POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

On the Challenges of Implementing Machine Learning Systems in Industry

EMILIO RIVERA-LANDOS
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2020

On the Challenges of Implementing Machine Learning Systems in Industry

ACKNOWLEDGEMENTS

I would like to first and foremost thank Foutse Khomh for his help, guidance and trust during my long journey throughout my work. His continuous support and desire for seeing me accomplish my goals have allowed me to present this work.

I also would like to thank Md Saidur Rahman, for his help and consistency during our work with the industry partner: his availability and critical thinking have helped me question my own decisions in a more critical way during our work together – perhaps more than he knows.

I would also like to thank my friends who incited me to purse my master's and with whom I shared good experiences. In addition, I would like to thank all the people involved with AÉCSP throughout the years who made my journey special.

Finally, I want to thank everyone who has helped me directly or indirectly, either in technical or relational aspects, without whom this work would not have come to fruition.

RÉSUMÉ

Dans l'optique de ce mémoire, nous nous concentrons sur les défis de l'implantation de systèmes d'apprentissage automatique dans le contexte de l'industrie. Notre travail est réparti sur deux volets : dans un premier temps, nous explorons des considérations fondamentales sur le processus d'ingénierie de systèmes d'apprentissage automatique et dans un second temps, nous explorons l'aspect pratique de l'ingénierie de tels systèmes dans un cadre industriel.

Pour le premier volet, nous explorons un des défis récemment mis en évidence par la communauté scientifique : la reproducibilité. Nous expliquons les défis qui s'y rattachent et, à la lueur de cette nouvelle compréhension, nous explorons un des effets rattachés, omniprésent dans l'ingénierie logicielle : la présence de défaut logiciels.

À l'aide d'une méthodologie rigoureuse nous cherchons à savoir si la présence de défauts logiciels, parmis un échantillon de taille fixe, dans un cadriciel d'apprentissage automatique impacte le résultat d'un processus d'apprentissage. Nous cherchons à quantifier l'impact qu'a la présence, ou absence, de défaut logiciel sur le potentiel d'apprentissage.

L'application de la méthodologie nous mène à la création d'un cadriciel pour permettre de quantifier avec le moins de variables confondantes les effets du non-déterminisme : `ReproduceML`. À la suite de l'application de la méthodologie, nous n'avons pas été en mesure d'accepter l'hypothèse du test statistique qui permettrait d'établir que les défauts logiciels ont un impact sur la qualité d'un modèle d'apprentissage automatique.

Finalement, nous discutons des effets induisant le non déterminisme dans les sytèmes d'apprentissage automatique et formulons des conseils qui s'appliquent à l'industrie et au monde de la recherche en général.

Pour le second volet, nous faisons un rapport sur les considérations pratiques que nous avons rencontrées dans le contexte d'un travail collaboratif entre notre équipe de Polytechnique Montréal et un partenaire Industriel. Plus spécifiquement, nous expliquons les problématiques auxquelles le partneraire d'affaire a été confronté et détaillons les systèmes d'apprentissage automatique que nous avons conçus afin de régler leurs problématiques.

Nous levons aussi le voile sur les défis techniques et relationels rencontrés lors de l'exécution de ce travail. Suite à quoi nous formulons des recommendations qui se répartissent sur trois axes afin d'assurer une meilleure implantation des systèmes d'apprentissage automatiques dans un contexte industriel, mais qui bénificiera l'ensemble des acteurs qui mettent en place de tels systèmes.

ABSTRACT

Software engineering projects face a number of challenges, ranging from managing their lifecycle to ensuring proper testing methodologies, dealing with defects, building, deploying, among others. As machine learning is becoming more prominent, introducing machine learning in new environments requires skills and considerations from software engineering, machine learning and computer engineering, while also sharing their challenges from these disciplines.

As democratization of machine learning has increased by the presence of open-source projects led by both academia and industry, industry practitioners and researchers share one thing in common: the tools they use. In machine learning, tools are represented by libraries and frameworks used as software for the various steps necessary in a machine learning project.

In this work, we investigate the challenges in implementing machine learning systems in the industry. We first look at one of the computational challenges dealing with machine learning systems: reproducibility and why it represents a significant crisis for current scientific research. We establish the fundamental factors that cause non-determinism in machine learning systems and propose that academia and industry both understand its impact. We then explore a rarely discussed source of non-determinism introducing factors: software versions.

Software products and thus machine learning tools have multiple changes introduced over their development. These changes are made for various reasons: performance gain, new features, defect corrections, among others. As these changes end up in versions and some not, different versions behave differently due to these changes being present or absent.

We explore this source of non-determinism by conducting an empirical study on the impact that one of these changes – bugs – has on machine learning systems. This study aims to quantify the impact that the occurrence of bugs (in a sample of fixed size) in a machine learning framework has on the performance of trained models trained using said framework. Counterintuitively we find that there is no current evidence based on our limited dataset to show that bugs which occurred in the framework PyTorch affect the performance of models.

We then look at the practical challenges based on our experiences in implementing machine learning systems for our industry partner with whom we collaborated to solve transactional problems. We describe the problems that we studied and our machine learning engineering approach to solve them. We share the lessons that we learned during the multiple months over which this project spanned. We also formulate recommendations for researchers and practitioners, to help ease the engineering of machine learning based software systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
ML	Machine Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
API	Application Programming Interface
RL	Reinforcement Learning
DL	Deep Learning
GPU	Graphics Processing Unit
CPU	Central Processing Unit
VCS	Version Control System
NDIF	Non-deterministic Introducing Factor

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

There has been a rise in popularity of machine learning, deep learning and data science in the last few years, due to the increase in reported performance for many real-world tasks. This newly found popularity pressures industries to create and adopt tooling for machine learning in a more data-centric world. However, while models are getting better for canonical tasks, both research and industry face their own set of challenges that directly impact their respective core mission. We examine these challenges of machine learning systems on a theoretical/empirical and an applied standpoint.

Reproducibility and empirical study

In the first part, we explore the theoretical shortcomings and the reproducibility crisis facing scientific workloads, with a focus on machine learning.

We first outline the problem and attempted solutions to the core scientific characteristics — reproducibility, replicability and repeatability — and to software engineering characteristics — determinism — in the field of machine learning. We explore the existing solutions by a literature review and find that there is a need for a better scientific process in machine learning. We also establish the important concept of "Non-Determinism Introducing Factors" as part of the literature review.

With these concepts defined, we study the empirical effect of the presence or absence of bugs in the machine learning framework **PyTorch**. To support further research efforts, we propose a framework for deterministic evaluation of software configuration effects in a real-life, but controlled, environment. Furthermore, we establish a methodology in order to reduce the sources of non-determinism introducing factors in our study and to properly evaluate the effects of bugs in machine learning frameworks. Our study is also crafted to make sure it adheres strongly to aforementioned scientific principles, so that our research is reproducible and extensible.

We then discuss the implications that these non-determinism introducing factors have on academia and industry. Notably, we provide some recommendations for better reproducibility in the machine learning. Practical challenges and considerations of carrying out build processes are also briefly discussed. We also explain the challenges of carrying out empirical version control history mining in open-source software repositories.

Engineering machine learning systems in the Industry

In the second part, we discuss the challenges and recommendations in engineering machine learning systems for the industry, in light of a collaboration between our team from Polytechnique Montréal and our industry partner SAP.

To do so, we first begin by reviewing the literature on the challenges of Industry-Academia Collaborations in the field of software engineering, at multiple levels. We then present the collaboration with our industry partner, by explaining the problematic they faced and how it was addressed by our team by using machine learning models.

Furthermore, we discuss the challenges we encountered during our collaboration in the perspectives of (1) software engineering, (2) machine learning and (3) industry-academia relations. Following these, we propose a set of recommendations for each perspective in order to efficiently engineer machine learning systems.

In the context of this thesis, we also provide new recommendations to the original set that was submitted and adopted by SAP. These additional recommendations pertain to reproducibility and its impact on machine learning workflows, in light of our work.

Chronology

The collaboration between our team and SAP started in fall 2017 and is still ongoing, albeit with different team members and scope of work.

The collaboration effort here expressed is situated between fall 2017 and fall 2018: the contact was first established in 2017, with initial data being delivered. A team member was then added and the official work as described began in 2018, in light of the early prototyping work done in fall 2017.

This period starting in beginning of 2018 up until summer 2018 is the work described for the proposed solution. The recommendations highlighted, however, are the accumulation of multiple years of collaboration and work not explicitly tied to the collaborations, *i.e.,* personal experience.

The empirical study was done in fall 2020.

CHAPTER 2 BACKGROUND

2.1 Machine learning

Machine learning is a field of Artificial Intelligence in which the computer, which is referred as a model, learns to accomplish tasks by analyzing the patterns present in data [2]. Indeed, handcrafted heuristics (*e.g.,* intelligent agents in a video game that use A*) or algorithms to solve a problem can be considered Artificial Intelligence. In machine learning, data is the key ingredient: procedures and algorithms are designed so that the behaviour is modified according to the data itself. Machine Learning (ML) finds its root in statistics and applied mathematics.

2.1.1 Models

We employ the term “model” as a scientific construct that describes how a system behaves. A machine learning model can be described at a high-level as an equation to describe a reality. In the case of ML, a model is a mathematical function: this function can be as simple as a linear combination with a single dependent variable, but typically involves a richer and more complex set of functions and operators. The goal of a model is to serve as a representation of a reality or desired behaviour, in this case described as output data. Canonical and earliest examples of such models are statistical models, *e.g.,* linear regressions and ANOVA can be considered as models. Nowadays, machine learning is mostly regarded as a discipline in the field of data science.

2.1.2 Training and inference

The mathematical equations underlying a model are expressed in terms of parameters and variables. In this context, we refer to “variables“ as traditional “dependent variables”. Parameters are numeric coefficients that interact with the variables to create equations. The process of finding the value for these parameters is known as model “training“ or “fitting”. Within the field of ML, all training processes aims to achieve best performance, *i.e.,* represent accurately a solution to a problem. Specific definition of this performance varies depending on the type of models and its complexity. Generally, this can be viewed as the optimization of a certain criteria; there are various optimization techniques across fields.

2.1.3 Architectures

Traditional statistical models use techniques based on purely statistical methods, *e.g.,* Maximum a posteriori (MAP) or Maximum Likelihood Estimation (MLE) algorithms, *etc.* In this case, a model is modelled as an estimator to which bias and variance are defined by mathematical constructs. Furthermore, when using a Bayesian approach, one needs to postulate a prior distribution for the parameters that govern the model inputs, *i.e.,* parameters. Multiple choices can be used for the prior distribution, depending on the available data. For example, in the task of natural language processing when training to classify texts containing sequences of words to know to which category they belong to, a plausible prior would be a function of the empirical frequency of words. One could also assign values sampled from a normal distribution or even uniform if no information is available or one wants no assumption.

It is important to note that some models have an optimal solution that can be determined mathematically, *i.e.,* finding the optimal solution of a certain task, *e.g.,* PCA or MAP under certain conditions. However, analytical solutions can become intractable as more data is given and complex equations are simply unsuitable for manual analytical computation. Therefore, training procedures can be modelled as an iterative process that minimizes a function. Traditional statistical models such as MAP or MLE can use an iterative process such as Expectation Maximization (EM) to achieve the most probable parameter configuration.

Often, Deep Learning (DL) networks are trained using the concept of *empirical risk minimization* [2]. The idea behind this concept is that we want to calculate the parameters that minimizes the risk of having outputs different that are not compatible from the dataset, *i.e.,* we want to have the output of our model as close to the dataset for the corresponding input. Instead of wanting to know the true underlying *true* distribution of the parameters, we compute an estimate of the divergence of our model by minimizing the *empirical* risk of these parameters giving diverging results: the measure of the diversion of results is known as our loss function. If the mathematical equation has computable derivatives, we can employ methods such as gradient descent to minimize this loss function. The efficient calculation of the gradient itself in DL is done by using backpropagation,

Moreover, the complexity of modern models as seen in DL is based on techniques developed to overcome the need for manual computation of derivatives. The use of auto-differentiation has been a major step in the availability to construct complicated networks.

The process of giving the model data and expecting a result is known as inference. The quality of a model is based on its inference capabilities among other factors.

2.1.4 Advent and differences of Deep Learning

While DL was present in the pre-2000 era, modern DL has been made possible by technological improvements over the years: the ability to ingest and store a vast quantity of data cheaply is one of the reasons DL has been made possible. Furthermore, ability to quickly make calculations on datasets using one or multiple Graphics Processing Unit (GPU) allows for faster training time that would have been order of magnitudes longer if done on Central Processing Unit (CPU). Contrary to traditional ML, DL can create powerful representations from data alone without too much need of the traditional manual feature engineering [2]. Instead, DL leverages its deep architecture to create intermediary representation of the data. Notably, Convolutional Neural Network (CNN)s on modern DL architectures create intermediary representations of edge detection that would have been manually created before, through the feature engineering process.

2.2 On measureability

The scientific process finds its core and credibility by laying down a framework to formulate hypotheses and verify them. An essential characteristic of this process is its validation and acceptance by reaching a consensus, based on thorough methodology and repeatable experiments. There are various levels at which an experiment can be conducted and reenacted by other scientists. We aim here to lay out some core terminology used throughout this work. As there has been discussion about this topic, in literature.

The Association for Computing Machinery (ACM) suggests the following terminology which we follow in this work (version 1.1) [3]:

- Repeatability: “The measurement can be obtained with stated precision by the same team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same location on multiple trials. For computational experiments, this means that a researcher can reliably repeat her own computation.”
- Reproducibility: “The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author’s own artifacts.”
- Replicability: “The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For

> computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently."

Interestingly, ACM did have to change their definitions [3] to provide consistency with the National Information Standards Organization (NISO) for a standard still in progress [4]. With this standard, these aforementioned definitions hold with the current definition from NISO [4], but their definition does not include repeatability. Therefore, throughout this work we will refer reproducibility as being "a peer using author's artifacts" and replicability as "a peer using new artifacts".

2.2.1 Software and computer engineering perspective

A similar but not equivalent concept in software and computer engineering is *determinism*: whether or not a system would give the same output with no changes in the inputs of said computational system, excluding possible external corruption factors. This concept can be applied to multiple problems: automatons, algorithms, verification, among others.

Determinism is a strongly desirable property for software systems and algorithms, as it makes them verifiable thus more maintainable.

As a demonstration, suppose a bug is reported for a certain software module, which affects purely arithmetical operations. This bug happens if the input data has certain properties, *e.g.,* if input's mean is zero (0). A software engineer is interested in fixing this bug: to do so, he *reproduces*[1] the condition, *i.e.,* software environment, in which the bug occurred. In this case, this means that the same input data needs to be used *and* the same version of the software module as reported having failed needs to be used. If the software module is deterministic, the software engineer should observe the same defect, as originally reported.

Let us assume that a second software module does similar calculations, but this time also introduces variance into the output value, by sampling a random value and adding it to the result. Such calculation uses a software module that requires a (pseudo)-random number generator to yield "random" values. At first glance, reproducibility looks harder to achieve since randomness in itself is non-deterministic. However since true randomness generation remains a problem in computer science, pseudo-random number generators are used. These can usually be configured, *seeded*, in order to get deterministic behaviour. Indeed, most programming languages have the concept of randomness seed, which allows to control the original value for a random number generator. The goal of the randomness seed is to ensure

[1]This is a reproducible environment and data scenario. "Replicability" as we defined would be if the arithmetical operations on the input data were coded as a new software module and tested upon.

a consistent initialization value for the random number generation process. That is, given a specific initialization seed, output from the random generation process is deterministic.

When using ML techniques to create models, we integrate a stochastic aspect to our software process. Our assumption is usually that our dataset is observed data coming from a unknown distribution. Having such random variables and making assumptions on data, *e.g.,* i.i.d allows us to leverage statistical methods to create some traditional ML models. Other models rely on other methods that still involve a stochastic process when sampling from an unknown distribution.

As a result, ML workflows are impacted by stochastic processes: in order to obtain a deterministic view of these workflows, there needs to be a control in the random seed of programs.

Multiple factors can impact reproducibility as demonstrated by our simplistic example. ML worklows and libraries are considerably more complex than such a simple setup. Moreover, several other factors can impact reproducibility in computational workflows. These factors are broad and range from low-level implementation to high-level constructs in the computer and software engineering fields. In the following sections, we will explore software and computer engineering concepts related to reproducibility: software, software versioning, data sampling, runtime configuration, computer architecture.

2.3 Engineering of Machine learning systems reproducibility

In this section, we present the various processes that influence the ecosystem of ML. Explanation and laying out this information is necessary to understand the various interactions between components of a ML process designed to be reproducible. ML is used to make observation and inference using random variables and random processes inherent to the world we live in. Modelling uncertainty and randomness has been a core problem in computer science since its creation. Thus, we currently rely on pseudo-random number generators in order to mimic randomness.

In order to properly implement deterministic capabilities in a framework, it is imperative to know how the entire stack of hardware and software modules interact to create an output. Consequently, in this section we offer a brief overview of software and computer engineering concepts and explain how interactions between components affect ML processes.

In the first subsection, we introduce some of the common software-related tools used in ML and how their use can impact reproducibility. Likewise, in the second subsection, we look at how hardware components for computations used in ML affect reproducibility. Finally, we present **PyTorch**, a framework for scientific computation and DL model definition and training.

2.3.1 Software

While ML in itself is not tied to a specific programming language, trends among practitioners has increased the popularity of certain programming languages in this field. Nowadays, the most used programming languages for ML are: Python, C++ and JavaScript according to activity on GitHub [5].

In our work on ML frameworks, we specifically focus on **PyTorch** which uses Python as its interface for building computational graphs. Therefore a brief explanation of Python is necessary.

Python

Python is an interpreted, dynamically-typed programming language. While it lacks performance in comparison to much closer to hardware and compiled languages such as C++, its ease of use makes it a good candidate for quick prototyping and teaching. It is also designed to be easily extendable by users, by building python extensions. The reference implementation of Python is known as `CPython`, implemented in C and Python.

In general, Python compiles to bytecode and is interpreted by a Python interpreter: we refer this environment as the *runtime* of the program. As such, runtime reproducibility entails runtime determinism. In other words, in order to ensure reproducibility in the ML workflow, and therefore in a computational workflow, determinism at the Python interpreter needs to be ensured. Mechanisms that affect the reproducibility in Python runtime are mostly related to the control of randomness within the program. It is partially possible to control determinism by fixing the seed of the built-in random number generator. However other mechanisms affecting determinism in runtimes include thread interactions: when threads within the same runtime create sequence operations whose end results depend on the order of these operations. We will look at such problems in Section 2.3.2.

However, usually, threads are managed and created by the frameworks, by implementing their thread management or by using external dependencies.

Dependencies

A dependency in regards to software engineering is an external module on which a system relies to accomplish some functionality. Indeed, as reusability of software is a desirable and use of external modules proven to be correct is possible, most non-trivial software projects integrate externally defined code into their code base. Using external dependencies therefore becomes a choice between effort, cost, complexity, security and time, among other factors.

In the context of ML, Python's ecosystem offers a wide variety of libraries for data processing and model creation. As most of these libraries and frameworks try to offer the best performance, implementation of ML algorithms exclusively written in *pure* Python are rare. By pure, we mean that there is no external library written in something other than Python called. Indeed, most Python packages for ML and data processing are built with libraries that contain bindings to libraries written in other programming languages – usually C++ – to offer vastly superior performance. However, using externally defined dependencies for a library or framework imposes a choice: using static or dynamic links to said libraries.

Using static linking allows for software releases to be packaged with all the necessary components and core packages at the cost of high binary size. Most software now ship with dynamically linked libraries for core components.

To avoid being overly restrictive and offering poor compatibility across various systems, software makers have the choice to allow certain versions of their applications to dynamically linked without breaking the API. These considerations raise the need for versioning artifacts.

Software releases and versioning

Depending on the type of software, software releases are managed differently. Nonetheless, versioning systems allow a form of contract between the software makers and its users. While this contract is not enforced by entities, it is in the best interest of software makers to make sure changes to the released software does not displease their users and clients if applicable.

A well-known application of this process is the versioning of releases in a semantic fashion: semantic versioning of a system. While formal definition and guidelines of semantic versioning exist, software makers decide how to implement this process. Generally, under this paradigm, software distributors have *major* and *minor* releases – versions. Changes in major version usually indicate that multiple major new features are available under the new version or that there are changes that would break existing systems if they were to replace without modification their external dependency (the newly released software). Such changes are known as breaking changes. As breaking changes break backwards compatibility, they are usually planned along a policy determined by the software makers. Consequently, software makers include documentation on releases to inform end users if an upgrade towards their new version would break existing applications. Traditionally, changes made throughout releases are collected into a `Changelog` that allows for documentation.

Each project deals differently with defects in their release process, but project maintainers usually explain their bug fixes throughout releases in the same manner as a changelog.

This approach is necessary on modern systems since software modules usually need multiple external dependencies. Each of these dependencies has its own development and release process. Consequently, management of dependencies can be complicated and lead to multiple issues if not managed properly: security vulnerabilities, low performance, incompatibility, *etc.*

On the difficulties of releases As such, for a single software module that contains a set number of dependencies, there is a combinatorial quantity of different configurations possible; this depends on various factors.

From the point of view of a software maker, this combinatorial number of configurations to be supported is unmanageable. As a result, software makers usually constrain the version of software they allow to be used with their own software systems. This process is called version constraint. Nevertheless, a software system that is released, *i.e.,* compiled – if applicable –, bundled and distributed represents a single instance of the multiple possible configurations.

While this distribution process is done by software makers with closed-sources, distribution of prebuilt packages is also present in open-source projects. Notably, when it comes to ML, libraries and frameworks that contain external dependencies are partially dynamically linked. By partially, we entail that there are some libraries that are statically linked while other libraries are dynamically linked. The process of building and linking binaries is done by the software makers or an external entity — whose trust ultimately goes to the end-user. However, users also have the choice of building their own version of software. This process is sometimes necessary for end users in order to customize the software, activate parts of the software that were not originally present or simply for security reasons. Most library and framework users do not build their own version of the software and rather rely on prepackaged versions made available by the software maker and/or distributor.

Developers of the software need to build versions for testing the software and implementing new functionalities. While projects have their own development workflow and process, usage of a Version Control System (VCS) is standard.

Version control system

For open-source project, defects are usually reported by opening *issues* on a bug tracking system to advise software developers that there is a problem. The quality and specificity of issues reported in the bug tracking system varies greatly with factors such as project maturity and the user reporting the bug. Consequently, the reported issue goes through a project-specific workflow for assigning a priority for the defect severity. This process is known as triage and culminates in a user working on the issue. This user can be a member of the community or a member of the project itself.

When and if a fix is developed and tested, the changes made by the user that fix the bug are incorporated in the main tree of development. Eventually this fix is integrated in a release, depending on the severity of the issue and the timeline of releases.

In most cases, the process of contribution, *e.g.,* providing a bug fix, goes through review, unit tests and eventually integration: this process is known as merge request or pull request, *i.e.,* integrating — traditionally merging — branches in git. It is important to note that there are various ways to integrate changes into a git history, but merging and rebasing are the main techniques. Each of these techniques has upsides and downsides when it comes to git history management. One other mechanism which is also used in some software workflows is cherry-picking.

All of these techniques also greatly influence the ability of performing searches in a project and thus understanding the workflow and history of a specific open-source project.

2.3.2 Hardware

In this section we present an overview of the use of computer hardware that impact ML workflows and reproducibility.

Acceleration

Several components for ML have been developed by hardware makers to facilitate training and offer a competitive edge. Notably, NVIDIA remains the main provider when it comes to GPU acceleration for DL workloads: various tooling has been provided by the hardware maker. Notably, DL frameworks support `CUDA` based workloads to provide strong acceleration, whereas direct support for OpenCL is non-existent in TensorFlow or PyTorch. `CUDA` is a computational library made by NVIDIA in order to launch highly parallel programatic computations on their GPUs, and is used throughout most DL frameworks to make computations.

When it comes to other types of workloads, hardware manufacturer Intel has released libraries to improve ML-based workflows with tools such as MKL [6], MKLDNN [7]. These respectively allow for optimized common operations of general mathematical functions and DL-related operations on CPU. By using functions from such libraries, programs can leverage optimized Single Instruction Multiple Data (SIMD) operations and specialized CPU instructions to gain performance. Furthermore, these libraries can use parallel thread-based computations based on non CPU-specific instructions: these parallel threads open the door for floating point operations non-determinism.

Open source projects aiming to leverage hardware acceleration for ML-workloads are also seen in projects such as openBLAS, CPU acceleration of linear algebra using architecture specific vectorized instructions.

Tools mentioned are almost all used in modern ML-based workflows for training and even data processing; they constitute external dependencies for software-based ML frameworks and libraries.

ML frameworks often opt to include support for multiple of the above tools in order to support a wide user-base. Consequently, when packages for such frameworks are built and distributed, they come linked to these externally provided vendor libraries, see Section 2.3.1. Dynamic linking to these libraries creates challenges, as there is no strong guarantee that everyone has the same version of these external libraries, *i.e.,* that symbols have not changed causing crashes upon execution. Fortunately, with the use of semantic versioning these difficulties can be reduced, provided libraries follow a documented approach to releases as described in Section 2.3.1.

Floating point operations

Floating-point operations can constitute a serious problem in ML and computational workflows, as calculations are prone to underflow or overflow, if care is not taken when implementing operations. Moreover, floating point operations can lose precision depending on the ordering of instructions.

Representation and precision Bit representation of decimal points in computers can vary from one computer architecture to another. However, a standard for representing floating point numbers for multiple binary sizes is described by IEEE-754 [8]. There are some problems when using floating points as described by this standard: precision of the result and sequencing of operations. Because there is a limit on the precision on the number themselves: when using a fix number of bits, it is impossible to represent an infinity of values using traditional computations. While the IEEE-754 standard offers a way to resolve rounding, it also entails that operations are non-associative [9]. In computational workloads, this means that the mapping of operation unto each core can affect the final result, *i.e.,* additions are not deterministic [9,10]. As ML frameworks use highly parallel environments to execute their operations, this means that the result of floating point arithmetic depends on the number of cores that are used. These are runtime and hardware-related sources of non-determinism.

