University of Passau
Faculty of Computer Science and Mathematics
Chair of Data Science

Bachelor Thesis in Internet Computing

Classification of Machine Learning Reproducibility Factors

submitted by

Martin Johannes Loos

Examiner: Prof. Dr. Michael Granitzer
Supervisor: Mehdi Ben Amor
Date: May 22, 2022

Contents

Abstract

The possible reproduction of scientific results is of the utmost importance. But there are numerous studies that came to the conclusion that we are in a reproducibility crisis. Nevertheless, we have not found any supporting tool through our research, which recognizes reproducibility problems in the machine learning area on the basis of a repository analysis and offers help to solve them. Hence, it is our goal with this thesis to develop and present a proof of concept tool that offers this functionality.

In order to be able to implement this tool, we first identified factors influencing the reproducibility of machine learning experiments through a literature search. Based on this, we have classified them according to whether these factors have properties that are software-detectable or not. We call these properties indicators. Then, for each identified indicator, the functionality to measure it was implemented in our tool. Using a factor-indicator-connection approach and with the use of threshold values that we have obtained from gathered data sets, a score can be calculated for each factor. The threshold values allow context to be given to a factor score as to whether it is rather good or bad.

Our *hypothesis* was that repositories linked to a machine learning experiment published on academic sites like "PapersWithCode" would, on average, perform better than ones found randomly on GitHub. Hence, we collected and evaluated representative data sets with 100 entities each. Both the performed K-Means cluster analysis and a traditional statistical evaluation support our *hypothesis*.

Assuming that the statement of the *hypothesis* is true, one can therefore assume that the developed tool assigns a meaningful score to the factors. By using this tool, researchers and reviewers of machine learning experiments should be able to quickly and standardized assess whether an associated repository meets the reproducibility guidelines. With the resulting simple identification of relevant problem areas, we hope to make a positive contribution to reducing the reproducibility crisis in the area of machine learning.

Acknowledgments

I am very grateful that I got the opportunity to create this thesis at the Chair of Data Science, which is led by Prof. Dr. Michael Granitzer, at the University of Passau. This endeavor would not have been possible without my mentor Mehdi Ben Amor. Thank you for your great mentoring during this exhausting but also very instructive time. I would have not been able to do this on this scale and in this quality without out.

I am also grateful for all the people I was able to get to know through this bachelor's degree and the friendships that have developed from it.

Lastly, I would be remiss in not mentioning my family, especially my parents, my grannies, and my brothers. Your belief in me has kept my spirits and motivation high throughout this process. I also want to thank my friends for all their emotional support.

> *"When I went to the trial to play for Barca, my father told me: 'If you come back and they haven't selected you because they had 20 better players than you, no problem. If they don't select you because they had 20 guys more determined than you, don't come home'."*
>
> – Carles Puyol

List of Figures

List of Tables

1 Introduction

Machine learning (ML) is a key technology for solving complex problems in various fields [ANK18]. Therefore, reproducing and verifying the results of ML experiments is of great importance. Thus, it must be clearly defined which factors influence reproducibility in this area.

In traditional programming [Zel78] software is created on the basis of a requirements analysis with the associated specifications. In this step, it is defined which input the program expects. The software architecture is then worked out in a design process and implemented in a coding phase. The goal of a program is to map the input to the expected output. Finally, the output is verified in a testing phase to verify that it operates the way it is intended to. In theory, you can test software to the point where you can guarantee correct mappings for all input data. In practice, this will not be feasible for large software programs due to many influencing factors [Pan99].

In contrast, machine learning takes a different approach [BD17]. For a machine, it is possible to build a model for a given data set, which maps the data for us. First, the data set is preprocessed to filter out superfluous information and reduce it to the required data. Several factors are then selected, including the algorithm to be used, or the ratio in which the data set is divided into training and test data (train-test split). In addition, relevant features of the data set are specified. Finally, the machine-made model is then validated to check how well the created model maps the test data of the selected data set. The advantage compared to traditional programming is that a machine can recognize relationships from the data that a human cannot recognize. However, it is possible that the machine-generated model will not be able to assign the input to the expected output in all cases. Therefore, one evaluates the quality of the representation of the model in this process and tries improving it incrementally until it meets the expectations.

Without the help of machine learning, many modern software solutions would not even be able to be implemented [ACP21]. In almost all areas such as healthcare [SSJ18], big data[Lhe+17], or intrusion detection [SP10], ML can improve software solutions or even enable them in the first place. Due to this variety of problems that are to be solved by ML, there is also the need for different, problem-specific algorithms [Dey16]. However, there are still numerous research challenges, such as the hardware design [Sze+17], ethical concerns [GHS19], or model management [Sch+18].

Reproducible experiments are a prerequisite for the acceptance of found solutions to scientific challenges by the respective community. This also applies to the area of machine learning. Since reproducibility is a so-called “confused” terminology, a precise definition of the term reproducibility [KS15] is needed. We define the terms reproducibility and replicability in 2.1 and distinguish them from each other. This is of particular importance as different definitions of reproducibility are used in different works. In the following citations, we mark with * if the definition differs from ours and refers to replicability instead.

In an article from 2014 McNutt states that "[...] just because a result is reproducible* does not necessarily make it right, and just because it is not reproducible* does not necessarily make it wrong." [McN14]. Although this statement is correct, experiments can only be verified if they can be replicated. Otherwise, the results from an academic paper can not be achieved by the community. Therefore, the replicability of experiments is not only desirable but also required to preserve research integrity [IT18]. A 2016 Nature survey, where 1,576 researchers from different scientific fields participated, found that "More than 70% of researchers have tried and failed to reproduce* another scientist's experiments, and more than half have failed to reproduce* their own experiments." [Bak16]. This indicates that we are in a reproducibility* crisis.

Aims and Objectives of the Thesis

In a report from the NeurIPS 2019 reproducibility program, processes and guidelines were listed that are intended to improve reproducibility. Software tools that would support researchers were not introduced in this article, but the authors noted that the "Standardization of such tools would [...] improve ease of reproducibility*." [Pin+20]. Hence, our goal is to create a supporting tool for researchers and reviewers that checks an ML repository whether the detectable reproducibility factors are within the guidelines or not.

Hypothesis

To test the functionality of the tool, we *hypothesize* that, in general, *a repository belonging to an ML experiment published by academic publishers can be expected to adhere to the reproducibility guidelines significantly better than an ML repository on GitHub* (on average). To check this, the following research questions must first be answered.

`Research Question 1:` Which factors influence the reproducibility of ML experiments?

`Research Question 2:` Which influencing factors can be detected with the help of a software tool and which can not?

`Research Question 3:` How can the detectable factors be analyzed using a software tool?

In Chapter 2 we first give a brief overview of important subject areas which help to understand the following contents. Chapter 3 provides an overview of the contributions that others have already made to address this problem. We want to use these findings to propose a new approach based on them. When creating a scientific work, the time available is one of the limiting factors. Therefore, we want to develop a software tool to reduce the time required to check whether the reproducibility guidelines have been complied with. By compliance with the guidelines, we mean that the identified reproducibility factors have been taken into account. This identification step is carried out in Chapter 4, in which research question 1 is dealt with, based on a literature search. Subsequently, we classify them according to whether they can be detected with the help of a software tool. This relates to research question 2 and is dealt with in Chapter 5. Then, in Chapter 6, the developed software tool is presented. The proposed proof of concept implementation answers research question 3. We examine our hypothesis in Chapter 7. For this purpose, we use the presented tool to evaluate two different sets, each containing 100 ML repositories. Finally, we discuss the knowledge gained in Chapter 8.

2 Background

This chapter contains basic knowledge on basic knowledge of the main concepts of this work. First, we want to define the concept of reproducibility precisely. In the literature, this term is often used with different meanings. Therefore, we will clearly distinguish it from the term replicability. We also provide an overview of other important concepts and technologies. Where necessary, we will place topics in the overall context to explain their influence on this thesis. In addition, at the end of this chapter, we give a brief overview of a typical machine learning workflow.

2.1 Reproducibility

In this section, we precisely define the term reproducibility and distinguish it from replicability. We also want to draw attention to the fact that different definitions are used in other works. This goes from slight deviations to the exact opposite meaning of the terms. Therefore, one speaks of a so-called "confused" terminology [Ple18]. We understand by reproducibility what Goodman et al. defined as results reproducibility.

Reproducibility: "Obtain the same results from an independent study with procedures as closely matched to the original study as possible. [GFI16]"

The term machine learning (ML) is explained in Section 2.5. The presented workflow makes it clear that an experiment in the ML area can only be repeated exactly under very specific conditions. However, results from scientific work must be able to be verified by third parties in order to confirm their validity. This is of paramount importance within the open-science guidelines. Reproducibility is desirable to increase the validity and impact of a scientific contribution since results are still the same under slightly different conditions. Replicability, on the other hand, is indispensable in order to be able to verify the results.

Reproducibility vs. Replicability

In order to keep this thesis consistent in terms of terminology, we adapt the definition of methods reproducibility from Goodman et al. and understand this to mean replicability.

Replicability: "Provide sufficient detail about procedures and data so that the same procedures could be exactly repeated. [GFI16]"

For replicability, all information, methods, and data must be accessible in order to be able to repeat the experiment exactly. If the results match those presented, the work can be replicated. For reproducibility, as much information, methods, and data as possible must be accessible in order to be able to repeat the experiment under slightly different conditions. If the results then match, they are reproducible.

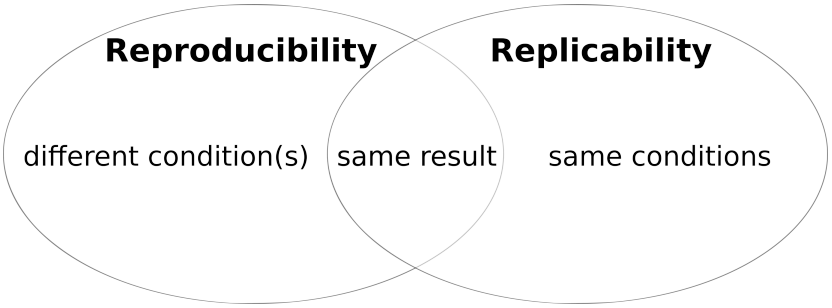


Figure 2.1: Reproducibility versus replicability. Since reproducibility and replicability are so-called "confused" terminologies, a precise distinction is required. An experiment is said to be replicable if repeating the experiment under the same conditions produces the same result. Reproducibility requires that the result remains the same even though some conditions have been (slightly) changed.

Drummond et al. state, that "Reproducibility requires changes; replicability avoids them. Although reproducibility is desirable [...] replicability, is one not worth having. [Dru09]". We only partially agree, because reproducibility also requires as much information as possible about the original work and the results that were achieved. If an author works transparently as far as possible, both replicability and reproducibility should ideally be achieved.

Challenges & Best Practices

Reproducibility challenges: There are various reasons why reproducibility and replicability are often not achieved. These criteria are usually not specified in the submission policies of most publishers. Furthermore, in order to achieve this, additional effort is required. It is also true that the more abstract the research field is, the more difficult it is to document the methods and processes in a comprehensible manner. In the area of machine learning, a model acquires the ability to assign the appropriate output for an input. Since this learning process is not necessarily subject to deterministic behavior, replicating an ML model is not trivial. This results in many influencing factors that need to be taken into account. These are identified and described in Chapter 4.

Olorisade et al. state that "[...] it may sometimes be hard or even impossible to reproduce computational studies [...]. [OBA17]". Furthermore, they point out that "[...] the minimum standard expected of any computational study is for it to be replicable.". It should be noted that we have changed the terminologies for reproducibility and replicability in the citations to match our definitions. This again shows the need to create at least replicable work. In the following sections, we present some key technologies that simplify this. In addition, we give an overview of the current state of affairs in Chapter 3. Also, we show which steps can be taken to increase the reproducibility probability without having to do a lot of additional work.

Reproducibility best practices: Naturally, there are already some best practices that have prevailed. Regardless of ML-specific factors, we consider complete and traceable documentation (both in terms of code and methodologies), encapsulation to eliminate dependencies, and avoidance of non-open-source technologies to be most important. We aim to point out these best practices, together with ML-specific features, and want to support improving the quality of a scientific contribution in this regard. In order to avoid additional overhead, we want to

analyze the reproducibility probability as automatically as possible and give recommendations for action based on this.

For this purpose, we first identify in Chapter 4 which influencing factors there are and in Chapter 5 which of them can be detected automatically.

2.2 Containerization

Containerization is important to this thesis for two reasons. First, this technique is useful from a reproducibility point of view. Second, BinderHub utilizes Docker and Kubernetes, among other technologies. For these reasons, we explain the basic concepts, starting with the definition of the term.

"Containerization is the process of creating, packaging, distributing, deploying, and executing applications in a lightweight and standardized process execution environment known as a container. [BSC17]"

Another common synonym for this is operating system (OS) virtualization.

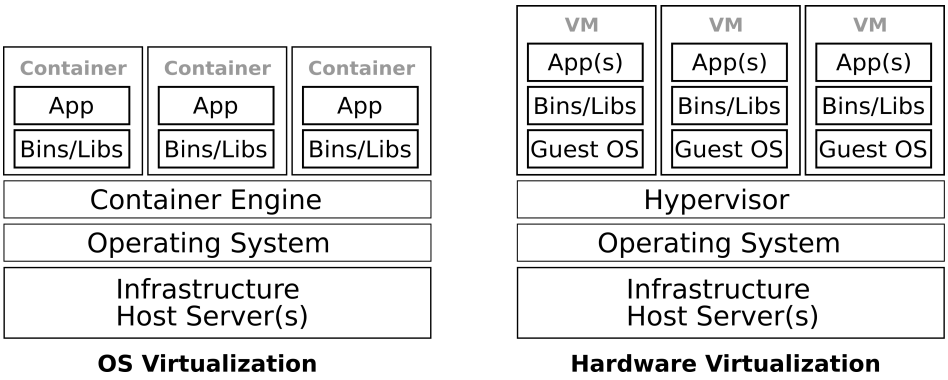


Figure 2.2: OS virtualization versus hardware virtualization. Both types of virtualization pursue a common goal, namely encapsulation. The difference lies in the layer on which this happens. OS virtualization occurs at the application layer, while hardware virtualization occurs at the hardware layer.

In Figure 2.2, we compare two different types of virtualization. Both have a common goal, namely encapsulation, but on different layers. Containerization takes place on the application layer and therefore has less overhead since only the application and its dependencies are simulated. With hardware virtualization, a complete computer environment is simulated.

Above all, containerization has enjoyed great popularity for years, especially because it "[...] provides efficient, scalable and cost-effective resource management solutions in cloud infrastructures [...]. [Wat+19]". This applies particularly to infrastructures that universities provide to their researchers and students. Thanks to open-source solutions such as Docker, this technology has also been established in the field of scientific work. Containerization can help eliminate the "works on my machine problem" and simplify the setup process. Since the de-

pendencies associated with the application are also stored in the container, this concept has a positive contribution to reproducibility.

The terms Docker and Kubernetes are explained below. These technologies are leveraged by Binder, which combines the benefits of containerization with other helpful features.

Docker

Without going into details about the inner workings of Docker, we want to explain some terms that should be clear, as they are of great importance from a reproducibility point of view.

Docker is a container engine used to create and manage containers on top of an operating system. Linux Kernel features like namespaces and control groups enable it to do so. It is an implementation of the concept of OS Virtualization.

A *Dockerfile* is a text document that describes the steps required to create a Docker image. This is the first step in creating a Docker container.

A *Docker image* is a snapshot of a virtual machine at a specific point in time. One can think of it as a digital image of a certain state. This image is immutable and can be duplicated and shared. It contains all the information and data needed to run as a container.

We have already listed possible areas of application for containerization. However, we would like to go into some properties that support reproducible research [Boe15].

Firstly, a Docker image provides other researchers with all the data they need to replicate the development environment. This avoids the dependency hell and simplifies the setup of an experiment. In addition, containers are lightweight and portable. Docker takes care of the packaging and running of a container, so it can run on different machines without any issues. Also, one can look up all the necessary dependencies and variables to create this image centrally in the Dockerfile. There are of course many other container engines such as LXC[1] or podman[2]. However, Docker is currently the de facto standard.

Kubernetes

Container orchestration systems are primarily required when large numbers of containers have to be managed. Kubernetes is a component of BinderHub, which is why we want to briefly explain this technology below.

Kubernetes is an open-source platform that enables the automated operation of Linux containers (e.g. Docker containers). Groups of hosts can be combined into so-called clusters, which are then managed by Kubernetes.

It allows computing-intensive applications to be distributed across multiple hosts, which is advantageous in different areas such as deep learning [Mao+20]. The self-healing ability is from an availability perspective also a reason for using Kubernetes, especially for microservice applications [Vay+19]. But numerous other services such as BinderHub, which will also be discussed in this thesis, use the combination of Kubernetes and Docker.

[1] https://linuxcontainers.org accessed: 16.05.2022
[2] https://podman.io accessed: 16.05.2022

2.3 Literate Programming

In our context, we have already explained that reproducibility requires that the methodologies, the code, and the results are documented in a clear and understandable way. Literate programming technologies, such as Jupyter notebooks, provide all the functionality required for this. The philosophy behind it was formulated by Donald E. Knuth in 1984.

He stated that the *"time is ripe for significantly better documentation of programs, and that we can achieve this best by considering programs to be works of literature"*[Knu84]. His idea was that instead of focusing on programming what the computer should do, one should shift to textually explaining to humans what we expect the computer should do.

Jupyter notebooks provide the functionality needed to implement this philosophy. It can not only execute the contained source code. In addition to the code, its output is saved. Also, it is possible to use markdown elements (e.g. paragraphs, figures, and links) to add human-readable documentation. So it combines the idea of being able to execute code quickly and easily with the concept of literate programming.

