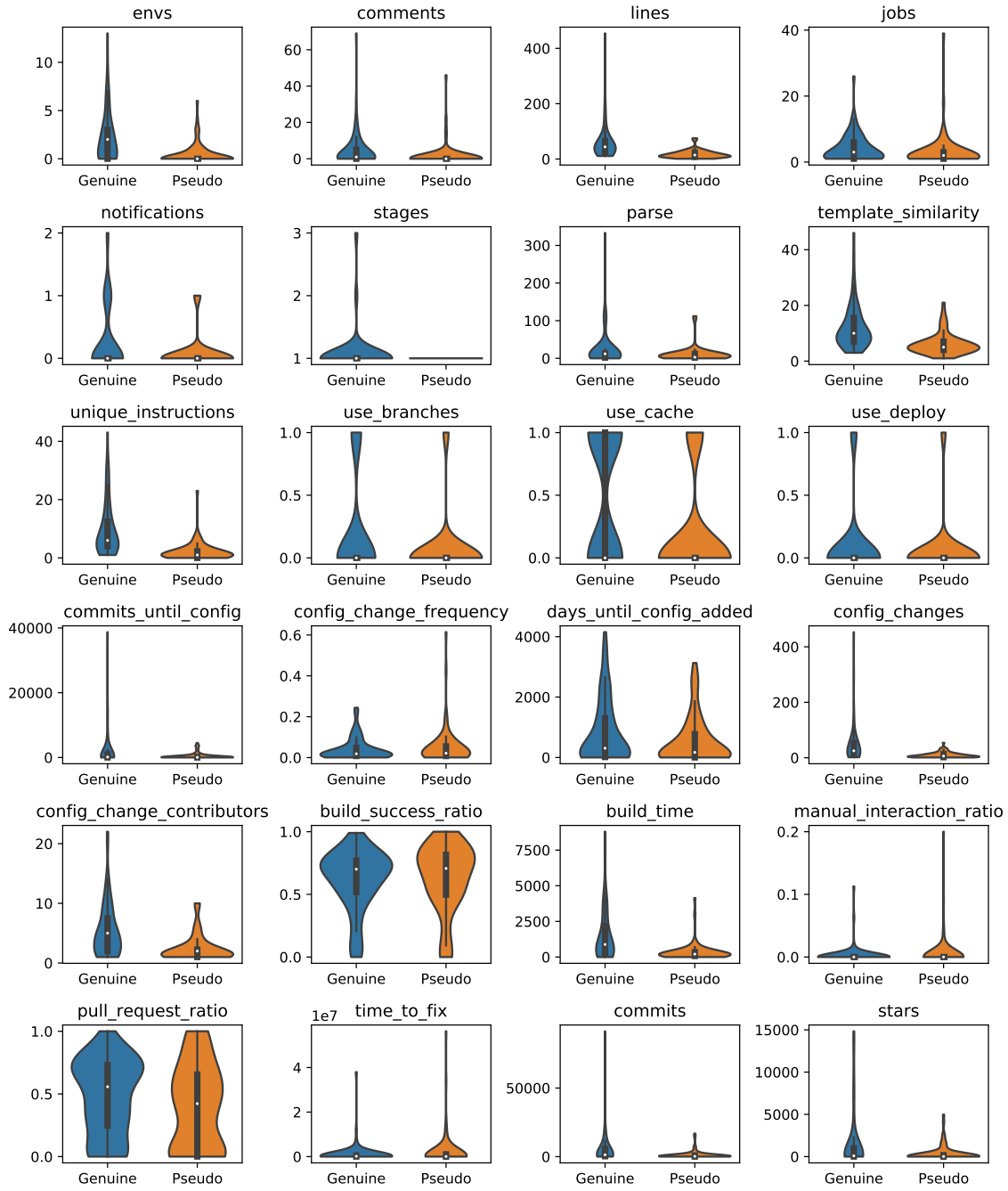

Figure 6.1: Feature distribution within the ground truth data set

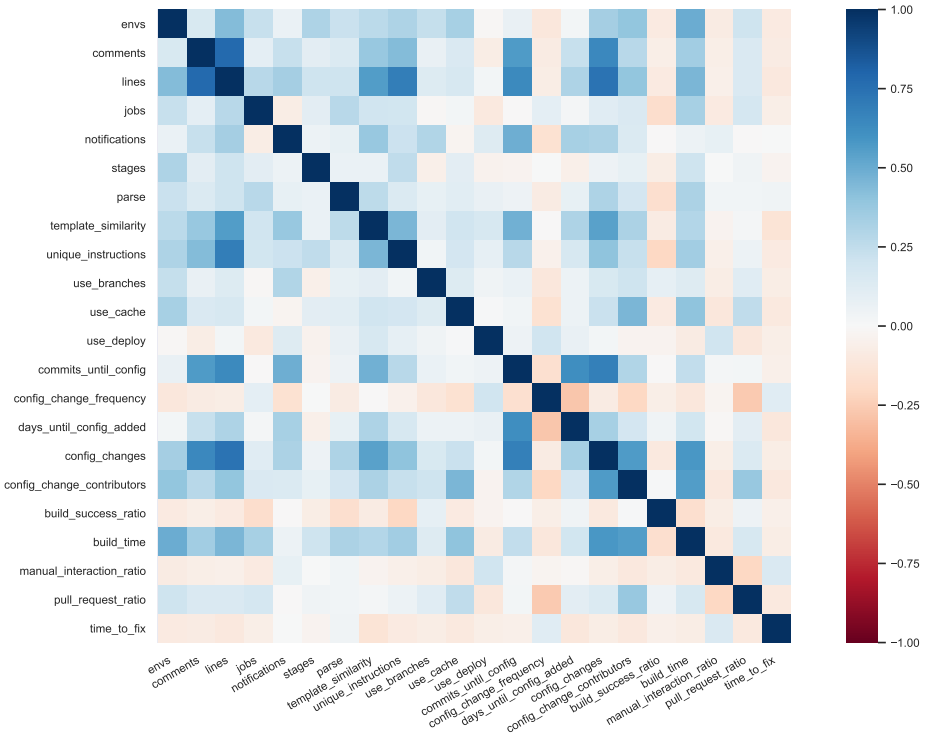

Figure 6.2: Feature correlation within the ground truth data set

6.1.3 Pseudo-Configurations

Based on the ground truth data set and the knowledge gained from it, we address the first research question: why do repositories contain pseudo configurations? To answer this, we inspect configuration files labeled as pseudo within the data set. In particular repositories that have made few changes to the configuration. In these, the configuration file as well as the commit messages are inspected in detail.