Quantization While IEEE-754 mandates a representation of floating point numbers and a range of multiple bit widths [8], there has been a push for bit width calculations lower than standard minimal 16-bit (half) precision. The main argument for using lower bit width in DL, is a lower total size of model parameters [11] and faster inference, due to faster operations on hardware [12]. Both of these can decrease the computational energy needed, thus is suited for mobile devices [13]. This, however impacts the precision of number possible to represent.

Underflows and overflows In the more specific case of ML, operations that act on computations with small and big values have the potential to overflow or underflow. For example, when normalizing values coming from a Deep Neural Network (DNN), *e.g.,* calculating a softmax, the operation can easily overflow, if not implemented properly. Moreover, the nature of the calculations done in DL can also exhibit some shortcomings: the *vanilla* RNN is famous for its vanishing gradient problem due to successive matrix multiplication of small values.

For most common operations, frameworks provide operations that have been created so that they leverage the best performance by using computer architecture-specific instructions and accuracy. However, for problems regarding floating point precision, the runtime is a big factor and cannot be controlled by frameworks.

2.3.3 Frameworks

In this section, we present the base idea of a ML framework. We describe **PyTorch** at a high-level as it is the framework that will be used for our study in Chapter 4.

Here, we define a ML framework as a package or set of packages, *i.e.,* artifacts, that allows for end-to-end training of ML models. Therefore, libraries to provide scientific calculations such as **numpy** are not considered to be frameworks since more tooling is necessary to create ML models in a streamlined way. Nonetheless, ML frameworks do use such libraries in order to leverage reusability, at a cost that was explained in Section 2.3.1.

Popular frameworks for machine learning are **TensorFlow**, **scikit-learn**, **PyTorch**, **MXNet**, **CNTK**, **Keras** among others. These frameworks allow for the definition, creation and realization of a training procedure of a ML model.

In this thesis, we analyze specifically **PyTorch**, but the considerations for non-reproducibility are similar for most other ML frameworks, as their goal remains similar.

PyTorch

PyTorch is an open-source computational and DL framework hosted on GitHub. The project's first release dates back to 2016 and currently contains 30,000 commits over its 4 years of development. While it is open-source, the project is maintained principally by Facebook employees.

At its core, **PyTorch** is composed of two parts, with support for hardware acceleration across its operations. Its first part consists of a C++ library for manipulating tensors, *i.e.,* multi-dimensional data, for computational purposes as what is referred as "computational workloads." Its second part is a module for creating and training DL models. The DL workload functionality is supported by their implementation of automatic differentiation [14].

Computations on **PyTorch** are done by manipulating dynamically a Control Flow Graph: a graph where nodes are operations and vertices are tensors (data). More importantly for our work, **PyTorch** supports multiple hardware to make sure computations on tensors are fast: they do so by referring them as "computational backends". GPU support is boasted on their GitHub page, since most DL workloads run much faster on GPUdue to their architecture. In Section 2.3.2, we explained how certain hardware acceleration software modules were designed by manufacturers. In order to accelerate their computations, **PyTorch** also uses other libraries such as **MKL**, **MKLDNN**, **OpenMP** to carry parallel computations, even when using a GPU, notably to accumulate (reduce) values in parallel. These computations are prone to the floating-point loss of precision and thus non-determinism explained in Section 2.3.2. Several other libraries are also used to make **PyTorch** work.

PyTorch makes use of these libraries in order to make sure that the tensor operations run optimally on most platforms. This also means that the codebase must interface with said external libraries: this is an important factor impacting reproducibility therefore our work, as the considerations about external libraries were explained in Section 2.3.1.

Furthermore, **PyTorch**'s release and management processes and special considerations will be discussed in Chapter 4, for our empirical study of the impact of bugs.

CHAPTER 3 LITERATURE REVIEW

This section aims to review the current literature on the two main themes of this work. First, we will review the state of reproducibility in ML, by explaining its importance, what factors affect reproducibility and how current tools and studies try to address the issue. In the second section, we review the literature on Industry-Academia Collaboration and present the general recommendations for efficient collaboration.

3.1 Reproducibility in machine learning

This section lays the foundation of our work for Chapter 4 in which reproducibility is at the core. Indeed this section will shed light on the various factors that affect determinism in a machine learning context. These factors directly impact the ability to conduct empirical studies in machine learning.

We first review the concept of reproducibility in science and showcase the "reproducibility crisis". Then we review the literature in order to demonstrate that there is a critical need specifically in machine learning for reproducibility and repeatability. Then we introduce the concept of "non-deterministic introducing factors" which affect the ML workflow at many levels and that is key to improving reproducibility in this field. We also review existing works that treat reproducibility on three different levels: (1) the existing works on ML Non-deterministic Introducing Factor (NDIF), (2) the benchmarking tools adapted to ML and (3) the tools that exist for reproducibility.

Finally, we showcase the existing works that are related to our study of machine learning bugs.

3.1.1 Reproducibility crisis

The reproducibility crisis is acknowledged by many contributions from the literature: these come from many fields [15, 16] such as ML [17, 18], medicine [19–22] and other disciplines such as psychology [23], archaeology [24], among others. The general terminology for reproducibility include: "replicability", "reproducibility" and "repeatability". These terms have been defined and used multiple times throughout the literature, but the desire for the paramount need of reproducibility in science is not questioned [20].

In light of the need for reproducibility, many propositions for research guidelines have emerged by showcasing reproducible research [17], creating guidelines for research formulation [25], guidelines for reproducible research [26–29] or even by modelling research [30, 31] and its workflows [32, 33]. We discuss many of these guidelines in Section 3.3.3. They are meant to advance the state of reproducibility in ML and science.

Multiple efforts have been made to enable a rigorous science for ML [25]. Furthermore, works by Vanschoren *et al.* try to standardize and centralize research results and processes for ML by constructing "experiment databases" [31], allowing for researchers to explore and query results from ML findings. Building on this work, Vanschoren *et al.* create **OpenML**, a platform allowing for sharing datasets, implementations and results among ML researchers [34]. Using such platform has the benefit of improving reusability and reproducibility of previous ML research.

Interestingly, the topic of reproducibility in "medical"-related fields has raised some relevant research: in the (bio)-medical field [19, 21, 22, 35–37] and neuroscience [27, 38].

There is a need for benchmarks in the machine learning community, as papers have shown that trivial changes in non-reported configuration for papers can lead to different results [37, 39]. There is also a call from scientists to use open source software for machine learning [40], as the use of undisclosed/private work hinders progress.

3.1.2 Calls for reproducibility in machine learning

Work by Pham *et al.* [41] studies the effects of non-determinism on ML training procedures: they show non-determinism can cause total failure of a training procedure (final accuracy of a model below 20%). Even when removing such extreme cases of training failure, they still report a potential difference in accuracy of up to 10.8%. Furthermore, when controlling as much as possible the sources of non-determinism – by setting special runtime configurations–, they still observed a 2.9% variation in accuracy across their experiments, which they attribute to "*implementation-level* non-determinism": non-determinism inherent to their libraries and computing platform. Therefore, they recommend that when papers state improvement of state-of-the-art models that are less than this measure, care should be taken".

Pham *et al.* also surveyed the literature to understand the state of reproducibility and repeatability in software engineering and artificial intelligence, and found that 19.5% of papers tried to use identical runs when reporting their results [41]. These findings outline the need for reproducibility in the field of machine learning, as well as the need for awareness on factors that influence the ML workflow. We discuss in more detail the work by Pham *et al.* in Section 3.3.1

Reproducibility has also been in the spotlight in reinforcement learning, as several papers aim to raise awareness about it. Notably, a literature survey conducted in 2019 on conference papers show that only 5% of the papers applied a significance testing methodology: this survey was reported by Alberti *et al.* [42] but the original source is a talk by Pineau [43] promoting reproducibility in reinforcement learning. Following the same trend, there are discussions for better workflows in reproducibility [44].

Similarly, a recent analysis of code inclusion in ML papers found that 25.8% of DL papers include code [45]. The proportion is 50% for reinforcement learning [42]. While code inclusion in a research paper goes one step further for reproducibility, it needs to be usable. Thus, a study by Gunderson [46] goes a step beyond to introduce degrees of reproducibility, and find that most papers in artificial intelligence are not fully reproducible. A similar finding in studies for ML applied in medicine shows a 9% reproducibility [19].

There are also papers calling for better reproducibility, that share the same desire to have better reproducibility in ML [18, 47] and DL [38, 48].

Most of the above papers from the literature measure one way or the other reproducibility based on the criteria of obtaining reproducible results or the availability of artifacts from research. Reproducibility of the results is directly impacted by the underlying causes of variance: “non-determinism introducing factors”.

3.2 Non-deterministic introducing factors

In this section, we present the various *controllable* factors that cause non-determinism specifically in regards to ML. These factors are referred as “non-determinism introducing factors” and are borrowed from Pham *et al.* [41] and adapted with findings based on Crane [49] and Guo *et al.* [50]. Each of these non-determinism introducing factors find their non-determinism in some root factor: we explain these in Section 3.2.1. Then we define a list (3.2.2) of factors influencing the ML workflow that can be controlled and that enhances reproducibility in Section 3.2.2.

In the context of machine learning, we define “Non-Deterministic Introducing Factors” as any change in the ML lifecycle that directly or indirectly affects the training or inference results due to non-deterministic behaviours.

In other words, NDIF are factors that are important to consider to be able to *reproduce* an experiment.

In this section, we discuss the causes of NDIF only in the context of *computational output* reproducibility. We do not consider other forms of reproducibility, *e.g.,* resource performance reproducibility. All the factors described below also do not mention obvious differences that can affect the output, such as running *drastically* different hardware, software versions and hyper-parameters.

3.2.1 Root causes

This section discusses the root factors that introduce non-determinism in computational, thus ML workflows. While they are causes for non-determinism, they do not carry a negative connotation and are by-products of how computer science, computer and software engineering have dealt with fundamental problems.

Randomness We explained a basis for random seed and determinism in Section 2.2.1. The random number generation has an impact on all aspects of random-based calculations, thus determinism.

Versioning As we saw in Section 2.3.1, the number of possible configurations for a specific set of dependencies is combinatorial, *i.e.,* it is the product of each dependency's allowed cardinality. As such, if no "pinned" versions are set, a specific version of a program might exhibit erroneous behaviour, while another valid configuration might not, as different software releases carry a variety of changes, such as bug fixes.

Floating point operations As we saw in Section 2.3.2, floating point operations are non-associative, thus the ordering on multiple threads affects the result of operations, potentially inducing non-determinism when parallel computations are done. Furthermore, the data itself has a representation with an associated precision: lower precision number and calculations affect the final output, thus influencing the capability to *reproduce* and possibly impeding the capability to *replicate.*

3.2.2 Controllable confounding factors

In light of the defined root causes for non-determinism in computational workloads, we propose a revised list for terminology to encourage reproducibility. We explain how these NDIF are tied to their root causes. Our list of controllable factors that introduce variability includes the following concepts:

- Random Seed
- Model definition

- Software versions
- Threading model
- Runtime
- Hardware
- Data

Random Seed In this context, it is important to remember that there are various random seeds: at least one for each runtime. Furthermore, a runtime's dependencies, such as external libraries, may need to be configured before or during the runtime, as it may not be attached to the program's runtime. Notably, when executing a program and using a random number generator, the program's associated runtime will check to see if a special configuration was done. This special configuration is called "seeding" the number generator. Providing a specific value for the seed should generate the same random number from the generator, thus enhancing reproducibility.

Model definition While a model is generally static in its definition, there are some parts that may be affected by the random number generation. In the case of DL, model definition as a NDIF can be showcased by the presence of some layers that are subject to randomness effects, such as Dropout methods [51].

Threading model The main reason why a threading model can affect non-determinism is the non-associativity of floating-point operations. Therefore, training a model with multiple threads can affect the output of floating point calculations and therefore resulting in non-deterministic behaviour.

Software versions Software versions encompasses two aspects that are different: (1) direct software dependencies of the program and (2) software versions that are available in the runtime. While versioning is also a cause for non-determinism, it is possible to mitigate its effect by "pinning" versions for an experiment.

Runtime A runtime is the environment in which a program is executed. This can span multiple "layers", *e.g.,* a Python interpreter and virtual machine are runtimes for the Python language, which runs on an operating system. Each of these layers might run with different configurations affecting the output. As an example, a specific runtime might be built with support for double precision while another might only perform standard precision operations. As previously stated, the computational runtime in which runs a program also dictates how the pseudo-random numbers will be generated.

Hardware The hardware used for a specific computation may yield different results in computational workloads for a variety of reasons. Fundamentally, different hardware might use different architectures which use different instructions which might yield different floating point precision. Specific implementation might offer different results due to specific instructions being available for specific hardware instructions, having specific hardware support for data types *e.g.,* half and double precision floating point support, or even using different than IEEE-754 representations [52, 53].

Data split While using a specific dataset already split in training, validation and test, there is another factor to take into account: the inner ordering of each of the splits. For analytical solutions, this might not pose a problem, but for networks trained with "mini-batches", the size [54, 55] and order [38] affect the training. Moreover, for algorithms using optimization techniques such as Stochastic Gradient Descent (SGD), ordering of data affects the convergence rate, as samples are stochastically sampled, thus dependent on the randomness. While SGD is demonstrably stable [56], it does not guarantee determinism in output model function; it rather bounds the error. Furthermore the data shuffle can affect the convergence rate when using SGD [57].

3.2.3 Summary

We establish three root causes of non-determinism in computational and ML workloads: (1) randomness, (2) versioning and (3) floating point operations. These impact multiple aspects of the ML worklow by potentially introducing non-determinism: we establish a list of 7 controllable factors that should be disclosed when publishing material as each of these factors are a source of potential non-determinism. Fig. 3.1 establishes a mapping between these factors.

While all these NDIF *can* cause non-deterministic computations, it can be infeasible to duplicate every aspect. In fact, an exact duplication is highly unlikely and is the reason why standards exist, *i.e.,* defining a standard and allowing implementations that guarantee they respect the standard. The goal for *reproducibility* is thus to reduce possible NDIF in order to obtain same results *within the stated precision* [3]. Consequently, a proper calculation of the results with statistical analyses is paramount for reproducibility in computational sciences

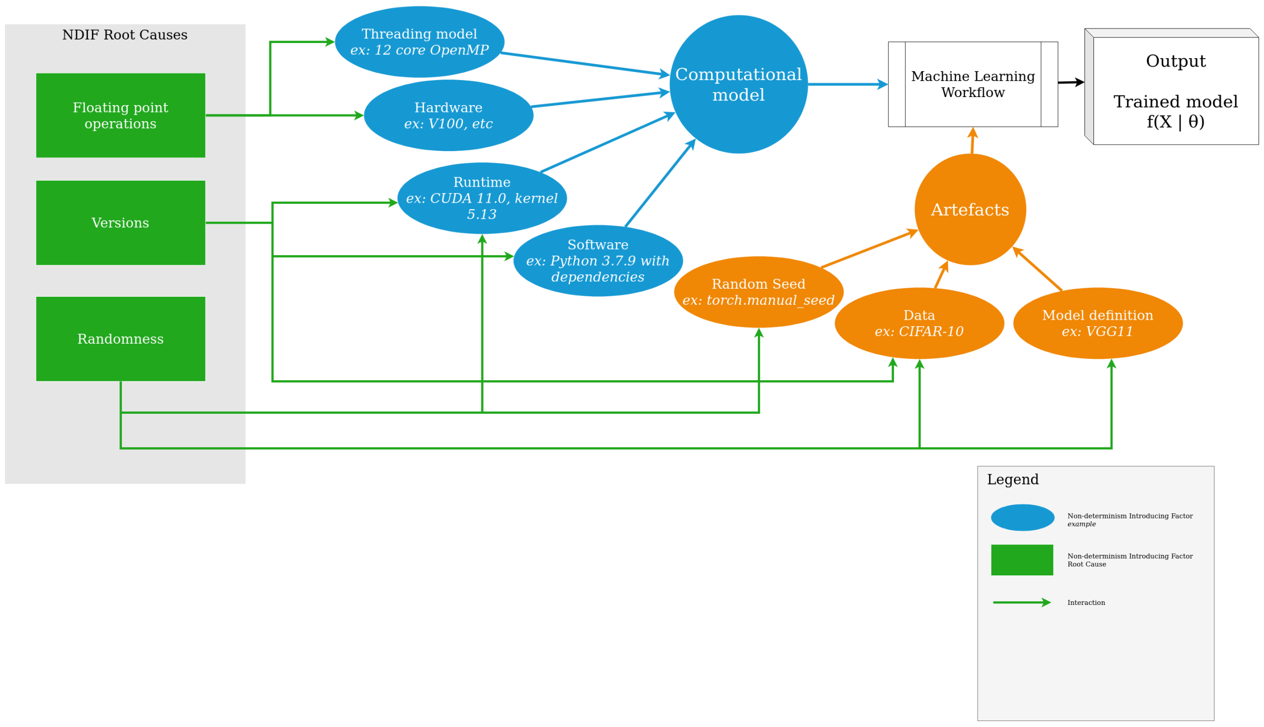


Figure 3.1 Non-determinism Introducing Factors sources and interactions in the machine learning workflow

and thus ML.

3.3 Existing work on reproducibility for Machine Learning

In this section, we detail the various works found in literature and in software that are closely related to ML reproducibility. First, we review studies that directly deal with NDIFs and measurement of their impact. Then we review studies and works that indirectly deal with NDIF , *i.e.,* benchmark tools. Finally, we look at the various tools that exist in research for opportunities in reproducible ML.

3.3.1 Measuring the effects of non-determinism

In this section, multiple works from the literature are presented. Some of them explicitly deal with the measurement of NDIF while others obtain measurements on NDIF indirectly. Several of the works introduced here are also explained in the benchmarking Section 3.3.2.

A recent study done by Pham *et al.* [41] directly showcases the problem of variance in

ML systems by measuring the impact that a random seed has on a model's performance and measuring the knowledge among practitioners on NDIF through a conducted survey. To do so, they run multiple models on the same task with a specific random seed. They introduce, to our knowledge, the term "Non-deterministic Introducing Factor" (NDIF), which are the various sources of randomness that can affect a ML workflow. First they measure performance differences of up to 10.8% in terms of accuracy when no effort is done to deal with NDIFs. But most interestingly, when taking precautions by setting the random seed and disabling other NDIF, the range is still substantial: up to 2.9% of difference can be found, with an empirical standard deviation ranging from 0.1% and 0.7% (depending on the model used). They also measured accuracy variance across ML frameworks when setting a specific seed, but did not find that each of the frameworks had different variances [41]. However, they did find that when using different low-level libraries (GPU-related) versions, difference in accuracy can reach 2%. The conclusion is thus that current level of NDIF can empirically produce up to 2.9% variation in accuracy even when using the same seed and configuring the code to act in the most deterministic way; two statistical tests are used in order to assert at a 90% confidence-level. As for the survey, the results show that 83.3% of participants were unaware or uncertain of implementation-level NDIF [41]. Similarly, they find that 31.9% of the surveyed people are unaware of variance in ML, while 21.8% are unsure about variance presence in DL. These survey findings indicate the need for more awareness in the community regarding variance and reproducibility.

Another study from Guo *et al.* [50] tries to measure the performance difference in multiple DL frameworks: in order to do so, they create similar experimental runs across frameworks and conduct a statistical analysis to measure the impact of choosing a specific framework on accuracy. They also investigate the robustness of models trained by each framework against adversarial inputs. As with Pham *et al.*, they select the best model out of multiple training runs. While Pham *et al.* used 16 runs to select the best model [41], the study by Guo *et al.* used the best out of 5. They also notice that across different frameworks, by using the same configuration, the weights and biases obtained by the models are different, thus their relative performance is different: they attribute this to differences in inner implementations of computations [50]. More specifically, **PyTorch**'s training results were more stable than **CNTK**, **TensorFlow** or **MXNet** [50]. This relates to the *computational model* definition of NDIF(see Section 3.2.3). Furthermore, they research the effect of quantization, *i.e.,* using lower precision tensors in order to calculate results, on the performance of models deployed on mobile devices. While they did not observe statistically significant impacts of quantization on the real dataset inference results, they did find statistically significant differences on *adversely generated examples* [50].

When it comes to variability in results stemming from non-deterministic behaviours, several works in reinforcement learning point to big differences in performance metrics [58] where "a single source of non determinism" could lead to drastically different results. Nagarajan shows that these impacts appear to be more accentuated in the later phases of learning. Interestingly, a similar study but with CNN contrasts these findings by stating that changes in random seed affects mostly the early phases [59]. This might indicate an avenue for further research.

A poster presented in the context of neuro-engineering data consortium also highlights the issue: authors set the same random seed in order to measure difference in performance between different random seeds, and between **PyTorch** and **Tensorflow** [38]. They show a 7% difference between the random seeds; which is similar to the aforementioned results. More interestingly, they look at the "sensitivity to Data Ordering" and show that there is as much as 3% difference when changing the order in which the data was fed.

Measurement of NDIF has also been conducted indirectly, *i.e.,* by works that aim to create benchmarking tools: we will discuss these in detail in the next section.

3.3.2 Benchmarking

Benchmarking is the process of evaluating and comparing several artifacts based on certain criteria. This is done to provide a fair comparison between artifacts. By its nature a benchmarking tool or suite is designed to provide a *fair* comparison by giving the same task to multiple artifacts. Multiple criteria can be assessed, but measurements and outputs are quantitative. Therefore, the act and process of benchmarking needs to be as *reproducible* as possible. Depending on the determinism and experiment variances that are present due to methodology, benchmark results are often repeated and averaged in order to compensate for these variances.

Other types of benchmarks measurements are also often considered. The performance evaluation, which captures a framework's ability to train models rapidly is a key metric at heart of benchmarking. It is in fact the main goal of benchmarking.

In the following section, we explore the various benchmarking tools available for ML and DL.

A recent comparison of popular DL frameworks is done in [60] in which they compare the performance on CPU and GPU as well as accuracy. Comparison is made with the same datasets and model architectures while leaving all default framework configurations. While authors disclose their hardware and use it consistently throughout their experiment, no random initialization control is made. To compensate, they average 3 runs for the same experiment.

Statistical relevance of results therefore is probably lacking. Moreover, as no control on randomness is done, some results about accuracy and convergence rate should be taken lightly. There is a GitHub repository that seems to contain the configuration and versions used to measure the results.

Another similar benchmark framework **DawnBENCH** [61] mainly focuses on *end-to-end* performance and high-level accuracy for the popular frameworks **PyTorch** and **TensorFlow**. The project aims to improve repeatability and proper benchmarking in DL workflows. In addition, they add *cost* as a metric. This project captures the essence of a benchmarking tool, but no control on randomness is done.

A study by Coleman et al. analyzed the output of runs executed on **DawnBENCH** to measure optimization repercussions [62]. The key metric in this work is a concept known as *time-to-accuracy* (TTA). This measure is the amount of time a model needs to be trained in order to achieve a certain level of accuracy. Within their analysis they did not use the same seed and observed that varying the randomness seed does not change the *end TTA* result, stating variance of 5.3%. Some may argue that 5.3% is in fact a big gap: indeed, in the search for deterministic and repeatable ML workflows, this variance needs to be diminished.

The same team behind **DawnBENCH** also created an updated benchmarking tool **MLPerf** whose purpose is to measure pure performance (*i.e.,* speed) to reach TTA [59]. In their initial findings, they conclude that hyperparameters sharing, *i.e.,* using the same hyper-parameters across different systems, is *relatively* portable.

Moreover, they state that small changes in the randomness seed with the same hyperparameters affect the final results, mostly *early* during the training phase.

In order to more adequately leverage the benchmarking process, they strictly prohibit data augmentation.

On the hardware side of tools, **Fathom** [63] was developed to measure difference in performance in order to help hardware makers optimize. Notably they implement reference DNNs *workloads* so that hardware makers can know if their changes have an impact on performance. Such a framework could also benefit from our work, since repeatability is at the core of benchmarking and measuring optimizations.

Similarly, a project called **BaiduDeepBench** [64] is created to correctly compare hardware in DL workloads. In this case, the main goal is to know which hardware gives the best performance for DL workloads. Rather than comparing models and frameworks, this is done by implementing as-close-to hardware-level instructions. Such project has the advantage of not being tied to any framework and measure a stricter set of operations. As DL frameworks use the same constructs to leverage hardware-acceleration, overall effectiveness should transpose to the real-world use cases.

Another tool called **TBD** aims at benchmarking performance out of DL workloads [65]. **TBD** implements state-of-the-art models for ML frameworks **TensorFlow**, **MXNet** and **CNTK**. Authors also build tools to perform "end-to-end analysis" and provide recommendations for optimization in implementation of the frameworks.

Liu *et al.* talk about design considerations that need to be studied when creating a benchmarking tool for deep learning [66]. They find that runtime configuration as well as data configuration impact the performance of a model. The goal of this paper was to exhibit this behaviour so that makers of benchmarking tools take into account that these factors do in fact impact the output and performance.

A comparison of popular DL frameworks by benchmarking the performance is done by Shi *et al.* [67]. They demonstrate and explain why there are differences in training time for the same model within different frameworks. Notably, the presence or absence of certain CPU instructions in a version of **TensorFlow** would drastically alter the performance in comparison to other frameworks. This confirms our intuition that framework dependencies have a direct impact on runs, based on solely compilation factors, as authors note that a subsequent version of **TensorFlow** contained the fix to this.