One possible use case is lab notebooks, which are a log of scientific activity. These should contain every detail (e.g. hypothesis, experiments, and interpretation of results) which later can be used to replicate the work [Ker+18]. This is also known as an executable paper. Jupyter notebooks offer a great opportunity not only for researchers but also for students. Due to their interactivity, complicated content can be conveyed well. This is particularly advantageous in the area of data science, artificial intelligence, and machine learning [OBM15]. In addition, Binder also supports this technology. But researchers are deterred by the additional effort that has to be expended to create a notebook and document it properly. They create no direct added value for the creator of a work, which is why it is often dispensed. However, we argue that the use of this technology makes sense for researchers as it can increase the probability of successful replication.

2.4 Binder

We assume that a researcher wants to pay attention to reproducibility and therefore undertakes both the specification of a configuration file (e.g. Dockerfile, requirements.txt, environment.yml, or conda.env) and the creation of a Jupyter notebook. Binder provides the functionality to combine both approaches.

"Binder is a free, open-source, and massively publicly available tool for easily creating sharable, interactive, replicable environments in the cloud [RW18]." In order to keep our definitions consistent, we modified the definition slightly.

It can thus combine the concepts of encapsulation that containerization offers with the possibilities of Jupyter notebooks. Since reproducing an experiment requires a change to some conditions, the interactivity that Binder offers is also perfectly suited. This also applies to the general workflow when developing an ML model, which is based on incremental improvements. Our tool will suggest using Binder to encourage the use of this technology. Of course, reproducibility does not require the use of these technologies, but they greatly simplify the verification process.

BinderHub

BinderHub is a cloud service based on Kubernetes that enables the sharing of replicable, interactive environments. These environments are generated from a repository using repo2docker. JupyterHub provides a scalable system with which users can authenticate themselves and interact with the created environment. It is not necessary to set up a BinderHub yourself, as a free infrastructure is provided at mybinder.org[3]. The generated Binder badge can then be integrated into the readme file so that third parties can use your repository quickly and easily. Figure 2.3 shows the architecture of this technology.

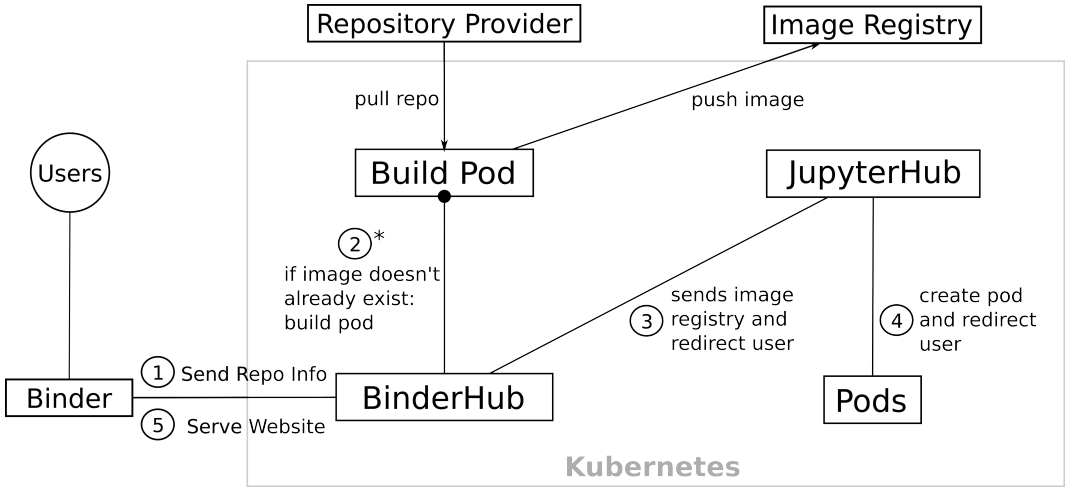


Figure 2.3: Simplified overview of the BinderHub architecture[4]. Shown here is the flow of information when a user sends a request to start a pod to BinderHub. Step 2 is omitted if an image is already stored in the registry for the requested repository.

repo2docker

BinderHub uses repo2docker to generate the Docker image if none is already present in the image registry.

"The core feature of repo2docker is to fetch a repository at an arbitrary URL, inspect the repository for configuration files that define the environment needed to run its content, and build a container image based on the files in the repository. [For+18]"

[3] https://mybinder.org accessed: 16.05.2022
[4] `https://binderhub.readthedocs.io/en/latest/overview.html` accessed: 16.05.2022

The following configuration files[5] are currently supported:

- environment.yml
- Pipfile
- requirements.txt
- setup.py
- Project.toml
- REQUIRE
- install.R
- apt.txt
- DESCRIPTION
- postBuild
- start
- Dockerfile

The Dockerfile has the highest priority. Other configuration files will be ignored if one is present. If no Dockerfile is present, but multiple others, they will be combined.

JupyterHub

"JupyterHub is the best way to serve Jupyter notebooks for multiple users. It can be used in a class of students, a corporate data science group, or a scientific research group. It is a multi-user hub that spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server.[6]" It not only works with Jupyter notebooks but also with Python code in general.

A JupyterHub consists of 4 subsystems. The hub is the core of the system and connects the other subsystems with each other. An HTTP proxy forwards requests from the client to the hub. After a user successfully logs into the authentication service, the hub spawns a single-user Jupyter notebook server. This server provides the functionality to run the container. The hub then tells the proxy to forward user requests to this server. When using JupyterHub within a BinderHub, the proxy does not receive the requests directly from the user. The BinderHub switches on beforehand to ensure that an image for the repository is stored in the registry.

2.5 Machine Learning Workflow

The goal of artificial intelligence (AI) is to be able to mimic intelligent human behavior. Machine learning is part of the more general field of AI. Rather than teaching a computer the human behavior directly, the concept here is to allow the computer to learn from data through experience and thus continually improve itself. ML is already being used today for things like chatbots, auto-correction, or automatic translations. We use the ML development workflow shown in Figure 2.4 to explain the general procedure. This is a simplified, modified version of the ML lifecycle that Ashmore et al. presented in their paper [ACP21]. After one has a general overview of this process, the ML-specific influencing factors identified in Chapter 4 can be better understood.

[5] https://repo2docker.readthedocs.io/en/2021.08.0/config_files.html accessed: 16.05.2022
[6] https://jupyterhub.readthedocs.io/en/2.1.1 accessed: 16.05.2022

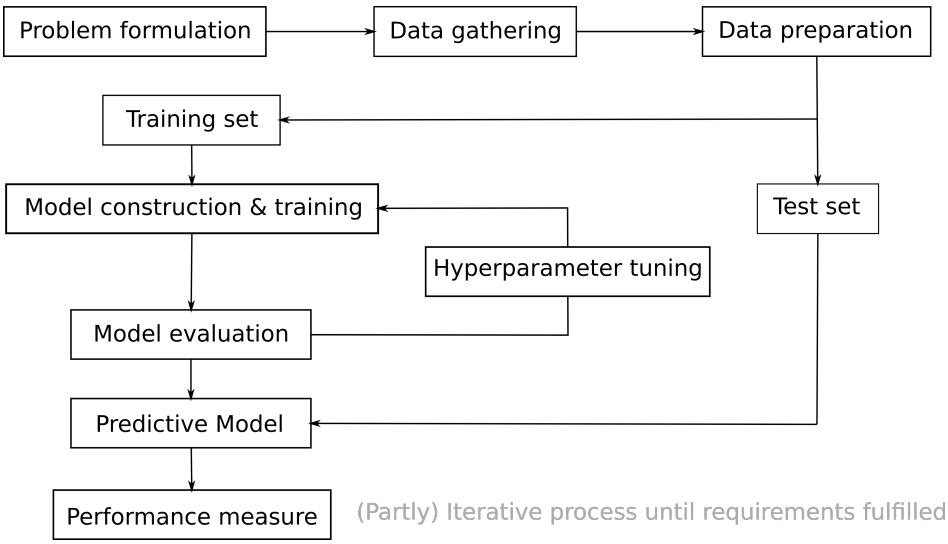


Figure 2.4: Abstracted, simplified machine learning workflow. A possible workflow for the development of a machine learning model is shown. Since a model is generally developed iteratively, some steps are run through several times. For this reason, it is important to document the changes made in order to be able to understand them afterward.

Data Gathering & Preparation

Matching the problem to be solved, a suitable data set is first selected as input. The content of the data set is of no further importance from the point of view of reproducibility. We are more interested in whether this data is publicly available.

Data preparation then ensures in several sub-steps that the data set is suitable as an input, as this significantly influences the ML model. As examples, we list the checking of the data quality, the comparison for uniform formatting, the substitution of missing values, or the elimination of superfluous features as possible intermediate steps. It is therefore not sufficient to give third parties access to the original data set. It is important to document which preparation steps have been taken.

`Feature Importance Ranking:`

The insight of feature importance ranking can be helpful for various reasons. For example, if the most important variables are known, less important ones can be deleted in order to increase the interpretability of the model at the expense of accuracy. This also results in a shorter training time, which is particularly noticeable with very large, complex data sets with many features. Feature importance can be determined using different approaches [KB21]. In Chapter 7 we use it to analyze the mean decrease of impurity with the random forest algorithm. Therefore, we want to explain this approach briefly.

Random forest is a supervised learning method that is often used in machine learning. The algorithm provides rules according to which uncorrelated decision trees are randomly created. The criteria by which each of these trees makes decisions are determined randomly and are different for all trees. An overall result is then created from the set of individual decisions of the random trees (using a special ensemble method). One advantage of random forest is, that

in contrast to other methods (e.g. neural networks), the decisions made are comprehensible and therefore easier to check. But although the individual trees are easy to understand, the contribution of the different features to the model is not so easy to assess.

The mean decrease of impurity is a measure of variable importance for estimating a target variable. It is the average of the total decrease of a variable in node impurity weighted by the proportion of samples reaching that node in each individual decision tree in the random forest. Hence, it is effectively a measure of how important a variable is in estimating the value of the target variable for all trees in the forest. This method requires the model to be trained to derive the importance scores. Because the mean decrease of impurity uses the statistics obtained from the set that trained the model, it may be limited in examining the important features of the test set. Due to the stochastic nature of this process, one should measure it multiple times and determine an averaged result.

`Data Analysis:`

Data analysis is not part of the workflow for creating an ML model but it can be used to study and understand the data used to create such a model. There are numerous data analysis methods [Zen+22]. In the context of this thesis, we use cluster analysis.

Clustering is the attempt to define groups within a set of entities. Entities belonging to the same group share some key characteristics. These groups can then be used to try to draw conclusions about the input data. We have decided to implement this method using the K-Means algorithm. It is an unsupervised, iterative machine learning algorithm, which is ideal for evaluating data sets that do not have a target variable. It assigns each entity to the cluster with the closest mean (centroid). In this algorithm, the k stands for the number of selected clusters and the *mean* for their centroid.

K-Means clustering has several advantages compared to alternatives. It is easy to implement and only requires the determination of the optimal number of clusters as this is the only significant parameter. In addition, K-Means performs so-called "hard" clustering, which means that each entity is assigned to a cluster [KP19], which simplifies the interpretation of the results. The relatively high impact of outliers on the result and the fact that certain patterns cannot be mapped well are disadvantages of this approach.

Train-Test Split

There are different ways to split the prepared data into different sets. The most common one is the train-test split. With this technique, the data is split into two subsets (e.g. 80/20 split). The training set is used for the model training process. After the model is trained it can then be evaluated using the test set. This split ensures that the model has never seen the data when evaluating. The train-test split is appropriate provided the data set is not too small. In general, this technique is used for regression or classification problems and is suitable for any supervised learning algorithm.

An adaptation of the train test split is the n-fold cross-validation method. The same technique is used here, but the process is repeated n times. For each run, a different block is selected as the test data set and an average is formed at the end. This reduces the risk of selecting non-representative test data. The result becomes more robust and the variance decreases.

Model Construction & Training

First, a machine learning algorithm is selected. In general, there are 3 different main categories of algorithms: supervised, unsupervised, and reinforcement learning [Wan+17]. In supervised learning, a mathematical relationship is established between an input x and an output y. These relationships are then used to create a model which can also predict the appropriate output from an input. In the case of unsupervised learning, the input x is not labeled. This means that the algorithm must match the input to a suitable output without obtaining any previously known patterns or relationships. In reinforcement learning, a reward system decides how the algorithm should proceed. The reward can be positive for good matching or negative for bad matching. The model continues to improve from past mistakes.

Different ML algorithms have different hyperparameters. By setting the hyperparameters one can control the learning process. By setting a specific random seed one ensures, that when rerunning the model with a different algorithm or changed hyperparameters, every time the same training and test data set will be used. In this way, we can eliminate the influence of different data split on the model performance.

Model Evaluation

After a model has been trained it can be evaluated using the test data set. This set consists of data that the model has never seen before. Based on specified requirements, such as a baseline or previous iterations, a decision is then made as to whether the model is good enough. If it does not meet the expectations, one can, for example, adjust hyperparameters in a new iteration or change the general learning strategy to achieve a better result.

Performance Measure

Based on the finished predictive model, new data can be evaluated. Typically, one notices the problem of underfitting in model evaluation, and overfitting when running the model on new data (see Figure 4.1). There are various performance metrics that can be measured. For example the F1 Score, Classification Accuracy, or Logarithmic Loss to name a few. While the model is deployed, these can be measured continuously. Based on these metrics, a future version of this ML model can be improved.

3 Related Work

In this chapter, we provide an overview of our findings related to the identification and classification of ML reproducibility factors. In addition, we present an existing approach for estimating the replicability probability of a paper and the state-of-the-art reviewing process.

`Reproducibilty Factors:`

Since our goal is to classify reproducibility factors that are relevant for machine learning experiments, we have to identify them beforehand. The identification of factors that affect reproducibility is an important subject of current research. NeurIPS, a machine learning, and computational neuroscience conference have even included reproducibility as part of its submission policy [Pin+19]. In the context of this thesis, we are interested in reproducibility factors that can occur in general or are specific to computer science or machine learning. Hence, we collected and merged the latest findings from these areas.

In their study, Samuel et al. [SK21] discussed different influencing factors for different research fields. They also made general recommendations that should help the scientific community to address these problems. Ivie et al. [IT18] examined computer science-specific reproducibility problems in their work. They described the causal factors and possible solution strategies in detail. Olorisade et al. [OBA17] have attempted to identify factors relevant to machine learning by manually reproducing several text mining studies. They came to the conclusion that the information on data sets, study parameters, and the software environment was missing in most of the examined studies, which hindered their reproducibility. Additionally, we took numerous other contributions into an account to identify the influencing factors [GGA18; Jan20; PF16; Rob10; Tsi+18; Hen+18; Liu+21; Hea+15; Eps+18; Cra17; Ioa05; Hud21]. We based the identification of reproducibility factors on the mentioned references in this paragraph.

In addition to identifying these factors, we also examined whether they had already been classified by others. Tatman et al. [TVD18] defined three different levels of reproducibility which differ in the amount of information available. However, they have used an alternate definition of reproducibility which corresponds to our definition of replicability. Goodman et al. [GFI16] proposed using a different definition in 2016. Because of the inconsistent use of the term reproducibility, they divided it into three different types: Methods, Results, and Inferential reproducibility. Methods reproducibility can be equated with replicability. Results and inferential reproducibility are understood to mean reproducibility. Based on this work, Isdahl et al. [IG19] investigated how well ML platforms support out-of-the-box reproducibility, and Gundersen et al. [GK18] manually surveyed 400 research papers using these metrics. While all of these classifications focus on conceptual delimitation, we want to achieve something different. Classifying the factors into either software-based detectable or undetectable factors should enable the implementation of a software tool. From our point of

view, this is important because it should be as straightforward as possible to check an ML experiment for reproducibility.

`Estimating the Replicability Probability of a Paper:`
Yang et al. [YYU20] examined how machine learning can be used to estimate the probability with which a study can be replicated. Their motivation was the same as ours. The automation of processes saves time and money and at the same time offers the possibility of scaling. Researchers could use this to prioritize work that is more likely than others to be replicated. They developed a machine learning model that can conclude the probability of replicability from the narrative content of a paper. Their result was that papers that are less likely to be replicated have a higher frequency of unusual n-grams and a lower frequency of common n-grams. N-grams are n connected words with which you can check the writing style. According to the authors, this ML model could provide results that are comparable to those of the best manual methods currently available.

`State-of-the-Art Reviewing Process:`
The use of checklists was recommended as part of the NeurIPS 2019 reproducibility program [Pin+20] and by Artrith et al. [Art+21]. This should enable the quality to be checked in a standardized manner concerning reproducibility before submitting a scientific paper. The downside to this is that this review is done manually. Hence, our goal is to reduce the use of manual review activities to the bare minimum. Nevertheless, checklists are a good standardized approach that we continue to recommend, especially for factors that we have classified as undetectable.

To the best of our knowledge, previous attempts at estimating the probability of replicating an ML experiment were not based on a repository analysis which is the core of our proposed method. However, there are several tools [Mor+21] that are supposed to improve the reproducibility probability in the area of machine learning such as Binder[1], CodeOcean[2], or Whole Tale[3].

[1] https://mybinder.org/ accessed: 16.05.2022
[2] https://codeocean.com/ accessed: 16.05.2022
[3] https://wholetale.org/ accessed: 16.05.2022

4 Identification of Reproducibility Factors

In this chapter, we identify factors affecting the reproducibility of machine learning experiments. To do this, we use relevant literature and collect this knowledge. For each of these factors, we explain what is meant by it and how it affects reproducibility.

This authentication step forms the basis of this thesis. The knowledge gained is used in Chapter 5 to classify the identified factors.

4.1 Source Code Availability & Documentation

Public access to the source code of a machine learning experiment is the cornerstone for many analysis options based on it. Without its publication, only the methods described in the work are available, which is generally not enough in terms of reproducibility. Source code can be shared in various ways. However, the most common way is to store relevant project files in a repository. In addition to the mere provision, it is also important that third parties can also understand and operate this code. Meaningful and helpful documentation is important for this [SK21]. Basically, code can be documented in different ways. In a readme file, for example, the basic repository structure, setup instructions, or a brief description of the areas of application can be described. In addition, a license helps to clarify the conditions for using and expanding the software [GGA18]. Additionally, detailed documentation in source code files and adherence to a standardized coding style can help make it more human-readable. This in turn leads to the fact that ambiguities are eliminated and the contents of the repository can be more easily understood, which increases the chances of successful reproduction.

