Investigating the Quality of Bindings for Machine Learning Libraries in Software Package Ecosystems

by

Hao Li

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical And Computer Engineering
University of Alberta

Abstract

Machine learning (ML) has revolutionized many domains, with developers often relying on open source ML libraries to integrate ML capabilities into their projects. However, these libraries primarily support a single programming language, limiting their availability for projects in other languages. Bindings serve as bridges between programming languages by providing interfaces to ML libraries. This thesis investigates the quality of bindings for ML libraries in software package ecosystems, focusing on their maintenance and software quality.

The first study presented in this thesis introduces `BindFind`, an automated approach to identify bindings and link them with their corresponding host libraries across various software package ecosystems. By analyzing 2,436 bindings for 546 ML libraries, we find that most bindings are community-maintained, with npm being the most popular choice for publishing these bindings. The analysis reveals that these bindings usually cover a limited range of releases from their host library and experience significant delays in supporting new releases.

In the second study, we investigate the usage and rationale behind release-level deprecation in bindings for ML libraries within the Cargo and npm ecosystems. We discover that bindings in Cargo have a higher percentage of deprecated releases compared to general packages, while the percentages of deprecated releases and general packages are similar in npm. The primary reasons for deprecation are package removal or replacement and defects in both ecosystems. We also identify the issue of implicitly deprecated releases in Cargo due to deprecation propagation through the dependency network.

The third study evaluates the impact of using different bindings on the software quality of ML systems through experiments on model training and inference using TensorFlow and PyTorch across four programming languages. The results show that models trained with one binding perform consistently in inference tasks when utilized with another binding. Furthermore, non-default bindings can outperform the default Python bindings in specific tasks without sacrificing accuracy. We also find significant differences in inference times across bindings, highlighting the benefits of choosing appropriate bindings based on specific performance requirements to maximize efficiency in ML projects.

The work presented in this thesis provides deep insights, actionable recommendations, and effective and thoroughly evaluated approaches for assessing and improving the quality of bindings for ML libraries in software package ecosystems.

Preface

The research work presented in this thesis has been conducted in the Analytics of Software, GAmes, And Repository Data (ASGAARD) lab led by Dr. Cor-Paul Bezemer. This thesis is an original work by Hao Li.

Chapter 2 has been submitted for review as: Hao Li and Cor-Paul Bezemer, "Bridging the Language Gap: An Empirical Study of Bindings for Open Source Machine Learning Libraries Across Software Package Ecosystems," to the *Empirical Software Engineering* journal. I was responsible for developing the research ideas, processing and analyzing the data, creating the algorithms, and writing the manuscript. Dr. Bezemer, as the supervisory author, was involved in concept formation and manuscript composition.

The majority of Chapter 3 has been published as: Hao Li, Filipe R. Cogo and Cor-Paul Bezemer, "An Empirical Study of Yanked Releases in the Rust Package Registry," in *IEEE Transactions on Software Engineering* journal, vol. 49, no. 1, Jan. 2023, pp. 437–449, doi: 10.1109/TSE.2022.3152148. I was responsible for developing the research ideas, collecting and processing the data of the Rust package registry, analyzing the data, and writing the manuscript. Dr. Cogo provided access to the collected data of packages in npm and contributed to manuscript edits. Dr. Bezemer, as the supervisory author, was involved in concept formation and manuscript composition.

Lastly, Chapter 4 has been accepted for publication as: Hao Li, Gopi Krishnan Rajbahadur and Cor-Paul Bezemer, "Studying the Impact of TensorFlow and PyTorch Bindings on Machine Learning Software Quality," to appear in the *ACM Trans-*

actions on Software Engineering and Methodology journal, doi: 10.1145/3678168. I was responsible for developing the research ideas, designing and conducting the experiments, analyzing the results, and writing the manuscript. Dr. Rajbahadur was involved in concept formation and contributed to manuscript edits. Dr. Bezemer, as the supervisory author, was involved in concept formation and manuscript composition.

Acknowledgements

I would like to express my sincere gratitude to Prof. Cor-Paul Bezemer for all the support, guidance, and encouragement throughout my PhD journey. I could not have embarked on this journey or completed this thesis without his mentorship. I deeply appreciate the freedom and opportunities Cor-Paul has provided.

I am also very thankful for the opportunity to collaborate with Dr. Filipe Roseiro Cogo and Dr. Gopi Krishnan Rajbahadur. I have learned a lot from both of them.

I would like to thank the members of my examination committee, Prof. Lei Ma, Prof. An Ran Chen, Prof. Sarah Nadi, Prof. Abram Hindle, Prof. Marek Reformat, and Prof. Benoit Baudry, for their insightful feedback and valuable advice.

I am grateful to the ECE-Huawei Research Initiative (HERI) at the University of Alberta for supporting the research presented in this thesis.

I would also like to thank my friends in the Analytics of Software, GAmes And Repository Data (ASGAARD) lab. The discussions and shared moments of laughter have been a supportive and enjoyable part of my research journey.

Last but not least, my deepest thanks go to my family for their unwavering love and support. A special thanks to my wife, Suzhen Zhong, for being my cheerleader and pillar of strength. I would not be the person I am today without your love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning (ML) has transformed many industries, enabling advancements in fields such as autonomous vehicles [62, 120], robotic surgery [4, 45, 148], and recommendation systems [84, 87]. Developers often rely on open source ML libraries such as TensorFlow [1] and PyTorch [124] to integrate ML capabilities into their projects. These libraries provide high-level interfaces and offer several benefits, including readily usable state-of-the-art algorithms, accelerated computing, and interactive visualization tools for data [119]. However, a major limitation is that these libraries are typically designed to primarily support a single programming language [57, 119, 133], which can be a barrier for developers working with other programming languages.

1.1 Bindings and software package ecosystems

Bindings [113, 158] serve as the bridge between a *host library* and a target programming language, thus allowing developers to utilize the functionality of the library within their preferred development environment. This concept of bridging language barriers is exemplified by bindings such as tfjs [152], which provides a JavaScript interface to the well-known TensorFlow library.

In software development, software package ecosystems play a vital role in simplifying the management and integration of open source libraries. Instead of manually downloading and building libraries from source code repositories like Git [16], de-

velopers can use package managers and registries to easily resolve dependencies and install libraries. Most modern programming languages have their own official package manager and registry, which form the core components of a software package ecosystem. Examples include Maven for Java, PyPI for Python, and npm for JavaScript. These ecosystems host both ML libraries and their bindings, facilitating their use across different programming languages.

1.2 Motivation

Although ML libraries primarily target Python due to its widespread adoption in the ML community [7, 57, 133], this focus limits the availability for projects developed in other languages. Through bindings, developers can leverage these state-of-the-art ML techniques across programming languages. However, the characteristics and challenges associated with bindings for ML libraries remain poorly understood. These include understanding how the bindings keep up with the rapid release cycles of ML libraries and how their releases are managed in software package ecosystems. Additionally, the quality of bindings for ML libraries is critical as it directly affects the quality of the software projects that depend on them.

1.3 Thesis objectives

This thesis investigates the quality of bindings for ML libraries in software package ecosystems, with a focus on two primary aspects: maintenance quality and software quality. The specific objectives are as follows:

- Objective 1: Examine the maintenance quality of bindings for ML libraries.
    a. Investigate the extent to which bindings keep pace with the releases of their host ML libraries.
    b. Investigate how bindings for ML libraries deprecate releases compared to general packages.

- Objective 2: Investigate the software quality of bindings for ML libraries.
    a. Assess the correctness of models trained or utilized by different bindings for the same ML library.
    b. Assess the time costs of training and utilizing models across different bindings for the same ML library.

To achieve these objectives, we conducted three research studies. The first study (targeting Objective 1.a) proposed an approach to identify bindings and collected data on bindings from 13 software package ecosystems to analyze their release coverage, delay, and technical lag. The second study (targeting Objective 1.b) explored the use and rationale behind release-level deprecation in bindings for ML libraries within the Cargo and npm ecosystems. The third study (targeting Objective 2.a and 2.b) performed experiments in model training and inference using bindings for TensorFlow and PyTorch across four programming languages, aiming to assess the impact of bindings on software quality in terms of correctness and time cost.

We summarize the motivation and findings of our research studies below:

Research Study 1: Studying the Development and Maintenance of Bindings for Machine Learning Libraries (Chapter 2)

Motivation: As ML libraries mature, developers using different programming languages may wish to leverage their capabilities. However, if the ML libraries do not natively support a particular language, developers cannot find the libraries in the corresponding software package ecosystem. Bindings operate as practical solutions to make state-of-the-art ML tools available across languages and ecosystems for software reuse. Nevertheless, the development and maintenance practices for library bindings can differ from their host libraries. If a binding only supports a small proportion of its host library's releases and experiences substantial delays in receiving updates, developers relying on the binding may be forced to use an outdated version for an extended period. This situation can be problematic when bugs are fixed or new

features are added in the host library but not propagated to the binding in a timely manner. Understanding the prevalence, distribution, and maintenance of bindings for ML libraries across ecosystems is crucial for developers when selecting a binding to use. On the other hand, identifying bindings for ML libraries and linking them with their host libraries is a complex task, as bindings can have different names from the host library and may not be maintained by the same organization. Therefore, an automated approach is necessary to enable the large-scale study of bindings for ML libraries in software package ecosystems.

Findings: Our proposed approach called `BindFind` can accurately identify bindings (with a 0.97 F1 score) and their host library names (with a 0.89 F1 score) in software package ecosystems. By using `BindFind`, we found that around 5% of ML libraries have bindings in software package ecosystems. We identified 2,436 bindings for 546 ML libraries across 13 software package ecosystems. Notably, 94% of these bindings are maintained by the community, indicating a vast reliance on non-official support. ML libraries with bindings typically span across more than two ecosystems, with npm and PyPI being the most common pair. One reason behind it could be that PyPI is the most popular ecosystem for ML libraries, while npm is the most popular for their bindings. However, we found that most bindings cover only a limited range of their host library's releases, often experiencing considerable delays in supporting new releases and widespread technical lag. The situation is generally worse for bindings that are not maintained by the official library organizations.

Research Study 2: Studying Release-Level Deprecation of Bindings for Machine Learning Libraries (Chapter 3)

Motivation: Some software package ecosystems (e.g., Cargo and npm) implement a release-level deprecation mechanism that allows library maintainers to deprecate published versions. This prevents others in the ecosystem from depending on deprecated releases. This mechanism is particularly relevant for bindings of ML libraries since rapid iteration and updates are common in ML library usage [40]. Deprecating

a buggy or outdated version can alert developers and discourage its use, potentially forcing them to migrate to newer versions or seek alternatives. However, the usage of release-level deprecation and its impact, especially in the context of ML library bindings, remains understudied. Cargo and npm have supported release-level deprecation since their inception in 2014 and 2010 respectively, providing ample usage data for analysis. Other ecosystems either lack this mechanism or introduced this mechanism recently, resulting in insufficient data. By investigating the usage of deprecation and the rationale behind it, we aim to understand the challenges faced by ML library binding maintainers in managing releases and the factors influencing their decision to deprecate certain versions.

Findings: Our analysis included a dataset of 48,823 packages from Cargo and was compared with data from a prior study on npm [138]. We found that bindings for ML libraries in Cargo have a higher percentage of deprecated releases compared to general packages. In contrast, the percentage of deprecated releases of bindings for ML libraries in npm is similar to that of general packages. We identified five patterns of deprecated releases and eleven rationales behind the deprecation. Package removal or replacement is the primary reason for deprecated releases in these bindings within Cargo and npm. Furthermore, we discovered that deprecated releases in Cargo can propagate through the dependency network, causing implicitly deprecated releases. This issue does not occur in npm since its deprecation mechanism is not forceful. Our analysis also shows that nearly half of all Cargo packages have adopted yanked releases, leading to unresolved dependencies in 1.4% of all packages. However, bindings for ML libraries contribute minimally to this issue. While the deprecation propagation of general packages in Cargo is usually caused by partially deprecated packages, the propagation among bindings for ML libraries is primarily caused by fully deprecated bindings, as their dependents cannot automatically migrate to any other versions. Our findings provide valuable insights into the usage of release-level deprecation in bindings for ML libraries within software package ecosystems.

Research Study 3: Studying the Correctness and Time Cost of Bindings for Machine Learning Libraries (Chapter 4)

Motivation: Bindings play a crucial role in integrating ML libraries into various programming environments, making it essential to assess their impact on the software quality of ML systems. Developers often choose bindings to utilize ML libraries in their preferred programming languages. However, this raises important questions about the correctness and time efficiency of models trained or utilized through these bindings. Our research is driven by the need to evaluate whether ML models maintain the same training accuracy when developed using different bindings and whether the accuracy of pre-trained models is preserved during inference across different bindings. These investigations are critical for developers to determine if bindings compromise the training and operational accuracy of their ML systems. Furthermore, given the resource-intensive nature of training and inference processes in ML libraries, it is crucial to consider the impact of bindings on time cost. Prior studies have highlighted variations in runtime efficiency and energy consumption across different programming languages, which could extend to the performance of bindings [118, 126, 128]. Therefore, we aim to quantify the time costs associated with training and performing inference using different bindings. These insights will help developers make informed decisions about the trade-offs between computational resources and performance when deploying ML models in production environments.

Findings: Our experiments with TensorFlow and PyTorch across bindings in C#, Rust, Python, and JavaScript revealed several key findings. We revealed that bindings can have different training accuracy curves under identical configurations. Remarkably, a model trained with one binding can be seamlessly used for inference with another binding for the same ML library without sacrificing correctness. Furthermore, non-default bindings can outperform the default Python bindings for certain tasks while maintaining the same correctness in both model training and inference processes. One important factor influencing the training cost of bindings for ML

libraries was the batch data loading time. In addition, significant differences were observed in inference times for the same pre-trained model across different bindings. In some instances, performing inference for the same model using a binding on CPU can be faster than using another binding for the same model on GPU. Variations in inference time among bindings were attributed to differences in both batch data loading and the speed of forward propagation. These results highlight the potential of non-default bindings to enhance ML software quality by reducing time costs while ensuring the preservation of model correctness.

Thesis Statement

Bindings for ML libraries enable developers to leverage state-of-the-art ML techniques across different programming languages. This thesis investigates the quality of bindings for ML in software package ecosystems, focusing on their maintenance quality and software quality, to provide insights for improving the development and utilization of these bindings.

1.4 Thesis outline

The remainder of this thesis is organized as follows: Chapter 2 presents a study on the development and maintenance of bindings for ML libraries. Chapter 3 investigates the release-level deprecation of bindings for ML libraries in Rust and npm ecosystems. Chapter 4 compares the correctness and time cost of bindings for ML libraries across different programming languages. Finally, Chapter 5 concludes the thesis by highlighting the findings and contributions of our research studies and discussing directions for future research.

Chapter 2

Studying the Development and Maintenance of Bindings for Machine Learning Libraries

2.1 Abstract

Open source machine learning (ML) libraries enable developers to integrate advanced ML functionality into their own applications. However, popular ML libraries, such as TensorFlow, are not available natively in all programming languages and software package ecosystems. Hence, developers who wish to use an ML library which is not available in their programming language or ecosystem of choice, may need to resort to using a so-called binding library (or *binding*). Bindings provide support across programming languages and package ecosystems for a host library. For example, the Keras .NET binding provides support for the Keras library in the NuGet (.NET) ecosystem even though the Keras library was written in Python. In this chapter, we collect 2,436 cross-ecosystem bindings for 546 ML libraries across 13 software package ecosystems by using an approach called `BindFind`, which can automatically identify bindings and link them to their host libraries. Furthermore, we conduct an in-depth study of 133 cross-ecosystem bindings and their development for 40 popular open source ML libraries. Our findings reveal that the majority of ML library bindings are maintained by the community, with npm being the most popular ecosystem for these

bindings. Our study also indicates that most bindings cover only a limited range of the host library's releases, often experience considerable delays in supporting new releases, and have widespread technical lag. Our findings highlight key factors to consider for developers integrating bindings for ML libraries and open avenues for researchers to further investigate bindings in software package ecosystems.

2.2 Introduction

Machine learning (ML) has become extremely popular in the last decade. Nowadays, there exist many ML applications in our daily lives, such as email spam filters, recommendation systems, and voice assistants. To provide ML features in an application, most developers rely on well-developed open source ML libraries, such as TensorFlow [1] or PyTorch [124]. These open source ML libraries provide easy-to-use interfaces for software developers to use ML techniques in their projects. However, these libraries often target only one programming language and publish to one software package ecosystem. For example, scikit-learn [125], a popular ML library which provides various ML algorithms, is written in Python and publishes to PyPI. Thus, Python developers can directly use the published scikit-learn package through PyPI but developers in other programming languages cannot use this library as easily.

There exist several workarounds that allow a developer to use a library that was not written in their preferred language. First, they could choose an alternative but similar library that is written in their preferred language. However, such a similar library may not exist, and even if it does, it may only provide a subset of the required functionality. Another workaround is to recode the library from scratch, but this approach is error-prone and requires a large amount of work. Finally, the developer can use a binding for the library in their preferred language, which would allow the developer to use the original library's functionality without recoding the library. However, there is not much known about this type of reuse.

Our research aims to fill this gap by conducting a large-scale investigation into

open-source ML libraries and their bindings across the ecosystems of 13 programming languages. Specifically, we seek to answer the following research questions (RQs):

RQ1. How effective is `BindFind` for identifying bindings, and what are the identified bindings for ML libraries?

We introduce an approach called `BindFind` for identifying bindings in software package ecosystems, which achieves a 0.970 F1 score in identifying bindings and a 0.889 F1 score in extracting the host library names. Using `BindFind`, we identified 2,436 bindings for 546 ML libraries across 13 software package ecosystems.

RQ2. How are ML libraries and their bindings distributed across ecosystems?

The most common combination of ecosystems that support an ML library is npm with PyPI. While PyPI is the most popular ecosystem for ML libraries, npm is the most popular ecosystem for the bindings of ML libraries.

RQ3. How are cross-ecosystem bindings for popular ML libraries maintained?

Cross-ecosystem bindings often offer low coverage of host library releases, and they suffer from high delays in supporting new releases, and considerable technical lag. The situation is worse for bindings that are not maintained by the official library organizations.

The main contributions of this chapter are as follows:

1. The first research to study bindings for ML libraries within software package ecosystems.

2. A replication package [98] containing our dataset of 250,668 bindings (together with their host library names) identified by `BindFind`. In addition, the replication

package includes details on 546 ML libraries and their 2,436 bindings, as well as the results of our analysis in which we matched 3,277 versions of 133 bindings for 40 popular ML libraries (including 3,785 tags).

3. A framework for understanding how well the bindings of ML libraries are maintained. Our findings offer a foundation for developers to make informed decisions when selecting bindings.

Chapter Organization. The rest of this chapter is organized as follows. Section 2.3 gives background information about our study. Section 2.4 discusses related work. Section 2.5 presents our methodology. Section 2.6 presents the findings of our three RQs. Section 2.7 discusses the implications of our findings. Section 2.8 outlines threats to the validity of our study. Section 2.9 concludes the chapter.

2.3 Background

In this section, we give background information about software package ecosystems and cross-ecosystem bindings for ML libraries.

2.3.1 Software Package Ecosystems

Traditionally, developers of open source libraries published their source code in a source code repository like Git [16]. Developers who wish to use those libraries could then download them directly from the source code repositories. However, developers had to resolve the library's dependencies and build the library manually. To help developers integrate a library more easily, the releases of software libraries can be published to a software package ecosystem. Open source libraries generally select a software ecosystem to publish their *main package*, making it the official distribution channel for releases. Alternatively, open source libraries might continue to release versions directly through their Git repositories.

Most modern programming languages come with an official package manager and a package registry. This package manager, the package registry and all the packages are the key components of a software package ecosystem. Usually, a package manager helps developers to manage the dependencies of their applications, for example, by downloading a specific version of a dependency when the application is installed. In addition, package managers help developers publish their applications to the software package ecosystem. Most software package ecosystems of programming languages will provide a website for developers to search and browse the information of stored packages. Some examples of software package ecosystems are Maven for Java, PyPI for Python, and npm for JavaScript.

2.3.2 Cross-Ecosystem Bindings for ML Libraries

In software development, bindings serve as bridges between different programming languages, enabling software written in one language to use functions and libraries developed in another [73]. This concept is especially useful in the context of ML, where libraries like TensorFlow and PyTorch are often tailored to specific languages and ecosystems [7]. Through bindings, developers can access these state-of-the-art ML tools across languages. This can be achieved through various mechanisms, including wrapper libraries that act as intermediaries, software development kits (SDKs) that bundle libraries with tools for development, or application programming interfaces (APIs) that facilitate interaction between different software components.

In our study, we use the term *host library* to refer to the original ML library, while *cross-ecosystem bindings* denote packages that allow the host library to be utilized in other software package ecosystems. For example, tensorflow in PyPI and tfjs in npm [152] are cross-ecosystem bindings for the same ML library (TensorFlow), even though they have different names and reside in different software package ecosystems. Moreover, cross-ecosystem bindings can be maintained by the community or the official organization of the host library. For instance, tensorflow in PyPI and tfjs in npm

are both maintained by the official organization of the TensorFlow host library (i.e., the owners of their source code repositories are the same), and we consider these two bindings as *officially-maintained bindings*. In contrast, TensorFlowSharp in NuGet, another binding for TensorFlow, is considered as a *community-maintained binding* since its owner is different from that of TensorFlow. Although the owner of TensorFlowSharp is an individual, the owner of a community-maintained binding can also be an organization that is not the same as the official organization of the host library.