Similarly, a comparison of different DL frameworks has been done by Guo *et al.* reporting that **PyTorch** performs better than others while using the same seed [50]. However, each model was trained 5 times as cross-validation and then benchmarked in inference of 10000 data points, which leads us to question the statistical significance of their results.

3.3.3 Tools

Each of the tools presented in this section aims to either encourage reproducible ML workflows for the ML's practitioner or to help conduct reproducible research. Each of these affects the ML workflow in one way or another.

Collaborative research There is a promising tool for managing the ML workflow in an ecosystem called **MLFlow** [68]. Its main goals are to simplify the lifecycle of a ML project by wrapping over existing processes. Such encapsulation of the lifecycle, they argue benefit the ML practitioner. Moreover, reproducibility is boasted as one of its four challenges addressed. **MLFlow** allows for tracking of experiments metrics and configuration via file configurations.

A similar idea for research experiments databases was proposed by Vanschoren [31] in order to promote collaboration. While **MLFlow** only adresses part of the requirements of Vanschoren, anecdotally it seems to have a higher popularity. Work on experiment databases by Vanschoren goes a step further by proposing to create ontologies for experiments, in *whichever* field of research. By using an ontology using RDF descriptors, they provide a powerful capability on meta knowledge of the ML workflow and propose a *sample* modelling language for describing it: **ExpML**. Using such ontology for describing the ML lifecycle would in our opinion greatly help reproducibility and overall knowledge of the applied ML workflow.

In the idea of promoting research reproducibility and disclosure of data **Zenodo** was created [69]. Zenodo encourages researchers in publishing artifacts from research on their platform in order to achieve "open science". Guidelines for reproducible research mention sharing artifacts as a guideline explicitly [70].

A survey was conducted by Isdahl comparing the existing ML "platforms". These platforms are ecosystems for managing the ML workflow in some way or another [71]. They rank these platforms by comparing their "out-of-the-box reproducibility". They show that most of these ecosystems offer some degree of reproducibility, but none of them achieves total reproducibility. Therefore, reproducibility is not yet completely managed by a single ML ecosystem and it is up to practitioners and researchers to make sure their work is reproducible.

Data versioning There has been some demand for data versioning systems [72, 73]. Data Version Control (DVC) is a tool that aims to facilitate scientific experimentation in ML, both in terms of shareability and reproducibility [73]. They use a variety of techniques and tools to make sure experiments can be easily shared and reproduced. They version data, models by using configuration files, without tying themselves with a specific run configuration, thus making it framework-agnostic. Our work, `ReproduceML`, resembles closely DVC as we also use configuration files to control reproducibility, notably for datasets. Our approach differs in that it acts more as a central database for collection and reproduction of runs. Moreover, we aim at ensuring that the *runtime* on which experiments are executed are tightly controlled, in order to collect measurements. Our work could use DVC in order to make sure data stays the same between experiments.

Other tools Other tools such as **COBRA** [74] aim to create reproducible setups of workflows by installing tools via a command line interface. A tool called **ReproZIP** is meant replicate experiments across various computers by capturing dependencies using tracing technologies [75]. They argue that configuration and moving from a research environment to a computational environment requires manual changes that discourage users from creating reproducible setups. There are certain ecosystems that have been developed for reproducible computations in the cloud, namely **MULTI-X** [35], **PRECIP** [76], and other unnamed solutions [77]. Other solutions such as **CoLoMoTo** are worth mentioning [36].

3.4 Research on machine learning bugs

In this section, we present the studies that are related to bugs in ML. Particularly, we will pay attention to the empirical studies that classify and measure the impact of bugs in ML programs. There have been two types of empirical studies conducted on the effect of bugs related to ML frameworks: studies on the bugs of the framework itself and the impact of the bugs on *end-user* code. Works on bugs in ML frameworks have been done on the past, but literature is rather sparse. There is, however, a vast literature for bug classification and prediction using ML techniques [78].

For research strictly analyzing only the bugs in frameworks, works by Sun *et al.* [79] and Thung *et al.* [80] are the main works on this topic in the literature. Both approaches mine the information from the source code and the associated version control system in order to create metrics and provide insights based on manual analysis. The former work considers **scikit-learn** while the latter examines Apache projects **Mahout**, **Lucene** and **OpenNLP**.

The other area of focus for research in ML is the impact on end users. Islam *et al.* have done research on understanding and classifying the "nature, root cause and impact of bugs" in DL frameworks [81]. They updated an existing taxonomy which was used for software engineering purposes. Zhang *et al.* classify the bug symptoms and causes [82] in *end-user* code, *i.e.,* repositories that use ML frameworks. Their classification of symptoms falls within the taxonomy of Islam *et al.*, if one does not consider their "unknown" category.

3.5 Industry-Academia Relations

The collaboration between industries and the academic world has been researched in multiple fields, not only software engineering. In this section, we present the takeaways from literature on collaboration between the Industry world and the Academic world. Such work has been labelled in many ways throughout the literature: Industry-Academia Collaboration, University-Industry collaborations and the most general research goal is to better to interaction between both parties, known as bridging the gap, or even "chasm" as the gap can be seen by many as considerable.

Improvements on collaborations between industry and academia have been made possible by several publication approaches. One of these approaches is the publication of case studies in which Academia publishes reports on the experiences and outcomes of the collaboration. Next, literature review of case studies is a way to acquire insight from all papers that were published in the context of Industry-Academia Collaboration. Researchers identify and output recommendations based on the content of the published material. Such works were conducted by Garousi *et al.* [83] and Carver *et al.* [84]. One key factor here is that there might be sampling bias in the findings reported by the resulting papers. Therefore, a more global approach tends to be survey-based. Surveys are created to explore a research question on Industry-Academia Collaboration and are sent out to researchers. Results are then gathered and analyzed to produce an answer to the research questions. Such work is conducted by Garousi *et al.* [1] and tend to summarize quite well the current state of Industry-Academia Collaboration since there is less selection bias and data is available to verify the results.

Another approach to measure and present results coming from Industry-Academia Collaboration projects is the Collaborative Practice Research introduced in 2002 by Mathiassen [85]. This approach fuelled academia to use it as ways to bolster better research and collaboration. Notably, this approach has been studied and applied in studies such as [86].

In the following sections, we present the general topics of research in Industry-Academia Collaboration pertaining that are linked to our own industrial collaboration.

Agile Methodologies A practitioner notes that the nature of an Industry-Academia collaboration project tend to be "waterfallish in nature" when they span over multiple years [87]. He notes that industries are more familiar with agile methods favouring shorter periods of work in contrast to a waterfall approach. Research on the length of Industry-Academia projects seem to vary depending on the location and other factors. Indeed, as seen in collaborations between industry and academia within countries like Canada and Turkey, most projects span several years [83]. Arguably, the number of projects in that study that reported length is quite low (12 out of 33). A more recent study done by the same authors on a wider set of industry-academia relations seem to indicate, however, that most projects are rather short [1]. Nonetheless, there is empirical evidence showing that collaboration between industry and academia are fortified by recurrent synchronized collaborations, whether it being by using an agile methodology or recurrent activities such as workshops [1, 83, 88]. This addresses one of the top challenges for a successful industry-academia project relation, *i.e.,* "integration into daily work" of the industry partner [89]. Sandberg outlines that collaboration between ITU and its industry partner Ericsson is most successful if researchers use an agile methodology by recommending that research questions aim to produce artifacts quickly [86]. They also state that delivering an answer to a research question seems to be optimal after 6 months using such agile workflows. Building on this work, guidelines for agile methodologies are presented by Sandberg [90]: the use of SCRUM methodology was seen by both parties as good by 89% of the concerned people. A common challenge for agile collaboration in research projects is the availability of research students participating in such collaboration environments [89, 90]. Notably, Runeson says on this topic having a more flexible schedule indirectly contributes to a more successful project, by ensuring time is spent by students in the industry [91]. This finding goes in hand with the challenges outlined by Wohlin [89] and a case study in which, as highlighted by Carver *et al.*, where a lesson learned was that "personal commitment was needed from both the researcher and the industrial test manager" [84].

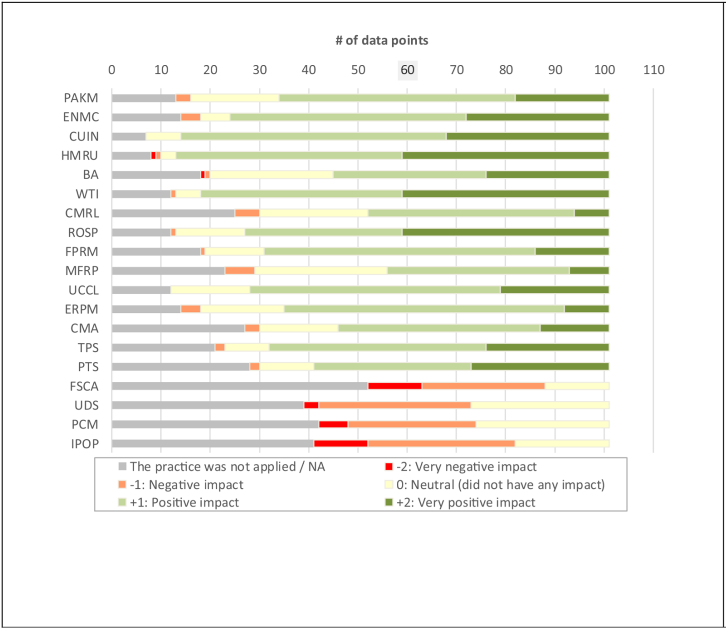
The need of champions Work done by Wohlin *et al.* establish that there needs to be a person on the industry side that drives the project: a "champion" [92]. This person must want the project to succeed and must provide the necessary help to the academic team. In this context, a "champion" differs from a "stakeholder": the champion must actively have stakes and will for the project's fruition, *i.e.,* the simple act of appointing a specific person as a stakeholder does not guarantee this. Evidence of the importance of a champion is presented in [88], where it is stated that "the active involvement of the company's CEO was of key importance for the success of the project."

Academia challenges Researchers are expected to have multiple qualities to ensure a better project outcome. Notably, soft [93] and social skills [89, 92] are seen as good factors towards building a successful relation and thus project. In projects that are geared towards solving business problems, Palkar indicates that students in computer science and engineering that participate would greatly benefit from knowledge in business [93]. We believe this statement is a specific instance of the more general concept of having non-technical skills.

Importance of non-technical factors Several challenges have been reported by practitioners and academia, and suggestions for improvement have been made, as seen throughout the previously mentioned paragraphs. It is interesting to notice that non-technical factors are considered to be the most impactful factors in the success of these industry-academia collaborations. While this may be a bias in literature, *i.e.,* the fact that technical challenges are not reported, a collaboration is purely a relationship construction. It thus makes sense that most challenges are on the non-technical side.

The largest current work on case studies reviews for software engineering Industry-Academia Collaboration is made by Garousi *et al.* [1]. In this work, 101 projects are analyzed to find and rank patterns and anti-patterns affecting Industry-Academia Collaboration. Moreover, much like Wohlin [92] they output evidence-based recommendations for successful collaboration on the basis of the observed data. A summary of answers can be found in Fig. 3.2. The top three practices that *on average* have a positive impact on collaboration are relational-based: "(1) working in (as) a team and involving the "right" practitioners, (2) having mutual respect, understanding and appreciation, and (3) considering and understanding industry's needs, and giving explicit industry benefits.". In the same work, for anti-patterns, one of the top anti-patterns can be tied to a technical aspect. Notably the following anti-patterns constitute the main three practices hindering collaboration: "(1) poor change management, (2) ignoring project, organization, or product characteristics and (3) following a self-centric approach". Unsurprisingly, the recommendations made by Garousi *et al.* in this work to avoid these anti-patterns are respectively to: (1) be flexible to change, (2) pay attention to both the industry partner and academia's needs and organizational factors by encouraging the use of soft skills, and (3) for researchers to be open to criticism, *i.e.,* avoiding a self-centred approach.

Thus, we can see throughout multiple sources that a practitioner needs to have not only technical skills, but also relational skills [1, 93]



Patterns:

- Proper and active knowledge management (PAKM)
- Ensuring engagement and managing commitment (ENMC)
- Considering and understanding industry's needs, and giving explicit industry benefits (CUIN)
- Having mutual respect, understanding and appreciation (HMRU)
- Being Agile (BA)
- Working in (as) a team and involving the "right" practitioners (WTI)
- Considering and manage risks and limitations (CMRL)
- Researcher's on-site presence and access (ROSP)
- Following a proper research/data collection method (FPRM)
- Managing funding/recruiting/partnerships and contracting privacy (MFRP)
- Understanding the context, constraints and language (UCCL)
- Efficient research project management (ERPM)
- Conducting measurement/ assessment (CMA)
- Testing pilot solutions before using them in industry (TPS)
- Providing tool support for solutions (PTS)

Anti-patterns:

- (Anti-pattern): Following self-centric approach (FSCA)
- Unstructured decision structures (UDS)
- Poor change management (PCM)
- Ignoring project, organizational, or product characteristics (IPOP)

Figure 3.2 Reported impact of practices in Industry-Academia Collaboration projects [1]

Carver *et al.* report specific instances in which ultimately successful Industry-Academia Collaboration projects progress can be hindered by non-technical factors; in fact, the lessons are almost all of a relational nature [84]. Moreover, they found that frequent communication with stakeholders was a key factor in establishing the relationship between the industry partner and the academic team. A case study with a medical industry partner also highlights some of the challenges in the industry -academia relationships: they give three recommendations in order to improve communication between the parties, encouraging training, documentation, and monitoring of the developed work [88].

Other factors to consider Data privacy and security are important factors to consider when starting a project since they can be the reason for cancellation of ongoing projects [1]. Moreover, these factors can potentially hinder research findings publications in academic journals because industry may want to retain intellectual property over the project and its artifacts. These are some of the reasons why reproducibility can also be hard to attain [70].

However, there is a growing need for a consolidated set of guidelines regarding software engineering for machine learning. We aim to contribute in filling this important gap by reporting about our experience building ML software components in an industrial context.

CHAPTER 4 EMPIRICAL STUDY OF THE IMPACT OF BUGS ON THE QUALITY OF MACHINE LEARNING SYSTEMS

In this chapter, we explore the theoretical challenges affecting the engineering of machine learning system and the ML workflow and how NDIF factors in. First, we assess the impact of bugs in the **PyTorch** framework on model performance. In order to do so, we create a generic methodology for measuring the effect of changes in ML frameworks and apply it on bugs present in **PyTorch**. The methodology has multiple steps in which systematic treatment of data is encouraged and is designed to promote reproducibility. We also discuss the various challenges in the reproducibility of ML workloads and their repercussion in the process of engineering ML systems.

We study and answer the following research questions:

RQ1 Given a sample of fixed size of recent bugs in a given machine learning framework, does the occurrence of bugs affect the performance metrics of machine learning models?

In order to answer this question, we also formulate the following research objectives:

RO1 Create a methodology to measure the impact of bugs in machine learning frameworks.

RO2 Create a framework that minimizes the effects of non-deterministic introducing factors.

RO3 Create the tooling towards systematic reproducible empirical software mining

Structure This chapter is organized as follows: In Section 4.1, we provide a brief summary of the context of this study. In Section 4.2 we present our designed methodology from data collection to the specific statistical analysis needed to answer our research questions. In Section 4.3, we introduce `ReproduceML`, a framework for reproducible machine learning training and inference. In Section 4.4, we apply the designed methodology on **PyTorch** and discuss the main considerations for applying the methodology. In Section 4.5, we conduct the statistical analysis of the results. In Section 4.6, we explain the main limitations and threats to validity of this study. In Section 4.7, we discuss the results, their impact on the ML workflow. We also comment on reproducibility of this study and the main challenges for conducting empirical studies of the same nature, and its impact of Industry and Academia. In Section 4.8, we list and explain a list of future works, mainly in light of the considerations and challenges discovered, while also providing improvements for the methodology.

4.1 Context

Frameworks and libraries in machine learning have various sources: products that were once used for internal research in companies are now distributed to the open-source community (e.g., Tensorflow and **PyTorch**). Such tools and frameworks are now becoming an essential tool for the ML practitioner. As these frameworks are developed and become stable, they are versioned and distributed, thereby becoming software products. As all software, they are susceptible to having bugs introduced in their codebase which by definition directly impact their usability.

Bugs in machine learning have been studied in empirical settings [79,80,82], but such studies have been limited to understanding the origin of the bugs and providing a classification [81].

The empirical impact of bugs on the machine learning process, however, has not been studied before, although works by Crane [49], Pham *et al.* [41] and Guo *et al.* [50] conducted studies on ML frameworks by using techniques to reduce the amount of NDIF.

Motivation In the machine learning research process, industry and academia both have a common task: improving the performance of their models rapidly and efficiently. Both use commonly available tools to carry out their experiments in ML: frameworks. However, as these products are software artifacts, they also exhibit their same potential for having defects. Quantifying the impact of these defects in the process of ML therefore allows to take educated decisions on the versions used to develop and deploy a ML system. Understanding the effect of defects in ML frameworks on the models will also allow us to gain understanding on the robustness of models as well as prioritize the bugs present in the codebase of such frameworks. The robustness of a model can be explained as its ability to adapt to defects. As ML models are mathematical equations, if the defects affect operations within the model, the resulting overall equation might behave similarly while some of the "inner" operations might change. However, while the model may behave differently internally, its output may remain the same. In a hypothetical case, we imagine a neural network that has L2 penalty applied to all the neuron's parameters. If a bug affects the distribution of weights very weakly, say for rounding errors in arithmetic and exponential calculation, the weight of the models might deviate from their correct calculation. However, after passing the past neurons' output into the current neuron, the resulting activation function threshold might still be met, therefore not affecting at all the model's ability to learn.

Such theoretical cases, while good for improving one's understanding of theoretical foundations of a model, have not been measured in practice and with changes that are empirical rather than purely theoretical. Therefore, the goal of this study is to shed light on these issues by providing empirical evidence with real bugs coming from a popular machine learning framework. A by-product of this study is also gaining knowledge on how open-source (but maintained by companies) software evolves over time.

Applicability on other frameworks While the methodology allows for any "change" to be used, a proper analysis for the effect of bugs needs access to the source code of the framework and access to the bug information. Having clear information on these aspects allows for better information gathering that will be used in the filtering part of the methodology. Moreover, the software development process of the framework using a VCS greatly affects the applicability of this methodology. Indeed, using best practices such as indexing and addressing issues in an issue tracking system and creating small well-defined fixes in commits to fix these issues allows for a systematic way to find where the bugs where fixed. We discuss the challenges and particularities of applying this methodology to **PyTorch** in Section 4.7.

4.2 Methodology

This section describes the general methodology applied in order to measure the impact of *code changes* (and specifically bug fixes) in a ML framework. The main idea is to obtain versions of a framework before and after a change is introduced and measure the impact of the change based on an evaluation procedure we define.

First, we define a *change* as the difference between two accessible *versions* of a software artifact: note that this definition does not mention releases. Indeed, a *change* is merely the difference of the software at two points in the lifetime of the software. Practically, a change is a difference between *versions* in a VCS, *i.e.,* a difference between two "commits" in `Git`. Note that a change can span multiple individual commits, as is the case for changes between releases.

The presence and absence of a bug, by the process of bug fixing can be seen as a change: therefore this methodology applies directly to bugs; since the overall idea is to quantify the impact that the presence or absence of a bug has on performance, we use two (2) different versions of the framework. The first version containing the bug is denoted as "buggy" whereas the version that does not have the bug is denoted as "corrected". While the steps taken are framework-agnostic, there might be subtle differences to take into considerations, notably in data collection and data filtering processes.

Generally, for an end-user of a library, the buggy and corrected versions would be separated by an official release of the software. However, while applying the methodology using official releases would be much faster and accessible, by doing so we would not measure *only* the bug presence, but also all the other changes that were bundled in the same version. Therefore, we opt not to use official releases. **We rather compare on the absence and presence of a bug by obtaining an artifact, *i.e.,* a build of version, with and without the defect's presence.** In either case, buggy and corrected versions are both `Git` revisions. Consequently, for each bug the key information is its buggy and corrected versions.

We use the terms "revisions", "commits" and "version" interchangeably without loss of generality. In the context of this study, acceptable can be different from state-of-the-art performance. Indeed, it is not currently uncommon to use various ensemble techniques to boost model performance. It is important to note that achieving the best possible performance is not the goal of this study nor a requirement: measuring the difference in performance is. Nonetheless, models are to be trained in the same way and achieve relatively close performance to their state-of-the-art counterparts.

In the current study methodology, we propose to set the "buggy" and "corrected" versions as respectively, the *last version* that contained the bug and the version that contains the bugfix. Therefore, the bugs are separated by a single commit

The general process consists of the following steps: (1) Data collection, (2) Data Filtering, (3) Artifact collection, (4) Experimental runs and (5) Statistical analysis. These steps are generic and can be applied to any ML framework, however special considerations might lead to small experimental differences from framework to framework.

The methodology is designed to reduce the amount of confounding variables and non-determinism, and thereby variance. The details of each step are found in Section 4.2.2.

4.2.1 Concept and Definitions

An experiment constitutes the whole metric collection process of a training procedure executed multiple times, each repetition is being denoted as a *run*, for a specific version of the framework. In this context, a training procedure aims to recreate as close as possible the process of learning used in modern machine learning; a run's output is a trained model that gives acceptable performance along with the metrics calculated against a test set. An *experiment* is uniquely characterized by the attribute in table 4.2. More specifically, a training procedure (run) is the set of steps taken to train a model from scratch. In the case of neural nets, the training procedure consists of multiple training epochs over the same dataset and updating the weights via gradient descent.

Table 4.1 Composition of experiment steps

Epoch	Ran multiple steps
Run	Ran multiple epochs
Experiment	Ran multiple runs

Table 4.2 Attributes of an experiment

Bug identifier	-	A user given name
Evaluation type	t	Either "**buggy**" or "**corrected**"
Model	m	Model used to train
Challenge	c	The dataset used to train the model
State	s	The random state used
Artifact	b	Specific build that showcases the bug
Software	d	Software dependencies used for the runtime
Epochs	t	Number of epochs used to train the model

For each bug, we will conduct two experiments: one for the **buggy** and one for the **corrected**. We use the term *performance* as various measurements of a model's capability to correctly accomplish its task. Notably, for classification problems, metrics used are Accuracy, Precision, Recall and F1-score, among others. In the context of this study, we are not interested in the traditional sense of performance, so we use "performance" to denote performance metrics.

4.2.2 Procedure

The following section describes the methodology procedure from extraction of the change information, to the actual measurements. The procedure can be summarized as:

Step 1: Collection: Gather bug information into set of all bugs B_a
Step 2: Filtering: Create a subset of B_a, B_f that only contains bugs that are relevant (non-user) and "silent"
Step 3: Artifacts: Obtain artifacts for **buggy** and **corrected** versions in B_f
Step 4: Evaluation: Get performance metrics with the least variance as possible for both versions of each bug of B_f by using a custom runtime
Step 5: Analysis: Conduct a statistical analysis to determine if the bug presence affects

the performance

Data collection The first step to data collection is to collect information about the past defects in the framework, *i.e.,* identify what bugs are present. There are various ways this can be done as a consequence of machine learning frameworks varying in size, software engineering complexity, documentation and release process.

In either case of manual or systematic bug collection, the core information that needs to be available at the end of this step is: (1) **buggy** and **corrected** versions of each bug, (2) enough information to build an artifact for each version and (3) enough information on the understanding of the bug to apply a decision on the filtering process.

As mentioned before, we take **buggy** and **corrected** versions to be before and after a bug fix is applied to the main branch of the project.

Data Filtering After having collected all the bug information, there is a necessary step of manual filtering. Each project being managed differently, specific methods, adapted for each framework's development life-cycle, to list potential bugs to be studied are to be considered. However, for each framework, the following conditions for a specific bug to be examined share the same rejection criteria:

For each bug $x \in B_f$

1. Reject x if it causes a compilation error
2. Reject x if it causes a runtime error, causing the application to exit (crash).
3. Reject x if it is caused by end-user code

Bugs that comply with the previous conditions are added to the set of issues to be examined, B_f. Our reasoning behind these criteria is to only study bugs that are *silent* and could easily be missed during the process of training ML models. In the first two cases, we therefore remove from evaluation all non-silent bugs. Lastly, we do not consider bugs that would be opened due to errors by end users, as they do not constitute a bug of the ML framework itself.

4.2.3 Artifacts generation

This step aims to create artifacts that can be used to measure the impact of the bug at both **buggy** and **corrected**. For each bug-revision pair, one needs to go through the entire build pipeline to produce an artifact that is suitable to be installed and measured in a "benchmarking" framework. The build process between a bug's **buggy** and **corrected** revisions have to be identical, while the build and linking dependencies need to be tightly controlled. Ideally, build dependencies are identical as well as the build process. While this specific step is to-the-point, there are magnitude of precautions and limitations that one needs to take into account. Section 4.6.5 discusses the precautions and limitations in more detail, notably on why a single build configuration environment is probably not enough. Depending on the framework at study, certain other precautions might apply.

For each bug $x \in B_f$

- Identify the build tool chain and configuration needed, namely T_x
- Build a version x_b for the **buggy** version using T_x at revision **buggy**
- Build a version x_c for the **corrected** version using T_x at revision **buggy** and **corrected**.

4.2.4 Evaluation

For a single bug $x \in B_f$, let e_b and e_c be its *experiment* for respectively **buggy** (b) version and **corrected** (c) version. The only experimental attributes (see Table 4.2) that differ between e_b and e_c are their *evaluation type* t and artifact b.

Additionally, the following criteria must be met identically for both e_b and e_c, which we denote by e:

Procedure The procedure is the collection of metrics for an experiment. Notably, it characterizes a model's performance given a specific set of experiment attributes.

- The experiment must consist of N runs
- At the start of each run r_i, all random configurations must be reset to specific state s
- Each r_i must consist of a training algorithm spreading over e_t epochs
- At the end of each r_i, measure performance metrics on test set and record metrics.