4.2 Hardware Environment

The influence of the hardware used on reproducibility is manifold. Available resources can be very different, especially in the machine learning area. It can happen that one has no way of reproducing an experiment because he does not have access to enough computing power [IT18]. In addition, there is also the long-term risk of hardware wear out in connection with extincted hardware. In the future, the components used may no longer be available, preventing an exact replication of the experiment. However, the description of the hardware used is still advisable, since the numerical accuracy varies between different CPUs, GPUs, and TPUs [Jan20]. But it is more advisable to use techniques from the field of virtualization (see 2.2) instead of a textual description of the components used since these minimize the effort to create the environment.

4.3 Software Environment

Specifying the software dependencies in a separate configuration file is recommended as it reduces the effort of replicating the environment used. Here, it is important to ensure that the dependencies used can also be accessed by third parties. They must therefore be publicly available. To avoid compatibility problems, a strict declaration of the version used is recommended [IT18]. In most cases, the creation of such a file does not represent a great deal of additional work. On the other hand, this simplifies the setup process and eliminates possible sources of error in this context, which could impede reproduction.

4.4 Out-of-the-Box Buildability

For us, if a repository is buildable out of the box, that means it can be used in a Binder environment without errors. We have shown the resulting benefits in 2.4. However, the use of this technology not only offers added value for third parties who want to examine the repository. The authors of an ML experiment are recommended to use it before or during the documentation and release phase [SK21]. Piccolo et al. have concluded that this technology can bring together the documentation capabilities of notebooks for literal programming with software encapsulation [PF16]. This allows quick and easy access to the software functionality to be replicated. The interactivity of Binder means that changes can be made directly to the code and the reproducibility can thus be tested.

4.5 Data Set Availability & Preprocessing

In 2.1, we have already found that replicability has a direct impact on reproducibility. For this reason, it is important that the original data set used is included. Where this data is stored only plays a minor role. It is important that it is permanently and publicly available. Hence, we advocate inclusion in the repository. However, in the ML space, it is not uncommon for data sets to take up a large amount of storage space. In these cases, it is better to describe and link to this in the readme within a separate data set section. The disadvantage here is that the link may no longer be accessible at some point in the future. A well-known problem when dealing with data in this context is authorization issues, which prevent publication for security or other reasons [IT18]. If possible, these data sets should be substituted with a publishable one.

If the data set was preprocessed, at least the methodology used must be explained, because an independent reviewer should be able to repeat this. However, it is better to either save the preprocessing mechanics in a file or even provide the modified data set. Robles et. al found out that there is still a need for action here because although publicly available data was mostly used, the data set after the preprocessing step was only rarely included [Rob10].

4.6 Random Seed Control

At first glance, the elimination of any randomness seems desirable. But for some challenges in the machine learning area, this assumption is not correct. Randomness counteracts inherent biases and is useful for developing a generalized machine learning model. In order to use these advantages and still be able to guarantee reproducibility, it is necessary to use so-called random seeds [OBA17]. These seeds initialize a pseudo-random number generator (PRNG) with a starting value. If one uses this value again, the generated number is the same. Randomness can appear in many different places. In 2.5, we briefly described a possible use case in connection with a seed declaration. In summary, it is not desirable to avoid randomness but to make it controllable and repeatable using fixed starting values (seeds).

4.7 Hyperparameter-Logging

Figure 2.4 shows an example machine learning workflow. Here it is important to understand that the development of an ML model is generally an iterative process. Before each round one can tweak some knobs to cause changes. Parameters that are manually chosen based on what has been learned from previous runs are called hyperparameters. The value of these parameters cannot simply be read from the data. Instead, these are chosen intuitively, so experience makes a well-chosen value more likely [Jan20]. After each iteration, it is measured whether the changes made have improved the accuracy of the predictions made by the new model compared to the previous configurations. Usually, only the final model remains at the end of the development process, the selected parameters are lost for all intermediate steps. But documenting each step in this iterative process may be necessary, especially for reproducibility [OBA17]. It is therefore advisable to record these parameters in log files, for example.

4.8 Model-Serialization

Since we want to discuss model-serialization (MS), we must first clarify what serialization means. The goal of serialization is to convert an object or data structure, in our case an ML model, into a specific format. This artifact can then be stored anywhere and transformed back into the original object or data structure via deserialization. In the context of machine learning, for example, this can be useful when training a model is very resource-intensive. This means that this already trained model can be ported and used on a less powerful machine using serialization and deserialization. There are numerous ways in which this can be implemented. Pickle[1], joblib[2], or framework-specific functionalities like *torch.save()* from PyTorch[3] are available for this, just to name a few. Based on this technology, there are also data version control systems (e.g. DVC[4]), which can document the evolution of changing ML models [Tsi+18]. This is particularly helpful from a reproducibility point of view, since the

[1] https://docs.python.org/3/library/pickle.html accessed: 16.05.2022
[2] https://joblib.readthedocs.io/en/latest/auto_examples/serialization_and_wrappers.html accessed: 16.05.2022
[3] https://pytorch.org/tutorials/beginner/saving_loading_models.html accessed: 16.05.2022
[4] https://dvc.org/doc accessed: 16.05.2022

final models, and in the best case also the intermediate models can be made accessible with it. This not only avoids possible sources of error during replication but also minimizes the time required for this.

4.9 Research Practices & Experimental Design

Successful reproducibility cannot be expected without comprehensible documentation of the research methods used and the basic experimental design. In the absence of a standardized reporting process for the results achieved, including meaningful metrics, it is difficult to assess the significance of the improvements indicated [Hen+18]. In addition, Gundersen et al. showed that in many cases the explanation of the research methods used, the specification of the problem and the associated hypothesis, the publication of the data set and source code, the results obtained and the prediction made was not discussed or specified in the paper [GGA18]. Based on these observations, they have defined the following recommendations as to what content should be included in a scientific paper:

Recommendations for Experiments:

- Hypotheses and Predictions
- Experimental Design
- Measure and Metrics
- Evaluation Protocol
- Results, Result Description & Analysis
- Workflow & Workflow Citation
- Executions
- Hardware Specification

Recommendations for AI Methods:

- Problem Description
- Conceptual Method
- Pseudocode

Compliance with these standards not only increases the likelihood that a scientific contribution can be reproduced. These can also be used as a checklist when creating a paper, which leads to a higher quality. This may also increase the scientific relevance and informative value in general.

4.10 Underfitting & Overfitting

Underfitting and overfitting are two problems that can arise in the field of ML that can be used to clarify the difference between replicability and reproducibility once again. Underfitting occurs when an ML model can neither model the training data nor generalize new data. This is a particular problem for replicability because even if the data set used is available, there is a high probability that the results cannot be produced.

Overfitting describes the problem that a machine learning model has adapted too much to the test data used. The consequence of this is that the performance of the model on new data deteriorates significantly. This is a problem from a reproducibility point of view because the results obtained will most likely change when the model has to process unseen data. Therefore,

the experimental findings of a paper should be based on performance results from different test data sets [Liu+21]. This ensures that the possible overfitting of the model is noticed before publication.

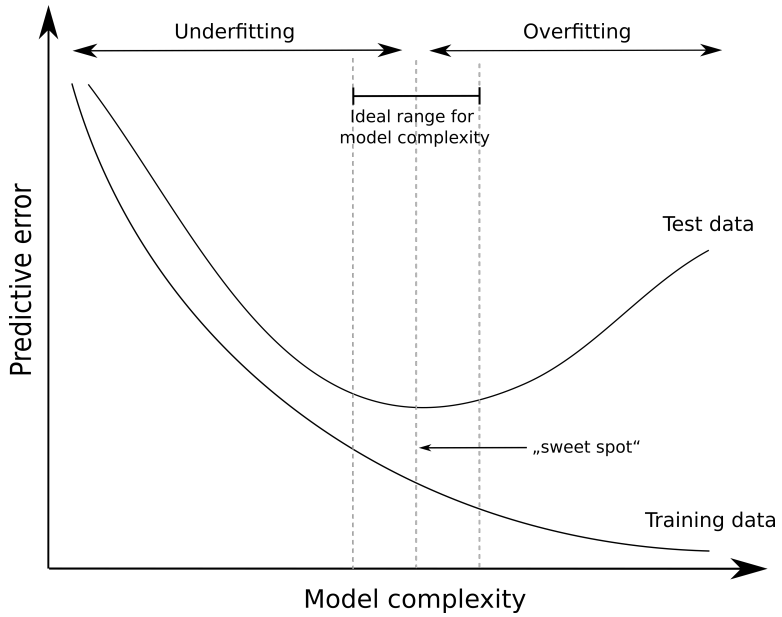


Figure 4.1: Adapted figure explaining overfitting & underfitting [Hea+15]. While neither the test nor the training data are well represented in an underfitting model due to its low complexity, it adapts too much to the training data in an overfitting model. A model complexity within the represented range close to the so-called "sweet spot" is ideal.

These problems can be viewed graphically in Figure 4.1. The optimal case is also shown here. An ML model should perform equally well on both the test data and new data. Hence, one is looking for the so-called "sweet spot" between underfitting and overfitting. There are some techniques that can be used in practice to find this "sweet spot", such as using a validation data set or resampling methods.

4.11 Knowledge Gap

In order to reproduce an experiment, it must first be understood. Of course, this assumes that a third party has the relevant expertise. In the field of machine learning, however, this trivial statement can become a problem. ML is often a very interdisciplinary field. Particularly in the case of very complex research projects, many people from different areas or disciplines are usually involved.

The term "knowledge gap" means that different people have different levels of knowledge. This can become an issue given the discussed imbalance between an individual reviewer and a research group. When creating a paper and documenting the experiments, care should be taken to ensure that an individual reviewer can understand the observations made as easily as possible. Otherwise, replication or reproduction can be very difficult or even impossible [Eps+18].

4.12 Probability Hacking

Probability hacking (p-hacking) means that prior to forming a hypothesis that one wishes to test, statistical tests are already performed on a data set until significant results are observed [Hea+15]. It does not matter whether and which underlying effects are present since the hypothesis can be adapted to the measured values. The main reason why this is done is that some scientific publishers like to publish significant results, as these are often particularly effective in the media. P-hacking is a problem mainly because it is so difficult to detect.

Whether this is done intentionally is not relevant to us, as the consequences are the same in both cases. Probability hacking has a negative impact on the scientific community because it allows third parties to draw false conclusions. In addition, these papers are inherently non-reproducible as the use of other underlying data will most likely not result in these reported significant results being observed [Cra17]. Significant evidence for the possible existence of p-hacking are results that differ greatly from previous work in a research area. Because it is so difficult to recognize, awareness is already an important first step.

4.13 Bias

Firstly, we want to point out that there is a difference between bias and chance variability [Ioa05]. Although the scientific approach is perfectly applied, chance variability can cause measurement results to be incorrect. Bias, on the other hand, refers to manipulations (intentional or unintentional) that can be made in both reporting and analyzing results.

There are many different types that can occur. Besides recall bias, there is publication bias, confirmation bias, and selection bias, just to name a few. Hudson et al. argue that unless any type of bias can be reliably identified through an assessment process, the reproducibility crisis is inevitable [Hud21]. The only way to counteract bias is to be aware of possible sources and try to eliminate them or at least reduce their influence.

5 Classification of Reproducibility Factors

After the influencing factors on the reproducibility were identified and explained in Chapter 4, we want to classify them in this chapter. Our goal is to present a classification approach that divides the factors into two different groups based on whether they can be detected by software or not.

We are particularly interested in the factors that a tool can evaluate. Building on this, in Chapter 6 we present a proof of concept (POC) implementation of an analysis tool that provides this functionality. In order to be able to understand the concepts discussed below, we first differentiate between the terms factor and indicator.

Factors have a direct correlation with reproducibility. By indicators, on the other hand, we understand software-based detectable properties of a factor. This relationship is shown in Figure 5.1.

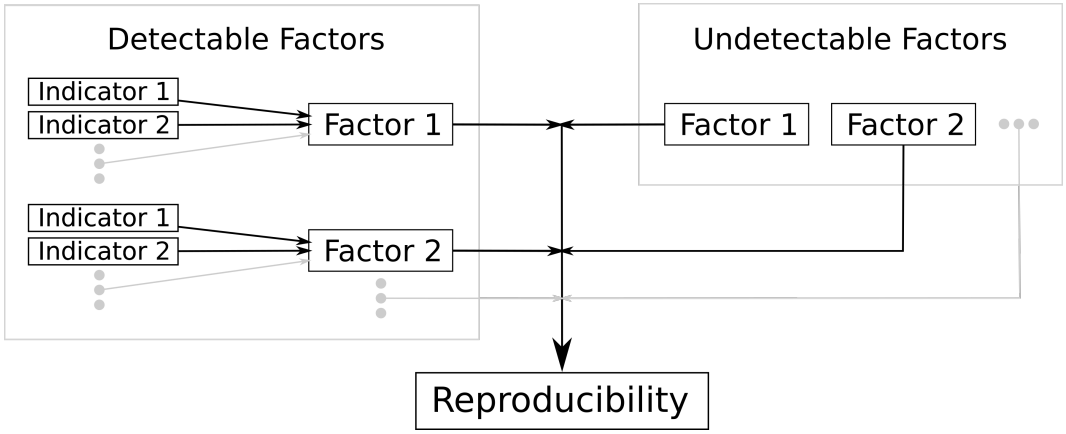


Figure 5.1: Connection of factors, indicators, and reproducibility. In order to be able to decide to which class a factor belongs, we use indicator-based reasoning. If at least one indicator that can be measured with a software tool can be assigned to a factor, then this is a detectable factor.

From this definition of the term, the basic concept of our classification becomes clear. We use indicator-based reasoning to classify the factors into one of the two groups. If at least one indicator can be identified for a factor, it can also be evaluated by a software tool. Otherwise, this is an undetectable factor. These will be covered in Section 5.2.

5.1 Detectable Factors

In the following, we deal with the factors that we classify as detectable. If at least one indicator can be found for a factor, we assign it to this group. In this section, we provide an overview of all identified indicators with their associated factors. In addition, we want to use a scale to clarify for each factor the degree of complexity to be expected with regard to the implementation.

Assignment of Indicators

For each factor, we considered what indicators, if any, it has. In the following, we give an overview of the results.

Table 5.1 contains all identified indicators and their respective factors.

Table 5.1: List of all identified reproducibility indicators and their corresponding factors.

Reproducibility Indicator	**Reproducibility Factor**
GitHub URL	Source code availability
Open-source license	Source code documentation
Readme length	Source code documentation
Readme hyperlinks	Source code documentation
Code comment ratio	Source code documentation
Pylint rating	Source code documentation
Notes on hardware environment	Hardware environment
Configuration file	Software environment
Strict declarations in the configuration file	Software environment
Imported libraries in the configuration file	Software environment
BinderHub-API build response	Out-of-the-box buildability
Binder badge in readme	Out-of-the-box buildability
Data set in the repository	Data set availability
Data set reference in the readme	Data set availability
Notes on data set preprocessing	Data set preprocessing
RS declaration (fixed) in the source code	Random seed control
HP-logging declaration in the source code	Hyperparameter declaration
Artifacts of model-serialization	Model-serialization
Model-serialization used in the source code	Model-serialization
Paper link in the readme	Research practices & experimental design

Complexity Scale

The presented scale estimates the complexity associated with the implementation of a factor's indicators.

To determine this complexity for each factor, we consider the associated prerequisites for computation and the expected level of implementation difficulty. The resulting scale is shown in Figure 5.2.

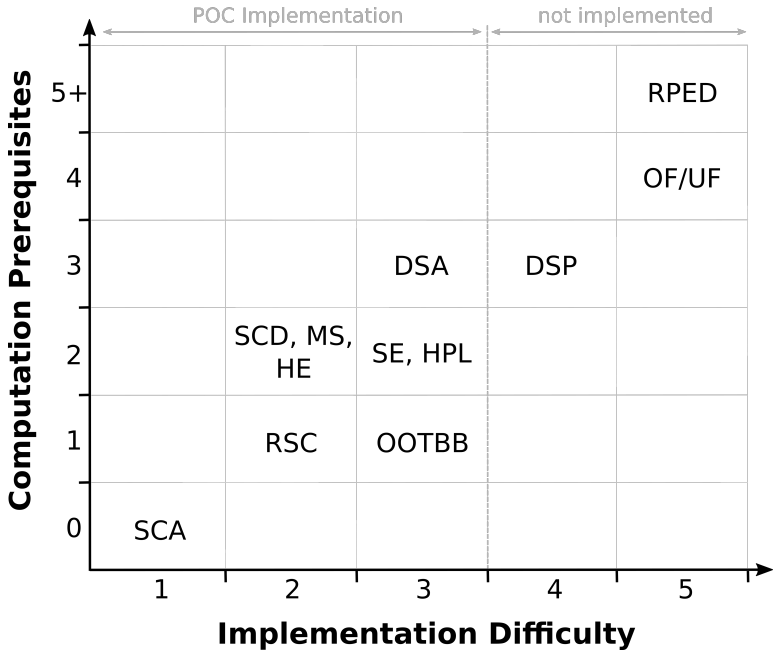


Figure 5.2: Classification of the detectable reproducibility factors in a complexity scale with regard to the number of their computation prerequisites and the implementation difficulty. Computation prerequisites refer to the number of required artifacts that have to be available, detected, or analyzed in order to examine all indicators of a factor. Implementation difficulty is a subjective assessment based on the author's skills and knowledge. Factors classified with an implementation difficulty above three are not included in our proof of concept implementation. The hardware environment factor is also not included, as we have found that if specified, it is mostly stated in the paper.

Computation prerequisites refer to the number of required artifacts that have to be available, detected, or analyzed in order to be able to examine the indicators of a factor in each case. Implementation difficulty is a subjective assessment that involved programming knowledge and theoretical concepts need to tackle the corresponding factor. We break this down into 5 different categories, with 1 being the lowest level and 5 being the highest level.

Due to their high complexity, the factors DSP, OF/UF, and RPED were not considered in our implementation.