Two reasons why repositories contain pseudo configurations can be identified. The first reason is that the repositories want to use the main function of the pipeline, the build process. Many repositories that works with template-like configurations were identified. This is also confirmed by the distribution of the feature *template_similarity* (Figure 6.1). We suspect that they simply want to be informed about the state of their software, *i.e.*, whether it is still executable after a newly introduced change.

On the other hand, we see that many projects have a badge in their *README.md*[1] file displaying the current status of the build process, *i.e.*, if the latest build was successful. This behavior is very common in the open source community. It can be observed that the badge is often added at the same time or directly after introducing Travis-CI. We deduce that the maintainers of the project wants to demonstrate quality with the badge, implying that a project that has a working pipeline (even if it is not very engineered) is functional and might be of interest to others. There is nothing more frustrating than using software that does not work at first attempt.

[1]The README.md file usually contains textual description about the repository and is displayed on the front page when browsing the repository on GitHub.

6.2 Detecting genuine CI configurations

A central question of our work (*i.e.*, RQ2) is whether it is achievable to detect genuine configuration files, and if so, how precise can they be detected. Using the scikit-learn package [PVG+11] and the ground truth data set, we train the model. In the following, the model is subjected to various analyses. We evaluate precision and recall, the importance of the variables, the influence of feature category as well as programming language and finally present the limitations of the model.

6.2.1 Performance

Using a repeated 10-fold cross validation, the model achieves an accuracy score of 80.00% ($std = 10.33\%$). Table 6.3 presents the summarized performance metrics of the classifier. A more detailed overview is provided in Figure 6.3 and Figure 6.4.

The precision-recall curve shows the trade off between true positive rate (precision) and the positive predictive value (recall) given different probability thresholds. The graph shows that up to a certain threshold (50% recall) the precision is very good and stable. With increasing recall, unfortunately, the precision decreases sharply. On the other hand, the ROC (receiver operating characteristic) shows the correct predictions versus the incorrect ones given different probability thresholds. The area under the curve (AUC) hereby gives an indication of the skill of the classifier - the larger the more skill he demonstrates. It can be seen that the classifier can work very precisely up to a certain point. However, with the increasing number of hits, the misclassifications also strongly increase.

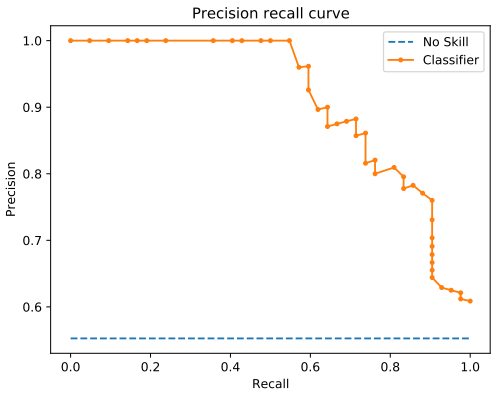

Figure 6.3: Precision recall curve

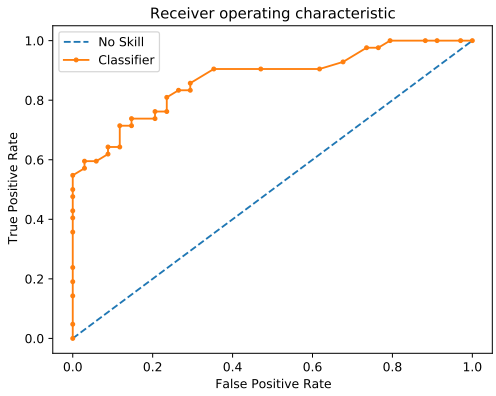

Figure 6.4: Receiver operating characteristics

The evaluation shows a good precision while also maintaining an acceptable rate of recall, demonstrating an overall good and acceptable performance of the model. What is particularly interesting is the difference between the recall of genuine and pseudo classifications. While the model may miss some genuine classifications, it shows strong performance in classifying pseudo configuration files.

	precision	recall	f1-score
genuine	0.815	0.789	0.795
pseudo	0.769	0.806	0.782
avg	0.792	0.798	
accuracy			0.800
std			+/- 0.103

Table 6.3: Classification report

Unfortunately, the evaluations also show that there are high uncertainties in the model. This is shown by the high standard deviation. This can be inferred from the fact that with 151 classifications the ground truth data set is rather limited. Depending on how the training set and test set is split, the performance may increase or worse. The question arises to what extent the extension of the data set affects performance and deviation.

To investigate the impact of an increased size of the data set on the model, we train the model with different sizes of the data set. Figure 6.5 shows the results of this analysis. While the accuracy does not change significantly, we observe that as the number of projects in the data set increases, the standard deviation is significantly reduced. From this we conclude that with a larger data set more reliable results can be obtained.