An appropriate number of runs N is to be set for analysis: this depends on various factors, but we recommend at least 30.

For each bug $x \in B_f$

- Choose experiment attributes e (4.2) representative instance of x

- Apply evaluation procedure 4.2.4 with experiment attributes for e_b, the **buggy** version.
- Apply evaluation procedure 4.2.4 with experiment attributes for e_c, the **corrected** version.

For a single experiment, e, a sample output can be seen in table 4.3:

Table 4.3 Metrics collection sample

	0	1	...	N
accuracy	a_0	a_1	...	a_n
precision	p_0	p_1	...	p_n
recall	r_0	r_1	...	r_n
f1	f_0	f_1	...	f_n

4.2.5 Analysis

This step aims to define the measurements of performance of machine learning models running on artifacts, for both **buggy** and **corrected** versions. The general idea is to train both versions on the same model, hyperparameters, data, randomness configuration, data dependencies and hardware and gather a series of point measurements during training and evaluation: these are *experiment attributes*.

For a said bug, in order to quantify the impact of the bug, different distributions are measured: the distributions of performance metrics when the bug is present (buggy) and the distributions of the metrics when the bug is corrected (corrected). The multiple distributions recorded for each model consist of data points representing each of the performance metrics. In order to reduce the variability, the same random seed is used for each experiment, that is both the **buggy** and **corrected** versions of a bug will be trained using exactly the same data and initial random seed. More generally, the **buggy** and **corrected** versions should run with identical NDIFs, see Section 3.2

Hypothesis In order to answer our research question, we define the following hypotheses:

H_0 For a training procedure, there are no differences in the f1-measure between the buggy and the corrected version of a bug fix.

H_1 For a training procedure, there is a difference in the f1-measure between the buggy and the corrected version of a bug fix.

We formulate the alternative hypothesis as a strict inequality since we have no prior knowledge of measurements, *i.e.,* we use a two-tailed test. In order to answer research question RQ1, we use a

Wilcoxon-Mann-Whitney test between the performance metrics of each version, *i.e.,* between the training procedure results using'e_b and e_c. Wilcoxon-Mann-Whitney is a nonparametric test that measures the difference in mean ranks between two samples [94]. There are two versions of this test: one in which observations are paired and one in observations are not paired. As the methodology prescribes that all initial factors be identical at each run, ordering of the runs is not important. Furthermore, the samples are not considered paired, as each run contains the same random seed configuration and is reinitialized to its initial state. In this case, as the metrics for runs are not paired we use the non-paired version Wilcoxon-Mann-Whitney's test: the *Wilcoxon-Mann-Whitney U-test.* We will use a 95% confidence interval for measuring the impacts.

Experimental Setup required In order to get these performance metrics, one needs to train models repeatedly. However, using a traditional environment for training does not suffice. Indeed there are a plethora of factors affecting the performance of a ML model. In this study, it is imperative that the runtime and its dependencies be the same for both experiments of a same bug. Therefore, a consistent runtime needs to be used: we advise using `ReproduceML`, presented in Section 4.3 in order to consistently collect the data needed in this sections' statistical analysis. While `ReproduceML` deals with the configuration of randomness and data across runs, there are other mechanisms that it cannot control, notably the runtime itself. Consequently, a virtual machine with predefined Docker [95] images provides a good compromise in runtime configuration and ease of configuration.

4.3 ReproduceML: a framework to support deterministic model performance assessment

The creation of `ReproduceML` is motivated by the need for reproducibility in ML and the lack of existing solution at the time of creation of the framework. It was originally designed to specifically run the experiments as explained in the Methodology Section 4.2 and thereby reducing the amount of variance between runs. At its core, `ReproduceML` is a Python training benchmarking framework focused on *repeatability* and striving for *reproducibility.* As there are technical limitations, prohibiting complete reproducibility, we will use the term "reproducibility" to indicate both "repeatability" and "reproducibility". It is important to note that `ReproduceML` is not a ML framework: rather, it configures the necessary runtime mechanism for other existing ML frameworks in order to have reproducible results: it focuses on reproducible data splits and reproducible random state configuration where possible. Another goal of `ReproduceML` is to have a simple usage with as simple as possible opt-in mechanism: this is done by defining clear interfaces. While `ReproduceML` is young and currently only supports Python and **PyTorch** for its runtime configuration, additional configuration for other ML frameworks is well within the realm of possibility.

It is primordial to consider that `ReproduceML` aims to address reproducibility at the data and randomness aspects of the machine learning workflow. The goal here is not to create an ecosystem to ensure reproducibility but rather to provide a framework for the specified parts: runtime configuration, data management and randomness seed archival. Other tools exist and their combined used with this framework is what will allow empirical studies to achieve better reduction of confounding factors.

`ReproduceML` performs the concept of "data versioning" but has some key differences to DVC [73]: our work is a benchmarking tool that integrates data versioning. DVC, however, is to be used in the ML workflow. The benchmarking system provides a centralized and clearly defined metric collection procedure. In short, it consists of two portions: a client and a server, each of which needs special configuration to achieve reproducibility. The client trains ML models in tightly controlled experiments which are given by a server. The sever is responsible for giving the training and test data subsets in a deterministic fashion and to collect results from multiple clients. What makes this portion of the framework stand out from most other available frameworks is its focus on reproducibility rather than performance as seen in Section 3.3.2.

Ensuring reproducibility

As previously stated, there are two main parts interacting within the framework. In this section, we explain how `ReproduceML` deals with the essential parts of the machine learning workflow in order to have better reproducibility and repeatability than standard ML workflows.

When it comes to the use of `ReproduceML`, the server can be viewed as static, whereas the client is dynamic. Indeed, the server acts as a central repository for data collection, challenges and most importantly deterministic data serving to clients. Clients, on the other hand, are expected to widely vary in configuration, hence there are two key aspects to consider: (1) reproducible random seed initialization and processes during training and (2) communication with the server in order to collect data and send metrics.

Client The client interfaces with libraries and has the responsibility to configure the runtime using reproducibility mechanisms available in the dependencies. Therefore, the client is strongly tied to machine learning frameworks and their versions. The client therefore uses the built-in mechanisms of dependencies and frameworks to correctly set the runtime. As an example, the core of the framework comes with support for **PyTorch** and interfaces with the runtime for model training. Interoperability with various frameworks is done by leveraging the interpreted nature of Python and its mechanism for dynamic imports. Using this, `ReproduceML` sets the several mechanisms to make sure the randomness is controlled.

Users are expected to import their models by inheriting from a base class or creating a class that contains the model. The training procedure is entirely controlled by the framework as the interface from the base class is representative of a typical DL training procedure. Currently, `ReproduceML` comes with two predefined models for image classification VGG [96], AlexNet [97] taken from the official implementation of **PyTorch**'s related module **torchvision** [98] which have been adapted. It also comes with a CNN inspired from VGG but adapted for working with 32x32 images, internally named `VGG-X`. When running a model with the framework, one specifies the runtime it wants to load and the model name, among other parameters. Therefore the goal is for a user to write their model with the ML frameworks of their choice, make sure it is supported by the framework and launch a client running procedure. Provided that the server is running, this setup will allow for reproducible runtime configuration done by the framework and the training sequence can begin.

Server The server responsibility of `ReproduceML` is two-fold: (1) keeping metadata and metrics about experiments and (2) serving data to a client in a deterministic and reproducible fashion. The server controls the metadata by holding and safekeeping seed values for an experiment. In this context, an experiment is a unique identifier that is supplied by the client. Deterministic random seeds are generated when a new identifier is transmitted by a client and are saved in a file by the server. These seeds are used both by the client and the server to configure the randomness on their end. On the server-side, data splitting reproducibility directly uses the random seed therefore allowing for consistent serving of data.

Moreover, the server collect metrics from runs by establishing a session with a simple protocol for communication. This protocol is followed by the current Python implementation provided in `ReproduceML`.

It is also possible to use the server as standalone and simply leverage the communication protocol defined within `ReproduceML`, as the underlying protocol is TCP. Therefore `ReproduceML` could be used to make a simple metric collection server and dashboard for comparison based solely on the identifier. This allows for other works to provide similar work of controlling randomness on the clients not using a Python runtime.

Communication The client and the server communicate over TCP to exchange training data, seed information and metric collection. There are some rudimentary safeguards currently implemented in `ReproduceML`'s communication protocol to make sure data is correctly received and unchanged. Notably, as a precautionary action, both parts of the communication check data integrity when receiving information. Security of the communication channel was not studied nor implemented in the context of this work as the current communication is meant to be done within trusted machines and/or networks.

Another precautionary action taken is by sending the value of the seed when the client asks for data for a specific run. The seed is compared to the server's version and if a mismatch is found, the server stops. Moreover, a checksum is used on both sides to make sure data integrity and ordering are preserved.

Limitations

Notwithstanding, the limitation of work needed in order to add a ML framework, the main hurdles to usability is that `ReproduceML` is limited by the external ML frameworks' ability to control randomness and ensure reproducibility. Therefore, we strongly encourage ML framework developers and hardware makers to provide compatibility for reproducible workflows. One limitation of the current implementation of this framework is the backup mechanism and checkpoint system for seeds: there is a mechanism in place in order to save the values when the program ends or a signal is received, but this needs to be done more periodically, perhaps at each generation, as unexpected errors can cause seed values to be lost in environments where programs can end at any moment. Moreover, as of now there is no validation set generation, *i.e.,* data splits are made in "train" and "test". As the primary intended use was for controlling specifically the random seed for the methodology, and since validation sets are used to tune hyperparameters and provide cross-validation, this feature was not included. It is however trivial to add support for validation sets if one desires to use it as a metric collection and challenge database. There are multiple factors affecting the stochastic aspect of training a model that cannot be controlled by runtime configuration: *e.g.,* installed version dependencies and the runtime itself. These issues are meant to be addressed by other tools and techniques.

4.4 Execution and results

In this section, we first discuss how we carried out the methodology. We then briefly summarize the experimental setup used throughout the study. Finally, we present the results of our application of the methodology on **PyTorch**.

4.4.1 Methodology application

The current subsection presents the results and specific methodological considerations taken in order to carry out an empirical study for the bugs in **PyTorch**. The study analyzes the bugs on specific releases ranging from version *v1.1.0* up to version *1.6.0.* As short explanations on our study decisions will be discussed here, Section 4.7 provides a more thorough explanation on decisions taken here.

We divided the data collection step of the methodology into several sub-phases in order to provide a systematic approach whenever possible.

Data collection and processing In order to measure the impact of defects on **PyTorch**, we extracted 737 bugs following an automated approach to parsing of release notes published by the maintainers. This parsing yielded metadata such as the Pull Request number, links to description, etc. Out of this corpus, we analyzed 439 bugs out of which 115 seem to meet our criteria explained below.

Collection In the case of **PyTorch**, the software release process has been streamlined and changes are usually correctly reported in their **Release Notes** with corresponding bug fix information. Therefore, we use the bug fix sections of each release notes to construct our corpus. We followed this methodology since the bugs that are reported by the maintainers are guaranteed to be coming from the framework and are definitely labelled as bugs. As **PyTorch**'s repository is hosted on GitHub, it is possible to mine the issues to find the bugs since GitHub provides a web API to get information about issues opened and their labelling scheme. However, we found that in practice this approach leads to more complications, as a bug might be wrongly labelled by the triage system in place. Additionally, using an issue-mining approach is time dependent, as labels can be added and removed at any time. Following the guideline of never changing the content of an official release by using consistent semantic versioning principles allows us to mine the information directly from the release notes.

Filtering For this study, in addition to the clear criteria established in the methodology, we added the following constraints to limit the search of bugs. These constraints were applied in order to prioritize bugs that would have a clear impact on training and that would be easy to replicate. Thus, a manual filtering effort was made on the 439 bugs with the following additional criteria:

- Rejection criteria:
    1. Is on CPU (takes too much time)?
    2. Version <= 0.x (early versions are more difficult to make models for)
    3. Don't see an application that would be affected
- favourable acceptance:
    1. Related to gradient
    2. Related to math functions
    3. Recent version of **PyTorch** (>*v1.0.0*)

This effort was made by people possessing a working knowledge of ML and DL. We use "favourable acceptance" instead of "acceptance": this is done to avoid false negatives. This criterion still led to false positives and false negatives, we discuss this in Section 4.7.2.

4.4.2 Experimental setup

Multiple systems were used in order to conduct this study: in this section we overview the overall experimental configurations. For a complete list of our hardware and software used throughout the sections, see appendix A.

Building To build the various versions of **PyTorch**, we used Google Cloud Platform (GCP) virtual machines running *Docker* containers. Specifically, we use `n2-standard-16` machines that directly mount docker containers. We use two images to build the versions: the first is a container loaded with the tools needed to build **PyTorch** at Python *3.7.9* while the other is for the version *3.6.7*. This dual setup is to make sure that we replicate a plausible build: a more in-depth explanation of time-related builds and software archaeology explains this in Section 4.7.3.

Evaluation For running the experiments, we leverage virtual machines for hardware control and Docker containers for software and runtime dependency control. The virtual machine used is a `n1-standard-16` with virtualization on *Skylake* CPUs running Linux kernel version *5.4.0-1028-gcp*, on Ubuntu *18.04.5* equipped with a **NVIDIA V100 SXM2 16B rev a1** running driver *450.51.06* with `CUDA` *11.0*. Experiments ran on Docker *19.03.13* accelerated by the nvidia-container-runtime. One might notice that the `CUDA` version is different between the build developer tools and the runtime. This is partially a configuration error as the original idea was to run the NVIDIA `CUDA` *10.1*. This should not change the result as NVIDIA promises binary compatibility between versions [99]. This is indeed the case as our experiments can be run and still use the libraries against which they were built.

For `ReproduceML`, we use the code at revision *v0.1.0*. At this version, the optimizer is set by the framework and uses **SGD** with a learning rate of 0.01 and a momentum of 0.5.

Our current experimental runs use `VGG-X` from the default models provided by `ReproduceML` to run experiments on working on the CIFAR-10 dataset. A brief description of the architecture can be found in appendix A. This choice is motivated by a good balance between complexity, time to train, and relative novelty of the model. We run our model for 30 epochs over 50 experimental runs for each of the **buggy** and **corrected** versions.

We chose to experiment on a Docker container in order to provide a reliable and reproducible software configuration. The overhead induced by Docker is not noticeable and has been shown to be small when training models [100].

Notably we use the following common *experiment attributes*:

Table 4.4 Shared attributes for our runs

Model	VGG-X (no Batch Norm)
Challenge	CIFAR-10
State	*Depends on the bug*
Software	*Depends on the bug, but Docker image contains specifics*
Epochs	30

The random state used varied from a bug to another, as we are not interested in measuring the stability for a single random seed. Within a single bug, the random seed stays identical for all the runs.

4.4.3 Metrics calculation

Since our challenge is CIFAR-10, the problem is multiclass. Therefore, there are multiple ways to calculate the accuracy, precision and recall. In this study, the scores reported are using the unweighted mean of scores of each metric. That is, we average the metric for each class, without weighing by the number of representatives in each class.

4.5 Results and analysis

This section presents the results and the analysis conducted for 18 bugs in our corpus. We present the results after having processed the log files emitted by our experimental setup. There were manual steps needed to parse and verify that results were clean of any mistakes. Notably, we verified that each experiment was conducted under the same experimental setup as described in Section 4.4. Out of the 115 candidate bugs for our experiment, we have been able to measure successfully 18 and conduct the Wilcoxon-Mann-Whitney test on their buggy and corrected versions. We find that there is currently no statistically significant evidence that the occurrence of bugs in the studied ML framework affects the performance of models trained using the framework. Discussions about these results follow in Section 4.7.

Note that there are two bugs that did not complete the 50 runs mandated for both **buggy** and **corrected**, namely experiment `study-pr36832` for its buggy version completed 47 runs (and 50 for its corrected). The second is experiment `study-pr31433` which completed 43 runs for its corrected version and 50 for its buggy version. All other bugs completed 50 runs; we will mark with "†" results that did not complete full 50 runs for both versions. We kept these two bugs for full disclosure.

4.5.1 Statistical analysis

As mentioned in the methodology, we use a Wilcoxon-Mann-Whitney non-paired test to test whether or not the results did show an impact. Table 4.5 shows the p-values of the hypothesis test for all our of experiments.

The Wilcoxon-Mann-Whitney test can accept different number of samples in each population, but in this case, we choose to only compare the same number of samples in the population. That is, for the bugs that did not complete the full experiments, the values portrayed are for a Wilcoxon-Mann-Whitney test done with the minimal number of runs available by either buggy or corrected version.

Statistically significant result We can see that only a bug did show in fact a statistically significant result for our experiment: `study-pr31433`. This bug's corrected version did only complete 43 runs. Even when computing the p-value by using the maximum number of samples in each population (50 for buggy and 43 for corrected), we still obtain p-values less than 0.05 for each metric. We can see the distribution of each metric in Fig. 4.1. Interestingly, this bug is not related to our CNN: the bug fix is related to completely different files (about LSTMs).

We categorize this value as an outlier as there is no reason to suspect there was a difference between the two versions. Furthermore, we think that there is several aspects that introduced non-determinism in our experiments: these will be explained in Section 4.6.

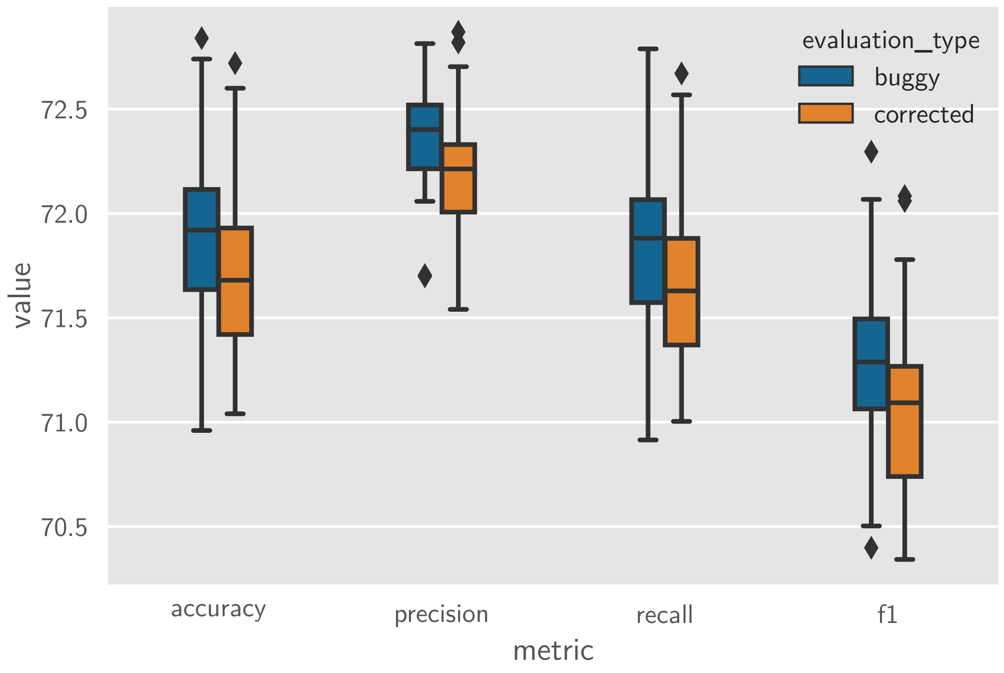


Figure 4.1 Distribution of metric values for bug `study-pr31433`

4.5.2 Deviation of runs

We here report the standard deviation of each metric for our runs, for each bug and metric. This allows us to observe the effects of NDIF on the runs. The table in Appendix B shows the complete result. Taking from this table, we can observe in Fig. 4.2 that the standard deviation for each metric is pretty small for all runs, indicating that some control of NDIF was done, without, however, quantifying the impact that is left to control.

Table 4.5 p-values of RQ1

	metric			
	accuracy	precision	recall	f1
study-pr31167	0.62929	0.42190	0.58365	0.88217
study-pr31433†	**0.03320**	**0.00243**	**0.03508**	**0.01057**
study-pr31552	0.92581	0.58365	0.97525	0.73292
study-pr31584	0.58356	0.56486	0.62208	0.57893
study-pr32044	0.84956	0.39074	0.78009	0.48842
study-pr32062	0.46259	0.87673	0.45033	0.77481
study-pr32350	0.48821	0.55096	0.53270	0.65657
study-pr32541	0.63418	0.63185	0.61237	0.39074
study-pr32829	0.50135	0.21085	0.50589	0.39455
study-pr32831	0.66144	0.94230	0.65657	0.95327
study-pr32978	0.31404	0.29949	0.35740	0.44620
study-pr33017	0.14363	0.21849	0.12005	0.10014
study-pr35022	0.28821	0.51031	0.29629	0.32932
study-pr36820	0.20198	0.53270	0.20097	0.57422
study-pr36832†	0.30004	0.37223	0.26631	0.26958
study-pr37214	0.67396	0.72257	0.69181	0.69181
study-pr38945	0.40400	0.97525	0.36465	0.58365
study-pr39903	0.97249	0.74854	0.98625	0.93681

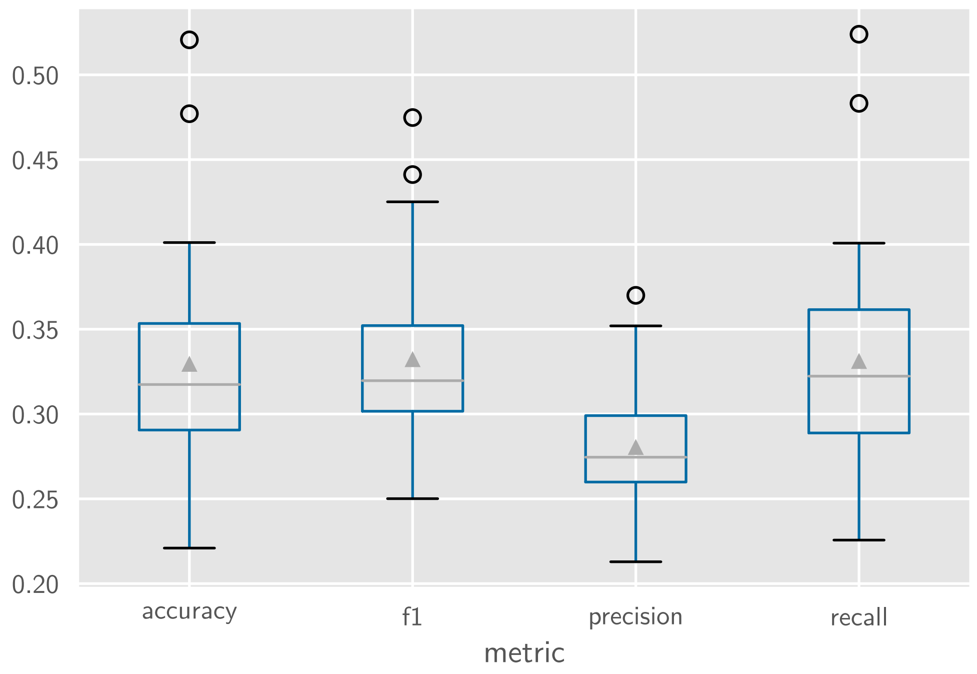


Figure 4.2 Distribution of metric standard deviations across all experiments

4.5.3 Unreported results

There are 3 bugs that have been excluded for not using the same seed: these are the consequences of ungraceful shutdowns due to GCP specificities, leading to data loss of seed value. We have therefore removed the 3 from our results. For the record, the values using different random seeds did show a statistically significant result, but were erroneous as different initialization seeds were used. While these are not reported here, box plots showing the results are visible in appendix C, their p-value is virtually null.

4.5.4 Summary

Out of the 18 bugs, only has shown statistically significant results allowing to reject our hypothesis H_0. However, we think that it is an outlier, as the buggy and corrected versions of the bugs do not seem do differ. **We can therefore say that with the current data, there is no evidence to reject H_0, thus we cannot conclude that bugs in ML framework have an impact on model performance.** For this, we have to take into account the multiple limitations that our study suffers.

4.6 Limitations and threats to validity

This section exposes the different limitations that are present in our realization of the methodology from a theoretical and practical standpoint. The different subsections are ordered on our intuition of the biggest limitations present in our work. The main limitation of our work was to establish if the bugs were actually triggered during the execution, we discuss the relevant resolution methods in Section 4.8.1.

4.6.1 Triggering the bug

The first and most important concern in this study is validating if bugs were actually triggered during the execution of our runs. We are aware that this is a huge limitation, as the core of this study relies on it. Time constraint was the biggest factor here. While the bugs could be triggered during manual execution, *e.g.,* by reproducing the bug with the code from the pull requests, this does not mean that when processing data in a training process this would happen. This is due to multiple factors, but the biggest is that **PyTorch** is a big framework with multiple "backends"; here, the key is building the right version, with the right build parameters to make sure that the bug is triggered. As mentioned before, we will discuss in Section 4.8.1 a way to validate the bug's presence *during execution of the training process.*

Indeed, if the bug was not triggered, this study only means to demonstrate the variability induced by NDIF even by controlling the seed, for within bug experiments. It can also be used to show the variance between multiple random seeds, although on a very small scale, since most of the runs (inter-bugs) did run on different seeds, as such, the split and the ordering of the data fed might affect the final training efficiency.

4.6.2 Model and dataset used

One might wonder why only a variation of VGG was used throughout this experiment: this goes in line with most of the considerations above. Indeed, there is no way of knowing during training if the workload contains the bug, as it is silent. Therefore, instead of using multiple models to accomplish the same task, we use the same model. The current model that was used that we named **VGG-X** is intended to work on the CIFAR-10 dataset: traditionally, the model works on the ImageNet [101] dataset, but as we wanted to reduce NDIF by reusing the same data split, we did not succeed in time to provide fast and deterministic serving of the ImageNet dataset. Furthermore training on such dataset would have taken considerably more time.