In the following, we give a brief justification for the respective assessment for each factor. Since all factors can theoretically also be dealt with in the paper, we do not see this as a prerequisite for computation but assume access to it.

`Source Code Availability (SCA):`

Source code availability has no prerequisites since a link in the paper is not counted as such. An implementation is superfluous but could be done by testing the hyperlink.

`Hardware Environment (HE):`

The factor hardware environment has two prerequisites. It can be specified in a separate file or within the readme. The difficulty with the implementation lies in the fact that there is no standard case as far as the documentation of this factor is concerned. Hence, we have decided not to include this factor as the only one with a difficulty level of less than 4 in our tool.

`Source Code Documentation (SCD):`

Source code is usually explained in the respective source code file through comments. Additionally, we consider the operation of all code here. This is usually described in the readme. Therefore we assign 2 prerequisites to this factor. In order to analyze the source code documentation in practice, all source code files and readmes must be identified. It should be noted here that we do not directly check the content of the readme in our implementation, but try to use quantitative evaluations to assess whether it may have any meaningfulness in this regard.

`Random Seed Control (RSC):`

In order to analyze this factor, all source code files must be identified. Within these, one can then search for random seed declarations. For the implementation, this of course assumes that code lines are recognized and these declarations methods are known (for different frameworks).

`Hyperparameter-Logging (HPL):`

In addition to analyzing all source code files, it may also be necessary to identify separate log files to identify hyperparameter-logging. We have not considered this log file detection in our tool, since there is no standardized methodology for this, which would make the implementation too time-consuming (compared to the benefit). For the implementation, it requires knowledge of the possible logging methods (for different frameworks) and in addition to code line also import line detection.

`Software Environment (SE):`

The software environment is usually defined in a configuration file. But sometimes the relevant libraries are also listed in the readme. In our implementation, however, we only want to cover the standardized case of the declaration in a separate configuration file. To do this, they must be parsed and the import lines identified within the source code.

`Model-Serialization (MS):`

Model-serialization detection requires that all source code files have been identified and relevant artifacts can be detected in the repository. The implementation requires knowledge of relevant file extensions and libraries that support this.

`Out-of-the-Box Buildability (OOTB):`

The complexity of the out-of-the-box buildability factor is due to the fact that a BinderHub deployment is required and that the implementation requires expertise in dealing with this technology and its API.

`Data Set Availability (DSA):`

In order to be able to track down data sets, we require the detection of a readme, the source code, and relevant artifacts in the repository. During implementation, it is then necessary to develop an algorithm that can use these files to estimate whether an external data set is referenced or stored locally in the repository.

`Data Set Preprocessing (DSP):`

Data set preprocessing can be documented in the readme or in the source code. Source code files or scripts could also exist which can automatically execute the steps taken. Because of the many ways in which preprocessing can be specified, we have divided this factor into difficulty level 4. Therefore our implementation does not include the detection of this factor.

`Overfitting/Underfitting (OF/UF):`

In order to be able to assess overfitting/underfitting, we need access to a similar or the same data set and the preprocessing steps. In addition, the code must be executable and the measured results must be documented in the paper. In our opinion, the implementation of an algorithm that can estimate the presence of overfitting or underfitting is generally possible. However, this is very complex and was not pursued further by us.

`Research Practices & Experimental Design (RPED):`

For the examination of this factor, it is assumed that the paper and repository can be examined with one or more suitable tools. With this, we mainly refer to the extraction of the required information from the paper. The software-based comparison of whether specific guidelines have been complied with is then also necessary. From our point of view, this is therefore the most complex of all identified and detectable factors.

5.2 Undetectable Factors

Since no indicator was found for these factors, they are not detectable with a software tool. This means that a manual review process is unavoidable for assessing whether and to what extent any of these factors influence an ML experiment. In the following, we briefly explain how to deal with the undetectable factors.

`Knowledge Gap:`

From the point of view of a researcher conducting an ML experiment, it is important to take this issue into account when creating it. Hence, everything should be documented as precisely and understandably as possible. In this way, a possible knowledge gap as described in 4.11 can possibly be reduced preventively or, in the best case, be closed. From a reviewer's point of view, due to the interdisciplinary nature of the ML field, it may still be unavoidable to work with reviewers from other disciplines to replicate or reproduce the experiment.

`Bias & Probability Hacking:`

The main issue here is that even with a manual review, these factors are very difficult, if not impossible, to detect. However, it is the ethical duty of a researcher to avoid these reproducibility problems. There are however types of bias that can arise subconsciously or cannot be avoided. Hence, it is important to document all steps as precisely as possible, explain the decisions and interpretations made, and publish the data used [SN14].

In general, these influencing factors are often difficult to avoid or arise subconsciously (apart from p-hacking). Therefore, researchers should always be aware that these factors can cause reproducibility and replicability difficulties.

6 Implementation

This chapter explains how we can use a software solution to automatically analyze and evaluate the factors identified in Chapter 4 and classified as detectable in Chapter 5. For this, we present our proof of concept (POC) implementation.

The tool described here will be available in a public repository[1]. The necessary requirements and installation steps are also documented there together with other helpful information in the provided readme file. In the following, we explain the conceptual considerations and thought processes that led to the creation of this implementation.

It should help authors and developers of machine learning experiments as well as reviewers of third-party work. A GUI is not required for this user group. We focused on the desired core functionality and opted for command-line operation. Since Python is the programming language mainly used in the field of machine learning, we decided to use it as well. The aim is to check a repository with regard to the reproducibility guidelines and to be able to assess whether there are any problem areas. This means that compliance with the recommendations can be checked during development or before publication with as less effort as possible. In addition, a reviewer can quickly and easily examine the properties of a machine learning repository in relation to reproducibility. This is made possible by the approach described in 6.2.

6.1 Architecture & Components

This section explains the chosen tool structure and the procedure to extract the values of the indicators listed in Table 5.1 from an ML repository. These measurements are required for the approach proposed in 6.2 in order to assign a representative evaluation value to each influencing factor. As mentioned, this is a POC implementation, which is why we currently only support public repositories hosted on GitHub and programmed in Python (file extensions ".py" or ".ipynb").

Flowchart 6.1 gives a visual overview of the implemented modules and how they interact with each other when the tool is executed.

[1] https://github.com/martinloos/ml_repository_reproducibility_analysis_tool

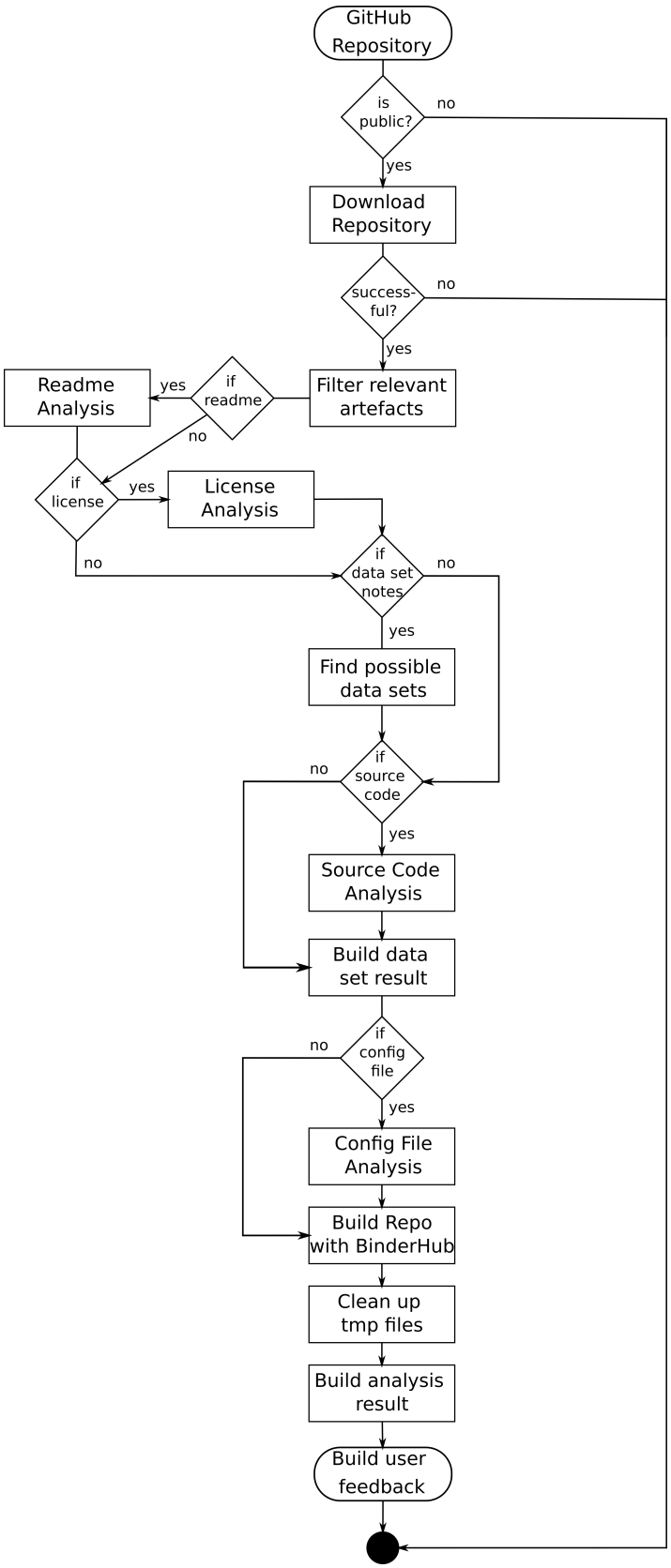


Figure 6.1: The simplified flowchart of the repository analysis tool shows the execution order of the python modules. Additionally, the conditions under which the tool terminates and when the execution of individual modules is skipped are displayed.

Repository Download & Preprocessing

Firstly, the URL of the GitHub repository to be downloaded locally is passed to the *repository_cloner* module. If it is already stored in the /tmp folder, downloading it again is superfluous. Otherwise, the download will be started if the repository is publicly available and also has either a *master* or *main* branch.

If this is successful, the *filter_repository_artifacts* module starts searching for relevant folders and files. These are then used by the respective analysis modules using *get* methods. At the end of each analysis file, the result is saved by the module so that it can be collected by the *result_builder* module.

Readme & License Analysis

To identify readmes, the *filter_repository_artifacts* module uses string matching to search for files that contain "readme" in the name. These are then analyzed by *readme_analysis.py*. The length and the links used are saved for each file. All hyperlinks are also tested for functionality. Large, memory-intensive data sets are often not stored locally in the repository but are referenced in the readme. In order to detect them, we saved the names of about 350 known data sets in a file. If one of these is mentioned, we save it as a data set reference. This also applies if a section containing "data" in the name and at least one hyperlink is present. In addition, it is checked whether a paper is linked or a binder badge has been specified. This is done for all files and the numerical results are averaged.

For the detection of licenses, *filter_repository_artifacts.py* also uses the same approach as explained in the readme part above. The contents of the recognized files are then checked by *license_analysis.py* using keyword matching to see whether the text mentions open source licenses defined by the Free Software Foundation[2]. Both open-source and non-open-source licenses are then stored in separate lists.

Retrieving Data Set File Candidates

Next, the search for included data sets in the repository is started. All folders that match a specified regex, along with files that have "data" in their name, are passed from the *filter_repository_artifacts* to the *dataset_analysis* module for further processing. Files found in the detected folders which do not end with ".py", ".ipynb" or ".md" are saved together with the files containing "data" in their name as so-called data set file candidates. For these candidates, it is then checked during the source code analysis whether they occur in the code. All candidates for which a match is found are assumed to be data sets.

Source Code Analysis

For the analysis of source code files, the *filter_repository_artifacts* module collects all files with the extension ".py" or ".ipynb". The *source_code_analysis.py* file calls the appropriate module for each file according to its extension. Since we currently only support Python, at this stage of development it is always *python_source_code_analysis.py*. If a Python notebook

[2] https://en.wikipedia.org/wiki/Comparison_of_free_and_open -source_software_licences accessed: 16.05.2022

is to be analyzed, it is converted into a ".py" file for further processing using nbconvert[3]. We are particularly interested in the code-comment-ratio, the pylint rating, the number of random seed declarations (and how many of them have a fixed seed) and a list of the library imports required by *config_ analysis.py*. We also check the import statements and lines of code to see whether they contain hyperparameter-logging relevant content. For imports, this affects the following libraries: "wandb", "neptune", "sacred" and "mlflow". In the code lines, we look for their respective method calls. These were taken from the documentation of the listed libraries. Finally, we also look for indicators of model-serialization. For this we check the code lines for calls of *pickle.dump()* or *torch.save()*. In addition, *source_ code_ analysis.py* also searches for model-serialization artifacts. We expect these to be either in a folder named ".dvc", that they have a file name equal to "model" or to have one of the following file extensions: ".dvc", ".h5", ".pkl" or ".model" (these artifacts are all collected by the *filter_ repository_ artifacts* module if found). Also, we filter out the Python Standard Libraries[4] and locally defined modules from all found imports and eliminate duplicates. We refer to the remaining libraries in the following as the relevant libraries. Finally, it is tested whether these are publicly accessible (using pip-search). The average of the numerical analysis results is then recorded and saved together with the other measured values in an overall result.

Software Environment Analysis

The *config_ analysis.py* module receives all from *filter_ repository_ artifacts.py* detected configuration files (which match a specified regex) as well as a preprocessed list of all used relevant library imports from *source_ code_ analysis.py*. The following configuration file types are currently supported: requirements.txt, environment.yml, conda.yml, and Dockerfile. From the files found, all library declarations are saved in a list. In addition, it is checked how many of them have been specified strictly (==). Then it is analyzed how many of the relevant imports used in the source code were specified in the configuration file and the results are saved.

Out-of-the-Box Buildability Analysis

In the *binderhub_ call* module, the user must specify a BinderHub IP or URL at the appropriate place in the Python file (if the associated out-of-the-box buildability factor should be checked) before the repository analysis is started. For this specification, it is first checked whether the BinderHub can be reached. If it is not available, further analysis is terminated and the result recorded. If successful, a build request is made via the API interface (*<BinderHub URL>/gh/build/<repository-owner>/<repository-name>*). The result of this request can be either "ready" if the build was successful or otherwise "failed".

Output

Finally, *result_ builder* collects the results of the analysis modules and forms an overall result from them, which is then saved in a list for creating the feedback. This is also persisted in a ".csv" file (e.g. for debugging purposes). The output of the tool is discussed in more detail in Section 6.3.

[3] https://nbconvert.readthedocs.io/en/latest/ accessed: 16.05.2022
[4] https://docs.python.org/3/library/ accessed: 16.05.2022

6.2 Factor-Indicator-Connection

In the previous section, we explained how the tool works and how the individual Python modules interact with each other (without taking into account the feedback). At this stage of development, the tool can analyze a repository and store the measured values in a ".csv" file for manual analysis. However, our goal is to provide helpful feedback on each identified reproducibility factor based on these measurements. Hence, it is necessary to connect the detectable factors theoretically defined in Chapter 5 with the associated indicators measured by the analysis tool. In order to achieve this goal, the following questions must be answered:

1. If a factor is determined by several indicators, how can the measured values be combined into an overall scoring? Each indicator has its own range of values. Therefore, each value must be converted to the same value range before weighting.
2. Which of the indicators examined are statistically significant? If the representative value of an indicator is very similar for both reproducible and non-reproducible repositories, no statement can be made about the probability of reproducibility with this indicator.
3. Does a measured indicator value indicate a reproducible or non-reproducible repository? In order to be able to answer this, we need comparative values.
4. If a factor is determined by several indicators, how much influence do they each have on the factor? Therefore, these indicators must be weighted. In order to determine these weights, we look at the comparative values measured for reproducible and non-reproducible repositories and look at the distances between the values. Indicators that show a higher deviation should have a higher weight.

As already mentioned, for each indicator we need comparative values for both reproducible and non-reproducible repositories. Therefore, twenty reproducible and twenty non-reproducible repositories were collected. Hence, we had to check them manually, and depending on whether we were able to reproduce them successfully they were assigned to one of the two sets. More repositories for each set would have been more representative, but due to the fact that we had to check them manually, that was not possible in the given time span. The two sets were then analyzed with our tool and the representative values and weights were determined based on the results obtained. We have documented in the appendix where these data sets can be accessed.

The *feedback_ builder* module is the implementation of the concepts presented in the following subsections, which assigns a range normalized and weighted value to each indicator. Based on the determined comparative values, it can be estimated for each indicator whether this indicates a reproducible or non-reproducible repository.

Using this assessment, a feedback file is created as an output. In addition to all factor and indicator scorings, this also contains the determined weightings, if applicable. In addition, textual feedback is given for each indicator, which enables users to assess the probability of successful reproducibility, suggest improvements, identify problem areas or explain the limits of the analysis tool.

6.2.1 Min-Max Normalization

Using normalization, we want to make it possible to merge all measured indicator values of a factor. The min-max normalization is particularly suitable for our problem since both the minimum and the maximum acceptable values of different indicators can be different. We adapt the formula used by Gajera et al. [Gaj+16] to calculate the value within a new range:

$$Y = \frac{(x - min(d)) * (max(n) - min(n))}{max(d) - min(d)} + min(n)$$

where

Y = Indicator value in new range
x = Original indicator value
$min(d)$ = Lowest value in original range
$max(d)$ = Highest value in original range
$min(n)$ = Lowest value in new range
$max(n)$ = Highest value in new range

Most important for our approach to connecting factors with indicators are *min(d)* and *max(d)*. In Section 6.2.2, we extract representative values for some of the indicators for both reproducible and non-reproducible repositories. We then use these values for the min-max normalization and assign them as the respective values *min(d)* or *max(d)*. In addition, we determine for *min(n) = 0* and *max(n) = 1*. Every measured value is within this new, uniform range after normalization.

6.2.2 Representative Indicator Values & Weights

In this section, we compare two sets each containing twenty ML repositories. One of the sets consists of exclusively reproducible the other of non-reproducible entities. We want to eliminate indicators that do not allow any conclusions about reproducibility and additionally determine representative values for all relevant indicators for both sets. Also, the difference between these values is used to determine a suitable weighting in each case.