Host libraries and their cross-ecosystem bindings do not necessarily follow the same release schedules and/or strategies. For instance, a cross-ecosystem binding may choose to only support a portion of the releases of its host library. Also, after the host library publishes a release, there may be a delay before a cross-ecosystem binding supports that release (if at all). In addition, cross-ecosystem bindings might lag in version compared to the latest release of the host library.

Cross-ecosystem bindings can be written in a different programming language (i.e., the primary supported language of the package ecosystem) than their host library. These bindings are often built on top of the host library and call the host library using a foreign function interface (FFI). For example, the tfjs-node binding in npm calls the TensorFlow C binary in the backend. Alternatively, some bindings choose to reimplement their host library in a different programming language. For instance, tfjs in npm is written in JavaScript and does not rely on the TensorFlow host library.

2.4 Related work

In this section, we discuss prior empirical studies of ML libraries, and related work on software ecosystems and foreign function interfaces.

2.4.1 Empirical Studies of ML Libraries

Many researchers have studied different concepts of ML libraries [40, 53, 58, 61, 63, 75, 95, 115, 167, 181]. However, this study is the first research to focus on bindings

for ML libraries.

The increasing popularity of ML has brought attention to the development and usage patterns of ML libraries. Dilhara et al. [40] conducted a large-scale empirical study, revealing a significant rise in ML library adoption and identifying common usage patterns and challenges. Furthermore, to support research in ML software reliability engineering, Morovati et al. [115] created a valuable benchmark of 100 reproducible ML bugs (originating from GitHub and Stack Overflow) in TensorFlow and Keras. Further analysis by Gao et al. [53] focused on the supply chain structure and evolution of the TensorFlow and PyTorch packages within PyPI, uncovering domain specialization, community clusters, and the reasons for packages leaving the supply chains.

Several studies focused on the problems that developers could face when using ML libraries. Islam et al. [75] mined Q&A of ten ML libraries on StackOverflow, and reported that three types of problems occurred frequently (i.e., type mismatch, data cleaning, and parameter selection). Zhang et al. [181] extracted bugs of TensorFlow applications from Q&A pages on StackOverflow and investigated the root causes of these problems. They provided strategies for developers to locate bugs and fix them. Wei et al. [167] introduced FIMAX to improve API recommendations for Python-based ML libraries based on extracted questions from Stack Overflow related to six popular ML libraries. Lei et al. [95] identified seven primary reasons why ML projects built on TensorFlow and PyTorch often encounter compatibility issues due to library version changes, causing code to fail in some projects even when using the same library API.

Several researchers have conducted comparison studies of multiple ML libraries. Grichi et al. [58] compared ten multi-language ML frameworks with ten multi-language traditional systems and reported that maintainers of these ML frameworks need more time to accept or reject a pull request than traditional systems. Guo et al. [61] compared the development and deployment processes of four ML libraries under the same

configuration for training of the same models. They found that using different ML libraries can lead to different levels of accuracy. Han et al. [63] collected projects that depend on PyTorch, TensorFlow, and Theano on GitHub, and observed four frequent applications (i.e., image and video, NLP, model theory, and acceleration). In addition, most projects depend on these three libraries directly instead of transitively.

2.4.2 Software Ecosystems

"Software ecosystems" are studied from several angles and even using different definitions [50, 109, 114]. The "software package ecosystems" term covers a subset of the software ecosystems term. In this chapter, we study ML libraries that can be found across multiple software package ecosystems. This study is the first to focus on cross-ecosystem bindings. Prior research has explored cross-ecosystem packages sharing the same source code repository [29, 81]. However, these studies have not directly addressed the concept of bindings. They might have unintentionally included some officially-maintained bindings within the same repository, but would have overlooked both community-maintained bindings and officially-maintained bindings hosted in different repositories. Our findings in Section 2.6.1 demonstrate that 94% of the ML libraries bindings are community-maintained, underscoring a key difference in our work.

Many studies have focused on software package ecosystems. In our prior work [99], we studied the release-level deprecation mechanism in Cargo (Rust) ecosystem and found that the deprecated releases propagate through the dependency network and lead to broken releases. German et al. [54] mined packages in CRAN and reported that most dependencies point to a core set of packages in the ecosystem. This phenomenon was also observed in another active software package ecosystem – npm [168]. Cogo et al. [27] studied dependency downgrades in npm and found three reasons behind the downgrades: defects, unexpected changes, and incompatibilities. Constantinou and Mens [30, 31] studied social aspects in ecosystems and found that the

developers are more likely to abandon an ecosystem if they do not participate in the community, and the Ruby ecosystem is being abandoned. Kula et al. [90] proposed a model for visualizing dependencies in ecosystems, and show that CRAN packages tend to use the latest releases, but Maven packages stay with the older versions. Decan and Mens [34] investigated package releases in three software package ecosystems and observed that most pre-releases do not become $\geq 1.0.0$ releases. Moreover, software package ecosystems have different practices, policies, and tools for handling breaking changes [10, 11].

In addition, researchers studied other types of software ecosystems. Blincoe et al. [8, 9] proposed a reference coupling method to identify software ecosystems in GitHub as well as the dependencies in the ecosystems. Osman and Baysal [123] identified the Bitcoin software ecosystem in GitHub and assessed it as a healthy ecosystem. Furthermore, many researchers studied the health of software ecosystems [77, 150]. Kula and Robles [89] investigated four abandoned software ecosystems and observed that all these ecosystems had a successor or their components were adopted by other systems. Bavota et al. [6] found that projects in the Apache ecosystem get updates when the dependencies published releases for breaking changes or bug fixes. de Souza et al. [153] studied social aspects in proprietary mobile software ecosystems and observed that most developers chose a specific ecosystem based on others' recommendation. Researchers also studied information security and business factors in mobile software ecosystems [155, 156].

2.4.3 Foreign Function Interfaces

Foreign function interfaces (FFI) bridge the gap between different programming languages and allow developers to reuse libraries written in other languages. To verify the correctness of existing bindings, Furr and Foster [52] presented a static checking system that analyzes both bindings and their host library. In addition, Lee et al. [94] built bug detection tools for the FFI in Jave and Python by performing dynamic

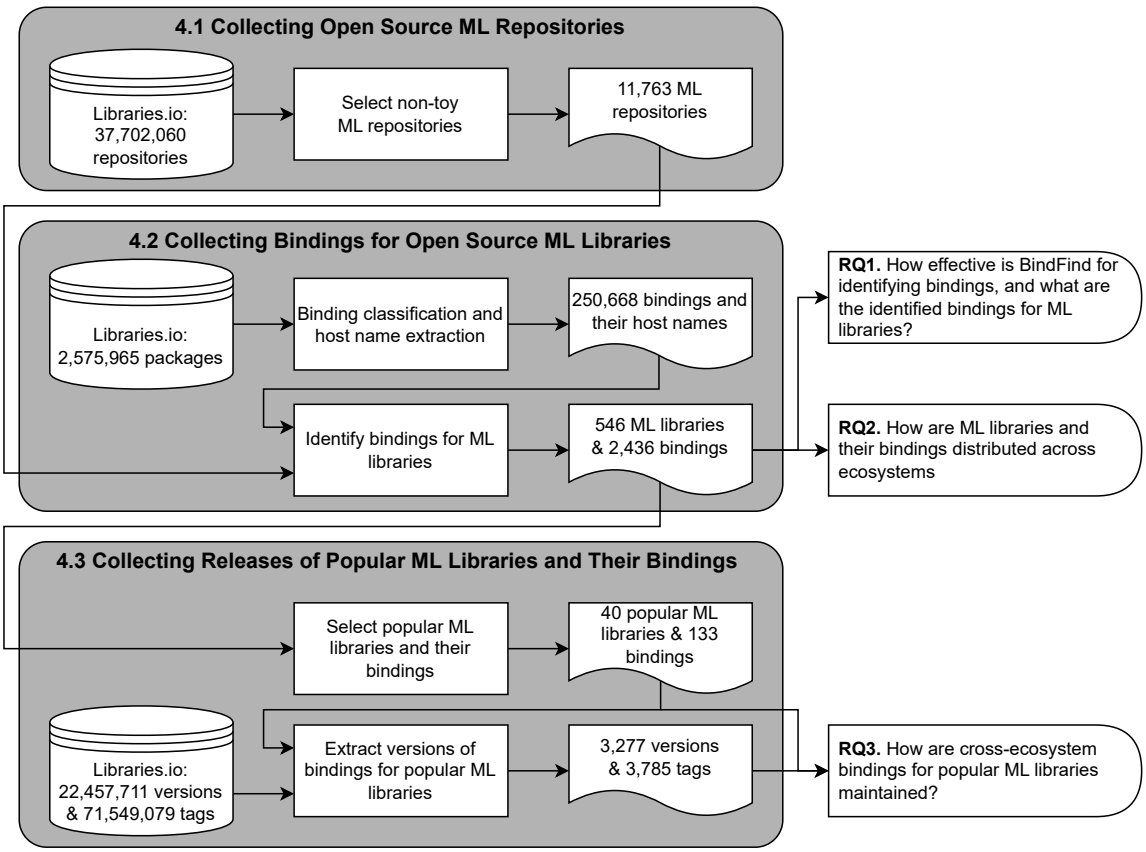


Figure 2.1: Overview of our methodology.

analysis. Nakata et al. [117] categorized link models and fault models of FFI and proposed a logging framework to track the information flow for each model.

Moreover, wrapping up a function to call a library from another programming language is not always applicable, Chiba [19] proposed a framework based on code migration to solve this problem. Besides writing the codes of FFI manually, Finne et al. [49] used an interface definition language to allow Haskell to communicate with both C and COM. In addition, Reppy and Song et al. [135] developed a tool to generate foreign interfaces for high-level languages to use the libraries written in C.

2.5 Methodology

In this section, we introduce the methodology of our study on popular ML libraries and their cross-ecosystem bindings. Figure 2.1 gives an overview of our methodology.

Table 2.1: Overview of the Libraries.io dataset

(a) Repositories and tags in Git hosts

Platform	**# Repos**	**# Tags**
GitHub	36,567,566	58,296,891
GitLab	864,563	12,248,518
Bitbucket	269,931	1,003,670
Total	37,702,060	71,549,079

(b) Packages and versions in software package ecosystems

Ecosystem	**Language**	**# Packs**	**# Versions**
npm	JavaScript	1,277,221	11,400,714
Packagist	PHP	313,575	1,766,576
PyPI	Python	232,050	1,752,770
NuGet	C#	199,671	2,445,003
Maven	Java	184,890	2,799,513
RubyGems	Ruby	161,650	1,055,874
CocoaPods	Objective-C	68,085	365,782
CPAN	Perl	37,496	290,847
Cargo	Rust	35,695	195,011
Clojars	Clojure	24,295	116,945
CRAN	R	16,710	94,716
Hackage	Haskell	14,484	98,572
Pub	Dart	10,143	75,388
Total		2,575,965	22,457,711

2.5.1 Collecting Open Source ML Repositories

We used the Libraries.io dataset [82] which was updated on January 12, 2020 as our primary data source. This dataset contains information (e.g., tags, owners, keywords) on 37,702,060 repositories from three prominent Git hosting services. These services host the actual code of open-source ML libraries and facilitate version control, developer collaboration, and other functions. Since these hosts are widely used, they may also contain personal projects, documentation, and experimental code. Table 2.1 outlines the distribution of repositories and tags among these Git hosts.

Following the keyword-matching approach proposed by Ben Braiek et al. [7] to extract ML projects from GitHub, we employed a similar approach to identify ML repositories. This approach involved scanning the "Description" and "Keywords" fields of repositories in the dataset for the following keywords: "machine learning", "deep learning", "statistical learning", "neural network", "supervised learning", "unsupervised learning", "reinforcement learning", and "artificial intelligence." We crafted regular expressions to accommodate variations in keyword formatting, including the presence of underscores, hyphens, and commas (e.g., "machine_learning"). To exclude toy repositories, we filtered out those with fewer than 5 stars, marked as inactive in the "Status" field, or indicated as forks. Ultimately, we extracted 11,763 ML repositories from the dataset.

2.5.2 Collecting Bindings for Open Source ML Libraries

The Libraries.io dataset includes information (e.g., released versions, creation dates, dependencies) on 4,612,919 packages from 38 software package ecosystems. As described in Section 2.3, ML libraries typically have their source code managed on Git hosts and publish packages to these ecosystems. Our analysis focused on 13 selected ecosystems, excluding ecosystems that: (1) focus on a specific domain, such as Sublime and WordPress, (2) those with a very small number of packages, such as Shards (33 packages) and PureScript (384 packages), and (3) those that do not store

information about releases, such as Go. Also, we excluded ecosystems that contain duplicated packages of other ecosystems, for example, most packages in Bower can be found in npm. Table 2.1 shows the supported programming language, the number of packages, and the number of releases in these 13 ecosystems.

Binding classification and host name extraction

To automatically identify bindings in package ecosystems and extract their corresponding host names, we propose `BindFind`. `BindFind` employs natural language processing (NLP) techniques, specifically leveraging BERT (Bidirectional Encoder Representations from Transformers) models [39]. As shown in Figure 2.2, we conceptualize the problem as an extractive question-answering (QA) task, akin to the methodology used in the Stanford Question Answering Dataset (SQuAD) v2.0 [131]. In this framework, the description of a package serves as the context for querying the model about the package's host library name, requiring the model to determine the precise locations (i.e., the start and end positions) of the host name within the context. Notably, all the inputs are tokenized before being fed into the model, and the start and end positions refer to the tokens instead of the original input. If the start and end positions point to the [CLS] token, or if the positions are invalid (e.g., the start position is after the end position), we conclude that the model did not identify the repository as a binding. This approach adeptly handles both scenarios where questions are answerable and unanswerable, reflecting real-world scenarios where some packages might not be bindings and thus not have a host name to extract.

Studied models. We selected several variations of BERT models, including the original BERT [39], DistilBERT [141], ALBERT [91], and RoBERTa [105]. These models were chosen for their proven efficacy in QA benchmarks [131, 132].

Data preparation. We manually labeled 2,546 packages to determine whether they are bindings and, if so, to identify their host names. The dataset was split into

training, validation, and test subsets. The training set included 2,346 samples, with 1,954 (83%) not being bindings and 392 (17%) identified as bindings (annotated with their host names). Both the validation and test sets were balanced, each containing 100 samples with an equal distribution of 50 (50%) bindings and 50 (50%) non-bindings. During the training process, we trained the studied models on the training set and used the validation set for hyperparameter tuning and model selection. After training, the performance of the models was evaluated on the test set.

Evaluation metrics. To assess the effectiveness of `BindFind`, we used the **F1 score**, **precision**, and **recall** [56] to evaluate the performance of classifying whether a package is a binding. For the extraction of host names within identified bindings, we applied **exact match** (EM) accuracy and (macro-average) **F1 score** specific to QA tasks [131]. EM is a strict metric where any deviation from the exact answer results in a score of 0 for that sample. The F1 score evaluates performance by considering predictions and ground truths as collections of tokens. The reported results are averaged over all of the samples.

$$EM = \frac{\text{Number of exact matches}}{\text{Number of examples that are bindings}} \tag{2.1}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{2.2}$$

where

$$Precision = \frac{\text{Number of accurately identified tokens}}{\text{Number of tokens in the prediction}} \tag{2.3}$$

$$Recall = \frac{\text{Number of accurately identified tokens}}{\text{Number of tokens in the ground truth}} \tag{2.4}$$

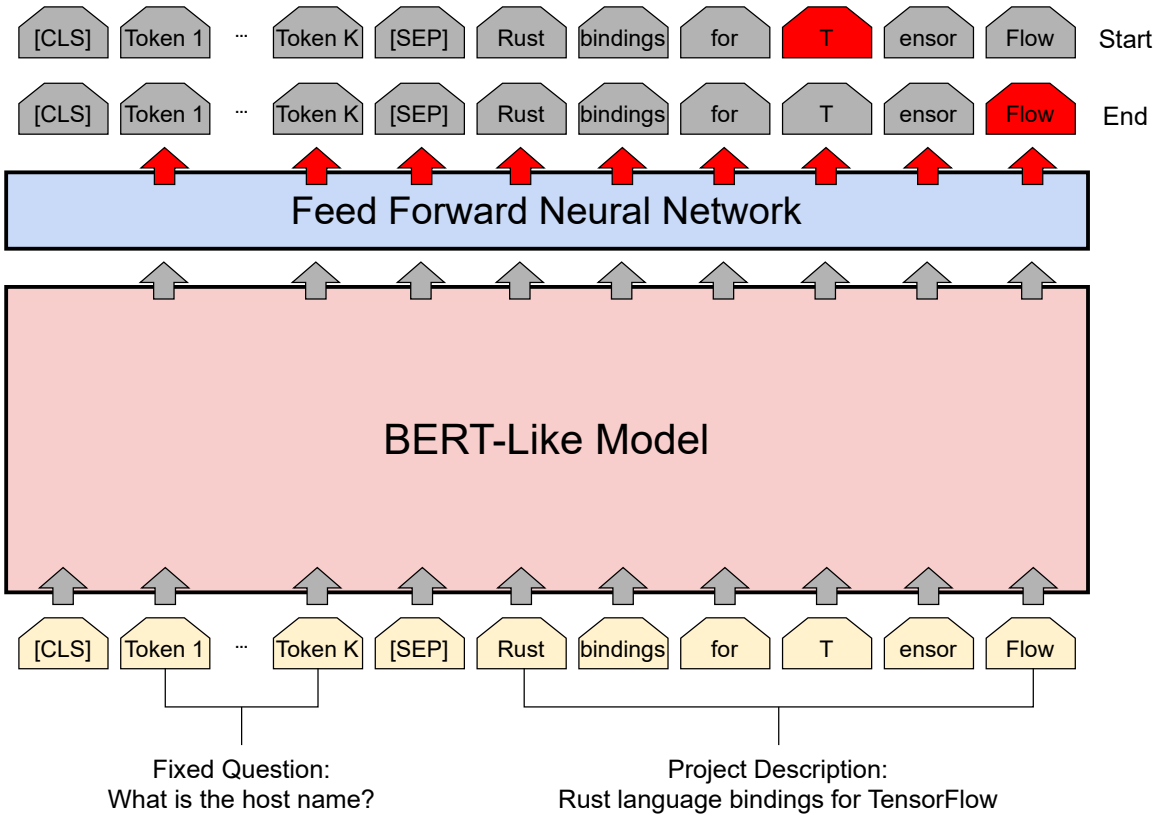


Figure 2.2: The model structure of `BindFind` for binding classification and host name extraction, illustrated using an example.

Identify bindings for ML libraries

We processed 2,575,965 packages from the 13 studied software package ecosystems and identified a total of 250,668 bindings with their corresponding host names. To identify bindings for the 11,763 ML repositories, we deployed a string-matching algorithm that compared the names of ML repositories with the extracted host names, resulting in 3,360 matches between bindings and 983 ML repositories. We manually reviewed these matches, removing repositories containing only tutorials or experimental code, and filtering out duplicate repositories of the same ML library. After refinement, we identified a total of 2,436 bindings for 546 ML libraries.

It is important to note that some of the 546 ML libraries might publish official packages into ecosystems. These published packages can be either normal packages or bindings (already identified by `BindFind`). For example, PyTorch publishes its

main package torch in PyPI,[1] which is not considered a binding as stated in their documentation [129]. To study the distribution of ML libraries and their bindings across ecosystems in RQ2, we included officially published packages alongside identified bindings. We used two criteria: (1) they specified the ML library's Git repository as their source code repository, and (2) they shared the same homepage URL as the ML library. Using these criteria, we identified 775 packages officially published packages by 202 out of the 546 ML libraries.

2.5.3 Collecting Releases of Popular ML Libraries and Their Bindings

To focus our analysis in RQ3 on widely adopted libraries, we filtered the 546 ML libraries with 2,436 bindings based on the number of stars and selected 127 ML libraries with more than 1,000 stars. Though we acknowledge that stars do not provide a complete picture of real-world usage, they are commonly seen as a proxy for the popularity of a project within the software engineering domain [12, 47, 64, 170, 172]. For instance, TensorFlow's binding tfjs has gained over 17,000 stars on GitHub,[2] suggesting significant attention from developers. For further analysis, we manually reviewed these libraries and their bindings to perform several refinements. We excluded supporting packages for the actual bindings within the same ecosystem, along with bindings that are either work-in-progress or have only placeholder/invalid releases, e.g., OpenCV's binding in Pub.[3] Following these steps, we obtained a final set of 40 popular ML libraries (as shown in Table 2.2) with 133 bindings.

[1] https://pypi.org/project/torch
[2] https://github.com/tensorflow/tfjs
[3] https://pub.dev/packages/flutter_opencv_plugin

Table 2.2: Basic information about the popular ML libraries that have cross-ecosystem bindings.