In summary, we adapted the original VGG model (**PyTorch**'s implementation in **torchvision** [98]), to work on 32x32 input images. While this does not change the results for our study, in the sense that comparison was made for the same model, which achieve acceptable good performance (around 70-74%) and considering that comparisons should be made only between buggy and corrected versions, our comparison holds. Notwithstanding the validity on a theoretical side, the neural network should achieve better performance considering its size. We believe that the adaptation of the model to fit the 32x32 input did in fact cause some non-optimal configuration for the network. Notably, there is one aspect of the network that caused some potential non-determinism: the Dropout layer. Two dropout layers with activation probability of 0.5 were in the network: this could have affected the determinism of the computation. This falls under the "Model definition" NDIF (see Section 3.2.2).

4.6.3 Non-deterministic inducing factors

There are multiple non-deterministic sources that have not been controlled.

Our work is currently limited and does not offer any guarantee against multiple confounding variables such as floating point calculations, possible thread interactions and GPU models. These are however mitigated by the fact that we carry the computations by using the same hardware, runtime, software versions and threading model for our experiments. Consequently, the results for a specific bug, *i.e.,* between a buggy and a corrected experiment, use the same experimental attributes, thus reducing the effects of NDIF.

Specifically for thread interactions, `ReproduceML` uses **PyTorch**'s built-in mechanisms in order to control its thread interactions. Consequently, our experiments use only one CPU core to do its work, thereby eliminating some thread interactions from sources of errors. However, there is not a complete guarantee that **PyTorch**'s thread interactions are complete, and our work is based on their suggestions for reproducibility [102, 103].

However, by detailing the current running hardware and related configurations, our work should be repeatable.

Random seed While we use `ReproduceML` in order to control the random seed on runtime, there are still variations present in the runs. There are multiple sources for this randomness that have not been controlled throughout the experiment. This indicates that additional precautions need to be taken when interpreting results. We think that they are the result of NDIF at the implementation level, as results by Pham *et al.* show a similar range of values for a similar model to ours [41].

Floating Point Calculations Our current work does not currently support detecting if floating point operations are the same between experimental runs. There has been some research on making float point operations more stable [104], but we are aware that our current framework values only hold for a specific architecture, which is why we disclose the hardware and software used in the study. Furthermore, adding support for such tools would most likely introduce more challenges and sources of error, as they are not integrated by default within ML frameworks.

4.6.4 Evaluation

The current experiments work on a sample of the dataset instead of the entire dataset: this poses a problem for reproducibility as this split is constructed dynamically. Practically, we used half of the dataset to train and test. Future work should improve over this limitation.

4.6.5 Build system

In the context of this study, we built versions of **PyTorch** using a common build process and using the default *suggested* parameters for build execution. As most software are distributed prebuilt through various channels, the upside from this process is that we reduce the possibility of introducing multiple build-related errors, as the recommended configurations reflect most widespread use of the framework. There are, of course, differences that can affect the final build artifacts from the ones we have built, notably because there might be some changes in the specific tooling used. Indeed, in the aim of reducing confounding variables, a single build system configuration was chosen to build all the versions with two variants on the Python version. Each variant represents a *Docker* container with the dependencies used to build **PyTorch** according to their guidelines.

Nonetheless, both containers use `CUDA` API *10.1*. Note that building does not use the GPU, but here the `CUDA` version refers to the `CUDA` toolkit, *i.e.,* the necessary developer dependencies needed to build application with support for `CUDA`.

Dependencies The methodology prescribes using a build system in order to build **buggy** and **corrected** versions. However great care must be taken when doing so: in the case of **PyTorch**, there are multiple computational "backends". These are libraries dependencies that exist in order to execute mathematical operations in order to allow different hardware and software configurations to run **PyTorch**. The limitation comes from the fact that some parts of the library can only be used, *i.e.,* compiled with one of these "backends". In an ideal world, one would need to build all the different versions of a framework with the different build options and then make an assessment of the impact of those while using statically linked libraries. Practically, this is infeasible and means that care has to be taken when building a version for a specific bug.

For reproducibility, the best way to build a framework such as **PyTorch** would be to statically link every dependency instead of using dynamic dependencies as this would mitigate improper or deviation of the methodology. Indeed, when statically linking an assembly, there would be no question of which runtime on the framework dependencies would be used. This would allow relaxation of the current constraint of running within the Docker container. Configuring static linking, however, is not trivial and would require a lot of effort. Furthermore, since we use *Docker* to control the runtime dependencies, we have tight constraints on the available runtime versions.

Docker and Python While it would have been possible to only use one Python version for all the bugs, there is no guarantee that a specific version of Python would be supported in a specific version of **PyTorch**. As an example, it is possible to build an old version of **PyTorch**, ex using *1.1.0* released in April 2019 with a Python version *3.7.9* released in August 2020, this configuration would not have been possible at the release of said **PyTorch** version and there is no practical use to ensure compatibility retroactively.

4.7 Discussion

Deviation of runs For each experiment, the standard deviation for all metrics across runs is situated between 0.2% and 0.51%, showing that controlling NDIF was done relatively well, considering the facts discussed in Section 4.7. We compare ourselves to work done by Pham *et al.* [41], in which we can observe that the larger models have a bigger standard deviation than smaller models when using a fixed seed. Notably, our model uses around 15M parameters, *with the Dropout effect encountered* and still shows mean standard deviations around 0.3%. Comparatively, for Pham*et al.*'s most comparable model in number of parameters, *i.e.,* ResetNet56 shows similar range, around 0.5% of standard deviation. While there is no hard evidence of the effect of size, *i.e.,* statistical test to show this, we think based on purely ad hoc observations that bigger models have a higher potential of being affected by NDIF. It would be interesting to see how Pham *et al.* constructed their models, since they say it was reviewed multiple times [41]. However, at this time, their repository does not seem to contain their implementation.

Impact of the random seed We briefly recall that we discarded some results because they did not fall under our methodology, namely some of them did not use the same random seed. While these results were discarded, they did prove statistically significant impact. Consequently, we can argue that when using different random seeds, a statistical test such as Wilcoxon-Mann-Whitney might positively reject H_0. In light of these findings and recommendations by Pham *et al.* [41], and this overall thesis, we recommend that publications and research in ML and DL use statistical tests to measure the performance of their work.

4.7.1 Reproducibility and its impact on the industry and academia

Reproducibility in computational workloads such as ML is becoming more common, as we have shown in Section 3.1. But what does it mean for the industry? And for academia?

We have seen that industries are interested in reproducibility, especially the ones in the life sciences. The work shown here along with work shown by Pham *et al.* [41], Crane [49], Guo *et al.* [50] and Jean-Paul *et al.* [38] show that the effect of randomness can cause drastic differences in the result of a ML workload.

Considering that ML and DL systems are being used for important medical systems, reproducibility is now more than ever paramount in the field of computational sciences [19]. This also translates into other industries, as having results varying from small values can have drastic repercussions, *e.g.,* an industry investing would never say no to 1% more performance in a market investment model. For the NDIF that is the random seed, there is already a huge impact on the industries, consequently, we suggest that industry models be checked with a statistical significance tests before being deployed. This will serve to more critically assess the performance of the model and show its underlying variance. While this might be more costly, by a multiplicative factor, we recall that such significance testing is the basis for any scientific rational decision. The exact same argument goes for academia when it comes to training and publishing results, perhaps even more strongly worded, as they constitute the basis for scientific progress.

In both cases, the artifacts for reproducibility should be available: for the industry this means keeping track of the NDIF that were present during the training part, and also providing the necessary artifacts for *repeatability* purposes. For the academia, this means publishing the results along with as many details as possible on the NDIF and other traditionally reported information; we aim to raise awareness on the issue of the reproducibility by listing a set of NDIF that should be published (see Section 3.2.2).

The effect of randomness is being researched by various authors, but there are other aspects that are sources of non-determinism. Notably, within all the factors established as being NDIF in Section 3.2.2, software versions are the subject of this study. Indeed, a bug fix is merely a change in version, although not necessarily consumed by users, as versions are only published in "nightly-builds". We have not been able to affirm that bugs in DL, in this case **PyTorch**, yielded different results, but we argue that this is mostly due to our limited number of samples and detecting the bug's presence at training time. We highlight the future possible studies that should be done with these NDIF in Section 4.8 and possible remediation solutions.

While the specific impact of these NDIF has not been measured, we highly suggest industry and academia to set their version constraints correctly in artifacts, as this will enhance reproducibility and thus the workflow. Doing so will be beneficial in the process of engineering ML systems, which has financial value.

In order to encourage the reuse and reproduction of this study, various efforts throughout the methodology were taken. Notably we provide the *Docker* images used to build and run the experiments as well as the build artifacts and logs of versions used. The full configuration can be found in Appendix A and the details explained in Section 4.4. Inside the *Docker* images, there are files containing the specific environment used for building, but specific verification can also be done by running a container based on the image and verifying the tools installed in the environment. Build artifacts and logs can be provided on demand.

4.7.2 Building

In hindsight of the labelling process of bugs and creating their corresponding artifact, we reach the conclusion that a solid understanding of the repository was needed in order to correctly assess where the bug would be triggered. Indeed, as **PyTorch** and arguably other ML frameworks have a lot of modules, knowing specifically how a bug affects the framework requires rigorous effort by a machine learning practitioner. Indeed, only after a *third* pass by an individual with average knowledge of the *internals* of **PyTorch** did we find that some bugs were misclassified: a peer-review process did not detect these false positives and false negatives.

Furthermore, a good ability to understand build systems and dependencies is needed, as there are multiple ways to build a single version. While **PyTorch** does offer some documentation for its build process by providing straight to the point explanations and offering a single build process, there are multiple cases where the bug could be triggered only by using a specific configuration, *e.g.,* bugs using **MKLDNN** when this option can be disabled.

Anecdotally, some tests have been performed after this study, to improve the quality of certain aspects. These improvement efforts have been hindered by build-related factors. Briefly, we did some test to use our environments outside a *Docker* environment on different hardware running a different operating system than the one in our study. Our artifacts were linked with the **C** runtime at a version higher than the one currently supported and no easy way was found to *replicate* an environment without a lot of work. This goes to further our point of packaging experiments into *Docker* containers for reproducibility purposes, as this greatly enhances reproducibility, thereby reducing the possible sources of non-determinism.

In addition to these difficulties, the availability of artifacts for builds might pose an additional problem. We preface the next subsection with a little overview of the difficulties encountered when building some artifacts versions for bugs.

Historical build During the process of building artifacts as part of the methodology, there were some cases in which manual intervention was needed. These situations happened because at some specific commit for which the build was created, the dependencies were pointing to a non-existing repository. Consequently, the build would always fail, as it was needed for the artifact to gather its dependency on a specific git repository. Upon manual investigation, it was found that the version was affected because of an older commit which introduced a dependency as a “hotfix” to an issue. Consequently, the following versions would get the temporary repository and when the root issue would be ultimately resolved by another mechanism, the original repository would be deleted. However, when we were trying to build, this repository would be non-existent, thus posing a problem. Fortunately, a similar version of the deleted repository was found and the build continued successfully, to the best of our knowledge this should not affect the results.

This showcases one of the limitations of our methodology albeit rare that can still happen. One might argue that there is no guarantee for build availability to be had when looking at past released versions, especially not specific commits. Nonetheless, these commits were on the main branch of the repository, in which a working version should be held at all time. There are multiple considerations for the advantages and disadvantages of keeping repositories clean and “buildable” for a long time, but adopting it would allow for easier search in software repositories.

4.7.3 Archaeology of software

This section aims to describe and provide explanation on design choices and difficulties of work that go back in time to analyze or build specific software artifacts: git archaeology.

When releasing software versions, considerations of future backtracking are rare: there is no expectation that external dependencies will remain available in the future. The only similar considerations mostly have to do with backwards compatibility, *i.e.,* if the next release would break old clients. The general assumption is that when a product is released at one time, support is given and build artifacts are distributed. If a change happened in the ability to create a past release, a new version of the software would be released and the old versions would be abandoned. While the release *artifact* might still be available, the process to do so might not, due to the possible disappearance of external dependencies. These are the basic of release support. When doing empirical work that deals with past versions of software, there are multiple special considerations that need to be taken.

Git archaeology, on the other hand, goes one step further and explores the availability of artifacts when going backwards in time. Indeed, as the goal of the study was to measure the impact of past releases at specific points in history, there are cases in which it is hard and sometimes impossible to recreate the exact version. A prime example of this is when “hotfixes” that reference separate channels are introduced and then deleted some time after. Channels are usually `Git` repositories. In this case, at the time the “hotfix” was applied, the repository pointed existed and artifacts could have been created. When the repository is deleted, some changes in the repository are made in order not to break the software again. The problem arises when we *later* try to position ourselves at a specific time, *i.e.,* commit at or between the introduction of the “hotfix” external channel dependency. That external dependency could have been deleted in the meantime, because the assumption is that there is no need for reproducibility at the commit-level. The same explanation can be extrapolated to releases, but there is usually – although not measured – more care given in keeping release dependencies available.

To mitigate such a problem, one would need to find a copy (archive) of the dependencies or functional equivalents. For reproducibility reasons, an archive would be best, but this is not always possible. Furthermore, finding such copies is not necessarily guaranteed. Therefore, the next possible solution is to use functional equivalents. However this implies changing the source thereby posing an additional hurdle to the practicality of the situation.

4.8 Future works

This section contains two parts. First we propose several improvements to the methodology, in light of our experience and findings. We also present some additional features of our codebase that will allow for a systematic approach to bug empirical analysis. Then we propose multiple other studies that can be conducted with the help of the current work and said additional features.

4.8.1 Improvements to our methodology

The factors that would most directly impact the methodology are those related to determinism in software and hardware configurations.

Tracing

The biggest threat to this study is knowing if the bugs were triggered when the training was in progress. The best way to verify such behaviour is to insert trace points inside the program and use a tracing software. Therefore, in next studies, when building artifacts we highly recommend inserting trace points inside the code. This was attempted in this study, but due to external constraints, we were forced to proceed with non-traced versions. However, the *Docker* images provided for building **PyTorch** do have a **LTTng** [105] tracer equipped.

The main challenge is to make sure the code correctly builds with the new shared libraries: such a solution was found during the building phase of our study for some commits. Indeed, it is not possible to insert trace points into every source file: notably, `CUDA` files that have functions defined in device functions proved hard to trace. Current tracing tools such as **LTTng** can only trace **OpenCL** programs. While successful attempts to use tracing within `CUDA` source files were made by using tools such as **Extrae** [106], they did not allow to insert code executed on the device; more work would then need to be done in order to put the tracing point outside such a function. Such work could be interesting for the developers of ML frameworks. There are other tools to be used when tracing `CUDA` applications, such as `NVIDIA`'s own tools, but these require a proprietary format which discourages the reusability and feasibility of systematic highly parallel workloads. We believe tracing GPU device functions is possible, but does not pertain to the scope of this work.

Nonetheless, the current tools that accompany this work have the ability to insert traces in the source code with **LTTng**. Thus, we recommend that each version be instrumented using the tools made available. For a solid complete study, this would need to be done, as it proves crucial to verifying this *when the model is running examples in training and in inference.* There is also the "trivial" way for trace execution, by inserting in the standard output a certain value, which might prove more convenient but might also introduce possible overhead and side effects.

Multiple models

In this work, we currently have used a variant of VGG, which was adapted from **PyTorch**'s repository, to work on the CIFAR-10 challenge. We argue that more models might need to be used to trigger each bug, as a model's architecture uses only a subset of **PyTorch**'s features. After our work labelling bugs, we can anecdotally say that most bugs are contained within specialized functions: triggering such functions might not be done by using a single model.

Moreover, in this work we have limited ourselves to using CNNs, hence only computer vision workloads. Several bugs might not be triggered by using such workloads: several bugs stem from functions that are rarely used in practice. Therefore, `ReproduceML` should be updated in order to work with different challenges and models.

While adding several models adds the possibility to explore more venues, it does not guarantee that the bugs will be triggered. Therefore we propose that after new models and challenges are introduced, a systematic search of bug triggering for models should be explored. This search assumes that the tracing tool is available in the versions; a quick heuristic to avoid building multiple versions of frameworks with traces embedded would be to instrument one of the versions, say before each minor release and from there make a search for the models triggering the bugs. This search would prove fast and would reduce the amount of instrumentation needed to be done: one on each release is more manageable than for each bug.

Injecting bugs

In the current methodology, we assessed the statistical difference in performance by building several versions. More specifically, in order to reduce the number of confounding variables, we built the versions which did contain a bug and its following revision. Therefore, the only difference between our comparisons would be the bug fixing process and the presence of NDIF which we tried to limit. There is another method to accomplish the same goal: by injecting the changes in a stable version.

This would partially solve the problems about possible missing versions, as the build process for a stable release has more chance to be robust and have longevity. However, this would still require building a specific version with that change: the goal of this approach would be to have better, cleaner and faster builds. This option becomes more interesting as we advance in time, as bugs present in the earlier versions of the frameworks would become more and more difficult to reproduce. This nonetheless runs the risk of being less representative of current advances in the field, as the maturity of ML frameworks progresses, there might be less opportunity to investigate such bugs. Another challenge with this approach is that while we can easily extract changes from one version to another, using "diffs", application of these might not hold for drastically different versions. Therefore, manual injection of faults might be needed when considering such an approach.

4.8.2 Other related studies

With the current tooling presented in this work, by following a similar methodology, several other empirical research on machine learning configurations can be made. Studies on these research questions are possible:

- How does the dependencies' versions in machine learning frameworks affect the performance metrics of machine learning models?
- How does the use of deprecated functions in machine learning frameworks affect the performance metrics of machine learning models?
- How does the occurrence of bugs in machine learning frameworks affect the raw performance (speed)?

4.9 Summary

In this Chapter, we partially answered the research question: "Given a sample of fixed size of recent bugs in a given machine learning framework, does the occurrence of bugs affect the performance metrics of machine learning models?" In order to do so we conducted an empirical study on the DL framework **PyTorch** by extracting, filtering, building and evaluating the artifacts. This study was based on a newly created methodology for the effect of software "changes". We have collected 737 from **PyTorch**'s repository, labelled 439, filtered 115, built 49 bugs (for their buggy and corrected version) and conducted our statistical analysis on 18 of them.

Based on these numbers and our dataset, we have the following answer to RQ1: "We cannot conclude that our sample of bugs in ML framework has an impact on model performance". Obviously, this conclusion is on the negative: we cannot conclude firmly based on our findings since there are not enough bugs studied. Furthermore, we have doubts with regards to bug triggering during training, among other limitations such as the tight control of NDIF.

Creating a methodology The first research objective, RO1 was about creating a methodology for measuring the impact of machine learning bugs. We did so by proposing a general methodology to measure the impact of any "change" in a repository by using the commits as the revision upon which a build process would be initiated. We executed this methodology by applying it to the **PyTorch** framework. As part of this methodology, we also explored the various factors that influence reproducibility in this context and devised the methodology and artifacts used in order to enhance reproducibility in the study. The aspect of reproducibility is showcased in both by (1) reducing the confounding variables of the methodology and measurements by reducing the NDIF and (2) by the reproducibility of this study.

Creating a framework reducing non-determinism introducing factors For the second research objective, the goal was to create a framework that minimizes the effects on non-determinism to accompany the methodology step of evaluation. This was accomplished by introducing `ReproduceML` which performs both random-seed control and data split control, a primitive form of data versioning.

Towards reproducible empirical software mining The last research objective was to create the tooling needed to conduct the proposed study. We aimed to create tools necessary to perform systematic reproducible software mining. This has been accomplished by the implementation of tools aimed at gathering the information for this study, which will be made available to the public under an open source license.

Towards machine learning reproducibility We also discussed the challenges that are present for reproducibility in ML, the impact of random seeds and NDIF on the output of the models and what it entails for academia and industry. We argue that each scientific publication of ML should be posted with a statistical test and the necessary artifacts to *reproduce* their work and therefore assess the true gains in a scientific manner. For the industry, we also recommend conducting statistical tests to ensure optimal model performance and stability.

We also propose a series of improvements and future works in light of the experience gained by executing this methodology.

CHAPTER 5 INDUSTRY CASE STUDY WITH SAP, AN OVERVIEW OF MACHINE LEARNING CHALLENGES IN THE INDUSTRY

In this chapter, we present an industrial case study to share our experience and insights from a research project on machine learning conducted in collaboration between Polytechnique Montréal and SAP. We first present our approach to developing ML-based components for automatic detection and correction of transaction errors. Here, we apply machine learning on retail transaction data to detect and correct transaction errors. We follow an agile methodology to develop ML-based solution. We identify the challenges in each step of our software development process. We present these challenges along two distinct perspectives: (1) software engineering, and (2) machine learning. From these perspectives, we explore the relationships and dependencies among the challenges to have better insights into the challenges in software engineering for machine learning. In particular, we answer the following two research questions:

RQ1 What are the key challenges of machine learning (ML) application development?

RQ2 What practices should the ML developers follow to overcome challenges of ML application development?

We synthesize our observations from our case study and existing knowledge from the literature on the common challenges and best practices to recommend a set of guidelines for developing ML applications in an industrial context.

The key contributions of this chapter are summarized below:

- We have identified important challenges in software engineering for machine learning. Our insights will help developers build ML applications.
- Based on our experience from our industrial case study, we propose guidelines for researchers and practitioners defining best practices in ML application development.
- Our proposed guidelines have been adopted by SAP as part of its internal guidelines for ML development.

The rest of the chapter is organized as follows: Section 5.1 offers a brief introduction for our work on a machine learning perspective and introduces our industry partner SAP. Section 5.2 presents our case study outlining our approach for detection and correction of transaction errors. We share important lessons learned from this case study in Section 5.3. Finally, Section 5.4 summarizes our work.

This chapter is part of a paper submission. There have been some modifications a posteriori. *Changes deemed important to the essence of this chapter will be marked with "†".*

5.1 Context

Machine Learning in industry Artificial Intelligence (AI) and Machine Learning (ML) have shown promising prospects for intelligent automation of diverse aspects of business and everyday life [107]. The rapid development of machine learning algorithms and tools and easier access to available frameworks and infrastructure have greatly fuelled this era of the development of ML-based software solutions for real-world problems. Software companies, big or small, are striving to adopt ML in software applications, in order to improve their products and services. However, the potential of machine learning accompany multifaceted challenges to the traditional software development processes and practices [107, 108]. The development of ML-based software applications may add challenges to all phases of the development life cycle [109]. The requirements for the ML models are dynamic in nature to adapt to the rapidly evolving user requirements and business cases. ML-based applications are data-driven. Thus, efficient pipelines and infrastructure are required for data-streaming; i.e., data acquisition, data storage, preprocessing and feature extraction, and ingestion by the ML models. Also, model building and deployment have different constraints regarding the type of problem, data, and the target environments. Software engineering for machine learning applications has distinct characteristics that render most traditional software engineering methodologies and practices inadequate [108]. The design of ML applications needs to be flexible to accommodate the rapidly evolving ML components. In addition, the testing of ML applications differs significantly from traditional software testing. So, new guidelines are needed to help ML developers cope with these challenges. Additional challenges are likely to surface when the ML applications are developed in collaboration of multiple teams with diverse backgrounds and expertise [90].

Industry Client SAP is the market leader in enterprise application software offering an end-to-end suite of applications and services to enable their customers worldwide to operate their business. Retail customers of SAP deal with millions of sales transactions for their day-to-day business. Transactions are created during retail sales at point-of-sale (POS) terminals and those transactions are then sent to some central servers for validations and other business operations. A considerable proportion of the retail transactions may have inconsistencies or anomalies due to technical and human errors. SAP provides an automated process for error detection but still requires a manual process by dedicated employees using workbench software for correction. However, manual corrections of these errors are time-consuming, labour-intensive, and might be prone to further errors due to incorrect modifications. Thus, automated detection and correction of transaction errors are very important regarding their potential business values and the improvement in the business workflow.

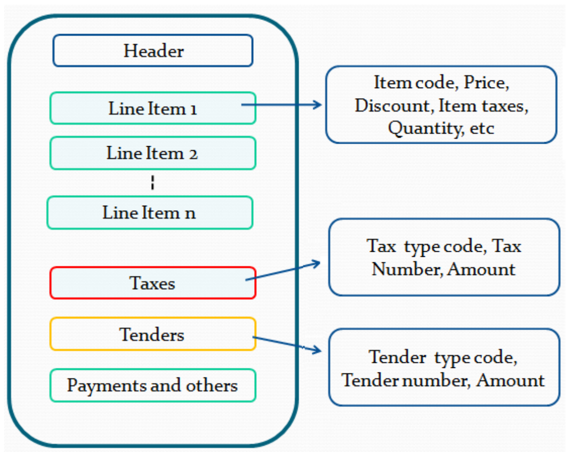

Header
Line Item 1
Line Item 2
Line Item n
Taxes
Tenders
Payments and others
Item code, Price, Discount, Item taxes, Quantity, etc
Tax type code, Tax Number, Amount
Tender type code, Tender number, Amount

Figure 5.1 Transaction data structure

5.2 ML-Based detection and correction of transaction errors

This section presents our case study. Here, first we briefly present the basic concepts of transaction errors and their correction. Then we describe our approach to develop the ML component for automatic detection and correction of errors in retail transactions.