Using Figure 6.2, we first consider which analyzable artifacts a repository of the respective set contains on average. For each of the artifact types, we observe that the set of reproducible repositories contains it significantly more often. The exception is the existence of a readme artifact. This is not surprising, since this file can also be created during repository creation by setting a check mark (which is often already preselected).

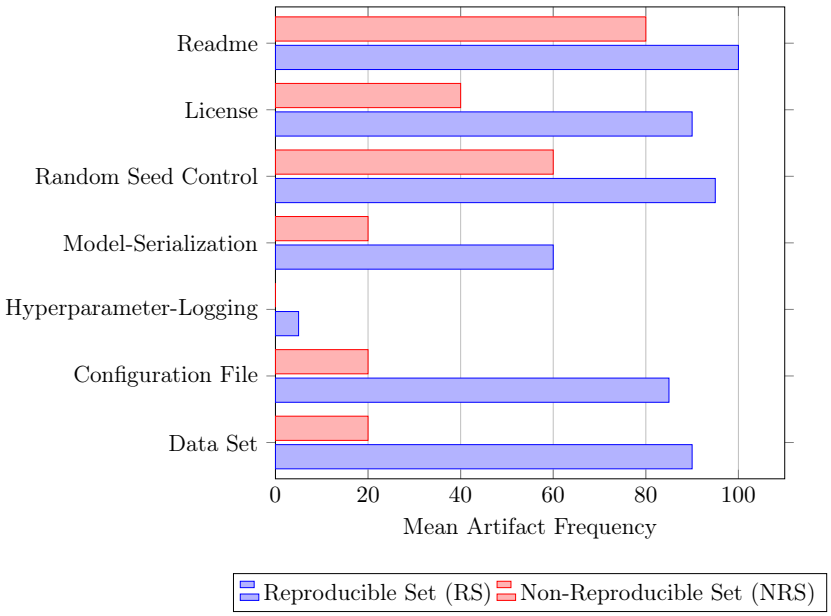


Figure 6.2: Statistical evaluation of the frequency with which relevant artifacts are present in the reproducible and non-reproducible ML repository set. From this figure it can be observed that the reproducible set contained all measured artifact types significantly more frequently on average. An exception is the readme artifact, where a similar frequency is given. This is probably due to the fact that this is usually generated when a repository is created by setting a check mark.

The following steps are considered for each indicator of a factor:

1. Assessment of whether there is a connection with reproducibility based on the observations made.
2. Determining whether representative values are necessary.
3. Assignment of *min(d)* and *max(d)*.
4. After steps 1 - 3 have been carried out for all indicators of a factor, we determine their respective weighting. If the factor is determined by only one indicator, its weight is 1 (maximum weight). If there are multiple ones their weight sums up to 1.

Source Code Availability & Documentation

To assess this factor, we examine a total of six indicators extracted from the following three artifact types:

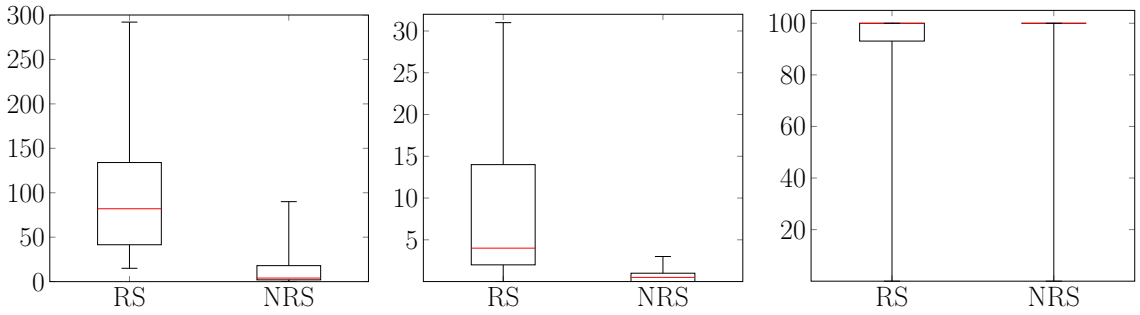
- *Readme:* The relevant indicators are the average length, the average number of hyperlinks as well as the percentage of accessible links.
- *License:* We want to check the presence of an (open-source) license.
- *Source code:* The average code comment ratio and the average pylint rating will be measured.

In order to simplify the weighting, we first summarize the indicators from the readme analysis and form a partial scoring from them.

`Readme Scoring:`

From Figure 6.2 we can see that the mere existence of a readme file does not mean anything. This artifact type was contained in almost all cases in both sets. However, differences become apparent as soon as we compare the measured values of both sets for the indicators.

Figure 6.3 shows that readmes in the reproducible set are significantly longer than those in the non-reproducible one. This observation also applies to the number of hyperlinks used, although the difference there is somewhat smaller. All found links of both sets were reachable. Since no differences could be determined here, this indicator probably cannot be used to draw any conclusions about reproducibility. However, since a hyperlink only makes sense if it works as intended, we combine these two indicators.



(a) Length (mean) **(b)** Links (mean) **(c)** Accessible links (%)

Figure 6.3: Readme indicators are measured to determine the representative values and weights. To obtain these values, we used our tool for all entities of the reproducible (RS) and non-reproducible (NRS) set and analyzed their readme files. Both *(a) Length* and *(b) Links* indicate a significant difference. From *(c) accessible links* it can be deduced that this indicator is probably not meaningful in terms of reproducibility.

`Determination of the Representative Values:`

Representative comparison values are required for the two remaining indicators in order to be able to make a statement about the measured values. We choose the value of the 3rd quantile of the non-reproducible set as *min(d)* for both the length and the number of accessible hyperlinks. The reason for this is that for all values below *min(d)* the score of the indicator will be 0 after normalization. It follows that most repositories from this set will get an appropriate score. The median of the reproducible set is chosen for *max(d)*. Our reasoning for this is that the assessment of an indicator should promote compliance with reproducibility guidelines. Therefore, the 1st quantile was not chosen because high standards are desirable. Since all entities are reproducible within this set, the 3rd quantile would not be a suitable value either, as most would then be below it. Therefore, we chose the median as a compromise.

`Determination of the Weights:`

When determining the weights, we could rely purely on the respective differences in the indicator values in the comparison of the two sets. However, since these are very similar, we add a conceptual consideration. In this context, the source code availability & documentation factor primarily refers to the documentation. Our assumption is that the contents of a readme serve precisely this purpose. Since a hyperlink is not necessarily included for documentation

purposes, we argue that the length should be given significantly more weight.

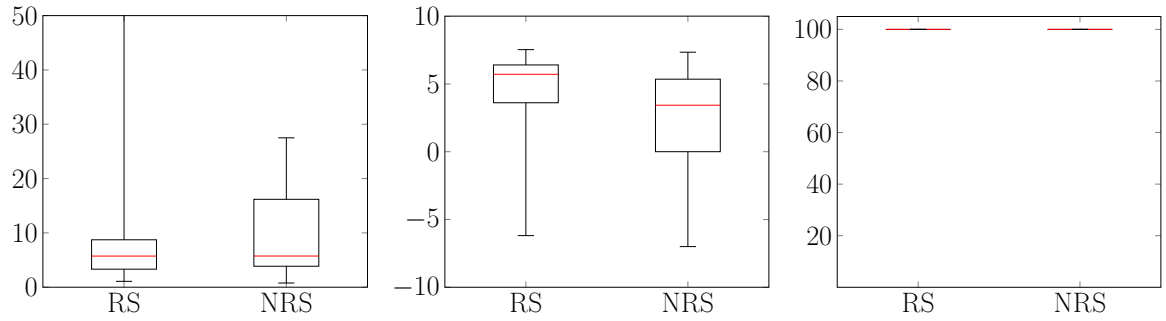
Table 6.1 shows the classification that results from these observations:

Table 6.1: Readme indicators with their assigned representative values and weights.

Indicator	min(d)	max(d)	Weight
Length	18	82	0.8
Accessible hyperlinks	1	4	0.2

`Source Code Availability & Documentation Scoring:`

Now let's turn our attention to the indicators of the other artifact types. Figure 6.4 shows that the median of the code-comment-ratio is almost identical for both sets. However, the ratio is significantly different as far as the 3rd quantile is concerned. It can therefore be assumed that this indicator has some significance. Since 100% of the licenses found in both sets were open-source (if they contained a license), no significance can be attributed to this indicator alone. In connection with the findings from Figure 6.2, we argue that a reproducible repository has a significantly higher probability of including an open-source license. Since not specifying a license is bad practice and specifying a non-open-source license can lead to reproducibility problems, we want to include this indicator in the scoring. Differences can also be observed in the pylint rating, especially with regard to the respective 1st quantile.



(a) Code comment ratio (mean) **(b)** Pylint rating (mean) **(c)** Open-source license (%)

Figure 6.4: Source code availability & documentation indicators were measured to determine the representative values and weights. In order to obtain these values, we used our tool for all entities of the reproducible (RS) and non-reproducible (NRS) sets of ML repositories and analyzed their source code files and all licenses found. From *(a) code comment ratio* it can be read that on average entities of the reproducible and non-reproducible set showed similar values. However, the values of the 3rd quantile show a significant difference. Equally clear is the difference of the 1st quantile as far as *(b) pylint rating* is concerned. Based on *(c) open source license*, it becomes obvious that if this artifact type was found, it could be assigned to the open-source category in all cases, which is why this indicator alone is probably not meaningful.

`Determination of the Representative Values:`

If no or a non-open-source license is detected, the score of this indicator is 0, otherwise 1. We choose these values because we only check the existence of this indicator.

While the median of the measured code comment ratios is almost the same for both sets, there is a significant difference when it comes to the 3rd quantile. A higher code comment ratio means there are fewer comment lines per line of code. We argue that a high ratio has a negative effect on the understandability of source code. Therefore, we assign a score of 1 if it is lower than the 3rd quantile of the reproducible set. For larger values we calculate the scoring with the formula:

1 - normalized value (with *min(d)* = 3rd quantile of the reproducible set and *max(d)* = 3rd quantile of the non-reproducible set).

Looking at the pylint evaluation, it is noticeable that the median of the sets is similar, but the 1st quantile differs significantly. For *min(d)* we choose the value of the 1st quantile of the non-reproducible set, as this represents the minimum rating we expect. To encourage a standardized coding style, we chose the median of the reproducible set as *max(d)* as this seems to be a high but achievable value.

`Determination of the Weights:`

When determining the readme weights, we already mentioned that the source code availability & documentation factor primarily refers to the documentation. For this reason, the readme score is given the highest weight for the overall factor score. It is also important that the content used is publicly available. Hence, we assigned a corresponding weight to the license indicator score. For the remaining indicators code comment ratio and pylint rating, the median of both sets was very similar. Therefore, in most cases, they are less meaningful than the indicators discussed previously. This was taken into account when determining the weights.

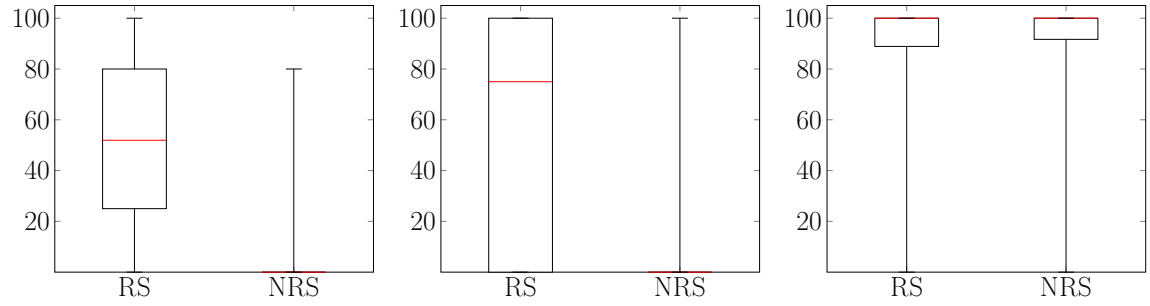
The following classification results from the observations made:

Table 6.2: Source code availability & documentation indicators with their assigned representative values and weights.

Indicator	**min(d)**	**max(d)**	**Weight**
Readme	-	-	0.5
License	0	Number of licenses	0.3
Code comment ratio	8.73	16.18	0.1
Pylint rating	0	5.71	0.1

Software Environment

As one can see in Figure 6.2, a repository from the reproducible set is much more likely to contain a configuration file. But we also want to check what, if any, influence the content and quality of these files have. In this context, relevant information from the source code files must also be used for the assessment. Therefore, we analyze the properties shown in Figure 6.5.



(a) Source code imports in the configuration file (%)

(b) Strict dependency declarations (%)

(c) Publicly accessible libraries in the source code (%)

Figure 6.5: Software environment indicators are measured to determine the representative values and weights. To obtain these values, we used our tool for all entities of the reproducible (RS) and non-reproducible (NRS) sets of ML repositories and analyzed their configuration and source code files. An important factor influencing this evaluation was that the majority of the NRS entities did not have a configuration file. Significant differences can be observed based on *(a) source code imports in the configuration file* and *(b) strict dependency declarations*. The evaluation of *(c) publicly accessible libraries in the source code* probably does not allow any conclusions to be drawn about the reproducibility property based on the measured values.

When comparing how many of the imports used in the source code (excluding standard Python libraries and local modules) were defined in the configuration file, a clear difference can be seen. The median for the reproducible set is around 50% coverage. Although this is not yet an optimal value, it is a significant deviation compared to the non-reproducible set. It can also be observed that the dependencies of reproducible repositories are usually declared with strict versions. The metrics of both sets of repositories in terms of how many of the libraries used in the source code are publicly available are very similar. Hence, this indicator is not suitable for estimating whether an analyzed repository is more likely to be reproducible or non-reproducible. Nevertheless, the use of libraries that are not open to the public is a hindrance and should therefore be reflected in the assessment of this factor.

`Determination of the Representative Values & Weights:`

Since all specified key indicators are calculated as percentage values, this results in $min(d) = 0$ and $max(d) = 100$. We argue that it is most important to include all relevant libraries used in the source code in the configuration file. In addition, a third party can only reliably access the functionality of these imports if they are publicly accessible. Finally, strict declarations avoid possible compatibility problems. These arguments are the basis of the chosen weights. This results in the following classification:

Table 6.3: Software environment indicators with their assigned representative values and weights.

Indicator	min(d)	max(d)	Weight
Source code imports in configuration file	0	100	0.6
Strict dependency declarations	0	100	0.2
Publicly accessible libraries in source code	0	100	0.2

Data Set Availability & Preprocessing

Firstly, we look at Figure 6.2 again. We found a data set for almost all repositories in the reproducible set. In contrast, this was included significantly less frequently in the non-reproducible set. From this, it can be deduced that the non-existence of a data set tends to indicate a non-reproducible repository. It is important to note at this point that this is a POC implementation. For this reason, the functions related to the preprocessing of the data set are not yet supported.

The *dataset_ analysis.py* module identifies possible data set files. We are referring to them as data set file candidates. The *source_ code_ analysis.py* module checks which of these candidates is used in the code. We assume that matches are relevant files containing the data set. Additionally, during readme analysis, we look for a data set reference (based on a list of known data sets and hyperlinks within a section that has "data" in its name). These are the identified meaningful indicators for this factor. Therefore, it is only examined whether at least one of the identified data set file candidates are mentioned in the source code or if a data set reference is found in the readme. If so, we score this factor with the maximum value of 1. Otherwise, we assume that no data set is included or referenced in the repository and we assign the minimum value of 0.

Random Seed Control

Almost all repositories from the reproducible set have taken this influencing factor into account (see Figure 6.2). For the non-reproducible set, this only applies to 60%. A statistical correlation can thus be observed. In addition to the declaration, we also measured how many of them contain a fixed random seed. Our analysis has shown that there are no differences in this respect when comparing these sets. If a random seed was defined, it was defined with a fixed value in all cases. However, from a reproducibility perspective, we want to ensure that a declared seed always has a fixed value assigned. Therefore, we want to keep this indicator although no statistical deviations could be determined. Hence, the input for the min-max normalization is only random seed declarations with a fixed seed.

To determine a factor scoring we choose: $min(d) = 0$ and $max(d) =$ number of all random seed declarations. If no seed was used, the score is 0, since then there is no control of the starting value of the random number generator.

Model-Serialization & Hyperparameter-Logging

As Figure 6.2 shows, most reproducible repositories used model-serialization. On the other hand, one can also see that hyperparameter-logging was not used in almost any repository of the two sets. We still want to keep this factor, because the tool should not only consider popular technologies or technologies that are often used by reproducible repositories but all helpful possibilities.

We summarize these two factors under this point, as they both use the same feedback generation mechanics. If at least one relevant artifact was detected, the score is 1, otherwise 0.

Out-of-the-Box Buildability

We left this factor out of the analysis because a repository can either be buildable or not buildable, regardless of the measured values. In any case, conclusions about the reproducibility property can only be made to a limited extent with this factor. The main benefit is the simplification of the reviewing process if a Binder build is available or at least possible. In 2.4 we discussed why the use is recommended from a reproducibility point of view. The inclusion in the feedback file is primarily intended to encourage the use of Binder or similar technologies.

6.2.3 Reproducibility Factor Thresholds

Threshold values are used for some factor ratings within the *feedback_builder.py* module to assess whether they correspond to the guidelines or not. This allows us to give context to these scores. Therefore, we calculated the average measured factor scores for both the reproducible and non-reproducible sets, shown in Figure 6.6, and use them as thresholds. This approach is not necessary for the factors of data set availability & preprocessing, model-serialization, hyperparameter-logging, and out-of-the-box buildability since these can only receive scorings that are either 0 or 1.