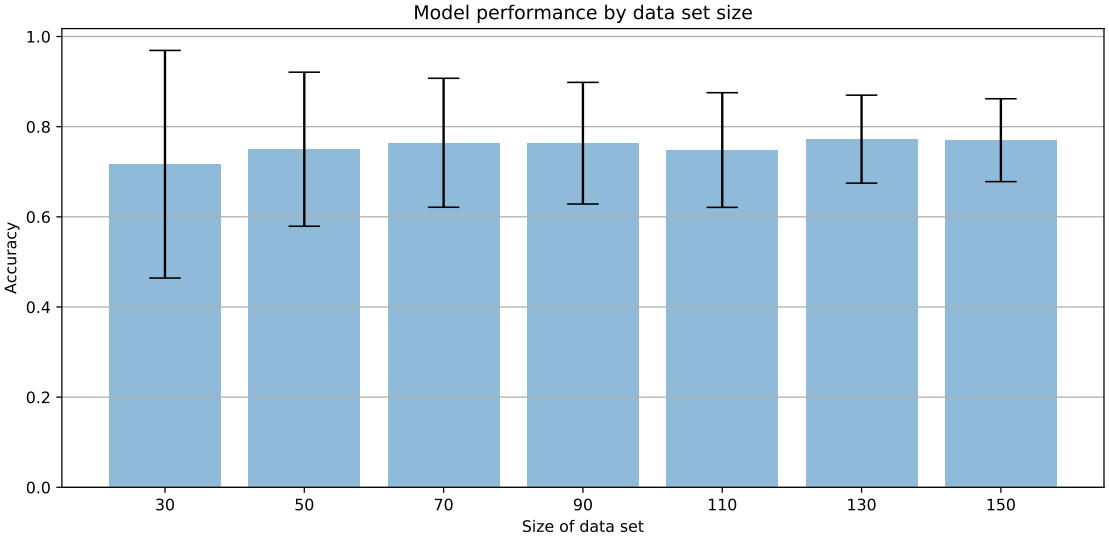

Figure 6.5: Model performance by using different feature sizes

6.2.2 Variable importance

In order to get more insights into the mechanics of the model, we perform a variable importance analysis. SHAP (SHapley Additive exPlanations) values provide a great framework for interpreting predictions, assigning each feature an importance value for a particular prediction [LL17].

Figure 6.6 presents a SHAP summary plot for each of the features, ordered by their importance. It displays the impact on the model based on their value. A positive SHAP value hints towards a genuine classification, while a negative points towards a pseudo configuration. Example: A high value (red) of unique instructions indicates that the configuration file is likely genuine, while a low value (blue) indicates a pseudo configuration. In addition, Figure 6.7 shows the weight of each variable on the model, as well as the cumulative ratio, *i.e.*, how many variables are required to explain a certain part of the model.

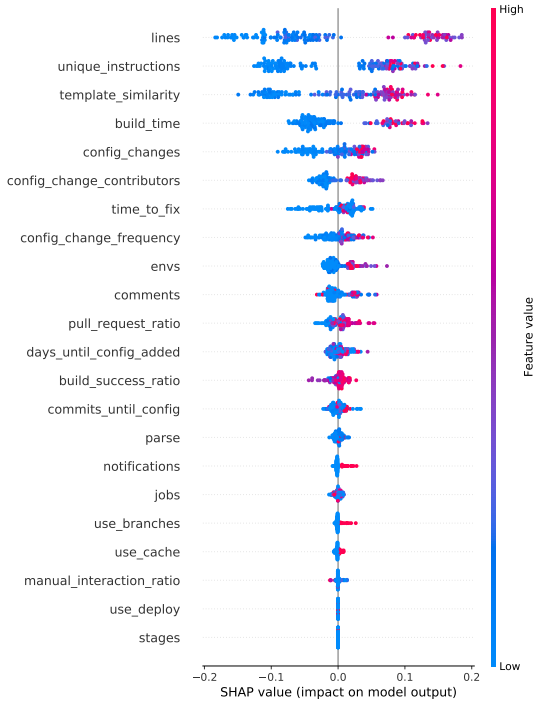

Figure 6.6: SHAP summary plot displaying the impact of each variable on the model

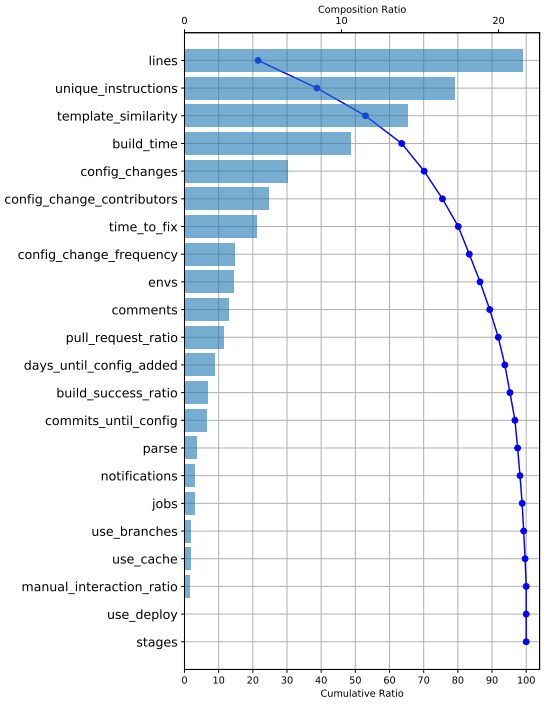

Figure 6.7: Importance of feature on the model including the cumulative ratio.

With only the top five variables, 70% of the models output can already be explained. Additionally, three out of these five can be assigned to the configuration category, clearly indicating a strong dominance of this category. Furthermore, there are many features that, contrary to our intentions, have hardly any influence on the model. These could be easily omitted without substantially affecting the performance of the model. For example it does not matter how many stages a pipeline has, whether it has a deployment mechanism or if notifications are enabled.

6.2.3 Feature category

Each feature can be assigned to one of the three different categories: Configuration, Repository and Pipeline, with the name indicating the origin of the feature. The third research question aims to find out to what extent these categories influence the classifier, in particular whether features directly extracted from the configuration are sufficient to yield accurate classifications or if more advanced features *i.e.*, features extracted from the repository or even the pipeline should be taken into consideration.

We examine this as follows: For each combination of the feature categories a new model with only these features is trained. The evaluation is performed using a repeated 10-fold cross validation. The result of this evaluation is visualized in Figure 6.8.