5.2.1 Transaction and Transaction Errors

By transaction (Fig. 5.1) we refer to a set of records (table rows) in the database related to a single order from a customer. A transaction consists of hierarchically organized components including metadata, item-level information (*e.g.,* price, product code, quantity, item-specific discounts) and the transaction-level information (*e.g.,* transaction type, taxes, discounts, payments). Each component can have zero or more sub-component(s). For example, an item can have none or several discounts applied on it. The relationships among the transaction components are governed by well-defined business rules. The transaction data are primarily contained in two (2) flattened tables: a Transaction Log (TLOG) representing the details of the transactions and a Process Log (PLOG) table logging the details of the changes to the transactions (in TLOG). A tuple of four (4) fields including the transaction index, transaction date, store number, and the timestamp, together uniquely identifying each transaction in TLOG. The same fields as in the TLOG table can be used as the key in the PLOG table for the identification of individual transactions. These key fields are used for mapping of transactions between TLOG and PLOG tables. Retail transactions created at POS terminals are sent to SAP's validation system. In case of validation failure, the cause of the failure (error) is logged in the process log table. These transactions are the candidates for corrections. All events (*e.g.,* creation, modification, *etc.*) related to transactions are logged in the PLOG table. Logs are written in batches and each row in the PLOG table can be associated with a range of transactions in TLOG. Transaction keys (from TLOG) are used to define the range of the transactions in PLOG. The PLOG table acts as a destination in which the logs for written for a transaction in TLOG, that is, it acts as a logging table. The error status column in PLOG can be used to detect the presence of an error while the changes to transaction column can identify its type. We cannot present example transactions due to a non-disclosure agreement.

5.2.2 The Problem

In SAP provided software solutions, retail transactions are created and processed in two primary phases. First, transactions are created in point-of-sale (POS) terminals and stored in the client company's server. These transactions are then forwarded to SAP's server for additional processing (*e.g.,* validation, audits, *etc.*) for different business functions. Transactions can be in an erroneous or inconsistent state for multiple reasons. More often than not, these errors stem from human errors (*e.g.,* incorrect manual input) rather than software or technical errors (*e.g.,* communication link failure). Additionally, transactions can have multiple errors at the same time, but SAP's software solutions only output one error at a time. Thus, multiple entries in the PLOG table can be recorded at different times, depending on the time of resolution of the previous error.

One might argue that these errors could be handled with perfect accuracy by hard-coded rules in transaction processing logic. However, the issue is that although SAP's solutions come with a rich set of features with core business rules, they give the client companies necessary flexibility to customize their software solutions to accommodate their business needs. For example, an individual store may need to customize product price, discount types and rates, loyalty benefits, *etc.* There might be different event and region specific offers which may be managed exclusively by individual stores. Similarly, there might be company-specific customization of the system that applies to all stores. While technically possible, these variabilities render hard-coded solutions infeasible in terms of cost. From a business operations points of view, this flexibility requirement cannot be compromised due to technical complexities of the transaction processing system.

Moreover, POS operations are real-time activities, and thus the technical issues need to be resolved immediately. For example, in the case of a communication link failure for a sales tax calculator service from a third party, the cashier may need complete transactions to keep service uninterrupted by compensating customers with reduced taxes or other similar measures. These transactions eventually need to be corrected to pass through the audit systems. Besides, there might be human errors in POS sales such as incorrect product code and price, discount code, amount *etc.* which may lead to unbalanced transactions. All these transactions become candidates for error detection and corrections. As mentioned earlier, defining hard-coded rules is not a viable solution for such error scenarios. Currently, tools exist in SAP's solutions to identify the root cause of the errors but the error resolution is a manual process and might become time consuming. Manual correction incurs high costs and it is a serious bottleneck to the business operations. Thus, we aim to develop ML-based solutions to leverage past correction examples for automatic detection and error resolution proposals with high accuracy. Here, our objective is to build machine learning models to detect (classify) and to correct (predict actions and the correct values) errors in retail transactions.

5.2.3 Overall Solution Architecture

As shown in Fig. 5.2, we extract transaction data from the database and extract features. We take a two-step approach. Our first model (Model 1) is a multi-label classifier that detects the types of errors in the erroneous input transaction. Each label in the classification corresponds to a specific error type (and location) of the error. Once we have the errors detected, we apply our second model (Model 2) to suggest values for the correction of the error. Model 2 is a multi-class classifier that predicts the correction value from the set of possible values for a transaction field. Model 2 is also fed the error information given by the first model.

5.2.4 Data Collection and Preprocessing

In our case, transaction data is stored in a large table (TLOG). The process log table (PLOG) captures all events and changes to transactions that are done either manually or automatically. Although there are millions of transactions in our data set, we are particularly interested in the erroneous transactions that were successfully corrected. Based on the error (task) status of the transactions from PLOG, we select corresponding erroneous transactions from the TLOG table. Again, the TLOG table contains the *final* state of the transactions. Thus, once corrected, only the corrected version of an erroneous transaction is available in the TLOG. However, to build a model that detects errors in transactions, transaction data needs to be in its erroneous state. Consequently, a transaction reversal process was designed and executed on transactions in TLOG based on the changelogs in the PLOG to extract the erroneous version of the transactions. This reversal process is also used to label our dataset for one of our models.

Data reception and quality As data comes from a SAP customer, said customer applies transformations to protect sensitive business and customer data before handing the data to SAP. The data as received by SAP the undergoes a secondary cleaning process before being transmitted to our team. Not all of the transactions with correction logs qualify for our analysis. In fact, out of the millions of rows received, only a small percentage qualifies as erroneous. Nevertheless, we filter out transactions that are missing necessary attributes or contain inconsistent field values, or that are outliers (with an excessive number of product items).

Pipeline As a step of data preprocessing, we developed data pipelines: our implementation strategy promoted code reusability and separation of concern, as rapid prototyping was needed to try on multiple features without losing data treated at a previous stage. We used a data extraction

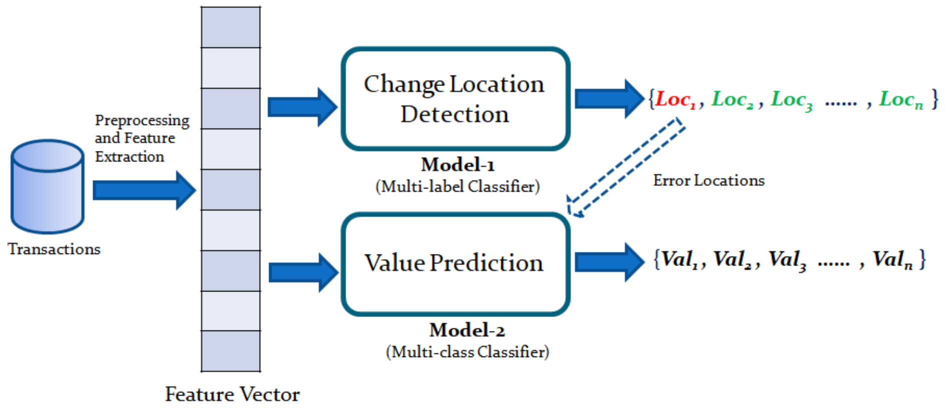

Preprocessing and Feature Extraction
Transactions
Feature Vector
Change Location Detection
Model-1
(Multi-label Classifier)
{Loc1, Loc2, Loc3 ...... , Locn }
Error Locations
Value Prediction
Model-2
(Multi-class Classifier)
{Val1, Val2, Val3 ...... , Valn }

Figure 5.2 Error detection and correction system architecture

module using User Defined SQL functions made with a connector to SAP's database. Data cleaning was done by scripts in a different module. Once settled and ready for model training, data underwent a final phase to create "model-ready" features. Each of these steps culminated in a different separate dataset.

5.2.5 Feature Selection

We analyzed the transactions in the TLOG and the corresponding correction logs in PLOG to understand the categories and characteristics of the errors. To infer the relationship between fields and how each field is related to the transaction errors, we analyze the error messages, associated field(s), correction steps and the changes in field values. We select table columns as the base features and we also compute some derived features. Our selection and derivation of base and derived features are based on the shared knowledge from domain experts and our understanding of the relationships between the components of the transactions. We iteratively refined our choice of the fields as features for the machine learning algorithms.

5.2.6 Feature Extraction

To extract features, we programmatically navigate the structures of the selected transactions to access transaction fields to extract base features. We also compute derived features from the extracted base features. Each feature vector is composed of features representing both transaction-level and item-level information. As transactions are variable in size depending on the number of items, we consider transactions only with maximum 20 retail items to avoid making the feature vector too sparse. This threshold is based on our analysis of the distribution of the number of retail items per transaction. We process all the qualifying transactions and store the features in Hierarchical Data Format (HDF) file for efficient processing. Our selection of features is based on knowledge from domain experts and our understanding of the relationships among the transaction components. We iteratively refined our choice of the transaction fields as features for ML algorithms.

5.2.7 Error Detection Approach

By error detection, we refer to the identification of the types and locations of errors. As transaction can have multiple errors, we model our error detection problem as a multi-label classification problem. Here, we label the feature vector for each transaction with a binary vector of length equal to the number of possible error classes. The label vector contains 1s in places where there is an error in the corresponding transaction field and 0s otherwise. We identify errors by examining the manual changes in the (PLOG) table. For example, if we detect a change in a tender type code, we infer that there was an erroneous tender type code. Thus, we set the bit in the label vector corresponding to the tender type code to 1. Once we have a labelled dataset, we build machine learning models to perform multi-label classification for error detection.

The dataset

For error detection, we need transaction data with correction examples. Therefore we select transactions that were detected as erroneous and extract their history in PLOG. We exclude erroneous transactions without corrections or that cannot be rebuilt to their original state by our reversal process. Then, we extract features from the erroneous version of the corrected transactions. Samples from other transactions without any errors are added to our dataset. These transactions satisfy the selection criteria (*e.g.,* number of items). We divide our dataset into three subsets for training, testing, and validation.

Multi-label Classifier

A first baseline was established by using a Decision Tree. We then investigated other model options, notably AdaBoost Classifiers, neural nets and Random Forests. In our case, Random Forests outperformed both AdaBoost and neural networks. Random Forest models can be applied for both classification and regression problems and can handle the possible overfitting problems present in Decision Trees.

A Random Forest is an ensemble learning method that constructs a multitude of decision trees. Classification and prediction decisions are based on the combination of outputs of the decision trees. Our Random Forest Decision Trees were built with the Gini impurity criterion.

We also attempted applying neural networks, specifically a Multilayer Perceptron (MLP), for error classification using the non-linearity sigmoid as the output layer. Various configurations of layers were attempted, with a hyperparameter search. However, because of our limited number of error correction samples (less than 100 to a few hundred for most error types), the neural net could achieve performance above baseline for most of the error types. Finally, we opted for Random Forest which achieved the best performance.

5.2.8 Error Correction Approach

By error correction, we refer to predicting the correct values for the erroneous transaction fields. The predicted class refers to a value from a known possible set of values. We model the value prediction problem as a multi-class classification problem and train a specialized model for each type of error. The features are extracted from the erroneous transactions in TLOG and the corresponding logs in the PLOG. We use the data reversal process described in 5.2.4 to gather old and new values. The label is determined by examining the new value and old value of the field related to the transaction error, changes that are recorded in the PLOG. The new value of a changed field of an erroneous transaction is set as the target label for the algorithm to predict. For the categorical field values (*e.g.,*, Tender Type Code), we need to predict the correct value from the set of alternative values for a given field. In such case, the target label is a one-hot encoded vector of possible values for the associated field.

The dataset

We build our error correction model based on the transactions with corrections available in the transaction database. We split the features into balanced data sets for training, testing, and validation.

Multi-class classifier

We formulate the error correction problem as a multi-class classification problem. Here, the classifier picks one of the values from all possible alternative values as the correct value. The label vector for each data sample represents a binary vector with '1' in the index of the correct value and 0's in all other places. We apply the Logistic Regression (one versus all) algorithm for the value-prediction problem. Hyperparameters are chosen using a grid search with cross validation. We also apply a Multilayer Perceptron (MLP) neural networks to predict values for error correction. Neural network models perform poorly compared to a simple Logistic Regression, due to limited correction data samples for most of the error types. Limited data size and class-imbalance issues are likely to blame for the poor performance of neural nets.

To measure performance of the models, we measure precision, recall, accuracy score and Jaccard similarity score of the models [110, 111] to evaluate model performance.. Metrics are defined as follows:

Precision Precision refers to the ratio of the number of correctly predicted or classified cases to the total number of cases. That is, we can define precision as,

$$P = \frac{TP}{TP+FP}$$

where, TP is the number of true positives and FP the number of false positive inferences.

Recall Recall refers to the ratio of the total number of correctly predicted or classified cases to the total number of true cases. Recall is defined as,

$$R = \frac{TP}{TP+FN}$$

where TN is the number of false negatives.

Accuracy Accuracy represents the ratio of the correct prediction relative to the total number of predictions. In a multi-label classification, the accuracy score refers to the subset accuracy. Let there are N samples and $y\prime_i$ be the predicted value for y_i, the i^{th} sample. Then the accuracy score can be defined as,

$$ACC = \frac{1}{N}\sum_{i=0}^{N-1} 1(y\prime_i = y_i)$$

Jaccard Similarity Jaccard Similarity or Jaccard coefficient is the ratio of the cardinalities of the intersection and the union of two sets of labels (predicted and true set). For binary classification this metric is equivalent to the accuracy score while they differ for multi-label classification. Let Y be the set of true labels and $Y\prime$ be the set of predicted labels. Then, Jaccard similarity is defined as,

$$J(Y, Y\prime) = \frac{|Y \cap Y\prime|}{|Y \cup Y\prime|}$$

We also measure the top-k accuracy score for the prediction of the correct values for recommendation. Here, top-5 accuracy score refers to the accuracy when the expected correct value appears in the top-5 ranked list of predictions from the model.

As each of these metrics reflects a different insight regarding the prediction or classification performance of the models, we measure all these metrics for our ML models to quantify model performance.

5.2.9 Performance

We measure precision, recall, accuracy score and Jaccard similarity score previously defined in order to measure performance of the models. Here, we present the performance of the models that detect a specific error: "Tender Type code" errors.

Error Detection

Fig. 5.3 presents the error detection performances of four different model configurations regarding error types. Here, Model1s, Model2s, and Model3s are to predict errors only in the first, second and third tender type code respectively. Model3a is trained to predict all of the first three tender type code errors. We present the error detection performance based on the Random Forest algorithm. Here, the first group of bars in Fig. 5.3 (Model1s) represents the error detection performance when

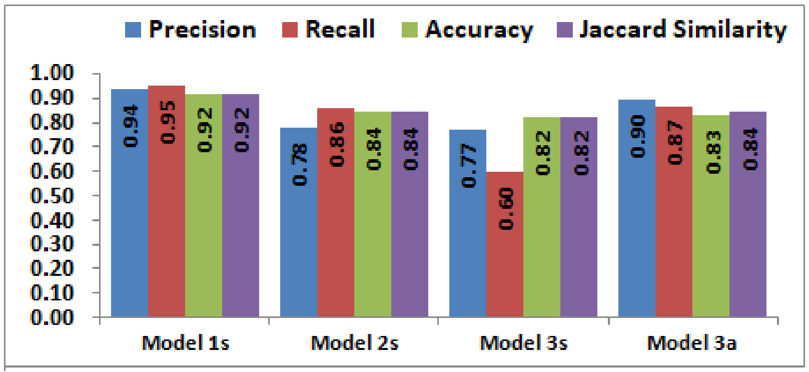


Figure 5.3 Error detection performance

only errors in the first tender type code are considered. Here, we observe that detection accuracy of the model are above 90%. However, for the second (Model 2s) and third (Model 3s) models, the detection performance drops. This decline in performance might be related to the comparatively lower number of training samples for each category. Again, when we include all the first three tender type codes in the detection model (Model 3a), we see a slight increase in the detection performance (above 80%).

Value Prediction

For value prediction we compute the accuracy of top-k recommendations of correction values. The performance is based on the results obtained on our best Logistic Regression model. Here, the value prediction accuracy of the model is 76% if we consider a single predicted value with the highest probability (*i.e.,* k=1). However, we also evaluate the prediction accuracy for multiple ranks, *i.e.,* top-k (k = [1, 5]). With $k = 5$ we achieve 93% accuracy.

5.2.10 Agile approach for research and development

For this work, we adopted the agile methodology (SCRUM [112]) in response the incremental and iterative nature of the problem and the need for periodic updates with SAP. SCRUM accelerates software development with faster delivery cycles while offering flexibility [90]. Our team comprises

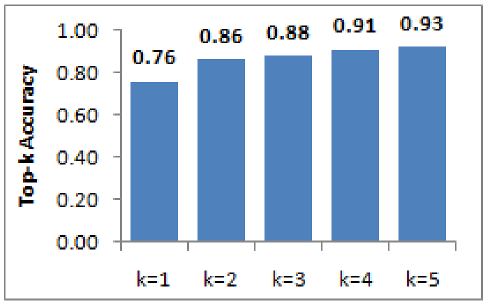


Figure 5.4 Top-k recommendation performance

members from academic and industry partners with diverse expertise in software engineering, machine learning, target software ecosystems and the business model of SAP. In our agile process, the length of each 'sprint' was 30 days with 'daily standup meetings' to provide updates on changes relating to the sprint goals.

We documented the challenges and discussed solutions to those challenges during our meetings. We later analyzed documentation to derive lessons on the challenges and best practices in each phase of ML application development. Each sprint ended with a "sprint demo" and a "retrospective" to address the backlog. The whole project spanned six sprints.

5.3 Lessons Learned

Machine learning applications have some distinct characteristics compared to traditional software applications. Thus, software developers and practitioners should be aware of a number of challenges or risk factors regarding Machine Learning applications [113]. We use the following methodology to reach our findings:

Methodology: We adopt our methodology based on the guidelines from Wohlin *et al.* [114] and Yin [115] to take a systematic approach for our case study. Here, our goal is to identify the challenges and to propose a set of guidelines for ML application development based on the synthesis of knowledge from the existing literature, and our practical insights from the case study. First, we review existing literature to explore software engineering methodology [68, 109, 116] for machine learning applications, common challenges [108, 117] and best practices [118, 119] for ML application development. Then we analyze the domain knowledge shared by the industry experts on the existing transaction processing workflow to understand the problem. The knowledge from the existing literature and domain experts helps us set up our study methodology and objectives. Then in each step of the development of the transaction error detection and correction system, we weigh the theoretical challenges against practical challenges in our problem context. This allows us to identify the gaps between theory and practices regarding the challenges in ML application development. Finally, we synthesize the extracted knowledge and our experience from the project towards the formulation of recommendations for best practices for practitioners for ML application development. We analyze the developers' feedback on the recommendations to qualitatively evaluate our proposed recommendations. Here, the development of the error detection and correction system is not our key research contribution and rather serves as an example case to derive insights for the practitioners.

To shed light on the common challenges in ML application development, we describe our findings from the case study along two different perspectives: software engineering (S) and machine learning (M). We explore their relationships to gain insights into the software engineering principles and practices for machine learning applications. We focus on each dimension and present different important challenges (RQ1) that need to be considered in a ML application development process.

5.3.1 Software Engineering Perspectives

Machine learning applications, like other software systems, need a well-defined software engineering process for their development and maintenance. However, given the distinct characteristics of ML applications, the phases of the software engineering process may need adjustments to accommodate the ML specific requirements. To mention, as ML application development is a special case of software development, many characteristics and challenges of ML application development phases are similar to that of traditional software engineering, which is not surprising. We discuss different phases of the software development life cycle (SDLC) for ML applications in as follows:

S1: ***Requirements Engineering*****:** Poor quality of requirements can lead to many issues in the phases of software development [120]. Requirements engineering for ML applications involves both ML specific and traditional requirements engineering activities such as feasibility analysis, requirements gathering, requirement specifications and validation. As the requirements in ML applications may change frequently, requirements specification for ML applications is a challenging task [121]. In our project, we gathered requirements from the discussions and demonstrations by the domain experts. We analyzed the transaction data structures, example error types, and their correction procedures. This is important since the functionalities of a ML component depend on information contained in the data. We iteratively refined our requirements specification based on the results of our different prototype models and the feedback from domain experts.

In our project, the requirements were gathered and shared to us by the industry partner. We observed that direct communication with the end-user and on-site observation of error correction scenarios is important to speed up requirement elicitation.

Iterative refinement of the requirements based on the feedback on prototype models from the stakeholders can help eliminating the differences in perception of the requirements. Otherwise, delayed identification of requirements and the consequent changes to the ML applications can be costly.

S2: ***Design*****:** Software design specifies the details of the scope, functionalities, and interactions of the software components. Traditional software systems comprise a finite number of states and are usually deterministic. However, for ML applications, the behaviour of the program can be unpredictable and defined by the training data. This makes the design of ML application a challenging task. ML applications require a large number of good quality data in order to perform well. Thus, the design of ML applications needs to accommodate the constraints and overhead of data extraction, data cleaning and data processing. As the algorithms and ML frameworks are evolving rapidly, ML application design should be flexible to adopt these changes.

Again, ML applications are data-driven and the performance of the system may degrade (*i.e.,* concept drift) over time despite no changes in the requirements or without the presence of bugs. Thus, the maintenance requirement of the ML applications may be hard to predict. In cases of adding ML capabilities to an existing application, the design should aim for minimum restructuring of the existing system architecture. Similarly, the design of a new ML application should be flexible in order to adapt to future changes. In our case, we added functionalities to the existing application and focused on the functional requirements of the modules and interfaces for interaction with the existing application.

S3: ***Implementation*****:** Development of ML applications commonly involves the use of many different frameworks and libraries [122]. However, it is can be challenging to put together a diverse set of frameworks and libraries while ensuring compatibility and integrity of the system. Again, ML models are a "Black Box" [123] and can be difficult to explain, depending on the nature of the model. [25, 124]. Thus, it is hard to clearly understand why sometimes they work and why sometimes they do not. In addition, ML models may exhibit robustness to noises [125] making it harder to verify the implementation.

Furthermore, the development environment for ML models might be different from the production environment. Thus, the implementation of ML applications should consider the target platform requirements while choosing frameworks and libraries. As the hardware-software ecosystem for ML applications are rapidly evolving, the implementation choices should also consider maximizing portability, compatibility, and adaptability of ML applications at lower cost.

S4: ***Integration*****:** Integration processes must ensure the functional integrity of the system by implementing appropriate interfaces among the subsystems. The integration of ML applications can be viewed as a two-step process: first integrating the sub-components of the ML component and then the integration of the ML components with other non-ML components. Thus, the defined interface between ML components with other components may influence the process and complexity of the integration phase. Again, ML models are expected to evolve continuously. Thus, the ML workflow should facilitate continuous integration of ML models. In our case, we needed to define a new interface for SAP application so that inference from the containerized ML models hosted either locally or on remote cloud can be accessed as services through APIs.

S5: ***Testing*****:** As the outcomes of machine learning models can be stochastic in nature [125,126], there might not be unique results to compare and verify machine learning applications (except with a predefined seed and tightly controlled environment). The rules learned by the ML models depend on many parameters such as the selected features, the model architecture and configuration [54] and repartition of the training data. The rules generated by ML systems might even be unknown to the developers [127]. This makes it harder to identify the erroneous system behaviours and to pinpoint the source of bugs in comparison to a traditional software system.

Again, ML algorithms may sometimes exhibit robustness to some bugs and produce reasonable outcomes by compensating for noisy data or implementation errors [125]. Thus, bugs in ML applications can be tricky to detect and fix. Testing ML applications may involve large-scale training data and whose manual labellingis costly. Moreover, a random selection of subsets of data is likely to fail to identify many edge cases. All these issues make testing and fixing of erroneous behaviour in ML applications a very challenging task. For our ML components, we evaluated the models' accuracy with the evaluation data set.

S6: ***Deployment*****:** The deployment phase puts the software system into production either by updating or replacing the existing system. For ML applications, this phase is more likely to add new functional modules to the existing system. One important challenge to consider in ML system deployment is that the platform and infrastructure for the production system might be very different from the environment in which the ML model was trained and evaluated [122]. These differences can represent compatibility, portability and scalability challenges and may affect the performance.

Our design aimed at creating a ML-based service for the existing applications. We tested our model deployment as an API-based web service for error detection and correction. We took different factors into account for the deployment of our models, such as scalability, performance, resource requirements and model maintenance.

5.3.2 Machine Learning Perspectives

ML application development has some specific aspects to consider as a software development project. The distinct characteristics and requirements of the ML application requires a well-defined set of principles and guidelines as recommended by practitioners [118].

M1: ***Problem Formulation*****:** Machine Learning algorithms offer general-purpose solutions. We need to formulate problems appropriately to fit into the ML-based solution space. An incorrect problem formulation may lead to failure of a ML application. Some key challenges in formulating a ML problem are that one should clearly understand the problem, the algorithms and the mapping of the problem from the original domain to the ML solution space. This requires a ML solutions architect to have a diverse set of skills and expertise. The correct formulation of the problem is a prerequisite to the success of other phases of ML application development such as data acquisition, selection and extraction of features, as well as model building.

M2: ***Data acquisition*****:** Collection and processing of large volume of data are critical overheads for machine learning [128]. Insufficient data is also a problem for machine learning applications. The data acquisition must focus on the completeness (*i.e.,* covering all use cases), accuracy (correctness of the data), consistency (no contradictory data), and timeliness (relevant to the current state of the system) of data to ensure data quality [129]. However, maintaining data quality requires careful steps in collection, curation and maintenance. This is often very expensive regarding the time and associated manual labour [129].

In our study, the transaction data was provided by SAP on behalf of a client company. While SAP's software solutions provide a comprehensive set of features, clients have the flexibility to customize and extend them resulting in new data structure and values. These customizations add challenges to the generalization power of our models for the same uses cases as each client is a specific instance of a general problem. Our analysis suggests that some prior analysis of the data and the problem is important to set appropriate data requirements.

M3: ***Preprocessing*****:** Raw data may require preprocessing for cleaning, organization, completion and transformation. Noisy data is claimed to be the top challenge for ML practitioners. Noisy data can adversely affect the learning and thus the inference of the models. Data preprocessing is a challenging phase for machine learning [130] and may incur a significant proportion (>50%) of time and effort [131]. In our study, we first analyzed transaction data to understand the attributes of the transactions. We observed that the overall structure of the data, its types and the range of values of individual fields may demand careful attention and warrant specific preprocessing tasks to ensure correctness and consistency of the data.