ML library	# Eco	# Stars	Description
Alluxio	2	4,449	Data orchestration for ML in the cloud
BerryNet	1	1,150	Deep learning gateway on Raspberry Pi
bert-as-service	2	6,379	Sentence vector mapping with BERT
BigDL	2	3,177	Distributed deep learning library for Apache Spark
Bullet	2	5,798	Physics simulation for RL
Caffe	3	29,655	Deep learning library
CatBoost	2	3,760	Gradient Boosting on Decision Trees
Deeplearning4j	2	11,328	Deep learning library for Java
DeepSpeech	4	12,710	A speech-to-text engine based on TensorFlow
dlib	5	8,412	A toolkit for real-world ML applications
DyNet	2	2,867	Dynamic Neural Network Toolkit
H2O	3	4,513	A platform for distributed ML
ImageAI	2	4,252	A library for deep learning and computer vision
Keras	3	45,995	A framework to provide human-friendly APIs based on TensorFlow
libpostal	6	2,171	An NLP library for address parsing and normalizing
LightGBM	3	10,242	A gradient boosting framework
MITIE	2	1,820	An NLP library for information extraction
MLflow	3	5,459	A ML lifecycle platform
mlpack	2	3,016	A library to provide ML algorithms
ncnn	1	7,930	Neural network inference
NLTK	3	8,498	An NLP library
NNPACK	1	1,085	Neural network acceleration

Continued on next page

Table 2.2 continued from previous page

ML library	# Eco	# Stars	Description
NNVM	1	1,586	Compiler for neural nets
ONNX Runtime	4	1,561	A runtime engine for ONNX models
OpenAI Gym	7	19,351	A toolkit for developing and comparing RL algorithms
OpenCV	10	41,126	A computer vision library
OpenFace	1	12,847	Face recognition with deep learning
OpenPose	1	15,532	Multi-person keypoint detection
Porcupine	3	1,853	A library for lightweight wake word detection
PredictionIO	8	12,226	A ML server for infrastructure management
PyTorch	5	35,004	A ML framework
Rasa	2	7,436	A ML framework for automating conversations based on text and voice
scikit-learn	3	38,756	A framework to provide ML algorithms
Seldon Core	2	1,296	An MLOps framework based on Kubernetes
spaCy	2	15,161	An NLP library
TensorFlow	11	139,939	A ML framework
Tesseract OCR	8	32,078	An OCR engine that uses deep learning
Vowpal Wabbit	8	6,767	Techniques to solve interactive ML problems
Weld	2	1,261	A library for data analy
XGBoost	6	17,996	A gradient boosting framework

2.6 Results

This section presents the results of our three RQs. For each RQ, we present the motivation, approach, and findings.

2.6.1 RQ1: How effective is `BindFind` for identifying bindings, and what are the identified bindings for ML libraries?

Motivation. This research question investigates the effectiveness of `BindFind` in classifying bindings and extracting host names within software package ecosystems. In addition, we aim to study the proportion of bindings for ML libraries and examine whether these bindings are maintained by the official organization or by the community. The findings from this research question will provide an overview of the prevalence of bindings for ML libraries in software package ecosystems.

Approach. We conducted a comparative analysis of various BERT models as described in Section 2.5.2. Subsequently, the most effective model was selected and utilized by `BindFind` to perform inference across all the packages in the 13 studied software package ecosystems. The distinction between officially-maintained and community-maintained bindings was established through an automatic examination of their association with the host ML libraries. Bindings sharing the same source code repository, organization name, or homepage URL as the host ML library were categorized as officially-maintained. Conversely, those lacking such affiliations were categorized as community-maintained.

Findings. `BindFind`**, powered by the RoBERTa base model, demonstrates exceptional performance in the tasks of binding classification and host name extraction, achieving F1 scores of 0.970 and 0.889 respectively.** As illustrated in Table 2.3, the RoBERTa base model exhibits superior performance in both the binding classification and host name extraction tasks. Notably, all evaluated models achieved an F1 score above 0.870 in binding classification, indicating a generally high level of accuracy across different architectures. Furthermore, the results demonstrate that larger models do not always guarantee better performance. For instance, the BERT base (cased) model outperforms its larger variant, and the RoBERTa base model surpasses the RoBERTa large model in both tasks.

Table 2.3: Performance comparison of `BindFind` with different BERT models in bindings classification and host name extraction on the test set. (U: Uncased; C: Cased; Prec: Precision; Rec: Recall)

Family	**Variant**	**Binding Classification**			**Host Name Extraction**	
		F1	**Prec**	**Rec**	**F1**	**Exact Match**
DistilBERT	Base (U)	0.903	0.977	0.840	0.806	0.780
	Base (C)	0.936	1.000	0.880	0.802	0.740
BERT	Base (U)	0.925	1.000	0.860	0.770	0.760
	Large (U)	0.925	1.000	0.860	0.799	0.760
	Base (C)	0.947	1.000	0.900	0.803	0.760
	Large (C)	0.936	1.000	0.880	0.767	0.720
ALBERT	Base	0.918	0.938	0.900	0.824	0.760
	Large	0.925	1.000	0.860	0.818	0.800
	XLarge	0.876	1.000	0.780	0.696	0.640
	XXLarge	0.959	0.979	0.940	0.887	0.840
RoBERTa	**Base**	**0.970**	**0.980**	**0.960**	**0.889**	**0.860**
	Large	0.969	1.000	0.940	0.869	0.840

5% of the ML repositories have bindings in software package ecosystems and a vast majority (94%) of them are community-maintained bindings. We found that 5% (546 out of 11,763) of the ML repositories are ML libraries with bindings in software package ecosystems. Among the 2,436 identified bindings for these ML libraries, a staggering 94% (2,292 out of 2,436) of the bindings are maintained by the community. Conversely, only 6% (144 out of 2,436) of the bindings are maintained by the official organization. Notably, 58% (84 out of 144) of these officially-maintained bindings share the same source repository as their host ML libraries, while 42% (60 out of 144) are hosted under the same organizational umbrella but in separate repositories. For instance, PyTorch has officially published a binding [130] in Cocoapods for iOS, and tfjs-node serves as an official binding for TensorFlow within the NPM ecosystem [152], with its source repository is maintained by the tensorflow organization on GitHub. Additionally, we observed the transition of some community-maintained bindings to official organizations, such as the Python binding for OpenCV [122].

Takeaway of RQ1

Bindings and their host library names can be very accurately identified in software package ecosystems using `BindFind`, utilizing the RoBERTa base model. The vast majority of bindings for ML libraries are maintained by the community.

2.6.2 RQ2: How are ML libraries and their bindings distributed across ecosystems?

Motivation. As ML libraries mature, developers across various ecosystems may wish to utilize them. To understand the distribution of ML libraries and their bindings across different software package ecosystems, we conducted a comprehensive analysis. This research question aims to unveil the availability of ML libraries to developers working in various programming languages and ecosystems, highlighting the interoperability and reach of these libraries. Also, we aim to explore whether certain (com-

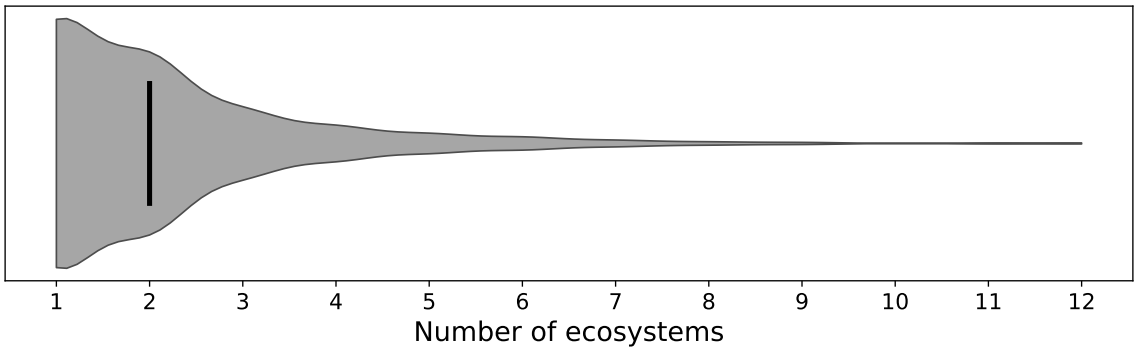


Figure 2.3: The distribution of the number of software package ecosystems supported by ML libraries with bindings.

binations of) ecosystems are favoured by the bindings for ML libraries.

Approach. We analyzed 2,436 bindings for 546 ML libraries, with a subset of 202 out of these 546 libraries having officially published packages (not bindings) across ecosystems (as detailed in Section 2.5.2). Our analysis focused on their distribution across 13 software package ecosystems. To better understand in which combinations of software package ecosystems these libraries reside, we counted the *ecosystem-pairs* for each library. An ecosystem-pair is counted for each pair of ecosystems supporting the same library. For example, if a library is supported in PyPI, npm, and NuGet, we count three ecosystem-pairs: PyPI-npm, PyPI-NuGet, and npm-NuGet. If an ecosystem-pair appears more frequently than others, it implies that those two ecosystems are more likely to be supported together by ML libraries.

Findings. **ML libraries with bindings typically span across a median of two software package ecosystems.** Our analysis revealed that ML libraries with bindings are typically supported across a median of two software package ecosystems. As illustrated in Figure 2.3, 55% (302 out of 546) of the studied ML libraries extend their reach by residing in multiple ecosystems. For example, the NLP library spaCy has bindings available in both PyPI and npm. Notably, the library with the broadest ecosystem presence is OpenCV which is available in 12 different ecosystems, followed by TensorFlow which is available in 11 ecosystems. Among the ML libraries with

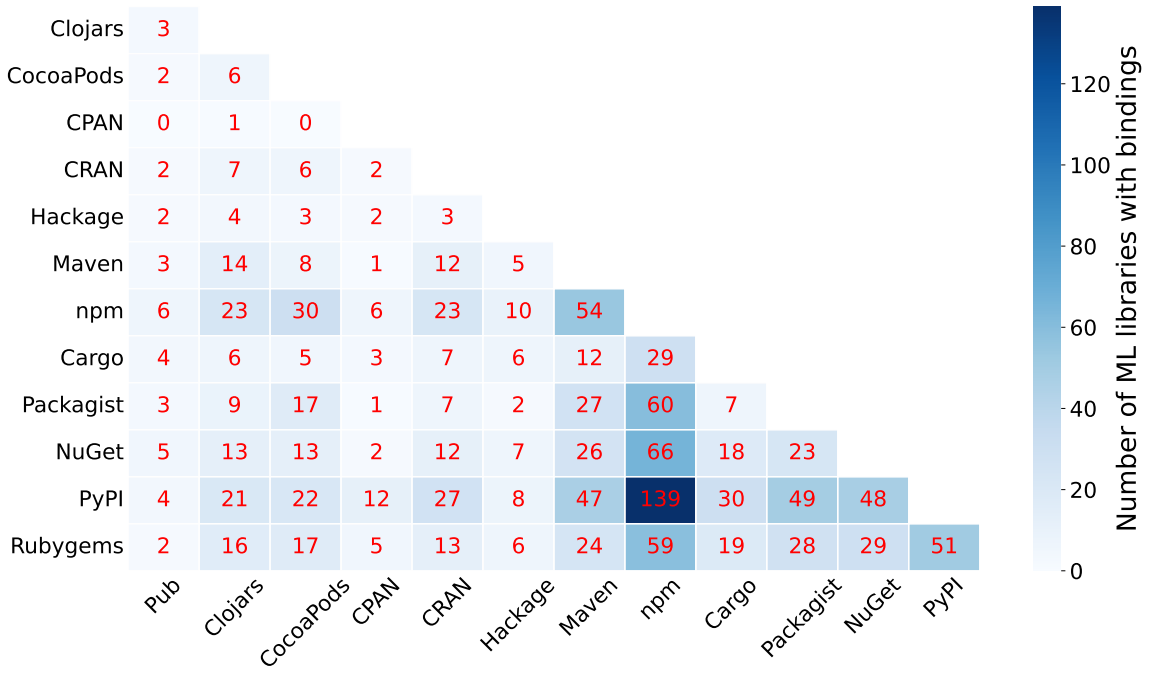


Figure 2.4: Combinations of software package ecosystems in which ML libraries with bindings are available. The elements represent the number of libraries that can be found in both ecosystems (i.e., ecosystems in the row and column).

bindings that are found in a single ecosystem (244 out of 546 libraries), 71% (174 out of 244) consists of libraries with only community-maintained bindings, lacking official packages or officially-maintained bindings. The remaining cases (70 out of 244) comprise libraries that have officially-maintained bindings.

Official organizations behind ML libraries with bindings prefer to focus on a single ecosystem, with PyPI being the most popular choice. Among ML libraries with bindings, 43% (236 out of 546) of them have official packages or officially-maintained bindings. When examining the publication behavior of official organizations behind these 236 ML libraries, we observed that 87% (205 out of 236) of the libraries prefer to focus on a single ecosystem, with PyPI being the predominant choice (69%). This preference aligns with prior research by Ben Braiek et al. [7], which reports Python's dominance in ML development.

npm is the leading ecosystem for hosting bindings for ML libraries and the most common combination of bindings is npm with PyPI. Regarding

bindings for ML libraries, we found that 53% (292 out of 546) of the libraries have bindings in npm. This is closely followed by PyPI in which bindings for 41% (225 out of 546) of the libraries are hosted. Figure 2.4 gives an overview of the ecosystem-pairs of ML libraries with bindings across multiple ecosystems (302 out of 546). The PyPI-npm pair is identified as the most prevalent combination, as it is supported by 139 out of 302 ML libraries with bindings. One reason could be that Python is the most popular language for ML development [7] and JavaScript has been the most commonly used programming language [154]. Hence, there could be a need for ML in npm, resulting in more support for such bindings. Other notable ecosystem pairs include npm-Packagist, Cargo-PyPI, and npm-Maven, reflecting a diverse landscape of ML library availability and collaboration.

Takeaway of RQ2

55% of ML libraries with bindings are distributed across at least two software package ecosystems, with npm being the most popular choice for publishing these bindings. Moreover, the most popular combination of ecosystems to support is PyPI and npm.

2.6.3 RQ3: How are cross-ecosystem bindings for popular ML libraries maintained?

Motivation. The popularity of certain ML libraries has led to increased development of bindings across software ecosystems. However, the development and maintenance practices for these bindings can be different from their host libraries. If a binding only supports a small proportion of the releases of its host library and has a high delay in getting an update, developers who rely on this binding might have to stick with a version for a long time. This situation could be problematic when a bug is fixed in the host library but not in the binding. As RQ1 showed, the vast majority of bindings for ML libraries are dependent on community support. In this research question, we investigate the releases of popular ML host libraries and their bindings to understand the development and maintenance of these bindings. The results can

be helpful for developers when they are choosing a binding.

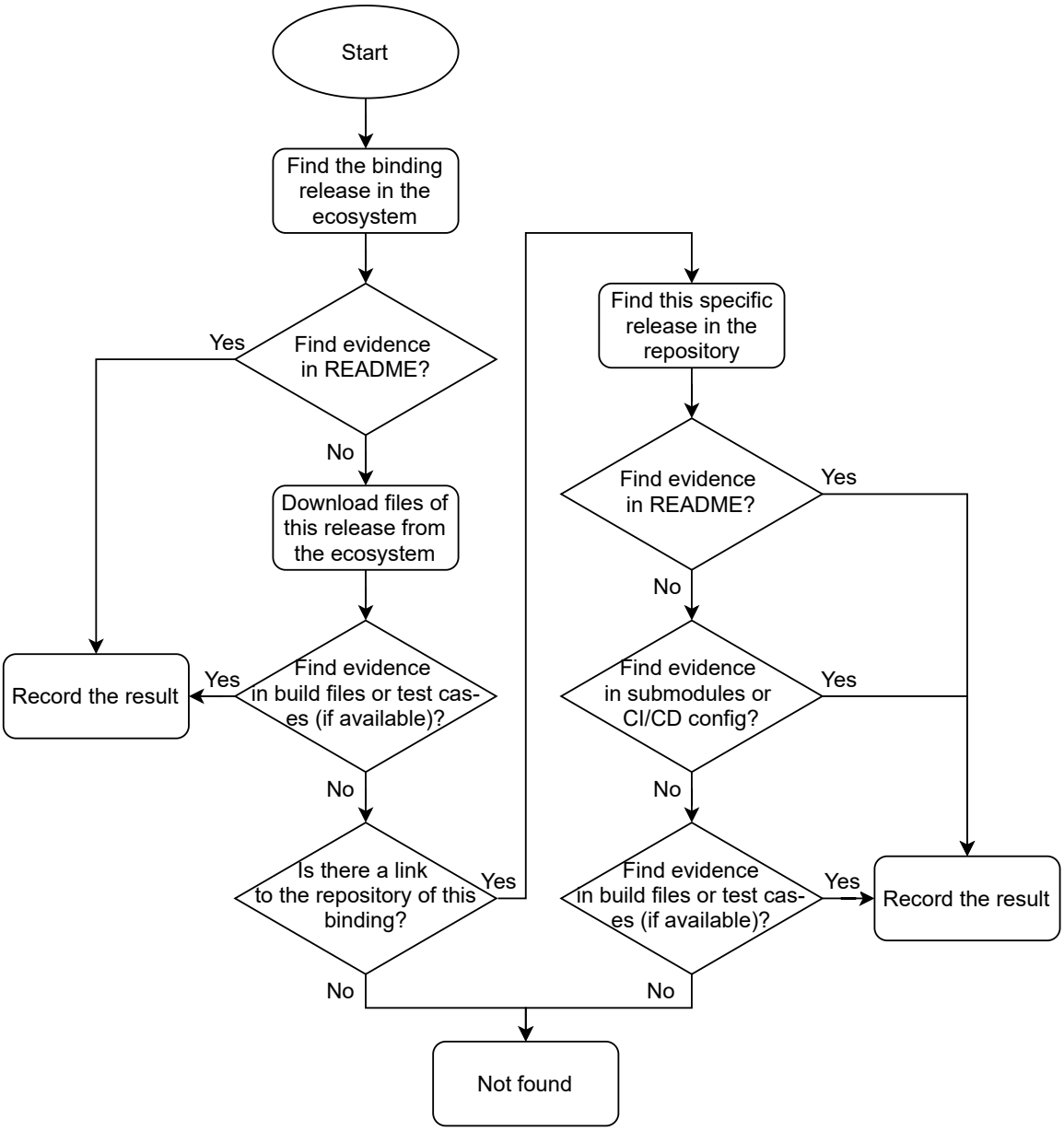


Figure 2.5: The process of identifying which version of the host library is supported by a specific binding version.

Approach. For libraries that primarily distribute their packages within a single software package ecosystem or designate a specific package as their main release channel, we considered the versions of these packages as the library's releases. For instance, PyTorch adopts a "Python First" approach [129], positioning its torch package in PyPI

not merely as a binding but as the main package. On the other hand, for libraries like OpenPose, which do not distribute packages through any specific package ecosystem,[4] we relied on source code repository tags as their releases. This method also applies to bindings that span multiple ecosystems, particularly when no main package is declared. In these cases, the source code repository tags often represent the most reliable source of library releases, such as the tags for XGBoost.[5] Subsequently, we matched all releases of each binding with the corresponding host library releases by searching for evidence of this matching in the following places of the binding releases:

1. **README:** When the supported version of the host library is mentioned explicitly.

2. **Git Submodules [16]:** When the source code of the supported version of the host library is included as a submodule.

3. **Build Files:** When the supported version of the host library that is going to be built for developers is mentioned explicitly, e.g., a binding might indicate the supported version in *CMakeLists.txt* or *Rakefile*.

4. **Test Cases:** When the supported version of the host library is verified explicitly by the tests.

5. **Configurations of continuous integration or delivery (CI/CD):** When the supported version of the host library is indicated explicitly in the configuration files, such as *.travis.yml*, to set up the CI/CD environment.

Figure 2.5 shows an overview of the identification process. For each release of a cross-ecosystem binding, we inspected the README and downloaded the files of this release from the software package ecosystem. If no evidence could be found, we checked out the source code repository of this binding. To locate the corresponding

[4]https://github.com/CMU-Perceptual-Computing-Lab/openpose/issues/1250
[5]https://github.com/dmlc/xgboost

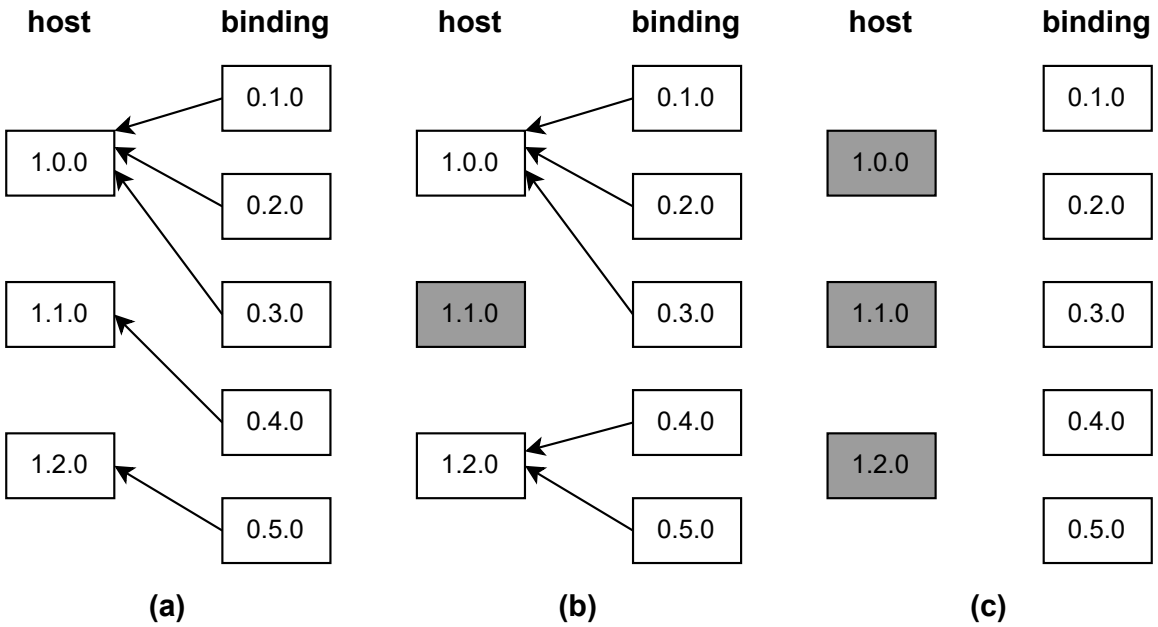


Figure 2.6: Three examples of matching binding releases and host releases: (a) all host releases are supported by the binding; (b) 2 out of 3 host releases are supported; (c) no host releases are supported.

checkpoint (i.e., a Git commit) of a specific release in the repository, we investigated all tags of the repository and extracted the one that had the same version as the release of the binding. However, some repositories do not have tags for published releases. In this case, we tracked the modification history of the meta-data file (which stores the version number) to locate the checkpoint. Some examples of meta-data files are *package.json* in npm, *setup.py* in PyPI, and *pom.xml* in Maven. After locating the checkpoint of the specific release, we inspected the files to find out which release of the host library is supported. We performed this process for all releases of cross-ecosystem bindings and matched them with the releases of the host library. The matching results can be found in our replication package [98].