By comparing the different models, the following becomes apparent. In order to achieve a high accuracy, the features of the level configuration are indispensable, since each model trained with configuration features significantly outperforms the other models. Likewise, the models with additional metrics (repository & pipeline) do not provide a significant accuracy boost compared to the single configuration model. In addition, the variance does not change with additional features. It can be concluded that additional metrics are not necessary and can potentially only dilute the model.

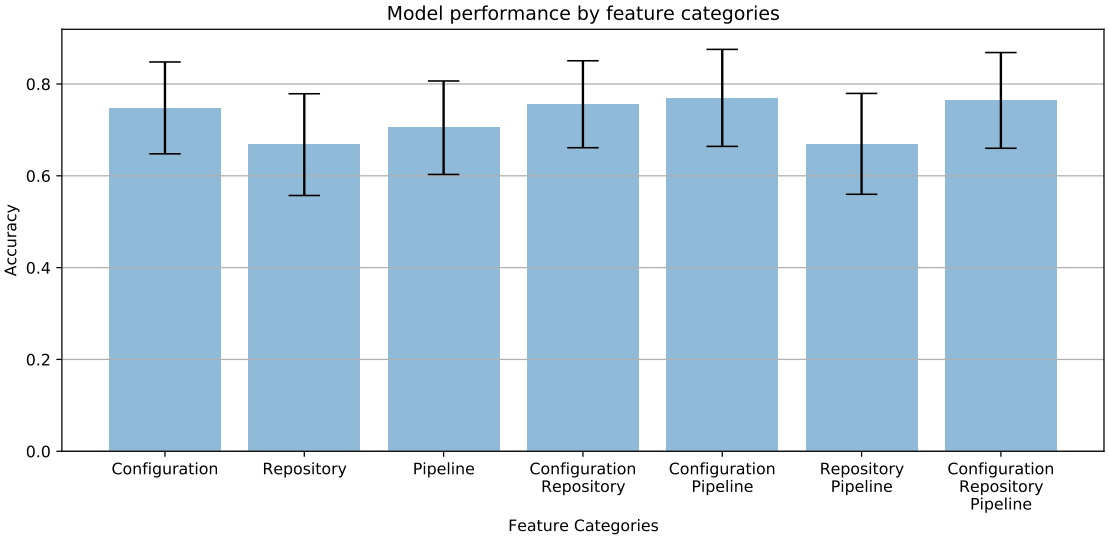

Figure 6.8: Model performance by using different feature categories

6.2.4 Programming languages

In selecting the data, we limited ourselves to 6 different programming languages. The question arises to what extent the programming language, *i.e.*, the underlying programming language of the software which is built by the pipeline, has an influence on the result of the classifier.

We evaluate this as follow. For each programming language, a new classifier with only a subset of the ground truth data set is trained. The subset consists of the original data set minus the data of the corresponding programming language. The programming language then acts as the test set. This will allow us to find out if certain characteristics of a programming language have a significant impact on the classification.

Figure 6.9 presents the finding of this evaluation. It can be seen that all programming languages basically perform quite well and the precision is similar to the previous findings. It is noticeable that Java performs slightly above average, while ruby and PHP slightly perform below average. If we look at the characteristics of the data set, it can bee seen that for Java the classifications are not evenly distributed and pseudo classifications dominate. The previous findings indicated that the classifier is stronger at identifying pseudo configuration, which therefore explains the stronger performance in the evaluation. However, we do not consider this deviation to be significant.

Thus, it can be concluded that the programming language has no significant influence on the classifier and all programming languages can be detected with about the same accuracy.

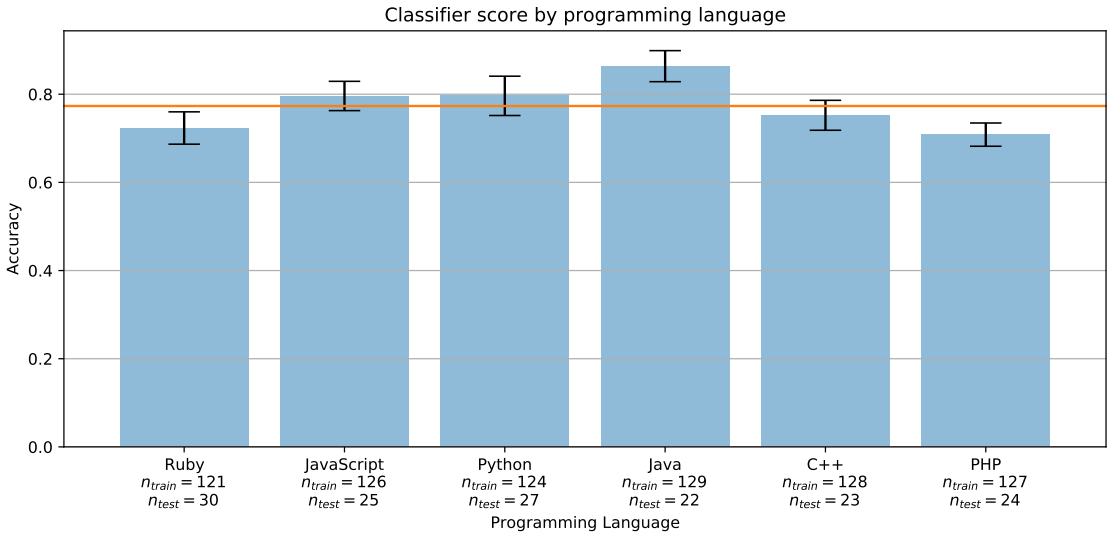

Figure 6.9: Model performance by using different programming languages

6.2.5 Limitations

Although the model demonstrates acceptable accuracy, it does have certain limitations. We show the limitations of the model with two examples, a false negative and a false positive, and explore the reasons for the misclassification.