To carry such preprocessing tasks, we needed tools and libraries that would allow for efficient exploration and experimentation for data processing and ML algorithms. Therefore, we implement our own data pipeline using these libraries in order to provide efficiency and flexibility to our machine learning workflow. We found these pipelines to be of great help.

We established multiple criteria on transaction data as a filtering step. Our preprocessing steps include filling missing values as well as normalization of data fields to correct formatting differences.

M4: ***Feature Extraction*****:** Feature extraction performs transformation of input data into feature vectors for ML algorithms [132]. Features should best represent the hidden characteristics of the data for machine learning. Feature extraction also aims to remove the noise and redundancy from the data [132]. Feature extraction can also be used to find lower-dimensional representations of the original data to improve the speed of the training and inference. However, it can be challenging to extract features when processing high volumes of data when prototyping. For example, finding a lower-dimensional representation with algorithms such as PCA or SVD is usually done with the whole dataset loaded. The large scale of the data makes a traditional non-iterative method of computation intractable. In our case, we considered the domain knowledge, the transaction processing workflow, and the transaction data structure and relationships as the basis of feature selection. In addition to base features, we also extracted derived features for our ML models.

M5: ***Model Building*****:** ML models are created based on specific ML algorithms depending on the problem and the characteristics of the data. One may use existing models from libraries or may create custom models. One frequent problem for ML models is 'overfitting'. In this case, a model performs well on the training dataset but does not generalize, thus being referred to as having overfitted on the training data. This might be because the model is too complex as a result of the higher-dimensional features and a complex or deeper architecture of the model such that every case of the training data is merely stored as a weight inside the model. The solution is to find the simplest model that achieves acceptable performance. However, overly simple models may fail to capture the hidden patterns in the data causing 'underfitting'. It is also important to make sure that data distribution is balanced across data and labels. Otherwise, the inference of the model might be biased towards the dominating class of the training data. In our project, we observed overfitting issues with neural networks of comparatively deeper architecture for error classification. We eliminated feature redundancies and tuned neural network architectures to avoid overfitting. Class imbalance was another important issue as the distribution of error samples were very skewed to a limited class of errors. To overcome this issue we trained using stratified sampling.

M6: ***Evaluation*****:** ML models are evaluated on their performance on the test dataset which is separated from the training and validation datasets. Both pre-deployment evaluation and post-deployment performance monitoring are important as the performance may drop with changes in the input data characteristics. Thus, ML models may need to be updated (*e.g.,* retrained) to adapt to changes. Since our solution comprises multiple models chained, with the output from the first give as input to the second, there is a direct effect on model performance going from the first model to the second. Consequently, the performance of each individual model may only represent a part of the end-to-end scenarios. Thus, it is important to have both model-level and system-level evaluations. In our study, we evaluated the individual models and also the overall performance after the integration of the models.

M7: ***Model integration and deployment*:** Trained models are integrated into the target application. This integration involves putting all necessary components (e.g., models, input-output pipelines) together. For multiple models, it may require to define and implement the interface for each model to interact with other models and system components. One common approach is to deploy ML models as services and access the services through an API. The deployment should consider the portability and compatibility of the ML models with respect to the target platform. In our project, we develop an interface between the ML-based components and the existing application. This interface allows us to provide the error detection and correction services.

M8: ***Model management*:** ML model management involving creation, deployment, monitoring, and documentation of ML models are a challenging task in ML workflows [108]. Model performance on new data may change over time in comparison to its performance when initially trained. It is thus important to monitor the performance of the deployed models, track changes in the data characteristics, and also, retrain and revalidate the models. These require iterations on the entirety of the ML model life-cycle activities which are often very expensive regarding time and resources. It is also important but hard to keep track of the model versions, dataset and configurations to allow *reproducibility* of ML models [68] and to make the management of ML workflows easier [133]. Model reproducibility helps us to analyze and compare model behaviour and performance, and also supports deployment or roll out decisions. We defined policies to version data and to keep track of models and configurations to support a comparative analysis of our ML models.

M9: ***Ethics in AI development*:** It is very important to ensure that the use AI and ML conforms with ethical standards to avoid any negative consequences. Researchers and practitioners should perform "responsible" use of AI [119]. The teams involved in the research and development of any ML application should adhere to the standard code of ethics for Software Engineering [134]. In our project, we maintained strict data privacy and security policies. Personal information was filtered out from the transactions prior to access and all the team members agreed to a non-disclosure agreement that protects the privacy and security of personal and business-sensitive information.

5.3.3 Research Collaboration Perspectives

Effective collaboration is a key requirement for the success of a collaborative project. Collaboration must aim to bring the best out of the partnering teams to advance towards their research and development goals. However, multi-partner (more specifically industry-academia) collaboration adds some inherent challenges [90] in terms of communication, interactions, and understanding of the problem. We shed some light on some collaboration challenges below:

C1: ***Problem Understanding*:** In a collaborative project, it is important for the partners to have the same or a similar understanding of the problem. Collaborations may be based on an academic research idea or a problem from the industry. The collaborative teams must understand the problem clearly to translate it into an appropriate research problem. The research aspects of the problem might be more transparent to the academic team(s) while the industry partner(s) may share domain knowledge as well as the business cases relevant to the problem.

C2: ***Focus on objectives*:** In an industry-academia collaborative project, there might be differences in prioritizing the objectives by individual partners. For example, the academic collaborators may naturally focus more on the research outcomes and may be more inclined to measure success in terms of research findings. Industry partners, on the other hand, might be more focused on maximizing the business values from the investment of time and resources. As the differences in objectives are not mutually exclusive, the differences in priorities may impact positively on the overall outcomes of the project. Thus, the heterogeneity in partnerships is more likely to enforce a balance between focuses on research and development. This, in turn, will translate the research outcomes into more practically usable deliverables as product and services to add business values.

C3: ***Knowledge Transfer*:** Knowledge transfer is an important driver of innovation and economic growth in industry-academia collaborations [1, 88, 135, 136]. In industry-academia collaboration in software engineering research, one of the key objectives is to transfer the research outcomes (*e.g.,* knowledge and technology) from the academic partner(s) to the industry partner(s) and vice versa. Such collaboration bridges the gap between research and practice. Studies show that technological relatedness and technological capability are important facilitators for knowledge transfer while tacitness and ambiguity may affect knowledge transfer [137]. Moreover, the differences in organizational cultures may influence the interpretation of behaviour, interactions, and knowledge.

C4: ***Professional Practice*:** In an industry-academia collaboration, the heterogeneity of team members with research and professional software development background is an asset. The blend of research, development skills and domain expertise add diversity in the overall team skill set. However, professional practices and work processes are expected to be different in academia and industry. The workplace practices might be different due to the nature of day-to-day work. In industry, the work process is generally more formal and might be influenced by corporate business cultures. While in academia, the work processes are more outcome-focused with a comparatively flexible or less formal process. Difference in skill sets of team members may be hard to transfer between members. Indeed ML is a discipline in which statistics and mathematics are priors needed to fully understand and leverage its power. Transferring this knowledge to partners that may not have same background knowledge can thus become more difficult. We find that explanation offered by each team with special skills sets is important. Many studies acknowledged these cultural differences relevant to the industry-academia collaboration [138].

C5: ***Data Privacy and Security*****:** Data privacy and security is crucial when business data with sensitive user information is subject to analysis for research and development. Application of machine learning may require access to a large volume of such data. Thus, the privacy and security of data are highly important. In addition to ensuring the privacy and security, use of data for AI must comply with ethical standards (see M8). Usually, data privacy and security are governed by the agreement among the partnering organizations. The team members need to be trained on the privacy and security of the data and associated resources (e.g., devices, networks).

5.3.4 Recommendations

Researchers and practitioners in software engineering and machine learning are in need of a consolidated set of guidelines and recommendations for research and development of ML applications. We summarize our guidelines or recommendations (RQ2) from the perspective of our identified dimensions of challenges for ML applications. It should be mentioned that many of the recommendations we propose for ML application development also apply to traditional software systems engineering. However, we present them in specific context of ML application development.

Software engineering perspective

From the software engineering perspective, we recommend the following guidelines to address the challenges we identified in Section 5.3.1.

- Requirements Engineering
  - ML models are data-driven and thus it is important to analyze the feasibility regarding data requirements for the intended ML-based solutions as early in the project as possible.
  - Applications can have ML-specific and non ML-specific requirements. Developers should be aware of any conflicts in the requirements and adapt to them accordingly.
  - A late discovery of a requirement can be costly. Thus, communication with the end users is important to keep track of frequently evolving requirements.
  - Iterative prototyping of multiple ML models and along with stakeholder feedback can help elucidate more requirements and improve data collection.
- Process†

  - Do use version control on works so that it is easy to track changes, collaborate and version artifacts.

- Design
  - ML applications are expected to be modular in design as modularity in order to offer separation of concerns and reusability. Each component can be developed with cohesive functionalities. The overall system is built by the integration of interacting modules.
  - ML components may evolve faster than other system components due to frequent changes in requirements and data. The design of ML components thus should be lightly coupled to accommodate changes with minimum effort and cost.
  - ML applications require a large volume of good quality data. Therefore, the design must accommodate the appropriate data handling mechanism in the system.
  - As the models or components need to be integrated through appropriate interfaces, integration strategies must be reflected in the design.
- Implementation
  - The implementation of ML applications should aim to use a cohesive set of frameworks and libraries.
  - The implementation should consider the target platform of the application for scalability, compatibility, and portability.
- Integration
  - The integration must ensure functional integrity of the system.
  - The integration process should reflect the design of the system.
- Testing
  - ML models can be opaque and it is usually hard to explain the model's behaviours. ML models need to be tested rigorously in a wide variety of settings and for all possible range of use case scenarios.
  - Bugs in ML application are hard to detect because of the stochastic nature of ML systems and the various non-determinism introducing factors that affect performance.
  - †Data processing and feature extraction should be unit-tested in order to avoid having noisy data that could affect the whole pipeline
- Deployment
  - The deployment should consider platform-specific differences in development and production environments.

  - The deployment of ML applications in a production environment must consider portability and scalability requirements.
  - †The deployment versions should be pinned to a verified set of dependencies upon which the statistical test was done.
  - †The deployment should consider having a complete self-contained artifact, such as containers.
  - The deployment and roll-out strategy should be carefully designed in order to avoid side effects in system users.

Machine learning perspective

From the ML perspective, we recommend the following guidelines to address the challenges we identified in Section 5.3.2.

- Problem Formulation
  - The correct formulation of a problem as a ML problem is a prerequisite for the success of ML applications. Formulation of ML problems must be based on a clear understanding of the target problem and the characteristics of the data.
  - ML algorithms offer general-purpose solutions. For this reason, developers must understand the available algorithms before selecting one or many for a particular problem.
- Data Acquisition
  - The data collection process should ensure the completeness, accuracy, consistency, and timeliness of the dataset.
  - As the structure of the data may evolve over time, the data acquisition process should be flexible in order to easily adapt to changes in the data structure and organization.
  - The data requirements should be set by necessary analysis prior – on a smaller subset of data – to the acquisition of a large volume of data. This may save considerable amount of time and resources in the ML workflow.
- Data Preprocessing
  - As the quality of a ML model heavily relies in the quality and abundance in data, raw data will need cleaning and preprocessing to remove noise, fill in the missing value. Additional transformations are then needed to make this data feedable to models.
  - It is possible that the required preprocessing processes be common to modules. Thus, the overall workflow should maximize the reusability of the preprocesed data.

- Feature Extraction
  - The quality of features greatly influences the performance of the ML models. Consequently the developers must find the best set of features for the appropriate type of ML algorithm.
  - An automated pipeline should be built for feature extraction as it is time and resource consuming.
  - Noises from the features should be cleaned as they have adverse effects on the model performance.
  - The complexity of the problem, model and data directly impact the complexity of designed features. Therefore, features may need to be represented in a lower-dimensional space while preserving hidden characteristics.
  - There should be a periodic review of the features [118].
- Model Building
  - The ML algorithms need to be tailored based on the problem and the characteristics of the data.
  - ML developers should start with a simpler solution (build a baseline model) and gradually adopt more complex solutions while considering the resource-performance trade-off.
  - The developers must ensure quality of data and its balanced distribution.
  - Whenever possible, developers should consider reusing existing solutions (for productivity) before opting for customized or new solutions.
- Model Evaluation
  - The evaluation dataset should be complete enough to represent all possible use case scenarios.
  - Model evaluation should focus on precision/recall/accuracy but also on other factors such as throughput, resource utilization, and scalability.
  - When different models interact, both model-level and system-level performances need to be evaluated.
  - †ML models need to be under some statistical tests to verify their generalization ability.
  - †Verify that the training process is not blocked by checking the gradients when problems arise
- Model Integration and Deployment

  - The Integration of ML components must ensure the functional integrity of the ML applications.
  - The deployment should consider the compatibility and portability issues regarding development and production platforms.
  - The deployment must ensure a smooth roll-out of the existing system without affecting users or the business process.

- Model Management
  - Post-deployment monitoring of models is important. Based on performance, models may need to be retrained which may initiate a maintenance cycle.
  - Newly ingested data needs to be monitored in order to quantify its characteristics such as distribution that may change over time.
  - Building a model with the desired performance requires exploration and experimentation. Versioning of the models, data, and configurations is important to facilitate reproducibility of ML models.
  - Each phase of the ML life-cycle should be well documented to support model maintenance and reproducibility.

- Ethics in AI development
  - Development of ML applications must adhere to AI ethics principles to ensure responsible use of AI.
  - Privacy and security of personal and business data must be preserved.
  - Collective well-being must get priority over business gains in the use of AI and ML.

Industry-academia collaboration perspective

We recommend the following guidelines to address the challenges from the industry-academia collaboration perspective presented in Section 5.3.3:

- Problem Understanding
  - All members of the team should have the same understanding of the problem. Perception sharing of perceptions can be useful to understand the problem from diverse perspectives.

- Focus on Objectives
  - The success of the collaboration largely depends on the collective focus of the teams. The focus should thus be on the commonalities of the objectives rather than differences.

  - Objectives need to be prioritized with consensus. Differences in objectives should be addressed in a cooperative manner rather than a competitive manner.
- Knowledge Transfer
  - Knowledge and technology transfer is a primary objective of industry-academia collaboration. Frequent interactions can be useful in easing knowledge transfer.
  - Identification of the differences in knowledge and in professional backgrounds can ease knowledge transfer.
  - Training on academic and industrial perspectives may bridge the knowledge gaps among the partners.
- Professional Practice
  - It is important to reach a common ground despite the differences in professional practices and institutional cultures in order to collectively focus on the objectives.
  - Interactions and collaborations can be useful to bridge the gap in professional cultures.
- Data Privacy and Security
  - Policies must be defined and communicated to the teams to ensure the privacy and security of sensitive data.

5.4 Conclusion

Artificial intelligence and ML are being increasingly adopted in modern software applications. Offering diverse benefits, ML is also adding new challenges in the software development process. We shared our experience on the development of ML-based automatic detection and correction of errors in retail transactions. We outlined our detailed approach to solve this important real-world problem in the retail business. We identified challenges in ML application development from software engineering and machine learning perspectives. Based on our experience building these ML components and the principles and practices in software engineering and machine learning, we report some important insights. We also highlight recommendations we believe will be useful to researchers and practitioners embarking in engineering ML applications.

CHAPTER 6 CONCLUSION

6.1 Summary of Works

In this thesis, we have proposed a methodology for measuring the impact of bug occurrence in a machine learning framework on its ability to train ML models. Since there is a lack of tooling for controlling NDIF in ML workflows for reproducibility, we have created a framework to control as much of these NDIF. That is, our framework, `ReproduceML` allows to configure a runtime in the most deterministic fashion possible for the purpose of running ML training procedures. Furthermore, `ReproduceML`'s ability to hold and serve as a metric collection server allows for reusability.

We followed this methodology on the **PyTorch** framework and gathered preliminary results on the effects of bugs on a model's performance after training. We applied a nonparametric statistical test on our experiments: the Wilcoxon-Mann-Whitney U test (non-paired). Our early results report that there is no statistically significant difference on the ability of a model's to perform well due to bugs, based on our limited dataset. We also highlighted the challenge of reproducibility throughout this thesis and provide an updated list of factors that impact non-determinism in the ML workflow.

We have also detailed our approach in implementing ML models in order to solve practical problems of an industry partner with whom a collaboration was established. In light of this industry-academia collaboration, we documented our approach to collaboration by identifying the challenges that inhibit efficient collaboration. Finally, we propose a set of recommendations based on our experience, for both parties in order to improve future industry-academia machine learning systems development.

These recommendations categorized into 3 main categories, *i.e.,* Software engineering, Machine learning, and Industry-Academia relations, should help ML practitioners who are building ML systems to build reliable, reproducible, high quality systems for the industry.

We have also discussed the implications of non-determinism for the Industry and the Academia in terms of publication and results.

In summary, we have created the tooling necessary in order to measure empirical data points of ML training procedures and conduct empirical studies based on such approaches while reducing confounding factors that introduce additional variance to the stochastic nature of ML training procedures.

Our works for bug extraction can be found at `https://github.com/swatlab/ml-framework-bugs` and the framework at `https://github.com/swatlab/ml-frameworks-evaluation`.

6.2 Limitations

The main limitation of our preliminary empirical study is knowing if bugs were triggered during runs by doing something other than a manual verification. This limitation is the main bottleneck for systematic automated research. Even then, this manual verification is done on bug reproducibility data, there is no guarantee that the bug is triggered. We have proposed an updated methodology and created the tooling necessary to carry such a second phase of the study.

Considering the above, great care must be taken when analyzing results. Indeed, similar empirical studies on the analysis of variance have indicated that there is an expected variance due to the NDIF, even by controlling their experimental seed and other factors. Notably, considering there is a variance that is still uncontrollable due to lack of appropriate tooling, even if the bugs could have had an impact, if this impact is considerably smaller than the NDIF induced variance, no effect would have been observed.

When it comes to `ReproduceML`, the server side could benefit from integrating recent tools such as **MLflow** [68] and **DVC** [73] that incorporates advanced configuration and data versioning aspects. The client side, responsible for runtime configuration and NDIF control is mainly tied to the reproducibility capabilities in the inner ML frameworks, but could integrate future works such as **GPUDet** [139] and **FlexPoint** [104].

6.3 Future Research

We highlighted the possible future research for the effects of NDIF in machine learning workflow by first providing improvements on our methodology, by using tools already developed. In essence, the future works for NDIF effects upon ML systems would be to measure the impact of software versioning, deprecation usage on the performance of models both on the metric and on the speed of execution. Future research can also be conducted on the various sources of non-determinism we established in Section 3.2.2, as each research in this direction will prove to help reproducibility in machine learning systems.

When it comes to deploying ML systems in the industry, future works would be to further understand the key problems to industry-academia collaborations by engaging in new partnerships while following our recommendations. Furthermore, these new collaborations will allow for a richer set of new recommendations to emerge.

REFERENCES

[1] V. Garousi *et al.*, “Characterizing industry-academia collaborations in software engineering: evidence from 101 projects,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 2540–2602, apr 2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[3] Association for Computing Machinery. Artifact review and badging - current. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[4] National Academies of Sciences, Engineering, and Medicine, *Reproducibility and Replicability in Science.* The National Academies Press, sep 2019. [Online]. Available: https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science

[5] T. Elliot. The state of the octoverse: machine learning. [Online]. Available: https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/

[6] Intel Corporation. Intel® math kernel library. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

[7] B. K. Brown. Intel® math kernel library for deep learning networks: Part... [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-dnn-part-1-library-overview-and-installation.html

[8] “Ieee standard for binary floating-point arithmetic,” *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985.

[9] O. Villa *et al.*, “Effects of floating-point non-associativity on numerical computations on massively multithreaded systems,” in *Proceedings of Cray User Group Meeting (CUG)*, 2009, p. 3.

[10] S. Collange *et al.*, “Full-speed deterministic bit-accurate parallel floating-point summation on multi-and many-core architectures,” *submitted Feb*, vol. 28, 2014.

[11] N. Zmora *et al.*, “Neural network distiller: A python package for dnn compression research,” October 2019. [Online]. Available: https://arxiv.org/abs/1910.12232

[12] PyTorch. Quantization - pytorch 1.7.0 documentation. [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[13] Y. Choukroun *et al.*, "Low-bit quantization of neural networks for efficient inference," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, oct 2019.

[14] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach *et al.*, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[15] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, pp. 452–454, may 2016.

[16] J. Freire, P. Bonnet, and D. Shasha, "Computational reproducibility," in *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. ACM Press, 2012.

[17] J. Gardner *et al.*, "Enabling end-to-end machine learning replicability: A case study in educational data mining," *arXiv preprint arXiv:1806.05208*, 2018.

[18] B. K. Olorisade, P. Brereton, and P. Andras, "Reproducibility in machine learning-based studies: An example of text mining," 2017.

[19] F. Renard *et al.*, "Variability and reproducibility in deep learning for medical image segmentation," *Scientific Reports*, vol. 10, no. 1, aug 2020.

[20] S. N. Goodman, D. Fanelli, and J. P. A. Ioannidis, "What does research reproducibility mean?" *Science Translational Medicine*, vol. 8, no. 341, pp. 341ps12–341ps12, jun 2016.

[21] C. Jansen *et al.*, "Reproducibility and performance of deep learning applications for cancer detection in pathological images," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, may 2019.

[22] S. Marrone *et al.*, "Reproducibility of deep CNN for biomedical image processing across frameworks and architectures," in *2019 27th European Signal Processing Conference (EUSIPCO)*. IEEE, 9 2019.

[23] L. K. John, G. Loewenstein, and D. Prelec, "Measuring the prevalence of questionable research practices with incentives for truth telling," *Psychological Science*, vol. 23, no. 5, pp. 524–532, apr 2012.

[24] B. Marwick, "Computational reproducibility in archaeological research: Basic principles and a case study of their implementation," *Journal of Archaeological Method and Theory*, vol. 24, no. 2, pp. 424–450, jan 2016.

[25] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[26] V. Stodden and S. Miguez, “Best practices for computational science: Software infrastructure and environments for reproducible and extensible research,” *SSRN Electronic Journal*, 2013.

[27] S. M. Crook, A. P. Davison, and H. E. Plesser, “Learning from the past: Approaches for reproducibility in computational neuroscience,” in *20 Years of Computational Neuroscience*. Springer New York, 2013, pp. 73–102.

[28] S. R. Piccolo and M. B. Frampton, “Tools and techniques for computational reproducibility,” *GigaScience*, vol. 5, no. 1, 07 2016, s13742-016-0135-4. [Online]. Available: https://doi.org/10.1186/s13742-016-0135-4

[29] A. Davison, “Automated capture of experiment context for easier reproducibility in computational research,” *Computing in Science & Engineering*, vol. 14, no. 4, pp. 48–56, jul 2012.

[30] V. Stodden, “Beyond open data: A model for linking digital artifacts to enable reproducibility of scientific claims,” in *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems*. ACM, jun 2020.

[31] J. Vanschoren *et al.*, “Experiment databases,” *Machine Learning*, vol. 87, no. 2, pp. 127–158, jan 2012.

[32] F. Z. Khan *et al.*, “Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv,” *GigaScience*, vol. 8, no. 11, 11 2019, giz095. [Online]. Available: https://doi.org/10.1093/gigascience/giz095

[33] S. Cohen-Boulakia *et al.*, “Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities,” *Future Generation Computer Systems*, vol. 75, pp. 284 – 298, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300316

[34] J. Vanschoren *et al.*, “Openml: Networked science in machine learning,” *SIGKDD Explor. Newsl.*, vol. 15, no. 2, p. 49–60, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2641190.2641198

[35] M. H. de Vila *et al.*, “MULTI-x, a state-of-the-art cloud-based ecosystem for biomedical research,” in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, dec 2018.

[36] A. Naldi *et al.*, “The colomoto interactive notebook: Accessible and reproducible computational analyses for qualitative biological networks,” *Frontiers in Physiology*, vol. 9, p. 680, 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fphys.2018.00680

[37] R. S. Olson *et al.*, “PMLB: a large benchmark suite for machine learning evaluation and comparison,” *BioData Mining*, vol. 10, no. 1, dec 2017.

[38] S. Jean-Paul *et al.*, “Issues in the reproducibility of deep learning results,” in *2019 IEEE Signal Processing in Medicine and Biology Symposium (SPMB)*, IEEE. IEEE, dec 2019, pp. 1–4.

[39] E. Keogh and S. Kasetty, “On the need for time series data mining benchmarks: A survey and empirical demonstration,” *Data Mining and Knowledge Discovery*, vol. 7, no. 4, pp. 349–371, 2003.

[40] S. Sonnenburg *et al.*, “The need for open source software in machine learning,” *Journal of Machine Learning Research*, vol. 8, no. Oct, pp. 2443–2466, 2007.

[41] H. V. Pham *et al.*, “Problems and opportunities in training deep learning software systems: An analysis of variance.”

[42] M. Alberti *et al.*, “Improving reproducible deep learning workflows with DeepDIVA,” in *2019 6th Swiss Conference on Data Science (SDS).* IEEE, jun 2019.

[43] J. Pineau, “Reproducible, reusable, and robust reinforcement learning (invited talk),” *Advances in Neural Information Processing Systems*, 2018.

[44] P. Henderson *et al.*, “Deep reinforcement learning that matters,” *arXiv preprint arXiv:1709.06560*, 2017.

[45] C. Liu *et al.*, “On the replicability and reproducibility of deep learning in software engineering,” 2020.

[46] O. E. Gundersen and S. Kjensmo, “State of the art: Reproducibility in artificial intelligence.” in *AAAI*, 2018, pp. 1644–1651.