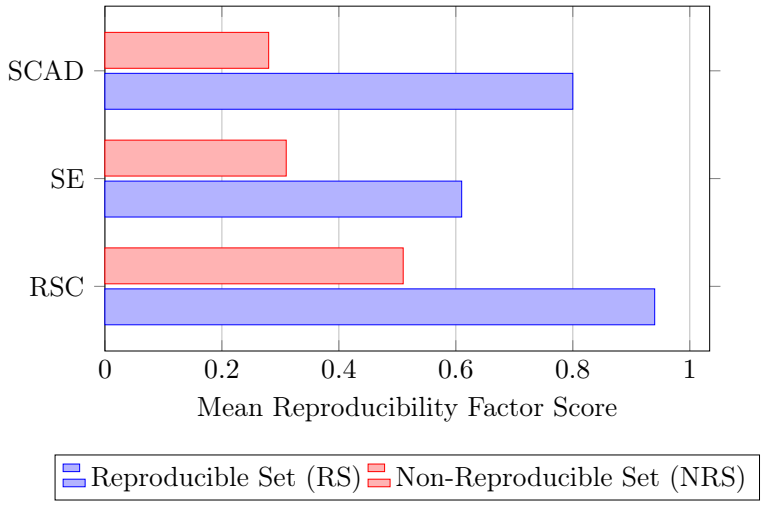


Figure 6.6: Statistical evaluation of the measured factor scores for source code availability & documentation (SCAD), software environment (SE), and random seed control (RSC) for the reproducible and non-reproducible ML repository set. The scores shown are average values of the respective factors in a set. It can be observed that all scorings of the reproducible set are significantly better. Based on these measurements, we set thresholds, which give context to measured values as to whether they are good or bad.

For the factors of source code availability & documentation, software environment, and random seed control we define the following thresholds:

- Top threshold (T): Scores that are above or equal to this threshold indicate compliance with the reproducibility guidelines, as these are at least as good as the average scores measured by repositories from the reproducible set.

- Lower threshold (L): Scores that are below or equal to this threshold indicate non-compliance with the reproducibility guidelines since these are at best as good as the average scores measured by repositories from the non-reproducible set.
- Average threshold (A): This threshold is calculated by the formula $\frac{T+L}{2}$. Values lower than T but above this threshold indicate a rather good factor score. Values that are at most as good as this threshold but are above L indicate a less desirable factor score. Therefore, if a value is between T and L, the average threshold is used to estimate whether the factor conforms more or less well to the guidelines.

Table 6.4 shows an overview of the determined threshold values which we assigned to their corresponding factor.

Table 6.4: Threshold values of the factors source code availability & documentation, software environment, and random seed control.

Factor	***T***	***A***	***L***
Source code availability & documentation	0.8	0.54	0.28
Software environment	0.61	0.46	0.31
Random seed control	0.94	0.73	0.51

6.3 Tool Output

This tool produces output in three different places. Each output serves a different purpose. Most important is the *feedback.md* file which assigns a score and context to the values measured by the tool and provides recommendations.

`Command Line:`

The intermediate results of each analysis step are displayed on the command line. This information can be used for debugging purposes in case something goes wrong.

`Result File (.csv):`

The measured values for all examined indicators as well as additional data points are saved in a ".csv" file. Hence, these values can be saved permanently if one needs them again at a later point in time.

`Feedback File (.md):`

A feedback file is created based on the measured indicator values, representative values, and weights. This includes textual feedback as well as scoring values for both factors and their indicators. The factor scorings can then be interpreted using their respective thresholds. Figure A.1 in the Appendix shows an example file for illustration. Here you can also see that not only the scorings of the factors and indicators are displayed, but also that recommendations are given.

By default, the two output files are saved in the \tmp folder. This can be configured differently in *main.py* if required.

7 Evaluation

In this chapter, we want to present the chosen methodologies with which we evaluate our developed ML repository analysis tool. This is an important step in order to be able to evaluate whether the chosen approach (scoring of reproducibility factors classified as software-based detectable) has a practical use.

We left out the reproducibility factor out-of-the-box buildability in our analysis since the local deployment of BinderHub proved to be too resource-heavy to evaluate the used data sets.

In order to be able to make an educated evaluation, we analyze different data sets generated with our tool. These are described in detail in Section 7.1.

Firstly, in Section 7.2, we evaluate the indicator weights chosen in Chapter 6. To do this, we compare the values we determined through statistical analysis and logical reasoning with those suggested by a factor importance ranking algorithm. This is to examine whether the selected weights are plausible and consistent with these findings.

Subsequently, in Section 7.3, cluster analysis is used to examine whether common key characteristics can be found in different data sets. This allows groups of ML repositories to be identified that were previously unknown.

Finally, in Section 7.4, these data sets are evaluated again using statistical methods to determine the differences between them.

7.1 Data Collection

In this section, the methods used to create the evaluation data sets are presented. We have documented where they can be accessed in the appendix.

`Data for the Evaluation of the Chosen Indicator Weights:`
In Chapter 6, sets containing only reproducible or non-reproducible ML repositories were used to determine representative indicator values and weights. We have merged these sets into one. In addition, some values were also merged. This applies to indicators for which, if any of these are present, the factor score is 1. For example, the indicators that evaluate whether a data set was found became a feature called “Data set found”. Also, we introduced a new label called “reproducible” in this new set, which indicates to which of the two sets an entity belongs. This is also used as the target variable for the feature importance ranking.

`Data for the Cluster Analysis and the Statistical Evaluation:`

The first data set we collected for this purpose contains only ML repositories that are hosted on GitHub. To find them, we searched for the keywords "machine learning experiment" using a crawler and saved the response in a list. These repositories were then checked manually to see whether they belong to a framework or if the source code was created in a programming language other than Python. If so, those entities have been removed from the list. Otherwise, the URL was stored in a separate list.

The second collected data set consists of ML repositories that were linked to an experiment on "PapersWithCode". We searched for experiments on this website that linked the code repository associated with the paper. We included any URL in the list that corresponds to a repository that is hosted on GitHub and where the source code is (mostly) written in Python.

After collecting 100 URLs each, we executed our tool on each entry. A result data set was created for both input lists. These data sets contain not only the measured indicator values for each entry but also their calculated factor scores.

As input for the cluster analysis, all factor scores for each entry of the two sets (GitHub and "PapersWithCode") were saved in a new list. We also included the label "from_set" to record which of the two sets an entry originated from.

For the statistical evaluation, the results of the GitHub set are compared with the "PapersWithCode" set. In addition to the set mentioned for the cluster analysis, we also use the measurements of some indicator values, which were saved separately in data sets.

7.2 Evaluation of the Indicator Weights

For the other evaluation approaches, the calculated factor scores of the tool are of fundamental importance. These scores are calculated by normalizing the measured indicator values, then multiplying them by a specific weight, and finally summing them up (if a factor score consists of multiple indicators). We chose these weights by statistically analyzing two ML repository sets, also taking logical arguments into account. In this section, we compare the chosen weights with those suggested by the feature importance ranking algorithm (on the same data) to ensure that they are well chosen.

We use the approach described in 2.5 for this. There we established that the mean decrease of impurity is effectively a measure of how important a variable is in estimating the value of the target variable. This means that a high mean decrease of impurity indicates a feature that is important for determining the target variable. In the context of this thesis, a feature is a synonym for an indicator.

The data set used for this as input is described in Section 7.1. In order to take stochastic fluctuations into account, we carried out this process 10 times and calculated the average value for each feature. The result is shown in Figure 7.1.

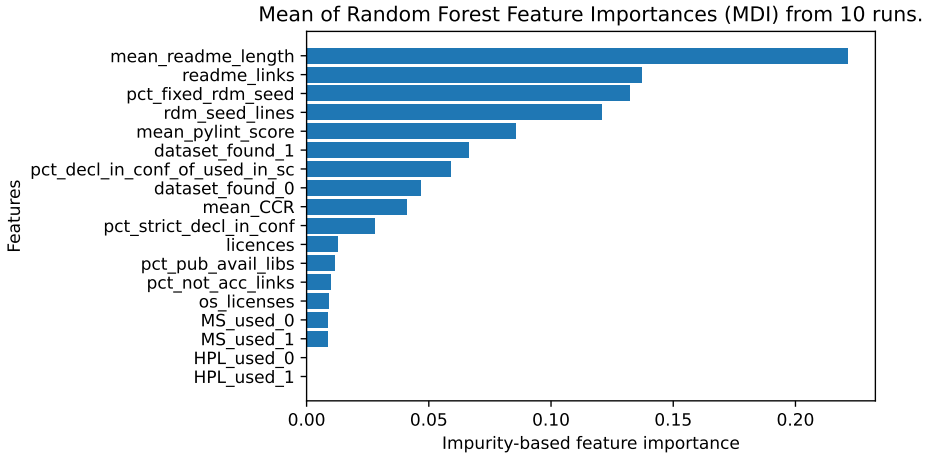


Figure 7.1: Mean decrease of impurity (MDI) for the measured indicators. MDI is a measure of variable importance for estimating a target variable. The higher the value, the greater the influence of the feature when determining the target variable. Categorical features, which in our case are indicators of factors whose score can be either 0 or 1, are presented separately for both cases.

A well-known problem with the mean decrease of impurity is the bias of higher-ranking numerical features. However, for our assessment, this bias is negligible since we want to use this approach to assess the chosen weights of the indicators and not the importance of an indicator on the overall result. But there are approaches to reduce this problem [Li+19]. Categorical features, which in our case are indicators of factors whose score can be either 0 or 1, are presented in the figure separately for both cases. This means that, for example, the value of "dataset_found_1" is the feature importance of "dataset_found" for entities that have a value of 1 for that feature.

It is not necessary to compare the rankings of all indicators to assess the selected weights. Rather, we want to make this comparison for indicators belonging to the same factor. In addition, not all indicators have weights, which is why we are not mentioning these in more detail below.

`Source Code Availability & Documentation:`

This factor is computed using the following indicators and their weights: readme (0.6), license (0.3), pylint rating (0.1), and code comment ratio (0.1). Since the indicators used to calculate the readme score are among the highest-ranked features, its weight appears to be consistent with the results observed here. A clear difference can be seen in the importance of the license compared to the pylint rating and the code comment ratio. However, we want to clearly "punish" non-inclusion or the use of a non-open-source license. For this reason, we consider our chosen weights to be valid despite this discrepancy.

`Software Environment:`

This factor is computed using the following indicators and their weights: % source code imports in the configuration file (0.6), % strict dependency declarations (0.2), and % public accessible libraries used in the source code (0.2). For these indicators, the measured feature importance score is broadly consistent with our chosen values.

7.3 Cluster Analysis

The theoretical foundation of the cluster analysis approach used is explained in 2.5. In this section, we are interested in whether, based on the calculated factor scores of the entities in the input set, different repository groups can be recognized by a clustering algorithm. In addition, we then want to show what insights one can gain from this using a suitable visualization of the calculated clusters.

We use the data set presented in 7.1 as input for the K-Means clustering algorithm.

The only significant parameter for this algorithm is the number of clusters into which the entities should be divided. There are many different methods to determine this parameter [KM13]. We chose the elbow method to determine the optimal number of clusters because it can be performed intuitively using a visual graph evaluation.

The Elbow Method

The elbow point is the k where the decrease in inertia when k is compared to k - *1* is more significant than at other points.

Inertia is the measure of how internally coherent clusters are. The K-Means algorithm aims to choose centroids (which are the mean value of the entities in a cluster) that minimize this inertia. This point can be determined very clearly in some cases. In our case, Figure 7.2 shows the largest decrease in inertia at $k = 2$, $k = 6$ and $k = 8$ (compared to k - *1* respectively). We chose $k = 2$ because this value is consistent with the elbow method and also useful in assessing the *hypothesis* of the thesis. By using this value for k, one can see how many entities of the GitHub or "PapersWithCode" set are assigned to which cluster.

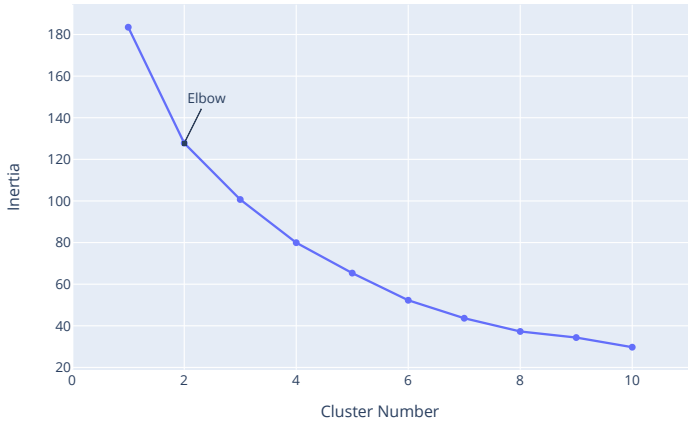


Figure 7.2: Determination of the number of clusters (k) for the k-means cluster analysis using the elbow method. The elbow point is the k where the decrease in inertia when k is compared to $k + 1$ is significantly less than at other points. Inertia is the measure of how internally coherent clusters are. The graph shows the largest decrease in inertia at $k = 2$, $k = 6$, and $k = 8$. We choose $k = 2$ because this value is useful in assessing the *hypothesis* of this thesis.

Result

Based on our findings, which are presented in Figure 7.3, the key difference seems to be the scoring of the random seed control. Also, significant differences in the source code analysis & documentation, the data set availability and the model-serialization factor scores can be observed. The scoring of the software environment is almost equal, with a slight tendency towards cluster 0. Hyperparameter-logging seems to be not relevant.

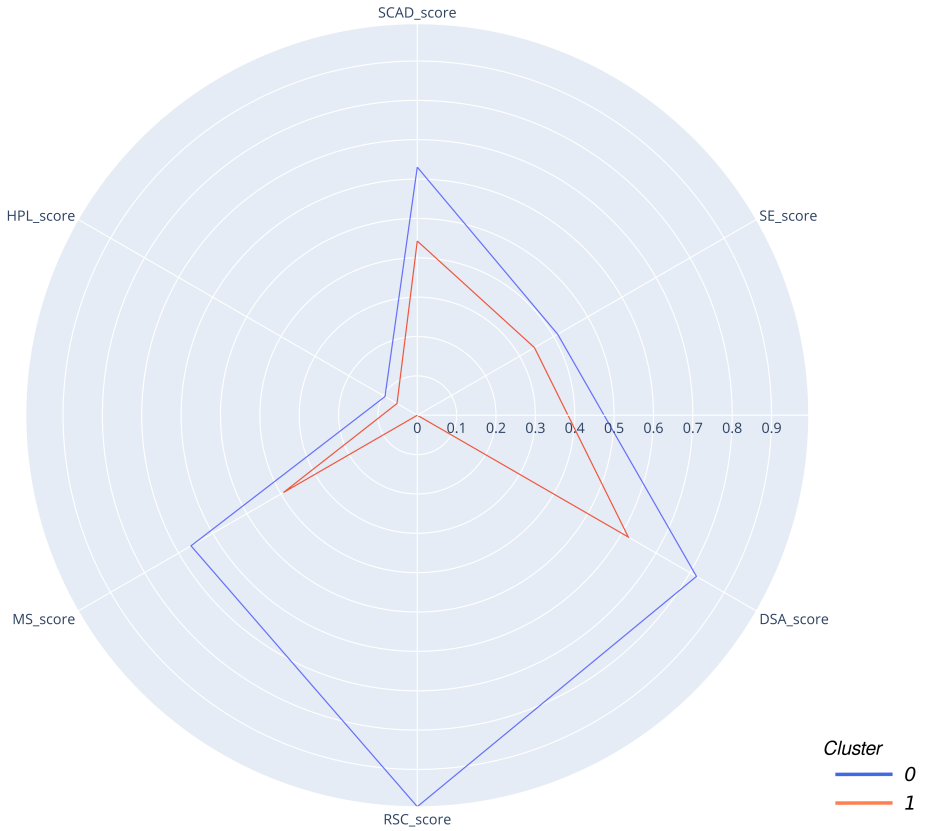


Figure 7.3: Visualization of the cluster analysis result using a polar line plot. Based on this, the key difference between the entities of the two clusters seems to be the scoring of the random seed control (RSC) reproducibility factor. Overall one can observe, that the mean of each factor score is higher in cluster 0. This indicates that these entities comply better with the reproducibility guidelines in comparison with the ones from cluster 1.

Overall one can observe, that the mean of each factor of a repository entity is higher in cluster 0. This indicates that the entities in this cluster might comply better with the reproducibility guidelines in comparison to the ones from cluster 1.

After observing that the entities in cluster 0 adhere better to the reproducibility guidelines, it is important to know how many entities from which sets were grouped into which cluster. The evaluation of this is shown in Figure 7.4. It can be observed that cluster 0 consists of about two-thirds of entities from the "PapersWithCode" set and one-third of the entities from the GitHub set. A similar distribution only with the inverse distribution is present at cluster 1.

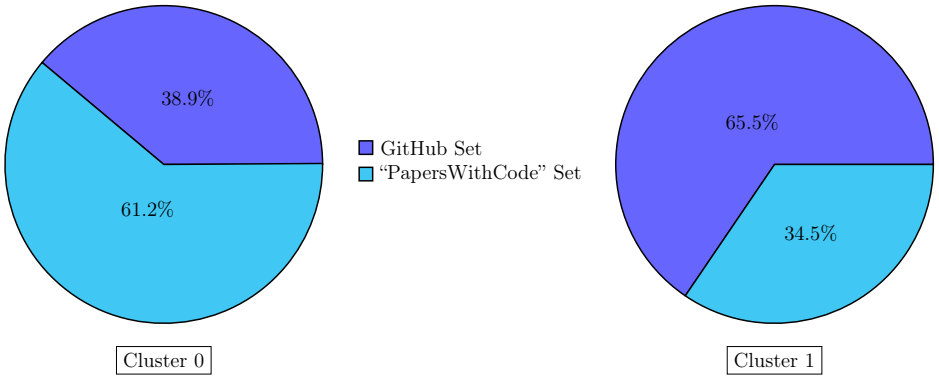


Figure 7.4: Distribution of the entities in terms of their set parentage within the identified clusters. Within cluster 0, about two-thirds of the entities are from the “PapersWithCode” set and one-third are from the GitHub set. This relationship is almost reversed in cluster 1. There, about two-thirds of the entities come from the GitHub set.

In addition, when comparing the mean factor scores of the clusters with the scores from the data sets used in Chapter 6, which contain reproducible or non-reproducible repositories, abnormalities can be observed. This comparison is shown in Figure 7.5. Since we have already established in the cluster analysis that hyperparameter-logging has no significant influence on the grouping, we have not included this factor in the figure.