To quantitatively assess the alignment between binding and host library releases, we extracted the **delay** between the matched releases and the host releases in days and calculated the **coverage** for each binding b in software package ecosystem e as follows:

$$coverage_{e,b} = \frac{\#matched\_host\_releases_{e,b}}{\#release_{host,b}} \tag{2.5}$$

where the numerator is the number of host releases that are supported by b and the denominator is the number of releases of the host library. We only consider the releases of the host library that were published since the binding started to provide support. The coverage of a binding b will be 100% if we can find a matched binding release for every host release. If we cannot find any matched release in b, the coverage will be 0%. The coverage metric captures the overall support that a binding offers for an ML library. Figure 2.6 presents three matching results, the denominators are 3 for all three examples and the numerators are 3, 2, and 0 respectively. Hence, the coverage values for these examples are 100%, 67%, and 0%. In addition, we investigated the **technical lag** [157, 179] of the latest release of each binding. Technical lag occurs when a binding does not support the most recent version of its host library. For example, a major version lag exists when the latest version of a binding supports version 1.1.0 of its host library, but the latest available version of the host library is 2.1.0. Similarly, minor and micro version lags exist when there are discrepancies in the minor or micro version numbers respectively.

Next, we compared our findings between officially-maintained bindings and community-maintained bindings by performing the Mann-Whitney U test [110] at a significance level of $\alpha = 0.05$ to determine whether the differences are significant. Also, we computed Cliff's delta d [106] effect size to quantify the difference. To explain the value of d, we use the thresholds which are provided by Romano et al. [137]:

$$\text{Effect size} = \begin{cases} negligible, & \text{if } |d| \leq 0.147 \\ small, & \text{if } 0.147 < |d| \leq 0.33 \\ medium, & \text{if } 0.33 < |d| \leq 0.474 \\ large, & \text{if } 0.474 < |d| \leq 1 \end{cases} \tag{2.6}$$

Findings. **Developers in PyPI are more likely to find a matched release of a**

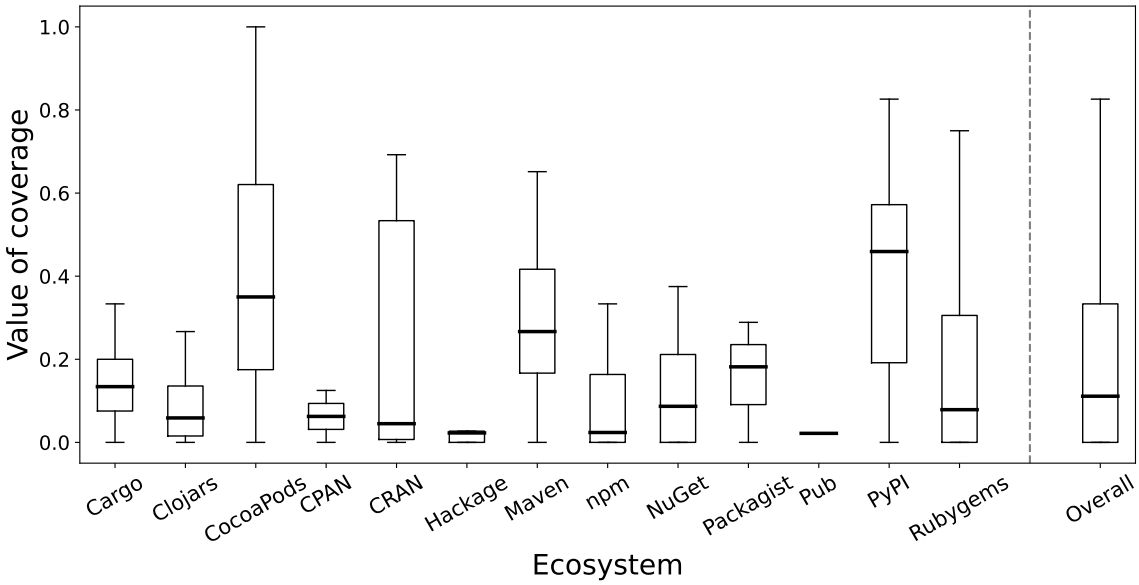


Figure 2.7: The distributions of the coverage of cross-ecosystem bindings for popular ML libraries across ecosystems.

cross-ecosystem binding for popular ML host libraries than developers in other software package ecosystems. Figure 2.7 shows that PyPI has the highest coverage among the 13 studied ecosystems, with a median value of 46%. In contrast, other ecosystems have relatively small coverage values. We observed that the main reason for the low coverage of these bindings is that they only provide support for a specific subset of the releases from their host libraries. For example, a binding of dlib in RubyGems has 13 releases but only supports 2 out of 50 versions of the host library. Specifically, releases 1.0.0 to 1.0.3 of this binding support version v18.13 of the host library, then it skipped nine versions (i.e., v18.14 to v19.3) of the host library and published 9 releases to support v19.4. This phenomenon can also be found in other bindings with low coverage. In addition, we noticed that the median coverage values of CRAN, Hackage, npm, and Pub are below 5%. This phenomenon may be attributed to a large portion of community-maintained bindings within these ecosystems (e.g., 84% in npm), where many bindings stick with a single host library version.

After a release of the ML library was published, their bindings in PyPI

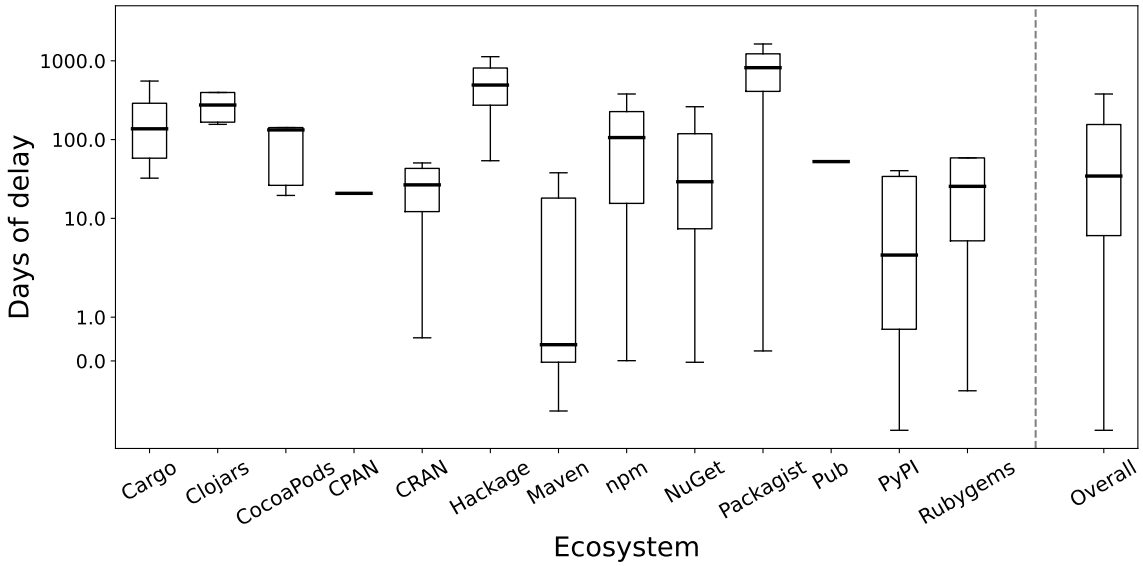


Figure 2.8: The distributions of the delay between releases of popular ML host libraries and their bindings.

and Maven received a corresponding update more quickly than the ones in other software ecosystems. Figure 2.8 shows that bindings within the PyPI and Maven ecosystems tend to be updated to match new releases of ML libraries more swiftly than those in other ecosystems, with median delays of less than 7 days. In contrast, bindings in ecosystems like Packagist and Hackage may experience median delays exceeding 365 days, indicating a slower pace of alignment with host library updates.

Compared to community-maintained bindings for popular ML libraries, the officially-maintained bindings have higher coverage, shorter delays, and smaller technical lags. Among the 133 bindings for the 40 popular ML libraries, 36 are officially-maintained bindings while 97 are community-maintained bindings (27% versus 73%). Figure 2.9 shows that the coverage of community-maintained bindings is mostly below 0.2 and the delays with the host libraries are mostly between 29 and 275 days. In contrast, the officially-maintained bindings have much more coverage (median of 11%) and less delay (median of 34 days). The Mann-Whitney U test demonstrates significant differences in both coverage and delay distributions

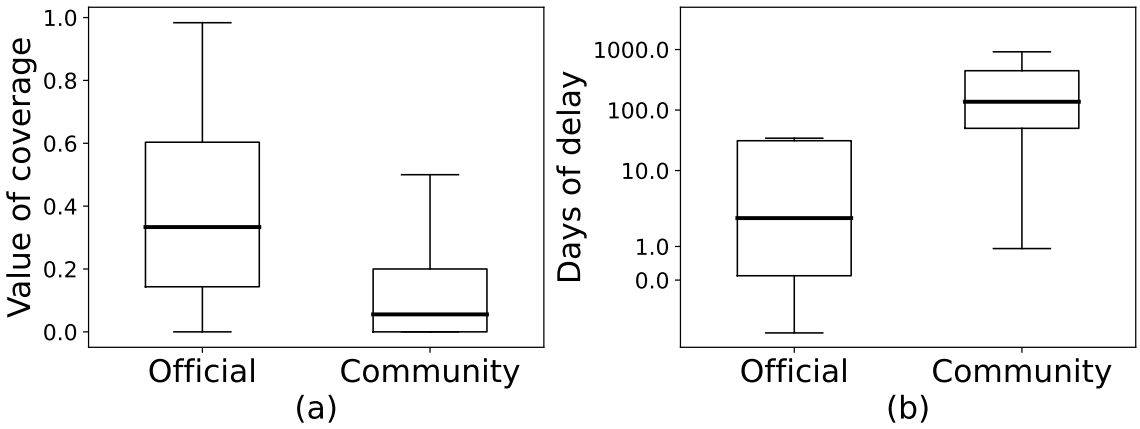


Figure 2.9: Comparisons of the cross-ecosystem bindings for ML libraries which are maintained by the official organization and the community: (a) the distributions of the coverage; (b) the distributions of the delay between a release of the host library and the corresponding release of the binding.

between officially-maintained bindings and community-maintained bindings. In addition, the values of Cliff's Delta d are 0.475 and -0.819 respectively, indicating large effect sizes. Furthermore, 45% of the officially-maintained bindings experience zero technical lag, compared to 24% of community-maintained bindings. Although officially-maintained bindings show lower proportions of major (21% vs. 32%) and minor (27% vs. 40%) technical lags compared to community-maintained bindings, they have a slightly higher proportion of micro lags (7% vs. 4%).

69% of the cross-ecosystem bindings do not follow the version numbers of their host library and 29% of cross-ecosystem bindings do not specify the matching information anywhere. During the process of matching binding releases with their host releases, we noticed that the majority (92 out of 133) of bindings do not reuse any version number from their host libraries. This misalignment in version numbers could lead to confusion for developers. Furthermore, our manual review process revealed that 29% (38 out of 133) of the bindings do not provide explicit matching information. Particularly, the absence of matching information in community-maintained bindings (89% of those are without matching information) creates an additional challenge, potentially forcing developers to resort to trial-and-

error methods to identify compatible versions.

Takeaway of RQ3

Generally, the release coverage of cross-ecosystem bindings for their host library is low and the delay is large. Officially-maintained bindings for popular ML libraries offer significantly better coverage, faster updates, and less technical lag than community-maintained alternatives.

2.7 Implications

In this section, we discuss the implications of our findings for developers, owners of ML host libraries and their cross-ecosystem bindings, and researchers.

2.7.1 Implications for Developers

Developers are not always limited to using the same source programming language as a popular ML library when they wish to use this library. PyPI is the most popular choice for ML libraries [7] and developers in PyPI are more likely to find an ML library for their own projects. However, developers might prefer to start a project in their most familiar ecosystem. Our findings show that 5% of the ML repositories are libraries that can be found in multiple ecosystems (Secion 2.6.1) and their bindings spread across different ecosystems, with npm being the most popular choice for bindings (Secion 2.6.2). Therefore, developers may find a binding of their desired ML library in the chosen software package ecosystem. We suggest that developers should put the choice of ecosystems into consideration before starting a project.

Developers should consider the number of supported releases and the delay of getting a corresponding update when choosing the binding for an ML library. Usually, cross-ecosystem bindings for popular ML libraries do not support all releases of their host library (Section 2.6.3). Our findings show that it is not sufficient to look at the number of binding releases only. Bindings with low coverage

could publish many releases but only support one or two versions of their host library. If developers are going to adopt such a binding, they should consider that it might not support a needed version in the future. For a binding with high coverage, developers should also consider how long it takes to update for a version of their host library and whether such delays are acceptable to them. By checking the maintenance history of the chosen binding, developers can have an expectation about the binding and consider whether they wish to adopt it or not.

2.7.2 Implications for ML Package Owners

Owners of cross-ecosystem bindings for popular ML libraries should explicitly indicate the matching between releases of the binding and releases of the host library. In Section 2.6.3, we found that some software package ecosystems have median coverage rates below 5% for bindings of popular ML host libraries. Such a low coverage indicates that either a small portion of the host releases are supported, or it is not possible to find out which versions are supported (even after our thorough investigation). In addition, we noticed that only 31% of cross-ecosystem bindings follow the version numbers of their host library. We recommend that owners of cross-ecosystem bindings use the same version number as their host libraries and indicate that in their README. For example, the binding of mlpack in CRAN mentions that "the version number of MLPACK is used as the version number of this package" [42]. Also, we recommend adding an extra number after the original version number, for example, changing the version number from "1.2.3" to "1.2.3.0". This way, the owner can increase the extra number when fixing bugs in the binding without causing confusion for developers.

Owners of popular ML host libraries should take notice of the communit y-maintained bindings for their libraries. 73% of the cross-ecosystem bindings for popular ML libraries are maintained by the community (Section 2.6.3). These community-maintained bindings help developers to use the functionalities from their

host library in an ecosystem which the official organization does not support. We recommend that official organizations keep an eye on the community-maintained bindings. For example, the official organization could inform popular community-maintained bindings about important updates, e.g., those that fix security vulnerabilities. We noticed that some libraries list the community-maintained bindings in their README or the official website. Furthermore, we observed that OpenCV even adopted a popular community-maintained binding [122].

2.7.3 Implications for Researchers

Researchers should expand research into bindings across software domains. The efficacy of `BindFind` in identifying and analyzing bindings (Section 2.6.1) underscores a rich avenue for further exploration. We suggest researchers apply `BindFind` in broader contexts, extending its use to examine areas such as web development frameworks. This extension could provide valuable insights into how bindings enhance software library interoperability across various domains. Researchers can reuse our replication package [98] which contains a dataset of 250,668 bindings and their host names identified by `BindFind`.

Researchers should further investigate the differences between officially-maintained bindings and community-maintained bindings. Our results show that community-maintained bindings and officially-maintained bindings have different coverages, delays, and technical lags (Section 2.6.3). Future studies should investigate what causes these differences. One factor could be simply financial incentives (e.g., because contributors to officially maintained bindings work for the company driving the binding), but there could also be socio-technical factors. For example, developers may be more motivated to contribute to officially-maintained bindings as such contributions are considered more valuable or prestigious. In addition, future studies should investigate how the communication between developers of community-maintained bindings and the host library can be improved.

Researchers should study automatic matching tools for releases of ecosystem bindings to match with releases of their host library. It is a complex and tedious task to identify which host library release is supported by an ecosystem binding. Automatic version matching tools for ecosystem bindings can help developers to find a suitable release without going through all the related files of a binding (like we did in Section 2.6.3) or trying the releases one by one in their project.

2.8 Threats to Validity

In this section, we discuss the threats to validity of our study about popular ML libraries and their cross-ecosystem bindings.

2.8.1 Internal Validity

We use release coverage as a metric to reason about the maintenance of a binding. A low coverage value does not always mean that a binding has a bad support. For example, a library can be forward-compatible, thereby making it easier to support newer versions without changing the binding. However, developers have to manually verify whether this binding will work for a specific version of the ML library that they are going to use, as it is not indicated anywhere.

We classified cross-ecosystem bindings of popular ML libraries into officially-ma intained bindings and community-maintained bindings. However, some community -maintained bindings could be adopted by the official organization and become an officially-maintained binding. In our study, we only consider the latest information and do not take into account the history of ownership. In addition, the owner of a community-maintained binding could be a member of the official organization of the host library. If the binding is not owned by the official organization and there is no evidence in its README, we consider such bindings as community-maintained bindings.

Some software package ecosystems like PyPI allow the owner of a package to delete

a published release. Libraries.io does not record the history of releases of a package, hence, we only consider the information of releases when the dataset was collected. In addition, some ecosystems also support the owner of a package to deprecate a release instead of deleting a release, e.g., npm and Cargo. In our study, we consider all releases which also include the deprecated ones.

2.8.2 External Validity

In our empirical study, we studied cross-ecosystem bindings for ML libraries. The results of our study might not apply directly to all cross-ecosystem bindings. Still, our methodology can be applied to analyze other cross-ecosystem bindings for libraries in other domains. In addition, we focused on open source ML libraries as many popular ML libraries are open source. Future studies should investigate if our findings hold for proprietary ML libraries.

2.9 Conclusion

In this study, we introduced `BindFind`, a novel approach for identifying bindings and extracting host library names within software package ecosystems. Applying `BindFind` to the libraries.io dataset allowed us to gather 2,436 bindings for 546 ML libraries across 13 software package ecosystems. We analyzed the population of bindings for ML libraries and the distribution of these libraries and their bindings across various ecosystems. In addition, we identified 40 popular ML libraries along with their 133 cross-ecosystem bindings, further examining their releases. Based on the data from these popular ML libraries, we conducted an in-depth analysis to investigate the development and maintenance of these cross-ecosystem bindings. We shared the collected data in our replication package [98]. The most important findings of our study are:

1. `BindFind` demonstrates high F1 scores in identifying bindings and extracting host

library names, indicating its robust capability to assist in the discovery of bindings within software package ecosystems.

2. npm is the most popular software package ecosystem for bindings of ML libraries, with npm and PyPI emerging as the predominant combination for publishing these bindings.

3. The study highlights significant challenges in the maintenance of cross-ecosystem bindings, including limited release coverage, delays in updating bindings to match new releases, and prevalent technical lag, especially among community-maintained bindings.

4. Most bindings do not follow their host library's version numbers, and many lack clear information on which host library versions they support.

Our findings show that developers who wish to use a popular ML library are not limited to using the programming language the library was written in, as there exist many cross-ecosystem bindings. However, they should carefully check the coverage, delay, and technical lag of these bindings before they commit to using one. In addition, we suggest that maintainers of cross-ecosystem bindings should follow the version number of their host library and add an extra number after it, to account for bug fixes, and make it easier for developers to identify which version of the host library is supported by the binding.

Chapter 3

Studying Release-Level Deprecation of Bindings for Machine Learning Libraries

3.1 Abstract

Release-level deprecation is a mechanism used in software package ecosystems to mark specific versions of packages as deprecated, thereby preventing other packages from depending on them. Most prior studies focused on code-level (i.e., deprecated APIs) and package-level deprecation (i.e., deprecated packages). However, few studies have focused on release-level deprecation, especially for bindings of machine learning (ML) libraries. In this study, we investigate release-level deprecation within the Cargo and npm ecosystems, focusing particularly on bindings for ML libraries. Our study shows that 12.6% of ML library bindings in Cargo have at least one deprecated release, compared to 9.6% for general packages. In contrast, npm has a similar percentage of deprecation between ML library bindings (3.6%) and general packages (3.7%). Furthermore, package owners deprecate releases for other reasons than withdrawing a defective release, such as indicating a package is removed or replaced. In Cargo, deprecated releases propagated through the dependency network due to the forceful deprecation mechanism, which leads to 1.4% of the releases in the ecosystem having unresolved dependencies. However, ML library bindings have a negligible impact

on the adoption of yanked releases and the propagation of unresolved dependencies compared to general packages.