[47] T. Raeder, T. R. Hoens, and N. V. Chawla, “Consequences of variability in classifier performance estimates,” in *2010 IEEE International Conference on Data Mining.* IEEE, dec 2010.

[48] K. Khetarpal *et al.*, “Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms,” 2018.

[49] M. Crane, “Questionable answers in question answering research: Reproducibility and variability of published results,” *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 241–252, dec 2018.

[50] Q. Guo *et al.*, “An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, nov 2019.

[51] A. Labach, H. Salehinejad, and S. Valaee, “Survey of dropout methods for deep neural networks,” *arXiv preprint arXiv:1904.13310*, 2019.

[52] Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-deep-learning-boost-new-instruction-bfloat16.html

[53] G. Henry, P. T. P. Tang, and A. Heinecke, “Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations,” in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 69–76.

[54] N. S. Keskar *et al.*, “On large-batch training for deep learning: Generalization gap and sharp minima,” *arXiv preprint arXiv:1609.04836*, 2016.

[55] D. Masters and C. Luschi, “Revisiting small batch training for deep neural networks,” *arXiv preprint arXiv:1804.07612*, 2018.

[56] M. Hardt, B. Recht, and Y. Singer, “Train faster, generalize better: Stability of stochastic gradient descent,” in *International Conference on Machine Learning*. PMLR, 2016, pp. 1225–1234.

[57] Q. Meng *et al.*, “Convergence analysis of distributed stochastic gradient descent with shuffling,” *Neurocomputing*, vol. 337, pp. 46–57, apr 2019.

[58] P. Nagarajan, G. Warnell, and P. Stone, “The impact of nondeterminism on reproducibility in deep reinforcement learning,” 2018.

[59] P. Mattson *et al.*, “Mlperf training benchmark,” *arXiv preprint arXiv:1910.01500*, 2019.

[60] N. Mahmoud *et al.*, “DLBench: An experimental evaluation of deep learning frameworks,” in *2019 IEEE International Congress on Big Data (BigDataCongress)*. IEEE, jul 2019.

[61] C. Coleman *et al.*, “Dawnbench: An end-to-end deep learning benchmark and competition,” *Training*, vol. 100, no. 101, p. 102, 2017.

[62] ——, “Analysis of the time-to-accuracy metric and entries in the dawnbench deep learning benchmark.”

[63] R. Adolf *et al.*, “Fathom: reference workloads for modern deep learning methods,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, sep 2016.

[64] baidu-research. (2016) Deepbench: Benchmarking deep learning operations on different hardware. [Online]. Available: https://github.com/baidu-research/DeepBench

[65] H. Zhu *et al.*, “Benchmarking and analyzing deep neural network training,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, sep 2018.

[66] L. Liu *et al.*, "Benchmarking deep learning frameworks: Design considerations, metrics and beyond," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).* IEEE, jul 2018.

[67] S. Shi *et al.*, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD).* IEEE, nov 2016.

[68] M. Zaharia *et al.*, "Accelerating the machine learning lifecycle with mlflow." *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[69] European Organization For Nuclear Research and OpenAIRE, "Zenodo," 2013. [Online]. Available: https://www.zenodo.org/

[70] V. Stodden *et al.*, "Enhancing reproducibility for computational methods," *Science*, vol. 354, no. 6317, pp. 1240–1241, dec 2016.

[71] R. Isdahl and O. E. Gundersen, "Out-of-the-box reproducibility: A survey of machine learning platforms," in *2019 15th International Conference on eScience (eScience).* IEEE, sep 2019, pp. 86–95.

[72] D. Atkins *et al.*, "Using version control data to evaluate the impact of software tools: a case study of the version editor," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 625–637, jul 2002.

[73] R. Kuprieiev *et al.*, "Dvc: Data version control - git for data & models," 2020.

[74] L. Vogtlin, V. Pondenkandath, and R. Ingold, "Cobra: A CLI tool to create and share reproducible projects," in *2020 7th Swiss Conference on Data Science (SDS).* IEEE, jun 2020.

[75] F. Chirigati *et al.*, "Reprozip: Computational reproducibility with ease," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2085–2088. [Online]. Available: https://doi.org/10.1145/2882903.2899401

[76] S. Azarnoosh *et al.*, "Introducing PRECIP: An API for managing repeatable experiments in the cloud," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science.* IEEE, dec 2013.

[77] M. Orzechowski *et al.*, "Reproducibility of computational experiments on kubernetes-managed container clouds with hyperflow," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya *et al.*, Eds. Cham: Springer International Publishing, 2020, pp. 220–233.

[78] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, feb 2015.

[79] X. Sun *et al.*, “An empirical study on real bugs for machine learning programs,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, dec 2017.

[80] F. Thung *et al.*, “An empirical study of bugs in machine learning systems,” in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, nov 2012.

[81] M. J. Islam *et al.*, “A comprehensive study on deep learning bug characteristics,” *arXiv preprint arXiv:1906.01388*, 2019, submitted for publication.

[82] Y. Zhang *et al.*, “An empirical study on tensorflow program bugs,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[83] V. Garousi, K. Petersen, and B. Ozkan, “Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review,” *Information and Software Technology*, vol. 79, pp. 106–127, nov 2016.

[84] J. C. Carver and R. Prikladnicki, “Industry–academia collaboration in software engineering,” *IEEE Software*, vol. 35, no. 5, pp. 120–124, sep 2018.

[85] L. Mathiassen, “Collaborative practice research,” *Information Technology & People*, vol. 15, no. 4, pp. 321–345, dec 2002.

[86] A. Sandberg, L. Pareto, and T. Arts, “Agile collaborative research: Action principles for industry-academia collaboration,” *IEEE Software*, vol. 28, no. 4, pp. 74–83, jul 2011.

[87] *SER & IP 2018 : 2018 ACM/IEEE 5th International Workshop on Software Engineering Research and Industrial Practice : proceedings : 29 May 2018, Gothenburg, Sweden*. New York, New York Los Alamitos, California: The Association for Computing Machinery IEEE Computer Society, Conference Publishing Services, 2018.

[88] M. R. Karim *et al.*, “Applying data analytics towards optimized issue management: An industrial case study,” in *2016 IEEE/ACM 4th International Workshop on Conducting Empirical Studies in Industry (CESI)*, 2016, pp. 7–13.

[89] C. Wohlin, “Empirical software engineering research with industry: Top 10 challenges,” in *2013 1st International Workshop on Conducting Empirical Studies in Industry (CESI)*. IEEE, may 2013.

[90] A. B. Sandberg and I. Crnkovic, “Meeting industry-academia research collaboration challenges with agile methodologies,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2017, pp. 73–82.

[91] P. Runeson, “It takes two to tango – an experience report on industry – academia collaboration,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, apr 2012.

[92] C. Wohlin *et al.*, “The success factors powering industry-academia collaboration,” *IEEE Software*, vol. 29, no. 2, pp. 67–73, mar 2012.

[93] S. Palkar, “Industry-academia collaboration, expectations, and experiences,” *ACM Inroads*, vol. 4, no. 4, pp. 56–58, dec 2013.

[94] R. Bergmann, J. Ludbrook, and W. P. J. M. Spooren, “Different outcomes of the wilcoxon—mann—whitney test from different statistics packages,” *The American Statistician*, vol. 54, no. 1, pp. 72–77, feb 2000.

[95] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[96] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[97] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[98] pytorch. pytorch/vision: Datasets, transforms and models specific to computer vision. [Online]. Available: https://github.com/pytorch/vision

[99] NVIDIA. Cuda compatibility :: Gpu deployment and management documentation. [Online]. Available: https://docs.nvidia.com/deploy/cuda-compatibility/index.html#binary-compatibility

[100] P. Xu, S. Shi, and X. Chu, “Performance evaluation of deep learning tools in docker containers,” in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, aug 2017.

[101] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[102] Torch Contributors. Reproducibility - pytorch 1.6.0 documentation. [Online]. Available: https://pytorch.org/docs/1.6.0/notes/randomness.html

[103] ——. Reproducibility - pytorch master documentation. [Online]. Available: https://pytorch.org/docs/1.1.0/notes/randomness.html

[104] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems 30*, I. Guyon *et al.*, Eds. Curran Associates, Inc., 2017, pp. 1742–1752. [Online]. Available: http://papers.nips.cc/paper/6771-flexpoint-an-adaptive-numerical-format-for-efficient-training-of-deep-neural-networks.pdf

[105] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[106] BSC. Extrae | bsc-tools. [Online]. Available: https://tools.bsc.es/extrae

[107] Red Hat blog. (2018) Bringing ai and machine learning data science into operation. [Online]. Available: https://www.redhat.com/en/blog/bringing-ai-and-machine-learning-data-science-operation/

[108] S. Schelter *et al.*, "On challenges in machine learning model management," 2018.

[109] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 291–300.

[110] "Precision and recall," in *Wikipedia*, Oct. 9 2020. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall

[111] "Jaccard index," in *Wikipedia*, Sep. 6 2020. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[112] K. Schwaber and M. Beedle, *Agile software development with Scrum.* Prentice Hall Upper Saddle River, 2002, vol. 1.

[113] D. Sculley *et al.*, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[114] C. Wohlin *et al.*, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[115] R. K. Yin, *Case study research and applications: Design and methods.* Sage publications, 2017.

[116] J. Jordan. (2018) Organizing machine learning projects: project management guidelines. [Online]. Available: https://www.jeremyjordan.me/ml-projects-guide/

[117] D. Storcheus, A. Rostamizadeh, and S. Kumar, “A survey of modern questions and challenges in feature extraction,” in *Feature Extraction: Modern Questions and Challenges*, 2015, pp. 1–18.

[118] M. Zinkevich, “Rules of machine learning: Best practices for ml engineering,” *URL: https://developers. google. com/machine-learning/guides/rules-of-ml*, 2017.

[119] Google AI. (2020) Responsible ai practices - google ai. [Online]. Available: https://ai.google/responsibilities/responsible-ai-practices/

[120] D. Firesmith, “Common requirements problems, their negative consequences, and the industry best practices to help solve them.” *The Journal of Object Technology*, vol. 6, no. 1, pp. 17–33, 2007.

[121] M. Attarha and N. Modiri, “Focusing on the importance and the role of requirement engineering,” in *The 4th International Conference on Interaction Sciences*, Aug 2011, pp. 181–184.

[122] C. E. Sapp, “Preparing and architecting for machine learning,” *Gartner Technical Professional Advice*, pp. 1–37, 2017. [Online]. Available: https://www.gartner.com/en/documents/3889770

[123] M. Honegger, “Shedding light on black box machine learning algorithms,” Master’s thesis, Department of Economics and Management, Karlsruhe Institute of Technology, Karlsruhe, Baden-Württemberg, Germany, 2018. [Online]. Available: https://arxiv.org/pdf/1808.05054.pdf

[124] A. Bibal and B. Frénay, “Interpretability of machine learning models and representations: an introduction.” in *ESANN*, 2016.

[125] R. B. Grosse and D. K. Duvenaud, “Testing mcmc code,” *arXiv preprint arXiv:1412.5218*, 2014.

[126] C. Murphy, G. E. Kaiser, and M. Arias, “An approach to software testing of machine learning applications,” 2007.

[127] K. Pei *et al.*, “DeepXplore,” in *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, oct 2017, pp. 1–18.

[128] Y. Roh, G. Heo, and S. E. Whang, “A survey on data collection for machine learning: A big data - AI integration perspective,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2019.

[129] P. Hebron, *Machine Learning for Designers.* O’Reilly Inc, 2016.

[130] S. Ramírez-Gallego *et al.*, “A survey on data preprocessing for data stream mining: Current status and future directions,” *Neurocomputing*, vol. 239, pp. 39 – 57, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217302631

[131] F. Herrera *et al.*, *Multiple Instance Learning.* Springer International Publishing, 2016.

[132] I. Guyon *et al.*, *Feature extraction: foundations and applications.* Springer, 2008, vol. 207.

[133] J. Dunn. (2016) Introducing fblearner flow: Facebook's ai backbone. [Online]. Available: https://code.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/

[134] Institute for Electrical and Electronics Engineers, Inc. and the Association for Computing Machinery, Inc. (1999) Code of ethics | ieee computer society. [Online]. Available: https://www.computer.org/web/education/code-of-ethics

[135] E. de Wit-de Vries *et al.*, "Knowledge transfer in university–industry research partnerships: a review," *The Journal of Technology Transfer*, vol. 44, no. 4, pp. 1236–1255, 2019.

[136] A. T. Alexander and S. J. Childe, "Innovation: a knowledge transfer perspective," *Production Planning & Control*, vol. 24, no. 2-3, pp. 208–225, 2013. [Online]. Available: https://doi.org/10.1080/09537287.2011.647875

[137] M. D. Santoro and P. E. Bierly, "Facilitators of knowledge transfer in university-industry collaborations: A knowledge-based perspective," *IEEE Transactions on Engineering Management*, vol. 53, no. 4, pp. 495–507, Nov 2006.

[138] J. Bruneel, P. D'Este, and A. Salter, "Investigating the factors that diminish the barriers to university–industry collaboration," *Research Policy*, vol. 39, no. 7, pp. 858–868, sep 2010.

[139] H. Jooybar *et al.*, "GPUDet," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 1–12, apr 2013.

APPENDIX A EXPERIMENT CONFIGURATION

Building These are the Docker images used to build the versions

```
emiliorivera/pytorch-lttng  py367-cu101 f56f9ff3befa
emiliorivera/pytorch-lttng  py379-cu101 25622dae1fa2
```

Evaluation These are the Docker images used to run the experiments:

```
emiliorivera/ml-frameworks-evaluation-server   latest       31a6603311ee
emiliorivera/ml-frameworks-evaluation-client   py367-cu101  7e51bb56f6c6
emiliorivera/ml-frameworks-evaluation-client   py379-cu101  65206552e0d5
```

VGG-X

Our version of VGG modified to run on 32x32 images has the following architecture:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           1,792
              ReLU-2           [-1, 64, 32, 32]               0
            Conv2d-3          [-1, 256, 32, 32]         147,712
              ReLU-4          [-1, 256, 32, 32]               0
         MaxPool2d-5          [-1, 256, 16, 16]               0
            Conv2d-6          [-1, 512, 16, 16]       1,180,160
              ReLU-7          [-1, 512, 16, 16]               0
 AdaptiveAvgPool2d-8            [-1, 512, 5, 5]               0
            Linear-9                 [-1, 1024]      13,108,224
             ReLU-10                 [-1, 1024]               0
          Dropout-11                 [-1, 1024]               0
           Linear-12                 [-1, 1024]       1,049,600
             ReLU-13                 [-1, 1024]               0
          Dropout-14                 [-1, 1024]               0
           Linear-15                   [-1, 10]          10,250
       LogSoftmax-16                   [-1, 10]               0
================================================================
Total params: 15,497,738
Trainable params: 15,497,738
Non-trainable params: 0
```

APPENDIX B RESULTS

The full standard deviation of our experiments are shown here:

			Standard deviation
bug_name	evaluation_type	metric	
study-pr31167	corrected	accuracy	0.349000
		precision	0.273460
		recall	0.360990
		f1	0.316318
	buggy	accuracy	0.347633
		precision	0.247173
		recall	0.354466
		f1	0.301058
study-pr31433	buggy	accuracy	0.401097
		precision	0.245187
		recall	0.400743
		f1	0.405464
	corrected	accuracy	0.384186
		precision	0.305089
		recall	0.384304
		f1	0.425116
study-pr31552	corrected	accuracy	0.274922
		precision	0.275623
		recall	0.272455
		f1	0.270807
	buggy	accuracy	0.317355
		precision	0.294632
		recall	0.314965
		f1	0.294290
study-pr31584	corrected	accuracy	0.520628
		precision	0.297059
		recall	0.523925
		f1	0.474808
	buggy	accuracy	0.477067
		precision	0.267144
		recall	0.483170

		f1	0.441195
study-pr32044	buggy	accuracy	0.310795
		precision	0.261660
		recall	0.311590
		f1	0.324546
	corrected	accuracy	0.275846
		precision	0.263358
		recall	0.271931
		f1	0.310229
study-pr32062	buggy	accuracy	0.317440
		precision	0.240009
		recall	0.315965
		f1	0.307069
	corrected	accuracy	0.327499
		precision	0.262634
		recall	0.323788
		f1	0.321101
study-pr32350	buggy	accuracy	0.311816
		precision	0.222810
		recall	0.311158
		f1	0.289974
	corrected	accuracy	0.363661
		precision	0.305160
		recall	0.363224
		f1	0.368758
study-pr32541	corrected	accuracy	0.373467
		precision	0.351968
		recall	0.374326
		f1	0.407222
	buggy	accuracy	0.365379
		precision	0.319897
		recall	0.366328
		f1	0.384371
study-pr32829	buggy	accuracy	0.338916
		precision	0.276703
		recall	0.345058
		f1	0.332161
	corrected	accuracy	0.304975

		precision	0.254600
		recall	0.313150
		f1	0.302676
study-pr32831	corrected	accuracy	0.220957
		precision	0.245364
		recall	0.225673
		f1	0.250097
	buggy	accuracy	0.315401
		precision	0.336513
		recall	0.321925
		f1	0.359283
study-pr32978	buggy	accuracy	0.376771
		precision	0.281298
		recall	0.379263
		f1	0.349762
	corrected	accuracy	0.340978
		precision	0.288742
		recall	0.344259
		f1	0.318935
study-pr33017	buggy	accuracy	0.290487
		precision	0.250763
		recall	0.289742
		f1	0.277894
	corrected	accuracy	0.262893
		precision	0.271642
		recall	0.259975
		f1	0.257889
study-pr35022	corrected	accuracy	0.274241
		precision	0.270380
		recall	0.273300
		f1	0.301711
	buggy	accuracy	0.271911
		precision	0.281770
		recall	0.273358
		f1	0.313745
study-pr36820	buggy	accuracy	0.259629
		precision	0.212889
		recall	0.264955

		f1	0.275142
	corrected	accuracy	0.316486
		precision	0.326837
		recall	0.322737
		f1	0.336985
study-pr36832	buggy	accuracy	0.349974
		precision	0.316834
		recall	0.346381
		f1	0.340095
	corrected	accuracy	0.304281
		precision	0.324330
		recall	0.301724
		f1	0.320067
study-pr37214	buggy	accuracy	0.328743
		precision	0.265165
		recall	0.331890
		f1	0.337470
	corrected	accuracy	0.321551
		precision	0.265107
		recall	0.322716
		f1	0.319260
study-pr38945	corrected	accuracy	0.290590
		precision	0.369977
		recall	0.286116
		f1	0.332605
	buggy	accuracy	0.283980
		precision	0.296394
		recall	0.285882
		f1	0.301431
study-pr39903	buggy	accuracy	0.380929
		precision	0.289519
		recall	0.388137
		f1	0.383754
	corrected	accuracy	0.315839
		precision	0.243343
		recall	0.318049
		f1	0.313139

APPENDIX C EXCLUDED RESULTS FROM STATISTICAL ANALYSIS

Here we present the 3 bugs that were removed from analysis because of their different seed values between a buggy and corrected version. There are 3 bugs excluded: `35253`, `32763` and `33050`

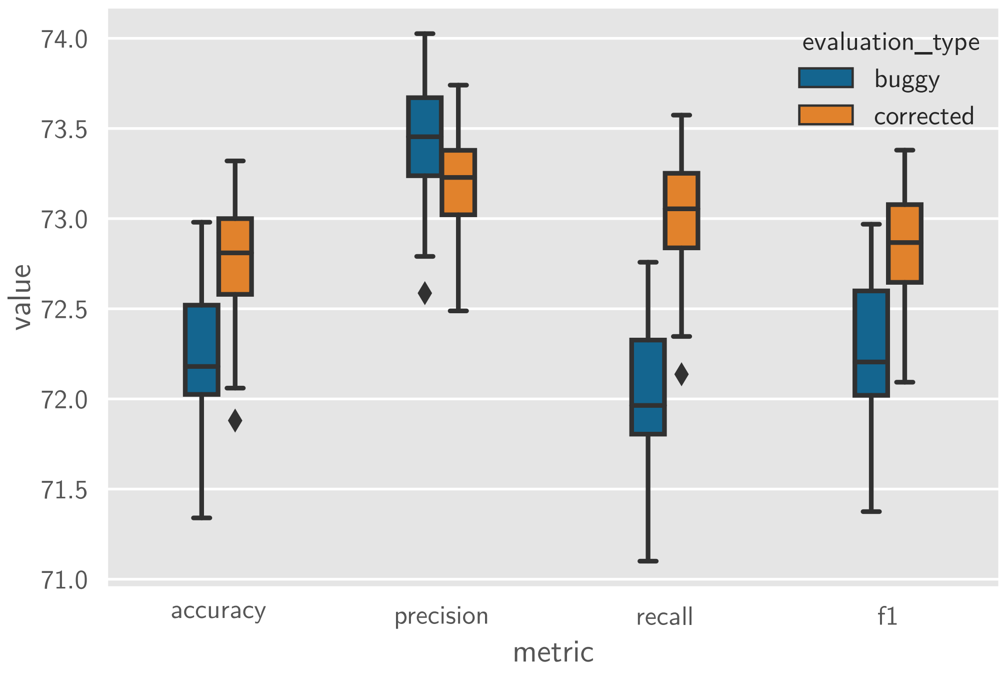


Figure C.1 Distribution of performance metrics for excluded result `study-pr35253` for using different seeds

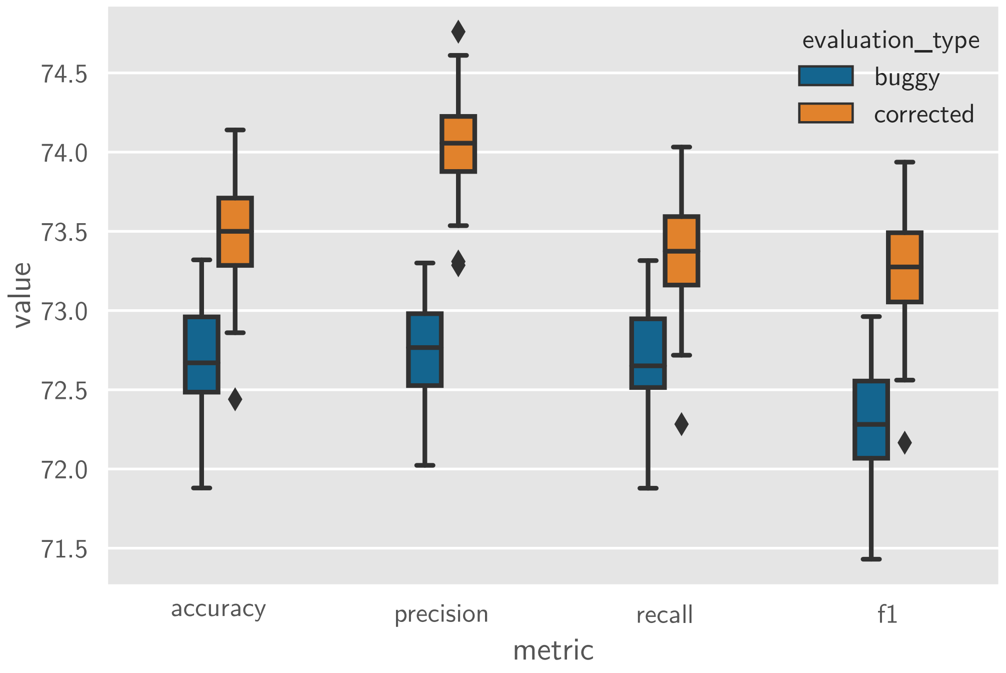


Figure C.2 Distribution of performance metrics for excluded result `study-pr32763` for using different seeds

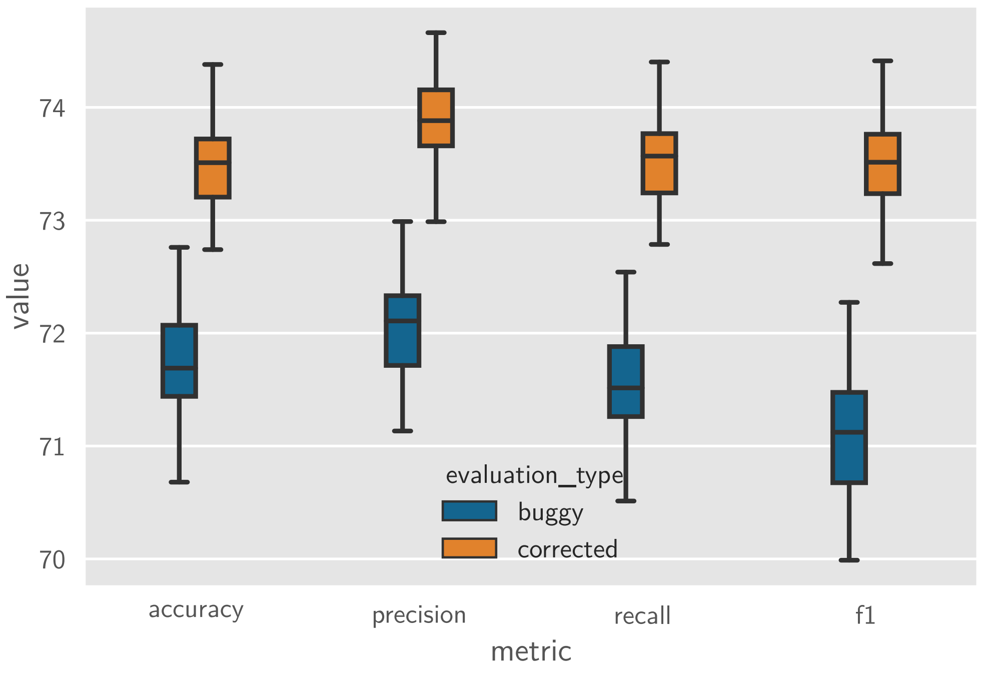


Figure C.3 Distribution of performance metrics for excluded result `study-pr33050` for using different seeds