False Negative

An example of a false negative is the configuration file from the project *integer-net/Anonymizer*[2]. While the manual classification tasks by the participants and the authors labeled the configuration file as genuine, the model indicates otherwise. With the help of the previously introduced SHAP values, we investigate the cause of this misclassification. The decision plot for the classification is visible in Figure 6.11

The decision plot is to be understood as follows: Starting from the bottom, the expected value of the model, the prediction line shows the effect of each variable on the classification. The features are sorted by importance weight, with the most important at the top. The final classification value is visible at the top.

```yaml
language: php
php:
  - 5.6
  - 7.0
  - 7.1
env:
  - MAGENTO_VERSION=magento-mirror-1.9.3.10
  - MAGENTO_VERSION=magento-mirror-1.9.2.2
  - MAGENTO_VERSION=magento-mirror-1.9.1.0
script:
  - composer install --dev --no-interaction
  - phpunit
  - curl -sSL https://raw.githubusercontent.com/schmengler/MageTestStand/master/setup.sh | bash
before_deploy: .travis/bundle.sh
deploy:
  provider: releases
  api_key:
    secure: ...
  file:
    - "integernet-anonymizer.zip"
    - "integernet-anonymizer.tar.gz"
  skip_cleanup: true
  on:
    tags: true
```

Figure 6.10: .travis-yml of the software repository *integer-net/Anonymizer*

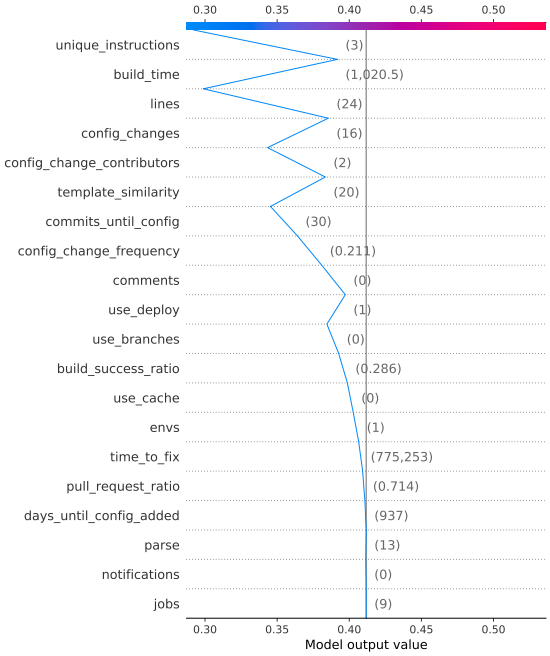

Figure 6.11: Decision plot false negative

The decision plot shows that the high build time (1020s), the frequent changes to the configuration file (16) as well as a high value for the template similarity (inverse property) classify the file as genuine, while the low number of lines (24) as well as few instructions (3) classify the file as pseudo.

A closer look reveals that the configuration file (Figure 6.10) intuitively looks very exciting.

[2]https://github.com/integer-net/Anonymizer

By combining different PHP versions as well as Magento versions, a job matrix with size 9 is formed. Additionally, dependencies are installed and unit tests are run. Furthermore, the versions are deployed for each git tag. However, the configuration lacks action. This action is actually included, but unfortunately it is hidden behind two bash scripts, called in Line 13 and 14 during the *script* respectively the *before_deploy* phase of the pipeline.

The model is not able to identify these additional actions behind the scripts and therefore the configuration file is unfortunately classified as pseudo.

False Positive

We show further limitations of the classifier by means of a false positive. To do this, we inspect the configuration file of project *crate/crate-jdbc*[3] more closely. The configuration file was marked as pseudo by the participants of the study as well as by the authors, the model however evaluates the file as genuine, as visible in the decision plot in Figure 6.13.

The decision of the evaluators can be understood by inspecting the configuration file (Figure 6.12) as follows: The configuration is rather sparse, uses an outdated entry (*sudo: false*), notifications are intentionally disabled, a deployment does not take place and a comment above the install phase indicates a workaround. All in all, a justified decision.

On the other hand, visible in the decision plot, the frequent number of changes (12), the many different contributors (6) and the long build time (1'053 s) contributes to an overall genuine classification, even though only by a small margin ($p_{genuine} = 0.584$). From this, we deduce that the model can be fragile for certain decisions, as already shown in the precision-recall (Figure 6.3) and RUC (Figure 6.4) curves.

```
1  dist: trusty
2  language: java
3  jdk:
4    - oraclejdk8
5  env:
6    - CRATE_VERSION=1.0.1
7    - CRATE_VERSION=2.2.0
8    - CRATE_VERSION=2.3.4
9    - CRATE_VERSION=3.0.7
10   - CRATE_VERSION=3.1.3
11
12
13 cache:
14   directories:
15     - $HOME/.m2
16
17 sudo: false
18
19 # Workaround for https://github.com/gradle/gradle/issues/2421
20 install:
21   - JAVA_HOME=$(jdk_switcher home openjdk8) ./gradlew classes testClasses
22
23 script: ./gradlew test -s
24
25 notifications:
26   email: false
```

Figure 6.12: .travis-yml of the software repository *crate/crate-jdbc*

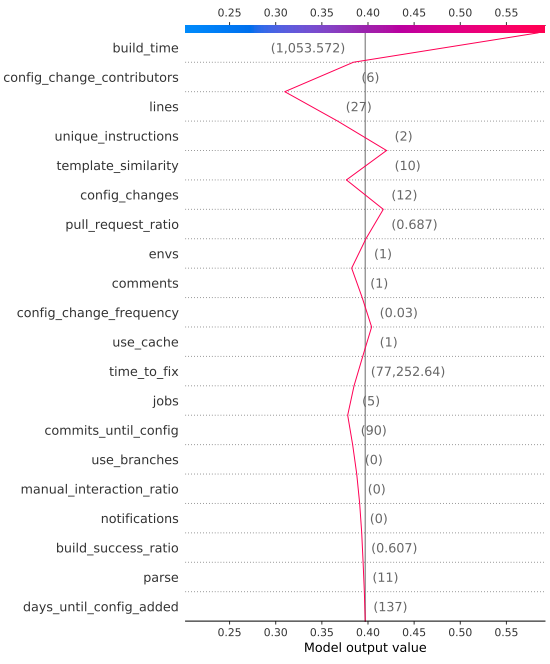

Figure 6.13: Decision plot false positive

[3]https://github.com/crate/crate-jdbc

6.3 Engineered repositories

As described in Chapter 3, Munaiah et al introduces REAPER, a tool that detects engineered repositories. The fourth research question addresses the extent to which there is a relationship between engineered repositories and genuine configuration files. In particular, we are interested in whether one can be inferred from the other, or whether the existence of two classifiers is justified.