3.2 Introduction

In a software ecosystem, deprecation can happen in APIs, releases, and packages. Usually, the owner of a deprecated API plans to remove this API in the future and attempts to warn the developers who are using these APIs. For example, the owner of a package plans to remove a `foo()` function in one year, and adds a warning message which will be printed when `foo()` is called, giving a developer time to deal with the prospective deprecation. Usually, the deprecation of releases and packages takes place unexpectedly. For example, when an owner of a package finds a critical bug in a release which was published a year ago, the owner can immediately deprecate this buggy release (without deleting the data) to make developers aware that the release is buggy, if the package manager supports release-level deprecation. This is particularly relevant in the context of ML libraries and their bindings since rapid iteration and updates are common in ML library usage [40]. However, release-level deprecation, particularly in the context of ML library bindings, remains understudied.

Prior studies have focused on the deprecation of APIs [136, 142, 162, 164, 171, 177] and packages [5, 15, 25, 26, 71, 85, 111]. Cogo et al. [138] studied the release-level deprecation mechanism in npm,[1] which has supported this mechanism since 2010.[2] Cargo[3] (for Rust) is another software packaging ecosystem that has supported release-level deprecation from the very beginning of its creation (since 2014)[4]. We also considered other five ecosystems that contain the largest number of packages,[5]

[1] https://www.npmjs.com/
[2] https://github.com/npm/cli/blob/v0.2.8/lib/deprecate.js
[3] https://doc.rust-lang.org/cargo
[4] https://github.com/rust-lang/crates.io/commit/663488fc0a0073d47402e61cf9cb999d054652c6
[5] http://www.modulecounts.com/

namely Maven[6] for Java, PyPI[7] for Python, Packagist[8] for PHP, NuGet[9] for .NET, and RubyGems[10] for Ruby. We observed that Maven and Packagist do not support release-level deprecation. RubyGems introduced release-level deprecation in 2012[11] but changed it to deletion in 2015 due to a too-heavy burden on the support team.[12] NuGet and PyPI have started to support release-level deprecation in 2019[13] and 2020[14] but there is not much data available.

In this study, we explore release-level deprecation in Cargo and npm. Cargo implements a forceful release-level deprecation (i.e., yanking). The yank mechanism in Cargo will remove yanked releases from the registry index without deleting the data, compared to the deprecation mechanism in npm which just provides warning messages for deprecated releases and continues to allow the package installation. This forceful deprecation mechanism in Cargo can lead to unresolved dependencies of certain releases of a package. In addition, Cargo records the date on which a release was yanked, which allows us to study how the number of deprecated releases evolves (in contrast to the prior work [138] in which they had to estimate the date of deprecation).

To better understand how release-level deprecation is utilized, particularly for bindings to ML libraries, we mine and analyze data from Rust's official package registry (crates.io[15]). We compare the results of bindings for ML libraries with general packages within the same ecosystem to assess the impact and management of deprecated releases. We collected data of all the 48,823 packages on crates.io for our analysis. In addition, we replicate the analysis on the npm dataset from prior

[6]https://maven.apache.org
[7]https://pypi.org/
[8]https://packagist.org
[9]https://www.nuget.org/
[10]https://rubygems.org
[11]https://github.com/rubygems/rubygems/commit/6f99254adf19a35850b1a9b81eb5356ff45f6090#diff-f1e136837461f7ea89d3c7442de42b0723bc7099fe3d77f76ba96823d8530378
[12]https://blog.rubygems.org/2015/04/13/permadelete-on-yank.html
[13]https://devblogs.microsoft.com/nuget/deprecating-packages-on-nuget-org/
[14]https://discuss.python.org/t/pep-592-support-for-yanked-files-in-the-simple-repository-api/1629
[15]https://crates.io

work [138] for comparison. We aim to answer the following research questions (RQs):

RQ1. What is the prevalence of deprecated releases in bindings for ML libraries compared to general packages?

While bindings for ML libraries have a higher percentage of packages with deprecated releases compared to general packages in Cargo, the percentages of packages with deprecated releases are similar in npm.

RQ2. What is the rationale behind using release-level deprecation in bindings for ML libraries compared to general packages?

We identified five patterns of deprecated releases and the rationale behind the deprecation. Notably, many releases are deprecated for reasons other than being defective, with package removal or replacement being the main rationale behind deprecated releases of bindings for ML libraries in both Cargo and npm.

RQ3. How many packages adopt yanked releases of bindings for ML libraries compared to general packages in Cargo?

Cargo implements a forceful deprecation mechanism, while the proportion of deprecated releases is small in Cargo, a relatively large proportion of packages adopt deprecated releases. Notably, the deprecated releases in bindings for ML libraries account for a very small fraction of these adoptions.

Chapter Organization. The rest of this chapter is organized as follows. Section 3.3 provides background information about the package manager of Rust and its yank mechanism. Section 3.4 discusses related work. Section 3.5 presents the method that we used in our study. Section 3.6 presents the findings of our three research questions. Section 3.7 discusses the implications of our findings. Section 3.8 discusses the threats to the validity of our study. Section 3.9 concludes this chapter.

3.3 Background

In this section, we describe how Rust manages packages, and we discuss the dependency requirements and yank mechanism in Cargo.

3.3.1 Package management in Rust

Cargo is the official package manager of Rust. Most developers use Cargo to compile their packages instead of using the compiler rustc[16] directly. Before performing a compilation, Cargo will resolve the dependencies and download specific versions of packages to satisfy the dependency requirements. After that, developers can run their package locally (as a standalone application) or publish their package to a package registry (as a library) using Cargo.

The Rust community's package registry is crates.io, which stores the packages online and provides a platform to search and browse the information of uploaded packages. Usually, developers interact with crates.io through the command-line interface of Cargo. For example, developers can use the `cargo search`[17] command to find packages in crates.io. In addition, package owners can publish releases to crates.io and manage their packages through the command-line interface. For instance, `cargo publish`[18] will upload the current package to a registry (which is set to crates.io by default). In this chapter, for simplicity we refer to crates.io as Cargo.

3.3.2 Dependencies in Cargo

Dependency requirements in Cargo are based on semantic versioning[19] and Cargo will determine the version of dependencies when developers build their projects. The semantic versioning specification defines that a version number consists of three parts: major, minor, and patch. For example, version number 1.2.3 has a major number 1,

[16] https://doc.rust-lang.org/rustc
[17] https://doc.rust-lang.org/cargo/commands/cargo-search.html
[18] https://doc.rust-lang.org/cargo/commands/cargo-publish.html
[19] https://github.com/steveklabnik/semver

a minor number 2, and a patch number 3. The packages should guarantee that patch updates only introduce "backwards compatible bug fixes", minor updates only add features which are backwards compatible, and only the major updates can introduce breaking changes. We refer to this guarantee as the *semantic versioning guarantee* in our study.

The semantic versioning specification is used by `Cargo.toml`,[20] a file under the directory of a Rust project, to store the dependency requirements. The interpretation of requirement statements is different across software ecosystems [36]. Table 3.1 shows the versioning specifications which are used in Cargo. It is notable that Cargo interprets 1.2.3 as a caret requirement ($\wedge$1.2.3) and the wildcard requirement statement "$\star$" (i.e., matching any version) was banned in January 2016.[21] Cargo searches the registry index to find versions which can satisfy the requirements and downloads dependencies. If there are multiple versions available that satisfy a requirement, Cargo will choose the version which has the largest version number. We call the owner of a dependency requirement a *client* and the package to which the dependency requirement points a *provider*. For example, a client package C has a dependency requirement 3.0.1 for a release from a provider package P. Cargo will choose the greatest version 3.5.1 from P which satisfies this requirement even though there exists an exactly matched version 3.0.1 (since Cargo interprets the requirement 3.0.1 as a caret requirement).

3.3.3 Yanked releases

Cargo provides a command called `cargo yank`[22] to deprecate a published release, which can also be unyanked with the `yank undo` command. After a developer calls the yank command for a certain release, this release will be indicated as yanked and is no longer available from the registry index for Cargo. Thus, when Cargo

[20] https://doc.rust-lang.org/cargo/reference/manifest.html
[21] https://doc.rust-lang.org/cargo/faq.html
[22] https://doc.rust-lang.org/cargo/commands/cargo-yank.html

Table 3.1: Five types of versioning specifications in Cargo

Types	**Statement**	**Interpretation**
Comparison	$=1.2.3$	$[1.2.3]$
	$>1.2.3$	$]1.2.3, +\infty[$
	$<1.2.3$	$[0.0.0, 1.2.3[$
	$\geq 1.2.3$	$[1.2.3, +\infty[$
Compound	$>1.2.3, \leq 2.3.4$	$]1.2.3, 2.3.4]$
Caret	$\wedge 1$	$[1.0.0, 2.0.0[$
	$\wedge 1.2$	$[1.2.0, 2.0.0[$
	$\wedge 1.2.3$	$[1.2.3, 2.0.0[$
	$\wedge 0$	$[0.0.0, 1.0.0[$
	$\wedge 0.1$	$[0.1.0, 0.2.0[$
	$\wedge 0.0.1$	$[0.0.1, 0.0.2[$
	$\wedge 0.1.2$	$[0.1.2, 0.2.0[$
Tilde	~ 1	$[1.0.0, 2.0.0[$
	~ 1.2	$[1.2.0, 1.3.0[$
	$\sim 1.2.3$	$[1.2.3, 1.3.0[$
Wildcard	$\star^a$	$[0.0.0, +\infty[$
	$1.\star$	$[1.0.0, 2.0.0[$
	$1.2.\star$	$[1.2.0, 1.3.0[$
	$1.2.3$	$[1.2.3, 2.0.0[$

[a]: removed in 2016

is trying to resolve dependencies for a project, it will automatically skip yanked releases and choose the release which has the largest version number that still satisfies the dependency requirement. For example, a client package C has a dependency requirement $\sim$2.5.1 for a package P. The latest releases of P are 2.5.5 and 2.5.6. However, the latter was yanked. Hence, Cargo will select release 2.5.5 of P. Due to the deletion of yanked releases from the registry index, Cargo cannot download a

yanked release even if the dependency requirement uses the "=" operator.

Notably, the yank command does not completely delete any data from the package registry, hence the yanked releases can still be downloaded through work arounds. One approach is using the download API which is provided by the package registry,[23] and another approach is through the locking mechanism of Cargo. Cargo will generate a `Cargo.lock`[24] file if the building process is successful. This file stores the versions of dependencies that were used during the build. When the developer compiles the project a second time, Cargo will reuse the versions of the dependencies that are stored in `Cargo.lock` (as long as the developers did not change a required version in `Cargo.toml`) even if the depended versions are yanked or a newer version is available. A standalone application will usually upload both `Cargo.toml` and `Cargo.lock` to its repository,[25] which assures the reproducibility of the building process. In contrast, a library will upload `Cargo.toml` but not `Cargo.lock`, so Cargo will help the clients of this library to determine a suitable version to use. However, as we show in Section 3.6.3, this could lead to unresolved dependencies when building a package that depends on a yanked release.

3.4 Related work

In this section, we discuss related work about software packaging ecosystems and the deprecation of APIs and packages.

3.4.1 Software packaging ecosystems

Most research on software packaging ecosystems has focused on npm [27, 37, 168, 178] of JavaScript, PyPI [72, 161, 166] of Python, and CRAN [23, 24, 33, 54] of R. In this chapter, we study the yank mechanism in the Cargo ecosystem, the packaging

[23] https://doc.rust-lang.org/cargo/reference/registries.html
[24] https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html
[25] https://doc.rust-lang.org/cargo/faq.html#why-do-binaries-have-cargolock-in-version-control-but-not-libraries

system of Rust.

Few studies have focused on the Cargo ecosystem. Evans et al. [46] studied the safety of packages in the Cargo packaging ecosystem of Rust. They found that 29% of the packages directly use the `unsafe` keyword, which is provided by Rust to avoid safety checking of the compiler. Furthermore, they observed that popular packages use `unsafe` more frequently.

Many studies include Cargo as a subject when comparing multiple software packaging ecosystems. Decan and Mens [35] investigated pre-releases of three packaging ecosystems and observed that more than 90% of the packages in Cargo published a pre-release as their latest release. In addition, they found that most dependencies that point to pre-releases allow patch updates in Cargo [36], which does not follow the semantic versioning specification. Constantinou et al. [29] studied packages which are distributed across multiple packaging ecosystems and found that these packages in Cargo have more stars on `GitHub` than in other ecosystems. Like other packaging systems, a relatively small proportion of packages are depended on by most of the packages in Cargo [38].

Many researchers have studied the npm ecosystem. In prior work [27], we investigated dependency downgrades in npm and observed that packages changed their dependency constraints for migrating away from defective dependencies. Decan et al. [37] also found that a proper dependency constraint can help a package migrate away quickly from a vulnerable dependency. In addition, they found that most of the security vulnerabilities are fixed before they are published in npm. Wittern et al. [168] studied dependencies in npm and found that the package dependencies keep increasing. However, most of the dependencies point to a small proportion of packages in the ecosystem. Zerouali et al. [178] analyzed various popularity metrics in npm and found that the results of identifying popular packages can be different based on the metrics used.

Imminni et al. [72] implemented a semantic search engine for the PyPI ecosystem

since PyPI has a limited ability to provide quality search results for developers. To detect dependency conflicts in PyPI, Wang et al. [166] developed a tool to monitor the ecosystem. Valiev et al. [161] built survival models for PyPI to analyze the risk of a package become dormant.

German et al. [54] studied the CRAN packaging ecosystem of R. They found that most dependencies point to popular packages and user-contributed packages need more time to grow their community than core packages. Claes et al.[24] observed that the time of fixing errors in CRAN packages differs across operating systems. They also [23] developed a tool to analyze the maintainability of a package in CRAN, which can visualize information such as release history, dependencies and namespace. Decan et al. [33] found that packages in CRAN also manage their repositories on `Github`, which can influence the dependency management.

3.4.2 Deprecated APIs and packages

Cogo et al. [138] studied the deprecation mechanism in npm. To the best of our knowledge, that was the first study that focused on the release-level deprecation mechanism of a software packaging ecosystem. This study focuses on the yank mechanism in Cargo and compares it with the deprecation mechanism in npm. The reason is threefold: 1) The yank mechanism in Cargo is more forceful than the deprecation mechanism in npm. 2) Cargo provides the date of yanking which supports a more in-depth analysis. 3) *"Comparative studies can be seen as a prerequisite for designing successful domain-specific ecosystem solutions"* [144]. Hence, the comparisons across these two software ecosystems can help us better understand the design of a release-level deprecation mechanism.

Many researchers have studied deprecated APIs at the code-level. Sawant et al. [142] interviewed Java API producers and surveyed Java developers, and suggested Java to provide a warning mechanism for developers. Wang et al. [164] investigated six popular packages in Python and observed that developers did not have a consistent

strategy to deprecate an API, and that about 25% of the deprecated APIs are not documented. Robbes et al. [136] analyzed deprecated functions and classes in Smalltalk, and found that about half of the deprecation messages cannot help the developers to migrate away from the deprecation.

Few studies have focused on the deprecation of web APIs. Yasmin et al. [177] analyzed 1,368 RESTful APIs and found that most of the removed APIs did not deprecate the interface to inform their users before introducing the deletion.

Unlike deprecated APIs, it is not easy to identify whether a package is deprecated because the owner may not indicate deprecation in the documentation. Coelho et al. [26] found that the important features to predict whether a package is deprecated or unmaintained include the number of commits and closed issue reports. Khondhu et al. [85] introduced the maintainability index to identify whether a package is inactive or abandoned on `SourceForge.net`. In contrast, Maqsood et al. [111] implemented eight machine learning algorithms to identify successful projects.

Many researchers conducted studies to understand the reasons behind the deprecated and abandoned packages. Coelho et al. [25] surveyed the owners of 104 deprecated `GitHub` packages, and showed that the reasons include environmental factors, project characteristics, and human factors. Iaffaldano et al. [71] interviewed developers from the open-source software community, and also found the reasons behind abandoned packages include human factors and project characteristics. Avelino et al. [5] found that the loss of core developers can increase the risk of a package becoming abandoned.

3.5 Methodology

In this section, we introduce the methodology of our study of yanked releases in the Rust package registry. Figure 3.1 gives an overview of our study.

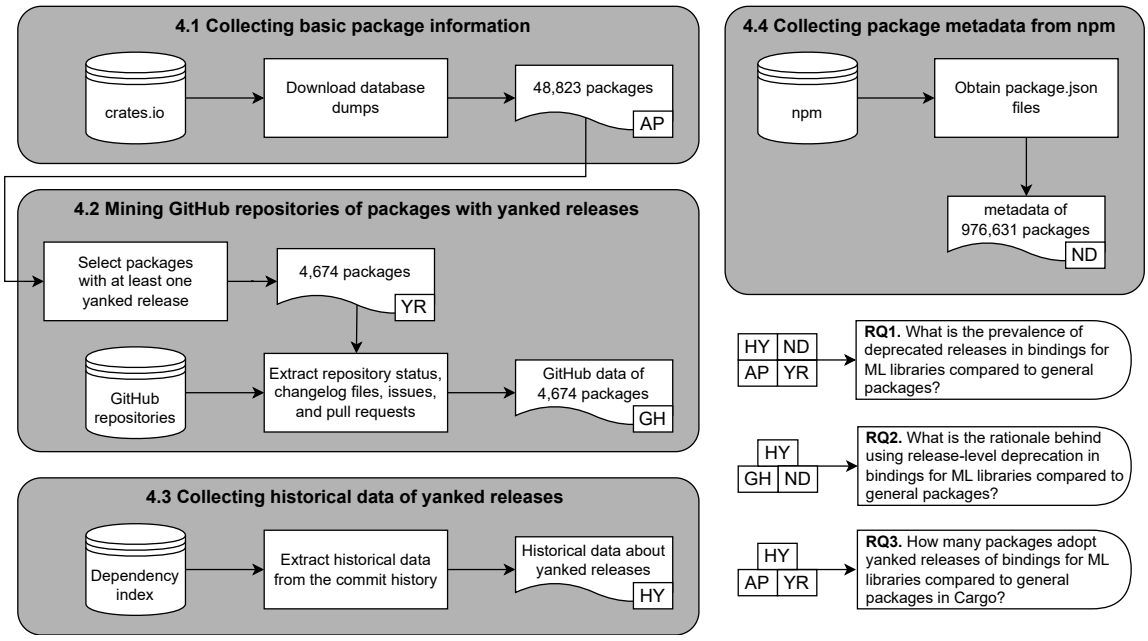


Figure 3.1: Overview of our methodology.

3.5.1 Collecting basic package information

Cargo provides database dumps[26] which contain all the information (e.g., dependencies, downloads, creation date) exposed through the official API. The database dumps are the primary data source of our study and we downloaded the dump that contains the information of 48,823 packages with 294,801 releases on October 29th, 2020. Table 3.2 shows the database fields which store important information for our study.

3.5.2 Mining GitHub repositories of packages with yanked releases

As Table 3.2 shows, yanked flags are stored in the `versions` table. We retrieved all entries that are indicated as yanked. Then, we selected the packages which have at least one yanked release and collected the links to their repositories. For the links which direct to a `GitHub` repository, we used the `GitHub` API[27] to extract the

[26]https://crates.io/data-access
[27]https://docs.github.com/en/rest

issue reports and pull requests of these repositories. In addition, we collected the status of these repositories (active, archived or forked) through the `GitHub` API. If the repository cannot be found, we marked its link as invalid. We distinguish general packages and bindings for ML libraries based on the dataset from Chapter 2.

Furthermore, we collected the changelogs of the packages which have at least one yanked release from their `readme` file and `GitHub` repository. The `readme` field of the `crates` table in the database contains the content of the `readme` file. We identified whether the `readme` contains a changelog by searching for the keywords "changelog", "change log", "release notes", and "release note" in the content, as well as searching "news" and "history" in the headings. For packages which provide a valid link to their `GitHub` repository, we queried the filenames in the root directory of the repository and collected the file if the filename matches the same keywords which we used above.

Table 3.2: Key information in the database.

Field	**Description**
versions.id	The identifier of a release.
versions.num	The semantic version number of a release such as 1.2.3 or 0.1.2-alpha.
versions.created_at	The creation date of a release.
versions.yanked	A flag to indicate whether a release is yanked.
crates.id	The identifier of a package.
crates.readme	The content of the `readme` file in a package.
crates.repository	The link to the repository of a package.
dependencies.version_id	The identifier of the release to which the dependency belongs.
dependencies.crate_id	The identifier of the package to which the dependency points.
dependencies.req	The dependency requirement (e.g., $\wedge$1.2.3 or $\sim$1.2.3).

3.5.3 Collecting historical data of yanked releases

Cargo determines the dependencies based on the registry index which is managed in a `Git`[16] repository. This index repository[28] contains the information (e.g., dependencies, version numbers, and yanking flags) of all published releases. The data of each package is stored in separate files, and the information is updated automatically whenever a change occurs (e.g., a new release is uploaded, yanked or unyanked). Because all the changes are managed in the `Git` repository, the commit history contains the date of each change. We mined the commit messages to extract when a release was yanked or unyanked.

However, we noticed that the commit history in the main branch is not complete because the maintainer of Cargo regularly squashed commits into one to speed up the cloning of the repository.[29] These squashed commits are stored in `snapshot` branches, hence we collected all the commits of these branches to obtain a complete historical overview of (un)yanked releases.