Cluster 0 has values very similar to the set consisting of reproducible repositories for most factor scores. The biggest differences between these sets are the factor scorings of source code availability & documentation and software environment. For these, however, the difference is clearly noticeable. The entities of cluster 0 have significantly worse scores there on average.

The evaluation of cluster 1 is less clear. For the factors source code availability & documentation and model-serialization, the mean factor scores for entities from cluster 1 are in the middle of the scores calculated for the non-reproducible and reproducible set. Since random seed control is the main grouping factor when assigning to one of the two clusters, the mean score for this is significantly worse than for the non-reproducible set (which has the second-lowest value). For the software environment factor, the score of cluster 1 is similar to that of the non-reproducible set. On the other hand, the score of data set availability is significantly closer to the value of the reproducible set. It is therefore not possible to classify cluster 1 unequivocally. In general, this cluster contains entities that do not comply as well with the reproducibility guidelines as the ones from cluster 0.

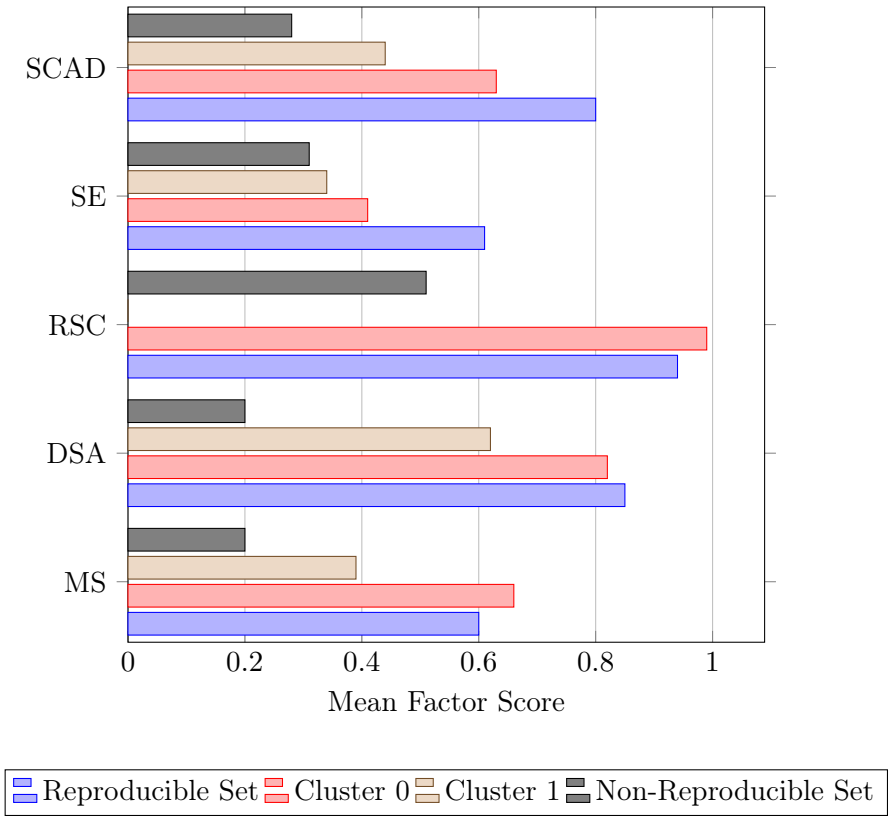


Figure 7.5: Comparison of the mean factor scores for the identified clusters with those of the reproducible and non-reproducible set. It can be seen that cluster 0 scores comparable to the reproducible set for most factors. Cluster 0 is less clearly assignable. In most cases, the values there are between the scores of these two sets.

`Cluster Analysis with Modified Data Sets:`

We performed similar evaluations with subsets of these factors in different combinations to gain additional insights. Excluding the hyperparameter-logging score had no significant impact.

If random seed control is not included in the input data set, model-serialization is the main influencing factor on the grouping. If this factor is also excluded, the grouping is primarily determined by the data set availability factor.

Finally, we evaluated only the factors of source code availability & documentation and software environment. The elbow method recommends a division into three clusters, as shown in Figure 7.6. Compared to the previously tested data sets, this time the optimal number of clusters could be determined with greater certainty using the elbow method since the elbow point is clearly visible in the graph.

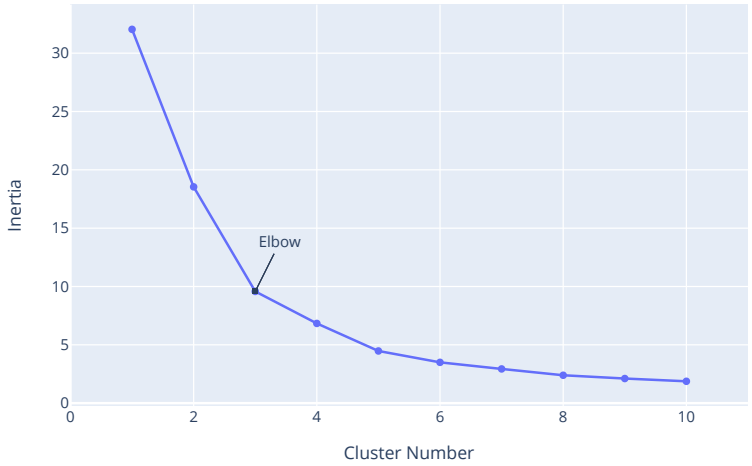


Figure 7.6: Determination of the number of clusters (k) for the k-means cluster analysis using the elbow method and a modified data set. The graph shows the largest decrease in inertia at $k = 3$.

A representation as a polar line plot is not graphically advantageous for this input set. Therefore, we present the results in Table 7.1, which shows the average factor scores for the entities in each cluster.

Table 7.1: Mean reproducibility factor scores for the clusters are determined by the K-Means algorithm. The input set included the factors source code availability & documentation (SCAD) and software environment (SE). It can be seen that cluster 0 contains the repositories that may be less compliant with the reproducibility guidelines.

Cluster	SCAD Score (Mean)	SE Score (Mean)
Cluster 0	0.32	0.22
Cluster 1	0.88	0.32
Cluster 2	0.52	0.80

From this, it can be seen that cluster 1 consists of repositories that score well regarding the availability of source code & documentation, but not so well on the software environment factor. In cluster 2, this relationship is reversed.

However, we were particularly interested in the distribution within cluster 0. Relatively poor scores for both factors were observed for the entities in this cluster in comparison with the other two clusters.

Figure 7.7 represents how many entities of cluster 0 originated from which subset. It can be observed that significantly more entities came from the GitHub set.

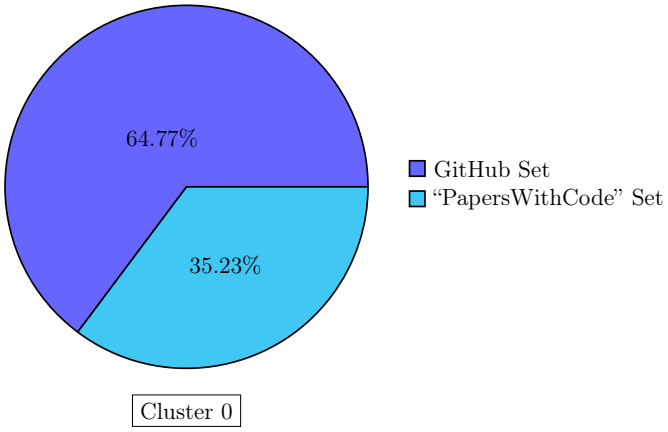


Figure 7.7: Distribution of the entities in terms of their set parentage within cluster 0. It is shown, that about two-thirds of the entities originated from the GitHub set.

7.4 Statistical Evaluation

In Section 7.3, we used cluster analysis to identify key characteristics of entities from the GitHub and the “PapersWithCode” set and to group the entities of these sets based on that. This section looks at the same data sets but with a different focus. Here, we want to show the key differences between the entities of these sets using statistical analysis.

The data sets used for this are explained in 7.1.

Result

In addition to the mean factor scores of the entities in the sets, we also consider the frequency with which certain artifact types are found in them. The results are shown in Figure 7.8. It can be observed that all artifact types are found significantly more frequently in the “PapersWithCode” set. The only exception is the readme artifact. We assume that the reason for the small difference is that it is usually generated automatically when a repository is created by marking a checkbox.

The clearest differences can be observed in the factors model-serialization, random seed control, and license. It is also noticeable that a configuration file was only found in about half of the entities, even in the “PapersWithCode” set.

It is important to mention that this evaluation only concerns the presence of these artifacts. In order to check compliance with the reproducibility guidelines, we also assess the factor scores determined by our tool.

Figure 7.8: Statistical evaluation of the frequency with which relevant artifacts are present in the “PapersWithCode” and GitHub ML repository set. From this, it can be observed that the “PapersWithCode” set contained all measured artifact types significantly more frequently on average.

Table 7.2 contains the mean factor score for all measured factors for the “PapersWithCode” and GitHub sets. The difference in the factor scores of model-serialization, source code availability & documentation, and random seed control in favor of the "PapersWithCode" set is most obvious here. The GitHub set also performs less well in comparison when it comes to data set availability. It was surprising that although the “PapersWithCode” set has a higher mean artifact frequency regarding the configuration file, the associated factor score is slightly better on average for the GitHub set. Overall, however, the "PapersWithCode" set performs significantly better.

Table 7.2: Mean reproducibility factor scores for the “PapersWithCode” and GitHub set.

Reproducibility Factor	**“PapersWithCode” Set**	**GitHub Set**
Source Code Availability & Documentation	0.69	0.41
Software Environment	0.35	0.41
Data Set Availability	0.81	0.66
Random Seed Control	0.71	0.45
Model-Serialization	0.76	0.34
Hyperparameter-Logging	0.14	0.02

8 Discussion

The goal of this thesis, formulated in Chapter 1, was to develop a support tool for researchers and reviewers in the field of machine learning, which checks influencing factors for compliance with the reproducibility guidelines.

This goal resulted in some research questions that had to be answered first. Which factors influence the reproducibility of ML experiments? Which of the identified factors can we analyze with the help of a software tool and which not? How can the factors classified as software-based detectable be measured? To test our created tool, we *hypothesized* that in general, a repository belonging to an ML experiment published by academic publishers ("PapersWithCode") can be expected to adhere to the reproducibility guidelines significantly better than an ML repository on GitHub (on average).

`Discussion of the Research Questions:`

The first research question was dealt with in Chapter 4. Based on a literature search, a total of 13 factors influencing the reproducibility of machine learning experiments could be identified.

Based on this, an approach was presented in Chapter 5, which assigns the identified factors either to the group of detectable or non-detectable factors. This classification is based on indicator-based reasoning. An indicator is a property of a factor that can be measured using a software tool. If an indicator can be determined for a factor, it is classified as detectable.

Based on this classification, we presented a proof of concept implementation of a machine learning repository analysis tool in Chapter 6. This checks a repository for the implemented indicators. Therefore, we had to develop an approach to combine the measured indicator values for the respective factors. Weights reflect the influence of an indicator on the associated factor. By using min-max normalization, the different values could be converted into a common value range. Thus, a score can be formed for each factor by adding the associated weighted, normalized indicator values. By means of comparative values (thresholds), a determined score could be given a context of how it should be interpreted. These comparative values are mean factor scores calculated from two sets containing reproducible or non-reproducible ML repositories. The tool also generates helpful user feedback based on all measurements and scores.

`Discussion of the Hypothesis:`

In order to be able to answer the *hypothesis*, we have carried out various analyzes in Chapter 7 using different data sets.

When performing the cluster analysis, we decided to divide the input set into two clusters. The input set consisted of the measured factor scores, which were calculated for 100 ML repositories randomly crawled on GitHub and 100 randomly collected on "PapersWithCode". We determined this number of clusters by using the elbow method. The difference between

these clusters was the mean factor score of their entities. This difference was most evident for the random seed control. One of the clusters (cluster 0) had significantly better values for all considered factor scores than the other. About two-thirds of the cluster containing the higher scored entities consisted of repositories taken from the "PapersWithCode" subset. For the other cluster (cluster 1), which contained the entities with the lower mean factor scores, the relationship was the inverse. This consisted of two-thirds of entities that were taken from the GitHub set.

When comparing the mean factor scores of these clusters with those that we had determined for the set of reproducible or non-reproducible repositories, it was noticeable that cluster 0 has comparable values to the reproducible set for most factors. For cluster 1, some values were significantly worse than for the non-reproducible set. Others, however, were also significantly better. No clear parallels could be drawn here.

Additional insights could be gained by using a modified input set that included only the factors source code availability & documentation and software environment. Analysis of the cluster containing the lowest-ranked repositories in the comparison revealed that about two-thirds of these entities came from the GitHub set.

After the cluster analysis revealed common key characteristics of entities from the GitHub and "PapersWithCode" sets, we performed a statistical analysis of the mean factor scores and the average frequencies of the relevant artifact types (such as the readme file) of these data sets. This allowed us to examine the key differences between them. Significant differences were also observed here. All examined artifact types were found more frequently, in most cases even significantly more frequently, in ML repositories of entities from the "PapersWithCode" set. The averaged scores were also significantly better for this set for almost all factors. Only the score for the software environment factor was worse compared to the GitHub set, although we have found a configuration file more frequently for entities of the "PapersWithCode" set. We suspect that this discrepancy is because the software environment is for entities of the "PapersWithCode" set disproportionately more frequently documented within the readme (which we do not check in this regard).

Those observations supported our *hypothesis*. After performing the described analyzes of the data sets generated by our tool, we conclude that it seems like a repository belonging to an ML experiment published by academic publishers ("PapersWithCode") can be expected to adhere to the reproducibility guidelines significantly better than an ML repository on GitHub (on average).

Overall, the research questions and the *hypothesis* could be answered in this thesis. Even though our implementation is a proof of concept and therefore does not include all detectable influencing factors, it can already generate helpful feedback when checking an ML repository about compliance with the reproducibility guidelines.

8.1 Limitations

From our point of view, the biggest limitation concerns the size of the data sets used. For the reproducible and non-reproducible sets, manual attempts had to be made to reproduce ML repositories. This is an extremely time-consuming task. Therefore, only 20 entities could be collected in each case in order to be able to comply with the time frame which was available for the creation of this work. This also applies to the data sets which contain random ML repositories from GitHub or "PapersWithCode". There, collecting entities is much easier, but evaluating them still takes some time. For small data sets, outliers have a relatively large impact.

The developed analysis tool can currently only evaluate machine learning experiments that are written in Python and hosted on GitHub. Although this simplifies the implementation, it limits the area of application of the tool.

The correct factor scores can only be generated if the artifacts required for the calculation are recognized by the tool and analyzed correctly. Due to the variety of possible repository structures, files that are not meaningfully named, or possible bugs in our tool, the calculated values may differ slightly from those that a manual check would have resulted in.

8.2 Future Work

The developed tool offers researchers a simple way to check their code repository with regard to the reproducibility guidelines and to identify possible problem areas before the experiments are published. This tool is also helpful for reviewers of third-party work in this area, as it can be quickly recognized whether a repository tends to comply with the reproducibility guidelines or not. This saves both parties time and provides a standardized approach.

In the future, the functionality of this tool can be improved and expanded. Some detectable factors are not yet implemented in our implementation. In addition to Python, other programming languages, or, in addition to GitHub, other repository hosting providers could also be supported.

By collecting more reproducible or non-reproducible repositories, the presented factor-indicator-connection approach could also be refined. Subsequently, the tool could be evaluated on larger data sets to either re-examine our *hypothesis* or to gain new insights.

Another possibility would be to test a combination of the approach mentioned in Chapter 3 to estimate the replicability probability of papers using a machine learning model with the tool we have developed. This could make it possible for not only a repository to be checked for compliance with the reproducibility guidelines but also the associated paper.

9 Conclusion

This chapter concludes this thesis by summarizing the key findings in relation to the research aim and the research questions. We will also discuss the contribution it provides to the machine learning community, including a brief review of the limitations of our findings and possible opportunities for future research.

This study aimed to develop a supporting tool for researchers and reviewers in the field of machine learning, which checks a repository belonging to an experiment for reproducibility. Therefore, we had to investigate which factors influence the reproducibility of machine learning experiments. The identified factors should be classified according to whether they are software-detectable or not. An analysis tool was then implemented for the detectable factors with which a scoring procedure can be used to check how well they comply with the reproducibility guidelines. Our *hypothesis* was that machine learning repositories related to experiments published by scientific publishers, such as "PapersWithCode", are on average better in compliance with the reproducibility guidelines than random ones found on GitHub. Our study results support this *hypothesis* which, assuming the statement made is true, means that our developed tool reflects this difference. Hence, we believe that this tool can meaningfully check the reproducibility guidelines.

So far, to our knowledge, there has not been a tool that analyzes a machine learning repository to make an initial assessment of the reproducibility factors and, based on this, proposes recommendations for action. This enables researchers to check their repository before it is published and, if necessary, eliminate problem areas. It can also help reviewers of third-party work by providing an initial overview of how well a repository to be examined complies with the reproducibility guidelines. An automated check naturally saves time and is therefore resource-saving. This standardized approach also enables the comparison of the metrics between different repositories.

However, the findings are also subject to some limitations. The data sets used were too small to be able to confidently draw generally valid conclusions. In addition, there is still room for improvement in terms of the functionality and scope of the tool. These limitations could be overcome with further research on the tool. In addition, there is another promising approach, with which one can examine a paper for its replicability probability using a machine learning model based on natural language processing. The combination of both approaches seems to us very promising.

In conclusion, one can say that the analysis tool we developed, which implements the approach presented in this thesis, creates seemingly meaningful scores for the detectable reproducibility factors. We believe that this automated way of identifying problem areas in this regard can have a positive contribution to reducing the reproducibility crisis.

Bibliography

[ACP21] Rob Ashmore, Radu Calinescu, and Colin Paterson. "Assuring the machine learning lifecycle: Desiderata, methods, and challenges". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–39.