6.3.1 Big data set

Since the ground truth data set is too small for this evaluation, a bigger data set with similar projects is compiled. However, this data set is not labeled manually, instead the model will take over this task. To obtain the projects for the larger data set, the same methodology as for the ground truth data set has been used (Chapter 5.1). The larger data set consists of 1'252 projects spanning across several programming languages. Since it is extracted in a similar manner, the large data set has roughly the same characteristics (Chapter 6.1.2) as the ground truth data set in terms of variable distribution. For each project it is evaluated whether it contains a genuine configuration or not using the model. Table 6.4 presents the findings of the evaluation.

	Ruby	JavaScript	Python	Java	C++	PHP	Total
Big	23/3	21/5	24/2	19/5	21/3	24/3	132/21
Medium	18/8	12/12	17/10	12/14	21/3	24/2	104/49
Small	46/101	48/100	88/58	56/92	82/45	109/37	429/433
Popular	14/11	10/11	15/1	5/7	0/2	7/1	51/33
Total	101/123	91/128	144/71	92/118	124/53	164/43	716/536

Table 6.4: Classification matrix per language and bucket. genuine/pseudo classification

6.3.2 Reaper

To identify a potential correlation with REAPER, we evaluate for each project in the large data set whether it is an engineered repository (according to REAPER) and compare the result with the classification of the model. Figure 6.14 visualizes the comparison. Each project is assigned to a quadrant depending on how it is classified by REAPER and the model.

Out of the 1'252 observations, we found a common classification in only 744 (59.42%) of the cases. The comparison shows that there is nearly no correlation between REAPER and CINDER, which is also shown by a very low Pearson correlation coefficient of $\rho(C, R) = 0.15077$. However, it is interesting to see that when the model has identified a configuration file as genuine, 72.77% of the time it is also found in an engineered repository. This result is not surprising, since one can assume that a project that is seriously managed and runs Continuous Integration, also runs it conscientiously with a genuine configuration file. Unfortunately, the opposite cannot be deduced. If a project has been declared as engineered by REAPER, it contains a genuine configuration file only in 62.47% of the cases.

We attribute this to the fact that REAPER only evaluates the existence of Continuous Integration, ignoring the quality of the process itself. In addition, Continuous Integration is weighted very weakly and only contributes at maximum 5% to the decision whether a repository is engineered or not.

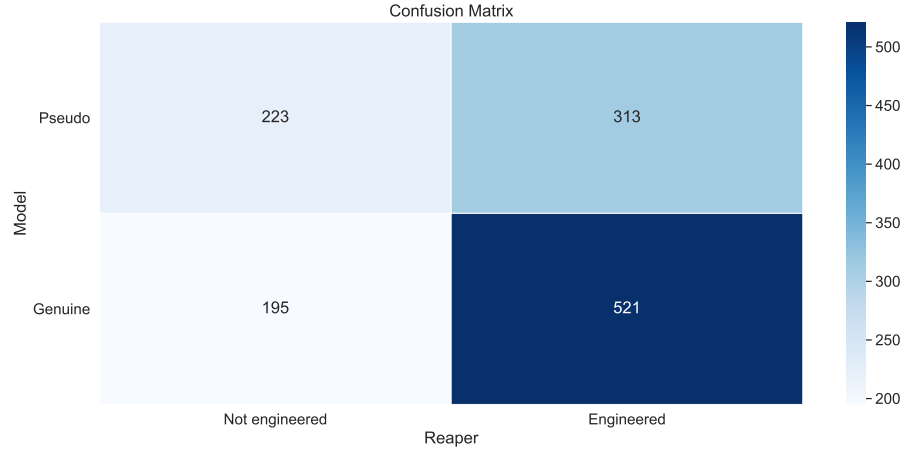

Figure 6.14: Confusion Matrix between the classification of the model and REAPER. $\rho(C, R) = 0.15077$

6.4 Impact on CI-based studies

The final research question (*i.e.*, RQ5) addresses the extent to which the model affects the results of CI-based studies, strictly speaking the validity of such studies. In doing so, we aim to show that clearer and more valid results can be generated through a better and more appropriate selection of projects.

We proceed as follows: In a first step, the study is reproduced on its original data. This ensures that once the reproduction on the cleaned subset is carried out, it is valid. Next, the data set is validated by CINDER and all projects that contains a pseudo configuration file are removed. This is accomplished by extracting the metrics for each project which are subsequently validated by the model. This process results in a new data set which consists only of projects with genuine configuration files. On this data set, the study is reproduced.

In an empirical study Widder et al. have investigated the reasons why open source projects are leaving Travis CI, one of the earliest and most popular CI tool [WVHK18]. They identified that the projects main programming language and the size of the project plays a dominant role when deciding to abandon Travis CI, while an increasing build complexity indicates the opposite. Their study has been conducted by analyzing 7,276 projects that have disabled Travis CI. The selected projects are part of a data set created by Zhao et al [ZSZ+17] in their study to explore the impact of Continuous Integration on the development process. The data set was originally obtained by iterating through GHTORRENT, validating for each project if Travis CI is used, and stopping at approximately half of the projects (165,549) that exist on Travis CI (approx. 318,000 project existed at the time of the gathering according to Travis CI). The replication of the study was performed

using the replication package[4].

When cleaning the data set, no features could be extracted for 1'434 projects (19.70%). On the one hand, certain projects were no longer publicly accessible, on the other hand, the configuration file was no longer part of the project. This is not surprising, since the data set is already around 3 years old. Features were successfully extracted for 5'842 projects. Of these projects, however, only 802 (11.02%) projects were able to provide evidence that they contained a genuine configuration file. The study was then replicated on this adjusted data set. The results are visible in Table 6.5.