3.5.4 Collecting package metadata from npm

We reused the dataset from the prior study [138] which contains the metadata of 976,613 packages from npm at May 5th, 2019. For each package, we collected the information of all releases and selected the dependencies in their latest releases. Finally, we extracted 7,829,362 releases and 6,178,019 dependencies from 976,613 packages. Similar as for Cargo, we group the packages as general packages and bindings for ML libraries based on the dataset from Chapter 2.

3.6 Results

In this section, we present the motivation, approach, and findings for each of our three research questions (RQs).

[28] https://github.com/rust-lang/crates.io-index
[29] https://internals.rust-lang.org/t/cargos-crate-index-upcoming-squash-into-one-commit

3.6.1 RQ1: What is the prevalence of deprecated releases in bindings for ML libraries compared to general packages?

Motivation. Deprecation can happen at the code-level, release-level, and package-level. Prior work [138] has studied the release-level deprecation mechanism in npm. Similarly, Cargo has a yank mechanism for release-level deprecation to allow the owner of a package to "remove a previously published crate's version from the server's index".[30] In contrast to npm, Cargo records the date on which a release was yanked or unyanked. Hence, we can study how often developers use the yank mechanism in the history of Cargo. In this RQ, we investigate the prevalence of release-level deprecation in bindings for ML libraries compared to general packages in the Cargo and npm ecosystems. By understanding the frequency of deprecated releases, we gain insights into how commonly this deprecation mechanism is employed.

Approach. To quantify the usage of the release-level deprecation mechanism, we calculated the proportion of packages that have at least one deprecated release in both Cargo and npm, distinguishing between bindings for ML libraries and general packages. In contrast to npm, Cargo records the date on which a release was yanked or unyanked. Hence, we can study how often developers use the yank mechanism in the history of Cargo. To analyze the trend of usage, we investigated the historical information of releases and yanked releases in Cargo. We collected the date on which a release was published from the `created_at` field of the `versions` table (as shown in Table 3.2). In addition, to count the yanked releases in a certain period more precisely, we also considered the date of unyanking a release. Finally, we calculated the number of releases and the proportion of yanked releases from November 2014 to October 2020.

Next, we calculated the deprecation rate (i.e., the percentage of deprecated releases in a package) for every package. For example, the deprecation rate is 100% for *fully*

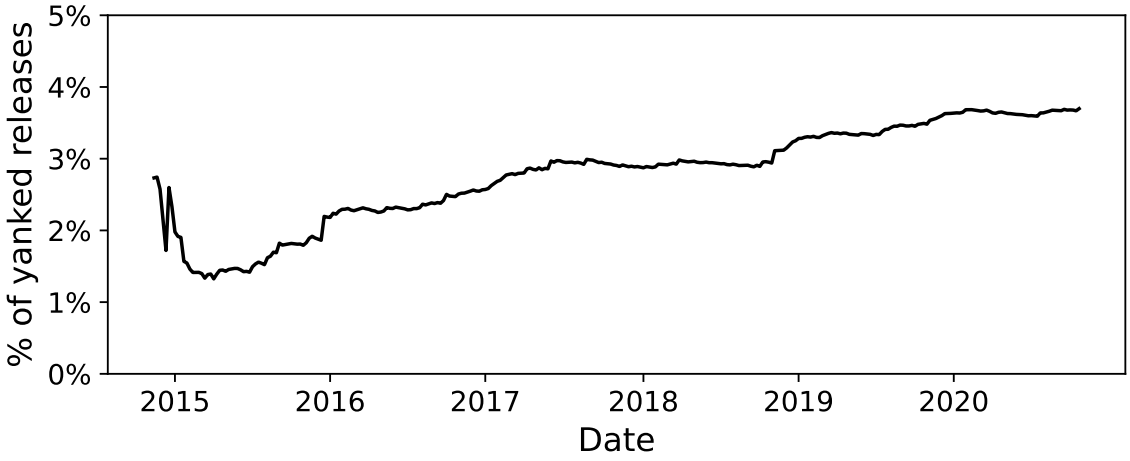
[30] https://doc.rust-lang.org/cargo/commands/cargo-yank.html

deprecated packages (i.e., packages of which all releases are deprecated), and 0% for packages which do not have any deprecated release. Then, we compared our findings between bindings for ML libraries and general packages in Cargo and npm by performing the Mann-Whitney U test [110] at a significance level of $\alpha = 0.05$ to determine whether the differences are significant. However, the Mann-Whitney U test only determines whether two distributions are different. Therefore, we computed Cliff's delta d [106] effect size to quantify the difference. To explain the value of d, we used the thresholds which are provided by Romano et al. [137]:

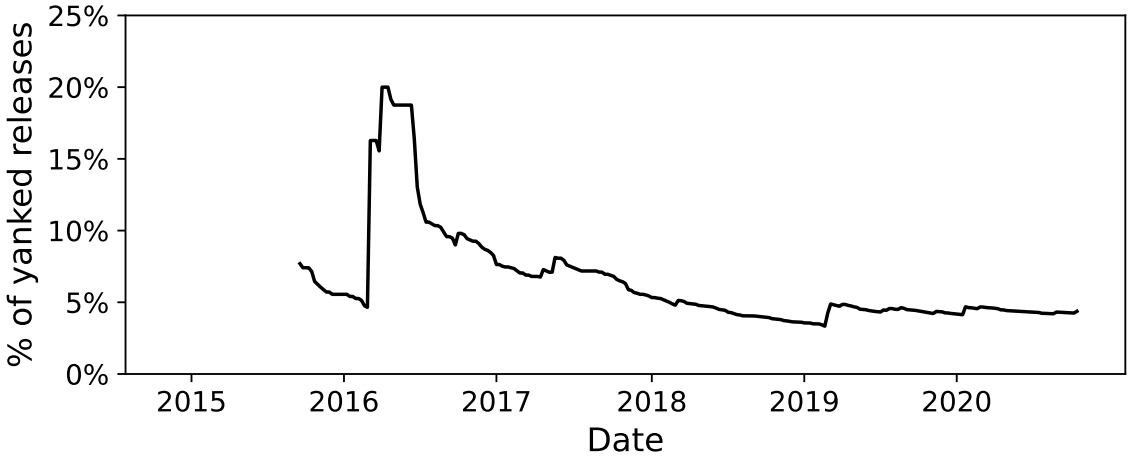
$$\text{Effect size} = \begin{cases} negligible, & \text{if } |d| \leq 0.147 \\ small, & \text{if } 0.147 < |d| \leq 0.33 \\ medium, & \text{if } 0.33 < |d| \leq 0.474 \\ large, & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

Findings. In Cargo, bindings for ML libraries have a higher percentage of packages with deprecated releases compared to general packages, but the percentages are similar in npm. We identified 103 bindings for ML libraries in Cargo and 1,251 in npm. In Cargo, 12.6% of these bindings have at least one yanked release, which is higher than the 9.6% observed among general packages. In contrast, the percentage of bindings for ML libraries with at least one deprecated release in npm is similar to that of general npm packages (3.6% vs. 3.7% [138]). Overall, the release-level deprecation mechanism is used more frequently in Cargo compared to npm.

Between 2014 and 2020, the percentage of yanked releases of bindings for ML libraries in Cargo has decreased from 7.2% to 4.4%, while the percentage of yanked releases of general packages has gradually increased from 1.4% to 3.7%. We found that unyanking only happened 725 times in the history, which is relatively uncommon compared to 10,761 yanked releases in Cargo. Figure 3.2a shows that the percentage of general packages with yanked releases in



(a) General packages



(b) Bindings for ML libraries

Figure 3.2: The percentage of yanked releases in Cargo from November 2014 to October 2020.

Cargo initially decreased but has been gradually increasing since then. The Cox-Stuart test [32] shows that the increasing trend of the number of releases is significant ($p \ll 0.05$). In contrast, Figure 3.2b shows that the percentage of bindings for ML libraries with yanked releases in Cargo decreased at first, then sharply increased to a peak of around 20% in mid-2016, and has been decreasing since then to 4.4%. The peak around mid-2016 is due to the fact that there were only 13 bindings for ML libraries at the time, and one of them yanked all releases (i.e., abandoned). The Cox-Stuart test confirms that this decreasing trend is significant ($p \ll 0.05$). This

trend suggests a maturing approach to the maintenance of bindings for ML libraries. We cannot analyze the trend of the proportion of packages with deprecated releases in npm because it does not provide historical information about deprecated releases.

The deprecation rates of bindings for ML libraries and general packages are not significantly different in Cargo, but they differ in npm. The Mann-Whitney U test indicates that the distributions of the deprecation rate are not significantly different between bindings for ML libraries and general packages in Cargo. However, we observe that most (75%) of the general packages with deprecated releases in Cargo are partially yanked, while the majority (54%) of the bindings for ML libraries are fully yanked. In contrast, the results of the Mann-Whitney U test and Cliff's Delta $|d|$ in npm reveal that the deprecation rate for bindings for ML libraries is significantly lower than for general packages, with a small effect size ($|d| = 0.181$). We also note that fully deprecating a package is less common among bindings for ML libraries (49%) compared to general packages (80% [138]) in npm.

RQ1 Summary

In Cargo, bindings for ML libraries have a higher percentage of packages with yanked releases compared to general packages, but their deprecation rates are similar. In npm, the percentages of packages with deprecated releases are similar, but fully deprecating a package is less common among bindings for ML libraries.

3.6.2 RQ2: What is the rationale behind using release-level deprecation in bindings for ML libraries compared to general packages?

Motivation. Prior work by Cogo et al. [138] has studied non-forceful release-level deprecation in npm. However, in Cargo, release-level deprecation (yanking) is forceful, which means that releases are no longer accessible once they are deprecated. In this research question, we investigate the usage and rationales behind deprecated releases in packages within Cargo and npm, with a specific focus on comparing bindings for ML libraries to general packages. The results can help us to understand the usage of

release-level deprecation from the developers' point of view.

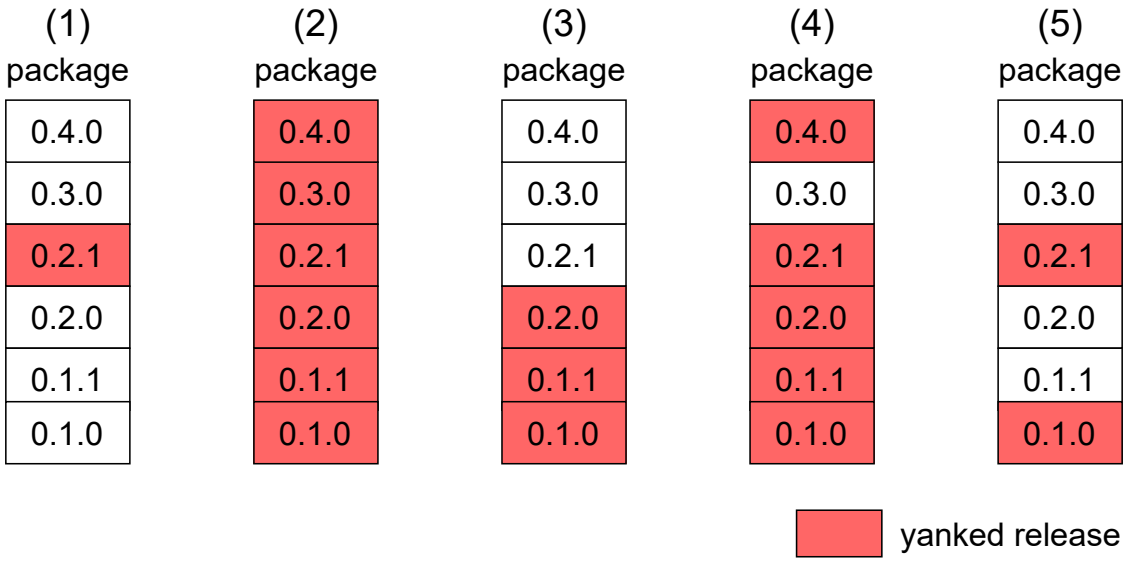


Figure 3.3: Five patterns of yanking: (1) A package yanked only one release; (2) A package yanked all releases; (3) A package yanked back-to-back releases; (4) A package yanked all releases except one; (5) A package yanked nonadjacent releases.

Approach. To understand the usage of the yank mechanism, we looked for five possible patterns in which releases are yanked in Cargo (as shown in Figure 3.3). First, we collected the packages which have at least one yanked release and sorted their releases based on the release date. Then, we went through the collected 4,674 packages to identify whether a package belongs to one of the five patterns.

In addition, we investigated the changelogs, issue reports, and pull requests from the 4,674 packages to analyze the rationales behind yanking. We selected the packages which contain the terms "yank" or "deprecate" in their changelogs, issue reports, or pull requests. We went through the selected 638 packages and filtered out 380 packages which did not provide information that is related to the yanked release. After that, we performed open card sorting together to identify the rationales behind the yanked releases of the remaining 258 packages. We could not identify the rationales for 9 out of 258 packages during the card sorting (i.e., 3.5% false positives of the filtering process). Hence, our results cover the remaining 249 packages.

For bindings of ML libraries with deprecated releases in both Cargo and npm, we followed the same approach to analyze the deprecation rationales. We compared the results for Cargo with the card sorting results for general packages. Regarding npm, we compared the results with the findings from prior work [138].

Findings. The usage of the yank mechanism among packages in Cargo follows one of the five patterns in Figure 3.3. In addition, developers yanked releases for 11 reasons (see Table 3.3). We summarize below the patterns that we were able to identify, together with some examples and the rationales we categorized in the card sort.

Table 3.3: Identified rationales behind yanked releases in the card sort

Rationale	**Description**	**Pkg**	**P1**	**P2**	**P3**	**P4**	**P5**
Breaking SemVer	The release introduces breaking changes and therefore does not follow the semantic versioning specification	43.0%	47.5%	25.0%	40.0%	-	42.9%
Defect	Release contains a defect/bug	36.9%	28.0%	-	46.7%	100.0%	50.0%
Fixing "*" dependencies	Release uses the wildcard dependency ("*") which has been prohibited since 2016	7.2%	4.2%	25.0%	8.3%	-	8.9%
Package removed or replaced	The whole package is removed or is replaced by another package	5.2%	5.1%	50.0%	-	33.3%	-
Broken dependencies	The release contains a broken dependency	4.8%	6.8%	-	1.7%	-	5.4%
Bump propagation	A patch release that includes a minor/major update of an existing dependency and therefore should be a minor/major release as well	3.2%	5.1%	-	1.7%	-	1.8%
MSRV policy	Upgrading the minimum supported rust version (which is a breaking change) in a patch update	2.8%	0.8%	-	6.7%	-	3.6%
Yanked dependencies	Dependencies are yanked	2.0%	3.4%	-	1.7%	-	-
Placeholder release	An initial release for holding the name in Cargo	1.2%	0.8%	-	1.7%	-	1.8%
License updated	Forgetting to update the license	0.4%	-	-	-	-	1.8%
Unsupported	Releases are no longer supported	0.4%	-	-	1.7%	-	-

Note: one package can have multiple rationales as it can have multiple yanked releases. We identified one rationale per yanked release. The "Pkg" column presents the percentage of packages that contain the corresponding rationales. The "P1" to "P5" columns present the percentage of packages that contain the corresponding rationales under the yanking patterns.

Pattern 1: Yanking only one release (40% of the packages). We found this pattern in 1,879 packages and *Breaking SemVer* is the main rationale behind this pattern (47.5%) in the card sort. One example is pyo3[31], an issue report of this project mentions that "[v]ersion 0.5.1 breaks SemVer guarantees"[32] because the owner of pyo3 accidentally merged a new feature to this patch update for 0.5.0. It introduced breaking changes and should be a minor or major update since the semantic versioning guarantee requires that patch updates only introduce backwards-compatible bug fixes. Hence, the owner yanked 0.5.1, backported the changes, and published 0.5.2 which did not include the new feature.

In addition, we noticed that developers yanked a release for *Broken dependencies* (6.8%) or *Yanked dependencies* (3.4%). One example of *Broken dependencies* is diesel_cli,[33] which yanked version 0.99.0 to restrict the dependency of clap from $\geq$2.27.0 to $\wedge$2.27.0 to prevent using 3.x.x versions of clap since major updates could introduce breaking changes. For *Yanked dependencies*, one example is version 0.9.2 of winping.[34] This version was yanked "due to a yanked dependency"[35] and winping had to update the dependency of the quote[36] package from 1.0.2 (yanked) to 1.0.3 in version 0.9.3.

Bump propagation (5.1%) is a notable rationale, which happens when updating the existing dependencies of a package. For instance, sdl2[37] updated the requirement of sdl2-sys from $\wedge$0.7.0 to $\wedge$0.8.0 and published a patch update 0.12.2 for 0.12.1. Since another package sdl2_image depends on $\wedge$0.12.1 of sdl2 and $\wedge$0.7.0 of sdl2-sys "which leads to conflicts",[38] sdl2_image was broken and sdl2 had to yank 0.12.2 and republished it as 0.13.0. The behaviour of bumping the required version of a

[31]https://crates.io/crates/pyo3
[32]https://github.com/PyO3/pyo3/issues/285
[33]https://crates.io/crates/diesel_cli
[34]https://crates.io/crates/winping
[35]https://github.com/TyPR124/winping/blob/master/RELEASES.md
[36]https://crates.io/crates/quote
[37]https://crates.io/crates/sdl2
[38]https://github.com/Rust-SDL2/rust-sdl2/issues/478

dependency (sdl2-sys) in sdl2 broke its dependent (sdl2_image) and sdl2 had to bump its version number as well.

Pattern 2: Yanking all releases (25% of the packages). We observed this pattern in 1,172 packages, with 60% of these packages having only one release. The main rationale behind this pattern (50.0%) in the card sort is *Package removed or replaced* since developers cannot point a new dependency to a fully yanked package. One example is ncollide,[39] the owner explained that "the overly generic crate ncollide has been replaced by two distinct crates: ncollide2d and ncollide3d which are dedicated to 2D and 3D respectively."[40] Another example is c,[41] which yanked all releases to "kill"[42] this package. We also identified *Breaking SemVer* (25.0%) and *Fixing "*" dependencies* (25.0%) rationales under this pattern, however, the identified rationales are not for the whole package, but for some specific releases.

Pattern 3: Yanking back-to-back releases (17% of the packages). There are 814 packages that followed this pattern and *Defect* (46.7%) is the most common rationale that we identified in the card sort under Pattern 3. We noticed that developers often yanked multiple releases due to the same defect. For example, clap[43] yanked the versions from 1.4.0 to 2.21.0 since these versions are all affected by "an erroneous definition of a macro".[44] In fact, one of the maintainers left a comment about Cargo not supporting "yank[ing] everything from X.X.X to Y.Y.Y". We also observed packages that yanked older releases for security purposes, such as untrusted[45] and bitvec.[46] Moreover, the only instance of the *Unsupported* rationale in the card sort was found under this pattern. Old releases of ring[47] were yanked because the owner no longer

[39]https://crates.io/crates/ncollide
[40]https://github.com/dimforge/ncollide/issues/322
[41]https://crates.io/crates/c
[42]https://github.com/hilbert-space/c/issues/1
[43]https://crates.io/crates/clap
[44]https://github.com/clap-rs/clap/issues/2076
[45]https://github.com/briansmith/untrusted/issues/29
[46]https://github.com/bitvecto-rs/bitvec/issues/59
[47]https://crates.io/crates/ring

supports these versions, even though they do not have any known vulnerabilities. Instead, the owner recommends that people only use the latest version.[48] In addition, we observed that packages yanked multiple releases due to the *MSRV*[49] *policy* (6.7%). For instance, block-buffer[50] yanked versions 0.7.0 to 0.7.2 since these versions used an interface that "was stabilized only in Rust 1.28",[51] hence, the dependents of block-buffer will be broken if they used a lower version of Rust.

Pattern 4: Yanking all releases except one (11% of the packages). There are 506 packages that follow this pattern and 94% of these packages left their newest release unyanked. We only found three packages that explained the rationales under this pattern in the card sort. All three packages explained that at least one of the yanked releases contained a defect. battery-cli[52] is the only package that explained that the rationale for yanking was *Package removed or replaced*. This package left the newest release unyanked to show the `readme` file which mentions that "this crate was yanked and replaced by battop crate" instead of the default page on the website.

Pattern 5: Yanking nonadjacent releases (7% of the packages). There are 383 packages that do not belong to any patterns which we proposed. Similar to Pattern 1 and Pattern 3, the two most common rationales that we identified in the card sort are *Defect* (50.0%) and *Breaking SemVer* (42.9%). In addition, the only instance of *License updated* we identified in the sort is from this group. Particularly, sic[53] yanked version 0.10.0 since the "dependency licenses [were] not updated".[54] sic also yanked version 0.7.1 for a defect which "fails to build from crates.io".[55]

***Package removed or replaced* is the main rationale behind deprecated releases of binding for ML libraries in both Cargo and npm.** We observe

[48]https://github.com/briansmith/ring/issues/774
[49]https://github.com/rust-lang/rfcs/pull/2495
[50]https://crates.io/crates/block-buffer
[51]https://github.com/RustCrypto/utils/issues/22
[52]https://crates.io/crates/battery-cli
[53]https://crates.io/crates/sic
[54]https://github.com/foresterre/sic/issues/193
[55]https://github.com/foresterre/sic/issues/50

that 61.5% of the bindings for ML libraries with yanked releases in Cargo fall under this category, exceeding the overall percentage observed in the card sort (i.e., 5.2%). 50% of the bindings for ML libraries used Pattern 2 to cooperate with the rationale. In contrast, for ML library bindings in npm, 60.0% of the deprecated releases are due to *Package removed or replaced*, which is less common compared to the 80% observed for general packages in prior work [138]. *Defect* remains a common rationale across bindings for ML libraries in both Cargo (30.8%) and npm (28.9%), ranking as the second most frequent rationale for the deprecation in bindings for ML libraries.