[ANK18] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. "Machine learning from theory to algorithms: an overview". In: *Journal of physics: conference series*. Vol. 1142. 1. IOP Publishing. 2018, p. 012012.

[Art+21] Nongnuch Artrith et al. "Best practices in machine learning for chemistry". In: *Nature Chemistry* 13.6 (2021), pp. 505–508.

[Bak16] Monya Baker. "Reproducibility crisis". In: *Nature* 533.26 (2016), pp. 353–66.

[BD17] Rabi Behera and Kajaree Das. "A Survey on Machine Learning: Concept, Algorithms and Applications". In: *International Journal of Innovative Research in Computer and Communication Engineering* 2 (Feb. 2017).

[Boe15] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[BSC17] Tom van den Berg, Barry Siegel, and Anthony Cramp. "Containerization of high level architecture-based simulations: A case study". In: *The Journal of Defense Modeling and Simulation* 14.2 (2017), pp. 115–138.

[Cra17] Harry Crane. "Why'Redefining statistical significance'will not improve reproducibility and could make the replication crisis worse". In: *Available at SSRN 3074083* (2017).

[Dey16] Ayon Dey. "Machine learning algorithms: a review". In: *International Journal of Computer Science and Information Technologies* 7.3 (2016), pp. 1174–1179.

[Dru09] Chris Drummond. "Replicability Is Not Reproducibility: Nor Is It Good Science". In: *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML* (2009).

[Eps+18] Ziv Epstein et al. "Closing the AI knowledge gap". In: *arXiv preprint arXiv:1803.07233* (2018).

[For+18] Jessica Forde et al. "Reproducible research environments with repo2docker". In: (2018).

[Gaj+16] Vatsal Gajera et al. "An effective Multi-Objective task scheduling algorithm using Min-Max normalization in cloud computing". In: *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. 2016, pp. 812–816. DOI: 10.1109/ICATCCT.2016.7912111.

[GFI16] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. "What does research reproducibility mean?" In: *Science translational medicine* 8.341 (2016), 341ps12–341ps12.

[GGA18] Odd Erik Gundersen, Yolanda Gil, and David W Aha. "On reproducible AI: Towards reproducible research, open science, and digital scholarship in AI publications". In: *AI magazine* 39.3 (2018), pp. 56–68.

[GHS19] Daniel Greene, Anna Lauren Hoffmann, and Luke Stark. "Better, nicer, clearer, fairer: A critical assessment of the movement for ethical artificial intelligence and machine learning". In: *Proceedings of the 52nd Hawaii international conference on system sciences*. 2019.

[GK18] Odd Erik Gundersen and Sigbjørn Kjensmo. "State of the art: Reproducibility in artificial intelligence". In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[Hea+15] Megan L Head et al. "The extent and consequences of p-hacking in science". In: *PLoS Biol* 13.3 (2015), e1002106.

[Hen+18] Peter Henderson et al. "Deep reinforcement learning that matters". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[Hud21] Robert Hudson. "Should we strive to make science bias-free? A philosophical assessment of the reproducibility crisis". In: *Journal for General Philosophy of Science* 52.3 (2021), pp. 389–405.

[IG19] Richard Isdahl and Odd Erik Gundersen. "Out-of-the-box reproducibility: A survey of machine learning platforms". In: *2019 15th international conference on eScience (eScience)*. IEEE. 2019, pp. 86–95.

[Ioa05] John PA Ioannidis. "Why most published research findings are false". In: *PLoS medicine* 2.8 (2005), e124.

[IT18] Peter Ivie and Douglas Thain. "Reproducibility in scientific computing". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–36.

[Jan20] PS Janardhanan. "Project repositories for machine learning with TensorFlow". In: *Procedia Computer Science* 171 (2020), pp. 188–196.

[KB21] Hyeyoung Koh and Hannah Beth Blum. "Machine learning-based feature importance approach for sensitivity analysis of steel frames". In: (2021).

[Ker+18] Mary Beth Kery et al. "The story in the notebook: Exploratory data science using a literate programming tool". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–11.

[KM13] Trupti M Kodinariya and Prashant R Makwana. "Review on determining number of Cluster in K-Means Clustering". In: *International Journal* 1.6 (2013), pp. 90–95.

[Knu84] Donald Ervin Knuth. "Literate programming". In: *The computer journal* 27.2 (1984), pp. 97–111.

[KP19] Sara Khalid and Daniel Prieto-Alhambra. "Machine learning for feature selection and cluster analysis in drug utilisation research". In: *Current Epidemiology Reports* 6.3 (2019), pp. 364–372.

[KS15] Ron S Kenett and Galit Shmueli. "Clarifying the terminology that describes scientific reproducibility". In: *Nature methods* 12.8 (2015), pp. 699–699.

[Lhe+17] Alexandra L'heureux et al. "Machine learning with big data: Challenges and approaches". In: *Ieee Access* 5 (2017), pp. 7776–7797.

[Li+19] Xiao Li et al. "A debiased MDI feature importance measure for random forests". In: *Advances in Neural Information Processing Systems* 32 (2019).

[Liu+21] Chao Liu et al. "On the Reproducibility and Replicability of Deep Learning in Software Engineering". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–46.

[Mao+20] Ying Mao et al. "Speculative container scheduling for deep learning applications in a kubernetes cluster". In: *arXiv preprint arXiv:2010.11307* (2020).

[McN14] Marcia McNutt. *Journals unite for reproducibility*. 2014.

[Mor+21] Marçal Mora-Cantallops et al. "Traceability for Trustworthy AI: A Review of Models and Tools". In: *Big Data and Cognitive Computing* 5.2 (2021), p. 20.

[OBA17] Babatunde K Olorisade, Pearl Brereton, and Peter Andras. "Reproducibility in machine Learning-Based studies: An example of text mining". In: (2017).

[OBM15] Keith O'Hara, Douglas Blank, and James Marshall. "Computational notebooks for AI education". In: *The Twenty-Eighth International Flairs Conference*. 2015.

[Pan99] Jiantao Pan. "Software testing". In: *Dependable Embedded Systems* 5 (1999), p. 2006.

[PF16] Stephen R Piccolo and Michael B Frampton. "Tools and techniques for computational reproducibility". In: *Gigascience* 5.1 (2016), s13742–016.

[Pin+19] Joelle Pineau et al. "ICLR Reproducibility Challenge 2019". In: *ReScience C* 5.2 (2019), p. 5.

[Pin+20] Joelle Pineau et al. "Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program)". In: *arXiv preprint arXiv:2003.12206* (2020).

[Ple18] Hans E. Plesser. "Reproducibility vs. Replicability: A Brief History of a Confused Terminology". In: *Frontiers in Neuroinformatics* 11 (2018). ISSN: 1662-5196. DOI: 10.3389/fninf.2017.00076. URL: https://www.frontiersin.org/article/10.3389/fninf.2017.00076.

[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)*. IEEE. 2010, pp. 171–180.

[RW18] Benjamin Ragan-Kelley and Carol Willing. "Binder 2.0-Reproducible, interactive, sharable environments for science at scale". In: *Proceedings of the 17th Python in Science Conference (F. Akici, D. Lippa, D. Niederhut, and M. Pacer, eds.)* 2018, pp. 113–120.

[Sch+18] Sebastian Schelter et al. "On challenges in machine learning model management". In: (2018).

[SK21] Sheeba Samuel and Birgitta König-Ries. "Understanding experiments and research practices for reproducibility: an exploratory study". In: *PeerJ* 9 (2021), e11140.

[SN14] Joanna Smith and Helen Noble. "Bias in research". In: *Evidence-based nursing* 17.4 (2014), pp. 100–101.

[SP10] Robin Sommer and Vern Paxson. "Outside the closed world: On using machine learning for network intrusion detection". In: *2010 IEEE symposium on security and privacy*. IEEE. 2010, pp. 305–316.

[SSJ18] K. Shailaja, B. Seetharamulu, and M. A. Jabbar. "Machine Learning in Healthcare: A Review". In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. 2018, pp. 910–914. DOI: 10.1109/ICECA.2018.8474918.

[Sze+17] Vivienne Sze et al. "Hardware for machine learning: Challenges and opportunities". In: *2017 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE. 2017, pp. 1–8.

[Tsi+18] Christos Tsirigotis et al. "Orion: Experiment version control for efficient hyperparameter optimization". In: (2018).

[TVD18] Rachael Tatman, Jake VanderPlas, and Sohier Dane. "A practical taxonomy of reproducibility for machine learning research". In: (2018).

[Vay+19] Leila Abdollahi Vayghan et al. "Kubernetes as an availability manager for microservice applications". In: *arXiv preprint arXiv:1901.04946* (2019).

[Wan+17] Mowei Wang et al. "Machine learning for networking: Workflow, advances and opportunities". In: *Ieee Network* 32.2 (2017), pp. 92–99.

[Wat+19] Junzo Watada et al. "Emerging trends, techniques and open issues of containerization: a review". In: *IEEE Access* 7 (2019), pp. 152443–152472.

[YYU20] Yang Yang, Wu Youyou, and Brian Uzzi. "Estimating the deep replicability of scientific findings using human and artificial intelligence". In: *Proceedings of the National Academy of Sciences* 117.20 (2020), pp. 10762–10768.

[Zel78] Marvin V. Zelkowitz. "Perspectives in Software Engineering". In: *ACM Comput. Surv.* 10.2 (June 1978), pp. 197–216. ISSN: 0360-0300. DOI: 10.1145/356725.356731. URL: https://doi.org/10.1145/356725.356731.

[Zen+22] Zexian Zeng et al. "Statistical and machine learning methods for spatially resolved transcriptomics data analysis". In: *Genome biology* 23.1 (2022), pp. 1–23.

A Code

The source code of the tool presented in this thesis is publicly available in our GitHub repository[1].

In order to visualize the output of the tool which we have discussed in Chapter 6, we have included an example feedback file in this thesis (see Figure A.1).

In addition, examples of all output files generated by our tool are available in the repository under */assets/output_examples*.

[1] https://github.com/martinloos/ml_repository_reproducibility_analysis_tool

Figure A.1: An exemplary feedback ".md" file.

REPRODUCIBILITY FACTOR SCORING (from 0 to 1):

Reproducibility factor	Score	Feedback	T*	A*	L*
Source-code availability and documentation	0.93	Score higher than top threshold (T). Very good.	0.8	0.54	0.28
Software environment	0.93	Score higher than top threshold (T). Very good.	0.61	0.46	0.31
Dataset availability and preprocessing	1	Score equal to top threshold (T). Very good.	1	-	0
Random seed control	1.0	Score higher than top threshold (T). Very good.	0.94	0.73	0.51
Model serialization	1	Score equal to top threshold (T). Very good.	1	-	0
Hyperparameter logging	0	Score equal to lower threshold (L). Major improvements should be made.	1	-	0
Out-of-the-box buildability	0	Score equal to lower threshold (L). Major improvements should be made.	1	-	0

> *: Thresholds computed from respective reproducible and non-reproducible sets of repositories. More information on this in the associated thesis in chapter 6.

SOURCE CODE AVAILABILITY AND DOCUMENTATION FEEDBACK

1. License scoring (score: 1.0 out of 1.0): All found licenses are open-source.
2. README overall scoring (score: 0.93 out of 1.0): See sub-ratings for more information.

- Average length scoring (score: 1.0 out of 1.0): We determined that 82 or more lines are best. We found 222.0 lines (in average) in the README file(s)
- Average accessible links scoring (score: 0.65 out of 1.0): We determined that 4 or more links are best. We found 3.0 accessible links (in average) in the README file(s)

3. Source-code pylint scoring (score: 0.63 out of 1.0): Not normalized pylint rating is 3.61 where 10.0 is best. We found that readable code is best for reproducibility and consider ratings over 5.71 as desirable.
4. Source-code-comment-ratio scoring (score: 1.0 out of 1.0): We measured a average ratio of 6.44. We consider a ratio below 8.73 as best. But more comments are fine too (which would make the ratio smaller). Well documented code is important.

> Source-code availability and documentation is calculated from the above identifiers. Since these have a different influence on the overall result, they are weighted as follows:

- License weight: 0.3
- Readme weight: 0.5
- Sub-readme: Length weight: 0.8
- Sub-readme: Average accessible links weight: 0.2
- Pylint rating weight: 0.1
- Code-comment-ratio weight: 0.1

SOFTWARE ENVIRONMENT FEEDBACK

1. Libraries used in the source-code and mentioned in config file scoring (score: 0.53 out of 0.6): We found that 87.5% of the used libraries are defined in the config file(s). We expect that 100% of the in the source-code used relevant libraries are defined in the config file(s).
2. Strictly defined libraries in config file(s) scoring (score: 0.2 out of 0.2): We expect that 100% of the defined libraries in the config file(s) are strictly (==) defined. We found 31 libraries in the config file(s). From these 100.0% were strictly specified.

- Public available libraries in source code file(s) scoring (score: 0.2 out of 0.2: We expect all of the not local modules or standard python libraries to be publicly available. We found that 100.0% are publicly available. We tested if the used library imports are accessible on https://pypi.org. If the score is not 0.2 (=100%): Please try avoiding the use of not public libraries as third parties may not be able to use your repository. Please note: It is also possible that, if the score is not the maxima, all libraries are publicly available, but we could not find a match. If you are unsure please recheck manually.

> Software environment is calculated from the above identifiers. Since these have a different influence on the overall result, they are weighted as follows:

- Source-code imports in config file weight: 0.6
- Strict dependency declarations in config file weight: 0.2
- Public libs in source code weight: 0.2

> Important note: For our analysis we exclude python standard libraries (see: https://docs.python.org/3/library/) as well as local file imports. We also eliminate duplicates (if one import occurs in multiple files we count it as one).

Figure A.1 Continued.

DATASET AVAILABILITY AND PREPROCESSING FEEDBACK

Dataset availability and preprocessing scoring (score: 1.0 out of 1.0): Dataset file candidate(s) were mentioned in the source code. This is best practice, because one should always provide a dataset (if possible) in the repository in order for others to reproduce your repository with the same input data.

> Important note: Dataset preprocessing detection is currently not implemented. But: If you preprocessed the dataset in any way please make sure to include either the final dataset or files to reproduce the steps taken.

RANDOM SEED FEEDBACK

Random seed lines with fixed seed scoring (score: 1.0 out of 1.0): We found that 100.0% of the 42 found random seed declaration lines had a fixed seed. If the value is not 100% make sure to fix the random seeds.

MODEL SERIALIZATION FEEDBACK

Model serialization scoring (score: 1.0 out of 1.0): Model serialization artifacts found. Model serialization helps in the field of machine learning, where incremental improvements should be documented in order for others to understand the steps taken. We look for one of the following: a) folders named ".dvc", b) files with the extensions ".dvc", ".h5", ".pkl" or ".model" c) one of the following keywords in the source code: "torch.save()", "pickle.dump" or "joblib".

HYPERPARAMETER LOGGING FEEDBACK

Hyperparameter logging scoring (score: 0 out of 1.0): No hyperparameter logging indicators found. We look out for imports of the following libraries in the source code: "wandb", "neptune", "mlflow" and "sacred". These libraries enable the logging of these parameters in order to document them. Also, we are looking for method calls of these libraries regarding logging. Logging changes of the hyper-parameters helps in the field of machine learning, where incremental improvements should be documented in order for others to understand the steps taken.

OUT-OF-THE-BOX BUILDABILITY FEEDBACK

Out-of-the-box buildability scoring (score: 0 out of 1.0): Tested building the repository with BinderHub resulted in an error. Repository is not buildable with BinderHub. Please try fixing this, as it greatly helps reproducing the found results. You can use the free and publicly available infrastructure accessible under https://www.mybinder.org to test. If you have included a Dockerfile in your repository it may not be compatible with BinderHub. Check: https://mybinder.readthedocs.io/en/latest/tutorials/dockerfile.html

B Data Sets

In this chapter, we briefly explain where the data sets used in this thesis are stored in our repository[1]. Also, we have removed the URLs of the repositories from the data sets and made them anonymous. The reason for this is that this data is used to improve the tool and to evaluate concepts or our *hypothesis*. We do not want to call out the authors or developers of these publicly.

The data used in Chapter 6 to calculate the representative indicator values and to justify the weights chosen can be found in the repository under */assets/data_for_factor_indicator_connection*.

The data used in Chapter 7 to perform the feature importance ranking, cluster analysis and statistical evaluation can be found in the repository under */assets/evaluation*.

[1] https://github.com/martinloos/ml_repository_reproducibility_analysis_tool

C Content of the CD

- This work as PDF file – in the folder *PDF*
- The source code of the implementation – in the folder *SRC*
- The LaTeXsource code – in the folder *LATEX*

Declaration of Academic Integrity / Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Mit der aktuell geltenden Fassung der Satzung der Universität Passau zur Sicherung guter wissenschaftlicher Praxis und für den Umgang mit wissenschaftlichem Fehlverhalten vom 31. Juli 2008 (vABlUP Seite 283) bin ich vertraut. Ich erkläre mich einverstanden mit einer Überprüfung der Arbeit unter Zuhilfenahme von Dienstleistungen Dritter (z.B. Anti-Plagiatssoftware) zur Gewährleistung der einwandfreien Kennzeichnung übernommener Ausführungen ohne Verletzung geistigen Eigentums an einem von anderen geschaffenen urheberrechtlich geschützten Werk oder von anderen stammenden wesentlichen wissenschaftlichen Erkenntnissen, Hypothesen, Lehren oder Forschungsansätzen.

Passau, 22. Mai 2022

Martin Johannes Loos

I hereby confirm that I have composed this scientific work independently, without anybody else's assistance, and utilizing no sources or resources other than those specified. I certify that any content adopted literally or in substance has been properly identified. Furthermore, I have familiarized myself with the University of Passau's most recent Guidelines for Good Scientific Practice and Scientific Misconduct Ramifications from 31 July 2008 (vABlUP Page 283). I declare my consent to the use of third-party services (e.g., anti-plagiarism software) for the examination of my work to verify the absence of impermissible representation of adopted content without adequate designation violating the intellectual property rights of others by claiming ownership of somebody else's work, scientific findings, hypotheses, teachings or research approaches.

Passau, 22. Mai 2022

Martin Johannes Loos