	Original Paper		Replication	
Predictor	Coeffs (Errors)	Deviance	Coeffs (Errors)	Deviance
log(Project age)	-0.68 (0.15)***	21.56***	0.48 (0.49)	0.9
log(Build duration)	-0.43 (0.03)***	226.27***	-0.16 (0.10)	2.7
log(Commits)	0.62 (0.04)***	334.69***	0.33 (0.11)**	8.2**
log(Contributors)	0.34 (0.05)***	58.86***	0.60 (0.16)***	14.3***
log(Build jobs)	-0.33 (0.06)***	27.92***	-0.44 (0.16)**	7.0**
log(Pull requests + 0.5)	-0.43 (0.02)***	423.17***	-0.41 (0.06)***	50.2***
log(.yml commits)	0.02 (0.04)	0.30	-0.21 (0.14)	2.3
log(.yml contributors)	0.02 (0.09)	0.07	-0.01 (0.24)	0.0
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$	$n = 7276$		$n = 802$	

Table 6.5: Results of the replication study. The left column are the original results obtained from the paper, the right column describes the results of the replication study.

Unfortunately, when reproducing the study on the original data set, we could not reproduce the results of the analysis of the programming languages (*e.g.*, JavaScript vs Mean) and obtained different values for them compared to those reported in the paper. Consequently, the evaluation refers only to a subset of the study.

From the results one can read the following: In the cleaned data the deviations (columns Deviance) become substantially smaller. We deduce this from the fact that the adjusted data set contains much less noise and the results are more meaningful. Recalling the core statement of the study: dominant factors for abandonment are build duration, the number of commits and pull requests. While the results for commits and pull requests can be validate, the statement in terms of build duration is significantly diminished. We attribute this result to the fact that build duration is also an important factor in the model. We therefore assume that this was taken into account when cleaning the data, and that the distribution of the build duration in the new data set is more homogeneous.

[4]https://github.com/CMUSTRUDEL/travis-abandonment-replication

Chapter 7

Discussion

In this chapter, we will summarize the main results into applicable findings and link them to the assumptions and intuitions we posed at the beginning. From this we finally derive our findings, uncover and discuss potential threats and pitfalls and suggest further work.

7.1 Classification task

Recall the initial problem statement: the detection of genuine configuration files (*i.e.*, RQ1 + RQ2). The classifier has shown to demonstrate good precision in detecting whether a given configuration is genuine or pseudo. Statistical analysis has also shown that, while the classifier may suffer some recall *i.e.*, it may not detect all genuine examples as genuine, it shows strong performance when it comes to detecting pseudo configuration files.

Further analysis has shown that additional features *i.e.*, from the repository level and the pipeline level, do not add additional value to the classifier. This result is very pleasing, as the extraction of these additional features is very time-consuming and error-prone. Thus the range of CINDER is extremely extended. This becomes especially important when the tool is extended to other pipeline providers. Often the access to the repository is limited or the API of the pipeline provider is restricted.

Moreover, we have shown that it does not matter which programming language is the dominant language within the pipeline. From this we can conclude that other programming languages can be included without any loss of precision. Furthermore, this also illustrates that the metrics are not specific towards certain programming languages and thus universally applicable to all programming languages within the Travis-CI universe. This leads us to the first conclusion:

> **Finding 1** CINDER is able to detect genuine configuration files based on simple metrics obtained by the configuration file itself.

Nevertheless, this conclusion has to be handled with great care. We identify the following threats:

Small Sample Set Due to the low number of participants and the strong filtering (to ensure the quality of the data set), the truth data set is rather small ($n = 151$). This is reflected by the high variance of the performance metrics, which naturally weaken the results. By evaluating the influence of different data set sizes on the performance metrics, we have shown that with

increasing the size of the data set, the error can be reduced. Therefore, the risk can be eliminated by expanding the data set in a future extension and providing more truth value to the classifier.

Classification Bias During the classification task, the participants took an average of 61 seconds to make a decision. In that time, they inspected the configuration file, the repository, and the Travis CI pipeline page and have subsequently made the classification. The high dropout rate of participants, as well as the high discard rate of classifications, indicate that determining whether a configuration date is genuine or not is not a trivial task. This struggle with the task becomes clear from the comment of one participant: *"I have a bit of a problem with the questions. [...] I was forced to answer every question, even when I wasn't able to find a good answer. [...] What if substantial parts are offloaded from travis.yml to subsequent bash scripts?*. Ultimately, the classification is a personal assessment in which the experience and preferences of the participant also play an important role. To address and mitigate this threat, we suggest that future research should deeper and more systematic address the notion of genuine configuration files. A systematic taxonomy would be a useful addition for future classification tasks.

7.2 Genuine configuration files

Through the help of 22 different metrics, we try to fathom the essence of a configuration file. With the examination of the influence of the different features on the model (Chapter 6.2.2), we determined that in particular with the features *lines*, *unique_instructions*, *build_time*, *template_similarity* and *config_changes* more than 70% of the model can be explained, showing that they significantly influence the decision whether a configuration is genuine or not. In turn, this means that many of the metrics have little to no impact on the model.

Looking at the meaning and interpretation of the most important metrics, the following pattern can be observed. A high value in the metrics *unique_instructions* as well as *lines* indicate that many actions take place within the pipeline and that the pipeline is specified more precisely. A high value in *build_duration* gives a similar indication. Conversely, a low value could be interpreted as little activity in the pipeline. And finally, a high number of changes to the configuration file indicates that the configuration is regularly maintained and adapted to the needs of the project. We reach the following summarized conclusion:

> **Finding 2** A configuration file demonstrate genuineness if there is sufficient action within the pipeline (*i.e.*, multiple tasks such as compiling, testing, deploying, ...) and if it is regularly adapted to the softwares needs.

This conclusion is also fraught with uncertainty. In addition to the classification bias, we identify another threat: missing potential important features.