5.3% of the packages with at least one yanked release explain the rationales in their changelogs, issue reports, and pull requests. During the pattern analysis, we observed that the proportion of packages which explain the rationales behind yanked releases is relatively small, compared to 64% of the deprecation messages in npm which explain the rationales for deprecating a package or release [138]. Among the packages with yanked releases, 4.1% explain the rationale in their GitHub issue reports or pull requests, while 1.5% include the explanation in a changelog. However, the majority (79%) of these packages do not maintain a changelog. Hence, it is not possible to identify the rationales behind all yanked releases.

RQ2 Summary

In Cargo, packages use the release-level deprecation for several reasons other than just to indicate a release is defective, with breaking SemVer being the most common rationale. For ML library bindings, package removed or replaced is the primary reason for deprecated releases in both Cargo and npm.

3.6.3 RQ3: How many packages adopt yanked releases of bindings for ML libraries compared to general packages in Cargo?

Motivation. Cargo does not allow the owner of a package to delete their releases, but it allows the owner to remove a release from the registry index by yanking. Hence, packages cannot point a new dependency to a yanked release because Cargo

cannot find it in the registry index. Moreover, the yank mechanism in Cargo is more forceful than the deprecation mechanism in npm. When npm resolves the dependency requirements for a package, it will use a deprecated release if needed and provide a warning message. However, Cargo will not choose a yanked release even if only this yanked release can satisfy the dependency requirements, which leads to *unresolved dependencies*. We call a release with unresolved dependencies an *implicitly yanked release*, because Cargo cannot use this release to resolve the dependency requirements of other packages (and hence build those packages) even though this release was not yanked explicitly. In this research question, we focus on Cargo to study how often packages directly adopted yanked releases, comparing the adoption rates between bindings for ML libraries and general packages. We also investigate how many releases have unresolved dependencies due to yanked releases in both categories.

Approach. We analyzed the `dependencies` table in the database (as shown in Table 3.2) to collect packages which directly adopted at least one yanked release. First, we selected the dependency requirements which can be satisfied by a yanked release. Then, we collected the information about the owners (i.e., releases from different packages) of these requirements, distinguishing between bindings for ML libraries and general packages. For each yanked release d, we calculated the proportion p_d of direct adoptions of yanked releases [138]. The value of p_d was calculated by:

$$p_d = \frac{a_d}{\sum_{d \in yanked\ releases} a_d}$$

where a_d is the number of times that release d is directly adopted by a package. For example, if packages directly adopted 100,000 yanked releases (sum of a_d) and a yanked release d accounts for 1,000 times of these adoptions (a_d), the value of p_d is 1%.

Next, we study how many releases have unresolved dependencies because of adopting yanked releases and compare the impact on bindings for ML libraries and general

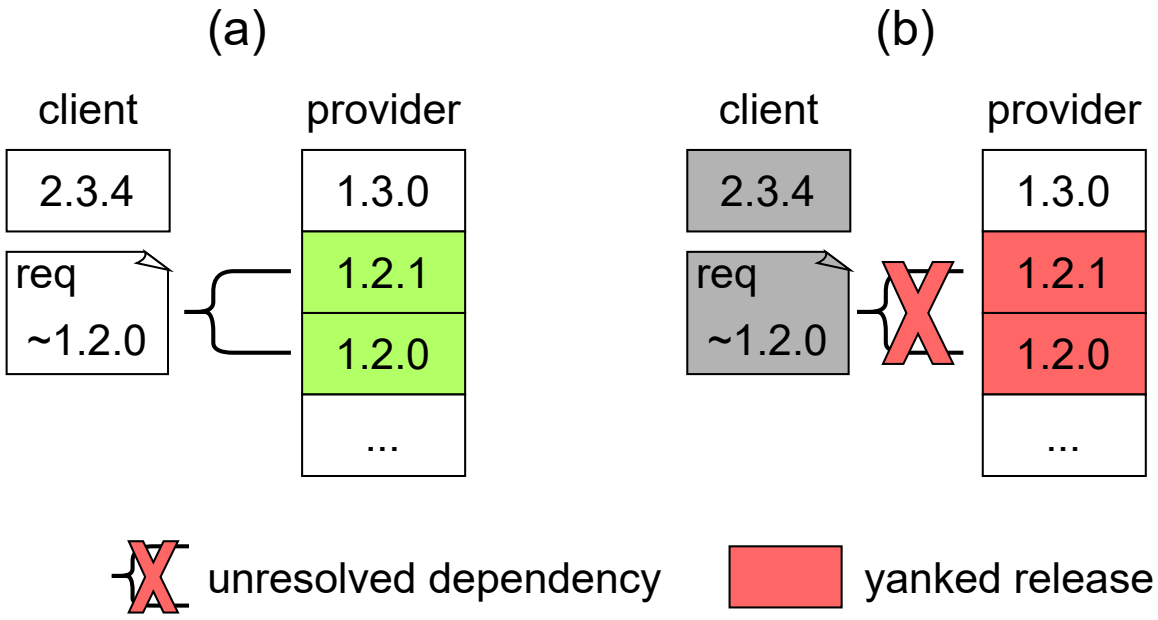


Figure 3.4: Two scenarios of resolving dependencies: (a) The dependency requirement can be resolved; (b) The dependency requirement cannot be satisfied because of yanking.

packages. Figure 3.4 shows two scenarios in which a dependency requirement of a client package points to a provider package. In the first scenario, version 2.3.4 of the client package has a requirement ∼1.2.0, and this requirement can be resolved by 1.2.0 or 1.2.1 of the provider package. In the second scenario, only versions 1.2.0 and 1.2.1 can satisfy the requirement ∼1.2.0 but these two versions are both yanked. We investigated all the dependency requirements and collected the releases which have unresolved dependencies, categorizing them into bindings for ML libraries and general packages.

In addition, we study how yanked releases propagate through the dependencies in the ecosystem. The propagation happens when yanked releases break dependency requirements and cause implicitly yanked releases (i.e., releases with unresolved dependencies). We collected the implicitly yanked releases which directly adopted yanked releases. However, the propagation continues if those implicitly yanked releases cause new unresolved dependencies. Hence, we performed the analysis recursively to collect all implicitly yanked releases.

Findings. While 46% of all packages in Cargo directly adopted at least one yanked release, bindings for ML libraries accounted for a very small fraction of these adoptions. There are 22,277 packages that directly adopted at least one yanked release of a partially yanked package, only 4 packages (less than 0.1%) adopted yanked releases from bindings for ML libraries. Similarly, among the 268 packages that directly adopted fully yanked packages, only 2 packages (0.7%) were related to bindings for ML libraries. In contrast, a small subset (2.4%) of the yanked releases from general packages accounted for 75% of all the adoptions. Although such adoptions do not directly put a package at risk (since there could be newer releases that satisfy a requirement), they could contribute to unresolved dependencies if another release is yanked.

1.4% of the releases in Cargo have unresolved dependency requirements which are related to yanked releases, with bindings for ML libraries having a minimal impact. 4,158 releases in Cargo have unresolved dependency requirements because it directly or transitively adopted a yanked release and these releases became implicitly yanked releases. However, bindings for ML libraries were involved in only 10 instances (0.2%) of these unresolved dependencies. We found that 65.2% (2,712 releases) of the implicitly yanked releases are caused by packages that follow Pattern 3 (yanking back-to-back releases) in Section 3.6.2 and 39.2% (1,631 releases) of the implicitly yanked releases are caused by the ring package which yanked unsupported old releases (see the example in Figure 3.5). The few unresolved dependencies caused by bindings for ML libraries were due to bindings following Pattern 2 with a rationale of *Package removed or replaced*. Furthermore, 54% of the implicitly yanked releases were caused by direct adoption of yanked releases rather than transitive adoption, with all the implicitly yanked releases propagated from bindings for ML libraries being caused by direct adoption.

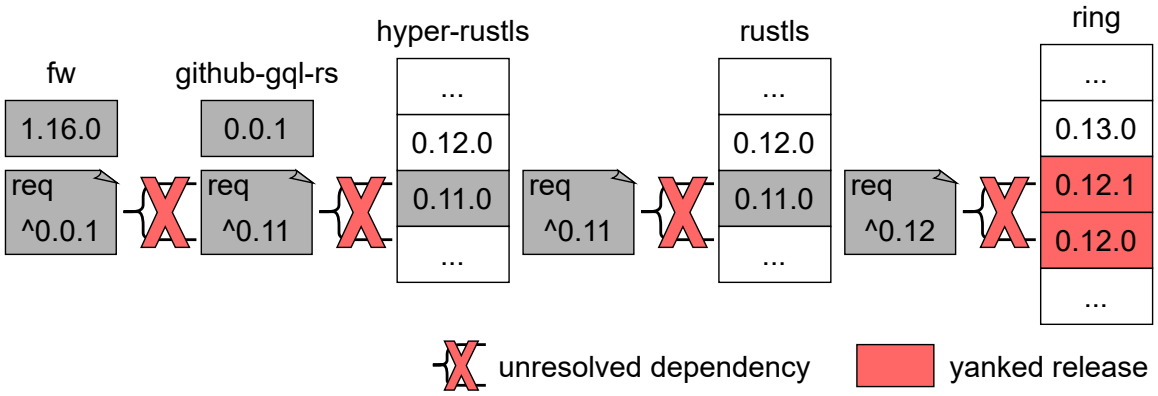


Figure 3.5: An example of yanking propagation of ring. 0.11.0 of rustls, 0.11.0 of hyper-rustls, 0.0.1 of github-gql-rs, and 1.16.0 of fw became implicitly yanked releases.

RQ3 Summary

46% of packages in Cargo directly adopted yanked releases and 1.4% of the releases that currently have unresolved dependencies due to the yanking propagation. Compared to general packages, bindings for ML libraries had a minimal impact on both the adoption of yanked releases and the propagation of unresolved dependencies.

3.7 Implications

In this section, we discuss our findings and implications for the maintainers of package managers, the package owners, the maintainers of Cargo, and researchers.

3.7.1 Implications for maintainers of package managers

Package managers should implement a release-level deprecation mechanism. The proportion of deprecated releases in Cargo and npm are 3.7% and 3.2% respectively, and the percentage of deprecated releases has gradually increased from 2014 to 2020 in Cargo (Section 3.6.1). Since the release-level deprecation mechanism has seen increased use in these two ecosystems, there is likely a need for it in other ecosystems as well. In recent years, PyPI and NuGet saw the need and have started to support release-level deprecation in April 2020 and September 2019. We suggest other package managers also implement the release-level deprecation mechanism.

Package managers should provide features to support the various ways in which developers deprecate releases. We observed five patterns of yanking in Cargo (Section 3.6.2). To support Pattern 2, we suggest that the deprecation mechanism of package managers should support package-level deprecation, which is also used by 80% of the deprecations in npm [138]. However, package managers should notice that deprecating packages can break the dependents of those packages if the managers implement a forceful deprecation mechanism. To support other patterns, we suggest package managers to support deprecating a single release or multiple releases at the same time to offer flexibility for developers.

Package managers should allow package owners to decide whether a deprecation is forceful or not. A forceful deprecation mechanism like yanking in Cargo can cause unresolved dependencies for releases of other packages (Section 3.6.3). We suggest that package managers leave the choice of forceful deprecation to the package owners. The package owners can deprecate a release forcefully (like in Cargo) when it is necessary, such as if they found security vulnerabilities in a cryptography package. Otherwise, the package owner can decide to deprecate a release non-forcefully (like in npm) and allow developers to decide whether they still want to adopt the deprecated release in their own packages. In addition, we suggest that package managers provide a warning with information about how many packages would break (similar to our analysis in Section 3.6.3) if a developer decides to deprecate forcefully.

3.7.2 Implications for package owners

Package owners should explain the rationales behind yanked releases in the documentation. 94.7% of the packages that have at least one yanked release in Cargo never explained the rationale behind yanking (Section 3.6.2). This percentage is high compared to npm in which for 64% of the deprecated releases a rationale is known [138]. Since Cargo does not support adding a message for a yanked release, we recommend that package owners record the reason for yanking a release in the

package's documentation. For example, package owners can create an issue report to track a yanked release and put its link into the readme or changelog. The issue report should contain detailed information about a yanked release, and provide a place for developers to discuss this release, or to give advice on how to deal with the yanking. In addition, we noticed that only 21% of the packages (Section 3.6.2) have a changelog in Cargo. We suggest package owners maintain a changelog to tell developers about the notable changes in each release, which can also be used to explain the rationale behind yanked releases.

Package owners should avoid yanking old releases which are no longer supported without providing an alternative release or migration guidelines. We found that 39.2% of the implicitly yanked releases in Cargo are caused by the ring package which yanked old unsupported releases (Pattern 3 in Section 3.6.2), even though these packages had no known vulnerabilities (Section 3.6.3). For the packages that insist on yanking unsupported releases, we recommend they indicate replacement releases or provide guidelines for developers to migrate away from yanked releases.

3.7.3 Implications for Cargo maintainers

We compare the yanked mechanism in Cargo with the deprecation mechanism in npm based on our findings. We summarize the difference in Table 3.4 for Cargo maintainers.

Table 3.4: Comparisons of the yanked mechanism in Cargo and the deprecation mechanism in npm.

Package manager	**Record deprecation date**	**Describe rationale**	**Deprecate a package**	**Ban yanked releases**
Cargo	✓			✓
npm		✓	✓	

Cargo should allow the owner of a package to add a note to a yanked release and provide a warning for packages that adopted it. We found that the percentage of packages that explain the rationales behind yanked releases is low in Cargo (Section 3.6.1) compared to npm (5.3% vs. 64%). One reason could be that npm allows the package owners to add a message for deprecated releases while Cargo does not. Moreover, we observed that it is much more common in Cargo to yank only a few releases instead of the whole package (Section 3.6.2) and the owners of packages yanked releases for various reasons. For fully yanked packages, at least developers know that these packages will probably no longer be maintained. However, developers who depend on a partially yanked package can hardly understand what is happening since there is no mechanism for describing why a release was yanked. Hence, we recommend that Cargo should allow the owner of a package to leave a message when they are yanking a release. There is an issue report[56] asking for the same functionality since April 23, 2016, but it is still not implemented. With the increasing proportion of yanked releases in the ecosystem (Section 3.6.1), more developers will be affected by this issue. In addition, we recommend that Cargo should provide a warning message for packages that adopted a yanked release.

Cargo should detect implicitly yanked releases and provide a warning for these releases. We found that 1.4% of releases in Cargo are implicitly yanked (Section 3.6.3). We recommend that Cargo should mention that a release is implicitly yanked on the webpage of a package. For example, there could be an "unresolved" label beside an implicitly yanked release on the webpage, hence developers can avoid using this release. In addition, for a package which adopted an implicitly yanked release, Cargo can show the dependency tree and indicate the release which breaks the dependencies for developers.

Cargo should warn the owner of a package which adopts a yanked release

[56]https://github.com/rust-lang/cargo/issues/2608

in its lock file. `Cargo.lock` stores the information about dependencies locally for a project if the project was compiled successfully. However, one of the dependencies can be yanked after the compilation and Cargo does not inform the developer. Since the proportion of packages which directly adopted yanked releases is 46% in Cargo (Section 3.6.3), we recommend that Cargo should check up the package registry when developers are building their project based on the `Cargo.lock`. Hence, Cargo can give a warning message to developers if one of the dependencies was yanked.

3.7.4 Implications for researchers

Researchers should study automatic semantic versioning guarantee checkers to detect whether a release follows the guarantee. In Section 3.6.2, we found that *Breaking SemVer* is the most common rationale behind yanking. This finding indicates the difficulty for package owners to decide whether an update follows the guarantee. Automatic semantic versioning guarantee checkers can help package owners by analyzing the code before a release is published. In addition, these checkers can be used to analyze how packages in different software ecosystems follow the semantic versioning guarantee.

3.8 Threats to validity

In this section, we discuss the threats to the validity of our study about yanked releases in Cargo.

Internal validity: We analyzed the percentage of yanked releases from 2014 to 2020 based on the `Git` history of the registry index. Ideally, this index repository is updated automatically by a program. However, we found eight records which show that the maintainer edited the index manually to delete some packages. Since these deletions are not considered, our results will include the yanked releases of these deleted packages.

The identification of changelogs is based on searching keywords in `readme` and

matching filenames under the root directory of packages' `GitHub` repository. It is possible that the owner of a package did not use a word in our keyword list to indicate their changelog. We randomly selected 341 readmes which correspond to a 95% confidence level and ±5% confidence interval. The manual analysis shows that our heuristic approach achieves a precision of 91% and a recall of 91%.

In addition, we filter out changelogs, issue reports, and pull requests which do not have "yank" and "deprecate" in the content. This filtering might exclude information that explains why a release was yanked. We randomly selected 100 changelogs and 100 issue reports/pull requests from the excluded samples to manually verify whether they contain the rationales for yanked releases. We found that the keywords searching approach missed 3 of the 100 changelogs (2 for *Package removed or replaced* and 1 for *Defect*) and 0 of the 100 issue reports/pull requests. Since few packages maintained a changelog and most rationales are identified from issue reports/pull requests (Section 3.6.2), the result of identifying rationales is considered reliable. Besides, we selected 638 packages which have the keywords in changelogs, issue reports, or pull requests and filtered out 380 out of the 638 for card sorting. The author who did not participate in the card sorting independently analyzed 100 samples of the 380 packages and reported the false negative rate is 9%.

External validity: We studied yanked releases in the Rust and npm package registries, but the findings of our study may not generalize to the package managers of other programming languages. One reason could be that other package managers may not provide a release-level deprecation mechanism for developers, and the identification of whether a release is yanked could be complicated (or not possible). Future studies should further investigate other software ecosystems with release-level deprecation.

We investigated the changelogs, issue reports, and pull requests of packages with at least one yanked release. However, we only looked at packages which provide a link to their `GitHub` repository. There are 15% of these packages that do not have a link

to their repository and 214 packages (4%) provide a link to other repository hosting platforms such as `Bitbucket`[57] and `GitLab`.[58] Hence, future studies are necessary to investigate if our results hold for packages that maintain their code outside of `GitHub`.

The results of our study might not apply directly to other software ecosystems, because the community and the development model of the programming language can also affect the results. However, our methodology can be applied to analyze other software ecosystems.

3.9 Conclusion

In this study, we investigated the prevalence and rationale behind release-level deprecation in bindings for machine learning libraries, comparing them to general packages in Cargo and npm ecosystems. We studied 48,823 packages in Cargo and utilized the dataset from a previous study on npm [138] to understand the deprecation mechanism. In particular, we studied the frequency in which the deprecation mechanism is used, the patterns and rationales behind deprecation, and the adoption of deprecated releases. The most important findings of our study are:

1. In Cargo, bindings for ML libraries have a higher percentage of deprecated releases compared to general packages, while in npm, the percentages of deprecated releases between the two are more similar. From 2014 to 2020, the proportion of yanked releases for ML bindings in Cargo decreased from 7.2% to 4.4%, whereas the proportion for general packages increased from 1.4% to 3.7%.

2. We observed 5 yanking patterns in Cargo and the rationales include withdrawing a defective release or a release that broke the semantic versioning guarantee, indicating a package is removed or replaced, or fixing dependencies. Notably, the primary reason for deprecating releases of bindings for ML libraries in both Cargo

[57] https://bitbucket.org/
[58] http://gitlab.com/

and npm is package removal or replacement.

3. 46% of packages in Cargo have adopted at least one yanked release. This widespread adoption has led to a substantial propagation of yanked releases, resulting in 1.4% of the releases in Cargo having unresolved dependencies. Although this issue is pervasive across Cargo, it is notable that ML library bindings contribute minimally to this problem.

Our findings provide valuable insights into the usage of release-level deprecation in bindings for ML libraries within software package ecosystems. Based on our findings, we suggest that Cargo should provide a package-level deprecation mechanism and allow package owners to leave a reason for yanking a release, and we recommend that other package managers integrate a release-level deprecation mechanism as well.

Chapter 4

Studying the Correctness and Time Cost of Bindings for Machine Learning Libraries

4.1 Abstract

Bindings for machine learning frameworks (such as TensorFlow and PyTorch) allow developers to integrate a framework's functionality using a programming language different from the framework's default language (usually Python). In this chapter, we study the impact of using TensorFlow and PyTorch bindings in C#, Rust, Python and JavaScript on the software quality in terms of correctness (training and test accuracy) and time cost (training and inference time) when training and performing inference on five widely used deep learning models. Our experiments show that a model can be trained in one binding and used for inference in another binding for the same framework without losing accuracy. Our study is the first to show that using a non-default binding can help improve machine learning software quality from the time cost perspective compared to the default Python binding while still achieving the same level of correctness.