Feature-Scope With the extracted features, we made a first attempt to systematically describe a configuration file and measure various features in terms of measurable numbers. However, we by no means consider this compilation to be sound and complete. To improve the model, future research should address the question of the extent to which additional metrics are discoverable when a systematic comparison of the different pipeline providers and their configuration options is created. We suspect that there are many more features to be discovered, especially those that occur within the pipeline process.

7.3 Impact on empirical research

After proving that CINDER can be used to detect genuine configuration files, and thus to qualitatively measure the CI process, we evaluated the implications of CINDER on the process of conducting empirical software engineering research. For this purpose, an existing study was reproduced and the impact of CINDER on the results was measured.

Already when filtering the data set, it became apparent that it contained a lot of noise. For only a small portion of the projects (11%) it could be shown that these meet the quality requirements for a genuine configuration file. This result is remarkable, but not surprising. A simple query on the latest GHTORRENT data set reveals that only 7.75% of projects hosted on GitHub have more than 20 commits. Consequently it is not surprising that this data set, with the absence of meaningful filtering mechanism, also contains a lot of noise.

While some of the results could be reproduced, a subset of the results could not be reproduced. Despite this unpleasant result, it was shown that in particular the deviations were significantly reduced. We conclude that this is due to the fact that the filtered data set contains a higher quality. This brings us to the last finding:

Finding 3 CINDER provides researchers with a useful tool to validate the quality of the research data set, ensuring to only include projects that practice Continuous Integration at a high level of quality.

Limited Expressiveness This result is obtained from the reproduction of a single study. To what extent these findings can be applied to further empirical studies concerning CI/CD cannot be conclusively stated, a possible error range must not be neglected. In our view, a more systematic and in-depth analysis of several other studies is necessary to reinforce the finding.

Chapter 8

Summary

This thesis presents a novel approach for project selection for empirical software engineering research in the context of Continuous Integration.

Inspired by REAPER, a classifier that detects engineered software repositories, we have presented CINDER, a tool to automatically detect genuine Continuous Integration configuration files. CINDER uses as input a configuration file or a software project in which the configuration file is located and outputs whether the given configuration file is genuine or not.

The basis for the tool is a random forest classifier, which is based on a ground truth data set. The ground truth data set consists of 22 different metrics that describe the characteristics of a configuration file, and a truth label. The truth label was assigned through a triple review process by experienced developers and the authors.

We have subjected the model to various evaluations. On the one hand, we have shown that the model performs acceptably in terms of precision and recall, both in the detection of genuine configuration files and in the detection of pseudo ones. On the other hand, we have also shown that there are certain limitations that need to be addressed.

Furthermore, we have investigated which features have the greatest impact on the model. It turned out that only the features of the configuration category are necessary and that the other feature categories do not provide any additional value. Our analysis has also shown that the choice of programming language does not play a role. Consequently, significant action within the pipeline and its regular adaptation is a strong indicator of the genuineness of a configuration file.

To check if there is a dependency between engineered repositories and genuine configuration files, we examined the correlation of *CInder* with *Reaper*. We found that when a genuine configuration file is detected, in most of the cases it is located in an engineered repository. By contrast it cannot be deduced that an engineered repository necessary contains a genuine configuration file.

To build the bridge with empirical software engineering research, we reproduced a study by Widder et al. on the reasons why repositories leave Travis CI. In the replica study, however, we only took projects with genuine configuration files. During filtering many projects had to be removed. The change in the data set affected the results of the study and consequently compromised the validity of the study. Thus CINDER provides researchers with a useful tool to validate the quality of the research data set, ensuring to only include projects that practice Continuous Integration at a high level of quality.

We have shown in this thesis that the categorization of configuration files and consequently of pipelines is an interesting and important topic in empirical software engineering research. With CINDER we provide researchers with a crucial tool to obtain perfect project matches for their next CI based study.

Bibliography

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[Boo90] Grady Booch. *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., 1990.

[BTL+13] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE, 2013.

[CI21] Travis CI. travis-yml readme. https://github.com/travis-ci/travis-yml, 2021. Accessed: 2021-02-11.

[CNM06] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[Duv10] Paul M. Duvall. Continuous delivery - patterns and anti-patterns in the software lifecycle. , 2010. Refcard #145, Accessed: 2020-02-05.

[FSS17] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–156. IEEE, 2017.

[GDCZ19] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 2019.

[Git21] GitHub. The 2020 state of the octoverse. https://octoverse.github.com/, 2021. Accessed: 2021-01-27.

[GM18] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50, 2018.

[Gou13] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[HF10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[HTH+16] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[HZ10] Kim Herzig and Andreas Zeller. *Mining Your Own Evidence*, chapter 27. O'Reilly Media, Inc., October 2010.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[KGB+14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[KKA14] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[LL17] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*, 2017.

[MAH12] Shane McIntosh, Bram Adams, and Ahmed E Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4):578–608, 2012.

[MBP15] Durjoy Sen Maitra, Ujjwal Bhattacharya, and Swapan K Parui. Cnn based common approach to handwritten character recognition of multiple scripts. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1021–1025. IEEE, 2015.

[MKCN17] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

[PB13] M. Nottingham P. Bryan. Javascript object notation (json) patch. RFC 6902, Internet Engineering Task Force (IETF), 4 2013.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Rob10] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180. IEEE, 2010.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.

[SB13] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, pages 736–743, 2013.

[skl21] sklearn. 1.4. support vector machines. https://scikit-learn.org/stable/modules/svm.html, 2021. Accessed: 2021-02-05.

[VPGDP19] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 105–115. IEEE, 2019.

[VPJ+20] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 327–337, 2020.

[VVSW+14] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE international conference on software maintenance and evolution*, pages 401–405. IEEE, 2014.

[VYW+15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816, 2015.

[WVHK18] David Widder, Bogdan Vasilescu, Michael Hilton, and Christian Kästner. I'm leaving you, travis: a continuous integration breakup story. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 165–169. IEEE, 2018.

[ZSZ+17] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017.

[ZVP+20] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 2020.