4.2 Introduction

The rapidly improving capabilities of Deep Learning (DL) and Machine Learning (ML) frameworks have been the main drivers that allow new intelligent software applications, such as self-driving cars [62, 120] and robotic surgeons [4, 45, 148]. These intelligent software systems all contain components that integrate one or more complex DL and/or ML algorithms. Fortunately, over the past decade, the need for coding these ML and DL algorithms from scratch has been largely eliminated by the availability of several mature ML frameworks and tools such as TensorFlow [1] and PyTorch [124]. These frameworks provide developers with a high-level interface to integrate ML functionality into their projects. Using such ML frameworks has several advantages including readily usable state-of-the-art algorithms, accelerated computing, and interactive visualization tools for data [119].

ML frameworks are typically accessed using Python, which is now the most popular programming language for ML applications [7, 119, 133]. Gonzalez et al. [57] show that more than 56% of the ML projects on GitHub are written in Python. However, many software projects do not use Python as their primary language[1] and the developers of these projects might be unfamiliar with Python. Since learning a new language is a non-trivial task even for experienced developers [147], these developers have to use a workaround to use the Python ML frameworks in their preferred programming language.

To help non-Python developers with the integration of an ML framework, 25% of the popular ML frameworks offer one or more *bindings* for other programming languages [97]. These bindings expose the functionality of the framework in the binding's language. For example, TensorFlow provides a JavaScript binding[2] that allows developers to integrate ML techniques directly in JavaScript. Because a binding adds an additional layer around the ML framework, it is important to investigate how the qual-

[1] https://githut.info
[2] https://github.com/tensorflow/tfjs

ity of the ML software created using these ML frameworks is impacted. For instance, different bindings may take different amounts of time to build a model.[3] In addition, bugs in the bindings can introduce inconsistencies for trained models. For example, TensorFlow's C# binding had different results than the Python binding when loading an already trained model due to incorrectly handling 'tf.keras.activations' functions.[4] However, no one has systematically investigated the impact of using bindings for ML frameworks on the ML software quality; typically, studies focus on the software quality of the ML frameworks themselves [18, 104, 149], or on the impact of the computing device on which the model executes [61].

To illustrate the potential impact and importance of our study, consider the following real-world scenario. Anna's team uses JavaScript as the primary programming language. Since the team lacks ML or Python expertise, they collaborate with the company's ML team to integrate DL techniques into their projects. They are now considering using an ML framework's JavaScript binding for their project. However, they are concerned about how their developed ML software's quality is impacted by the binding; in particular, they are concerned about the correctness and time cost. There are three possible scenarios for integration of the binding that our study can assist with choosing the best option:

- **Integration Scenario 1:** The ML team develops and trains the DL models and ships the pre-trained models to Anna. In this scenario, Anna needs to use the JavaScript binding to load the pre-trained models and perform model inference in her project.

- **Integration Scenario 2:** The ML team assists Anna in training DL models in the project's native language which is JavaScript, allowing Anna to alter and maintain the code more efficiently. After training the DL models, Anna needs

[3] As can be seen in this GitHub issue for TensorFlow: https://github.com/tensorflow/tensorflow/issues/55476

[4] https://github.com/SciSharp/TensorFlow.NET/issues/991 and https://github.com/SciSharp/TensorFlow.NET/pull/1001

to deploy the trained models to the production environment in JavaScript as well.

- **Integration Scenario 3:** Since computational resources for the project are very limited, Anna is also open to a third scenario, in which the ML team assists her in selecting the most efficient combination of training and inference bindings in any language. In this scenario, Anna is willing to hire an expert in the chosen language(s) to help with the integration of the binding(s) as long as the reduction in computational resources is large enough.

Therefore, in this chapter, we study the impact of bindings on two important ML software quality aspects:

- **Correctness:** We evaluate if models trained using different bindings for a given ML framework have the same accuracy. We study (1) training accuracy, which captures the model's classification performance on the train set during the training process, and (2) test accuracy, which captures the classification performance of the final trained model on the test set. In addition, we measure whether the test accuracy is the same after loading a pre-trained model in a binding that was not used to train the model (the *cross-binding* test accuracy).

- **Time cost:** We evaluate if models trained using different bindings for an ML framework take similar time for training and making inferences. Bindings that produce models with a high time cost are expensive (in terms of computational resources), which limits their applicability.

We conducted model training and model inference experiments using bindings for TensorFlow and PyTorch in C#, Rust, Python, and JavaScript. In the model training experiments, we trained LeNet-1, LeNet-5, VGG-16, LSTM, GRU, and BERT models on the GPU in every binding (excluding BERT which is only trained on the Python bindings) using the same data and as far as possible, the same framework

configuration. In the model inference experiments, we loaded pre-trained models and performed inference using every binding on the CPU and GPU. We do so to address the following research questions (RQs), with RQ1 and RQ2 focusing on correctness, and RQ3 and RQ4 focusing on time cost:

RQ1. How do the studied bindings impact the training accuracy and test accuracy of the studied DL models?

During the training process, bindings for the same ML framework can have different training accuracies for the same model as well as varying test accuracy values (2% difference) in the final trained models.

RQ2. How do the studied bindings impact the cross-binding test accuracy of pre-trained models?

The cross-binding test accuracy of the pre-trained models was not impacted by the bindings.

RQ3. How do the studied bindings impact the training time of the studied DL models?

Non-default bindings can be faster than the default Python bindings for ML frameworks. For instance, PyTorch's Python binding has the slowest training time for the studied models; PyTorch's C# binding is more than two times faster than the Python binding in training the LeNet-5 model.

RQ4. How do the studied bindings impact the inference time of pre-trained models?

Bindings can have very different inference times for the same pre-trained model, and the inference time of certain bindings on CPU can be faster than that of other bindings on GPU. For example, TensorFlow's Rust binding can perform inference faster for an LSTM model on CPU than the JavaScript binding on GPU (73.9 vs. 177.7 seconds).

The main contributions of our study are as follows:

1. We are the first to study the impact of using different bindings for ML frameworks on the ML software quality in terms of correctness and time cost.

2. We found that using a non-default binding can help improve ML software quality (from the time cost perspective) compared to the default Python binding of the studied frameworks in certain tasks, while still achieving the same level of correctness.

3. We provide a replication package [100], which consists of the implementation of the studied ML models in the studied bindings, scripts for running the experiments, and Jupyter Notebooks for analyzing the experiment results.

The remainder of this chapter is outlined as follows. Sections 4.3 provides background information. Section 4.4 describes the design of our study. Sections 4.5 and 4.6 present the results. Section 4.7 discusses the implications of our findings. Section 4.8 gives an overview of related work. Section 4.9 outlines threats to the validity of our study and Section 4.10 concludes the chapter.

4.3 Background

4.3.1 ML Frameworks

Machine learning frameworks are software libraries that provide ML techniques to developers for the development and deployment of ML systems. Most popular ML frameworks are supported by large companies such as Google and Facebook [7]. As shown in Figure 4.1, an ML framework provides interfaces to define the structure of a model, train the defined model using a selected optimizer, and save the trained model for later use. In addition, developers can deploy the trained models to the production environment by loading a saved (or *pre-trained*) model and performing inference. ML frameworks can load a pre-trained model using (1) the *model parameters* (e.g.,

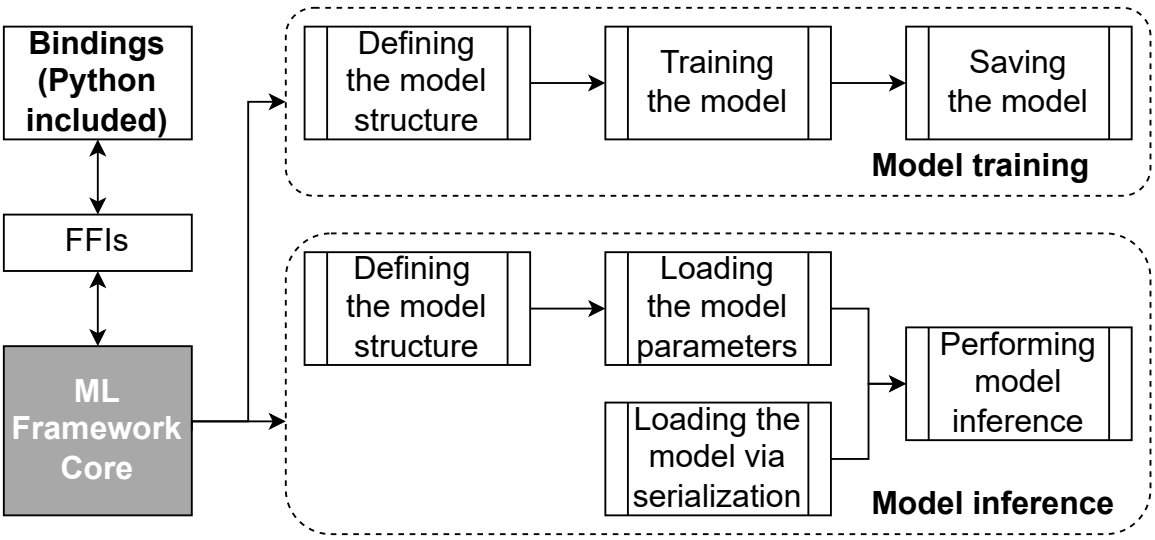


Figure 4.1: Bindings use the functionality of ML frameworks via foreign function interfaces (FFIs) to train models and perform model inference.

weights and hyperparameters) or (2) *serialization*. If only the model parameters are saved, developers first have to define the model structure before they can load the stored parameters into the defined model. When loading a serialized model, the ML framework can recreate the model from the saved file automatically since it contains both the structure and the weights of the pre-trained model.

Modern ML frameworks, such as TensorFlow and PyTorch, have been built upon a foundation that leverages parallel processing devices like GPUs. GPUs have proven to be highly efficient for tasks that demand parallel computation, especially in the realm of ML. Their architecture is inherently designed to handle multiple tasks simultaneously, allowing for massive parallelism. However, one significant characteristic of GPU computations that needs emphasis is their asynchronous nature. When a task is dispatched to a GPU, it does not always execute immediately. Instead, it often gets scheduled in a queue.[5] Consequently, a CPU might continue with its tasks believing that a GPU job is complete when, in fact, it has not even started. This asynchronous behaviour allows GPUs to optimize task execution but also necessitates careful synchronization when precise timing or task ordering is crucial.

[5]https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

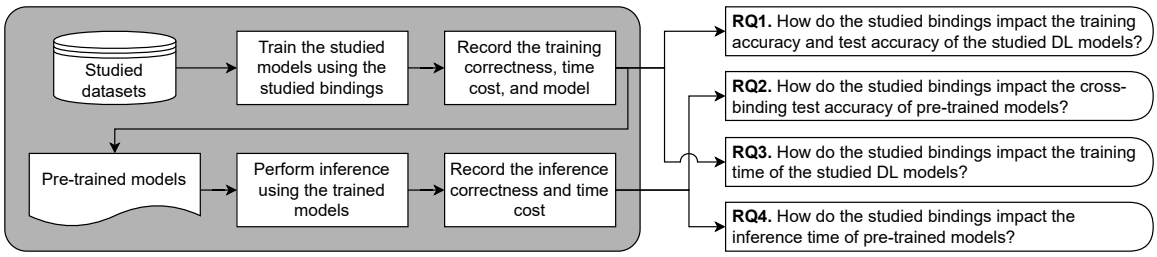


Figure 4.2: Overview of the study design.

4.3.2 Bindings for the ML frameworks

Python is the most popular programming language for ML applications [7, 119], but developers in other languages also have the need for using ML algorithms. Developers might choose an existing ML framework in their preferred language or they have to create a new one from scratch (which requires a large amount of work and is error-prone). Another alternative is to use a *binding* in their preferred language, which provides interfaces to the functionality of an existing ML framework in the language of the binding [97].

As shown in Figure 4.1, bindings access the functionality of the ML framework through foreign function interfaces (FFIs) without recoding the library. FFIs bridge the gap between programming languages, allowing developers to reuse code from other languages. For example, TensorFlow's Rust binding[6] uses the FFI provided by the Rust language[7] to access TensorFlow functionality. Since the GPU support is provided by the underlying C/C++ computational core of ML frameworks, bindings typically leverage FFIs to access these functionalities. For example, the Python bindings for TensorFlow and PyTorch make use of SWIG[8] (Simplified Wrapper and Interface Generator) and Pybind11[9] to generate FFIs for its Python binding to tap into the C++ backend which includes the ability to access the GPU. However, the efficiency in leveraging GPU resources may vary among different bindings.

[6]https://github.com/tensorflow/rust/tree/master/tensorflow-sys
[7]https://doc.rust-lang.org/rust-by-example/std_misc/ffi.html
[8]https://www.swig.org/
[9]https://github.com/pybind/pybind11

Table 4.1: Our studied datasets and models. (Each model is paired with a dataset for the experiments)

Dataset	#Samples*		Model	
	Train	Test	Name	#Parameters
MNIST	60,000	10,000	LeNet-1	4,326
			LeNet-5	61,706
CIFAR-10	50,000	10,000	VGG-16	33,650,890
IMDb	25,000	25,000	LSTM-based	4,665,537
			GRU-based	4,250,817
SQuAD	87,599	10,570	BERT (base)	108,893,186

* The split of the training and test set is provided by the dataset.

4.4 Study Design

In this section, we first describe our experimental environment and the studied datasets, models, ML frameworks, and bindings. Then, we discuss how we evaluate the correctness and time cost in the model training and model inference experiments. Finally, we introduce the experimental setup of our study. Figure 4.2 gives an overview of our study design.

4.4.1 Environment setting

We set up our experimental environment on a dedicated laboratory server provided by ISAIC[10], where we can control the execution of other running tasks. The server runs Ubuntu Linux 20.04 with Linux kernel 5.11.0. We used the CUDA 11.1.74 and cuDNN 8.1.0 GPU-related libraries. The hardware specifications of the server are as follows:

- GPU: 2x NVIDIA TU102 [TITAN RTX] (24 GB)

[10]https://isaic.ca/

- CPU: 3.30 GHz Intel(R) Core(TM) i9-9820X
- RAM: 100 GB

4.4.2 Studied datasets and models

Table 4.1 presents the datasets and models used in this study, specifically pairing each model with the dataset used in the experiments. The datasets we studied are MNIST [93], CIFAR-10 [88], IMDb review [108], and SQuAD [132]. These datasets are widely used as benchmarks in ML research [61, 69, 86, 101, 102, 119, 165, 173]. The models we studied are LeNet [92], VGG [151], LSTM [66], GRU [21], and BERT (the base model) [39] as all of them are typically paired with these datasets in various research domains [2, 22, 51, 61, 67, 69, 146, 163, 165, 175, 176].

MNIST and CIFAR-10 are datasets for image classification tasks. MNIST contains 70,000 grayscale images of handwritten digits, serving as a benchmark for evaluating classification models like LeNet-1 and LeNet-5. We used the CIFAR-10 dataset, which contains 60,000 colour images of 10 different objects, to train the VGG-16 model. The primary metric for these classification tasks is accuracy, reflecting the proportion of correctly identified images out of the total dataset.

The IMDb review dataset is utilized for sentiment analysis (text classification). The dataset contains 25,000 positive and 25,000 negative text reviews of movies. We used it to train the LSTM and GRU models to analyze the sequential nature of text data. Both LSTM and GRU models utilize a recurrent neural network (RNN) structure for handling sequential data, and we integrated a word embedding [28] on the IMDb dataset in our experiments. The performance is measured by accuracy which indicates the model's ability to correctly classify reviews.

SQuAD is a dataset for the extractive question-answering task. SQuAD contains around 100,000 question-answer pairs, where the questions are posed by crowdworkers on a set of Wikipedia articles and the answer to every question is a text span from the corresponding reading passage. We used SQuAD to train the BERT-base model,

Table 4.2: Studied bindings for TensorFlow and PyTorch in software package ecosystems.

Framework	Name	Ecosystem	Language	Version	# Stars†
TensorFlow	tensorflow	PyPI	Python	2.5.0	177,149
	TensorFlow.NET	NuGet	C#	0.60.4	2,906
	tensorflow	Cargo	Rust	0.17.0	4,627
	@tensorflow/tfjs-node	npm	JavaScript*	3.9.0	17,635
PyTorch	torch	PyPI	Python	1.9.0	70,021
	TorchSharp	NuGet	C#	0.93.9	946
	tch	Cargo	Rust	0.5.0	3,178
	@arition/torch-js	npm	JavaScript*	0.12.3	252

* We wrote TypeScript code when using the JavaScript bindings.

† The number of stars on GitHub recorded as of August 24, 2023.

leveraging the model's capability in language understanding. The task is to identify the exact text span (i.e., start and end positions) within the given passage that answers a question. The evaluation metric for SQuAD is the exact match score [132], which calculates the percentage of questions for which the model's answer exactly matches the annotated answer.

4.4.3 Studied ML frameworks

We study the latest stable versions (at the time of starting our study) of TensorFlow [1] (2.5.0) and PyTorch[11] (1.9.0), since they are two of the most popular ML frameworks. TensorFlow and PyTorch have recently grown in popularity as Caffe2 was merged into PyTorch in 2018[12] and Keras became "the high-level API of TensorFlow 2" [83].

[11] https://github.com/pytorch/pytorch/releases/tag/v0.1.1
[12] https://caffe2.ai/

4.4.4 Studied bindings

The studied TensorFlow and PyTorch bindings are shown in Table 4.2. These bindings are all based on the same version of the studied ML frameworks (i.e., TensorFlow 2.5.0 and PyTorch 1.9.0). Notably, TensorFlow and PyTorch both utilize the Python bindings by default. The reason behind selecting bindings in these four software package ecosystems is twofold: (1) Generally, PyPI (Python), npm (JavaScript), and NuGet (C#) are the three most popular software package ecosystems for cross-ecosystem ML bindings [97] and (2) specifically, the Cargo ecosystem (Rust) is popular (according to the number of stars on GitHub) for both TensorFlow[13] and PyTorch.[14] As shown in Table 4.2, the number of GitHub stars serves as a proxy for the popularity of a project in the software engineering domain [12, 47, 64, 170, 172], with TensorFlow's JavaScript binding being particularly notable. Although the number of stars for C# and JavaScript bindings for PyTorch may appear low, we included these to ensure a fair comparison with TensorFlow bindings in respective ecosystems.

4.4.5 Correctness evaluation

Training correctness. During the training process, the correctness is measured in each epoch using the training accuracy which is calculated by $Acc_{train} = N_{correct}/N_{train}$, where $N_{correct}$ is the number of correct predictions and N_{train} is the number of data samples in the training set. For the final trained models, we use the test accuracy $Acc_{test} = N_{correct}/N_{test}$ as the evaluation metric for comparison, which is the accuracy on the test set.

Inference correctness. When we finish training a model, we use the test accuracy Acc_{test} of this pre-trained model as a reference. Then, we perform inference with a studied binding for the pre-trained model on the test set to obtain the cross-binding test accuracy $Acc_{cross\_test} = N_{correct}/N_{test}$ using that binding. The difference between

[13]https://github.com/tensorflow/rust
[14]https://github.com/LaurentMazare/tch-rs

Acc_{test} and Acc_{cross_test} is that the inference correctness is measured in the studied binding. For BERT on SQuAD, we use the exact match score [132] instead of accuracy as the metric to evaluate the correctness.

4.4.6 Time cost evaluation

Training time cost. The training time cost measures the time spent training a model in seconds. Developers commonly train DL models on GPU rather than CPU since the training can be time-consuming and GPU can considerably shorten the training time [14, 96]. Hence, all model training experiments of bindings for ML frameworks are conducted on GPU and we measure the training time cost on GPU only.

Inference time cost. The inference time cost measures the time spent for performing inference with a pre-trained model on the test set in seconds. Since developers can deploy pre-trained models to a production environment which supports the CPU or GPU, the inference time cost of a binding is measured on both CPU and GPU.

Table 4.3: Supported features of studied bindings for TensorFlow (TF) and PyTorch (PT).

		Training	**Supported interfaces**			**Loading models**	
			CNNs	**RNNs**	**BERT**	**Parameters**	**Serialization**
TF	Python	✓	✓	✓	✓	✓	✓
	C#	✓	✓	✗†	✗	✓	✗
	Rust	✗*	✓	✓	✗	✗	✓
	JavaScript	✓	✓	✓	✗	✗	✓
PT	Python	✓	✓	✓	✓	✓	✓
	C#	✓	✓	✓	✗	✓	✗
	Rust	✓	✓	✓	✗	✓	✓
	JavaScript	✗	✗	✗	✗	✗	✓

* Unlike other bindings, TensorFlow's Rust binding does not support the API (Keras-like) of TensorFlow 2.

† TensorFlow's C# binding has only recently introduced support for RNNs based on TensorFlow 2.10, however, our study uses the C# binding for TensorFlow 2.5.0 for consistency across all bindings.

4.4.7 Experimental setup

In this section, we detail our experimental setup with a running example of how we computed the correctness and time cost of LeNet-1 when trained and inferenced using the studied bindings for the studied ML frameworks.

Step 1 – Train the studied models using the studied bindings: We conduct model training experiments for each supported model-dataset pair (as shown in Table 4.1). For a given model-dataset pair, each binding that supports the model's interface and training features (as shown in Table 4.3) trains the model from scratch on that dataset. For example, LeNet-1 and MNIST form one model-dataset pair and each supported binding trains LeNet-1 on MNIST independently. We repeat this process for each model-dataset pair in each binding that supports the model. For consistency, we ensure the following across all bindings for a given model-dataset pair:

- **Model structure.** We use interfaces that provide the same functionality in bindings to build up each layer of the studied models. However, not all bindings support model training, as indicated in Table 4.3. As a result, we do not conduct training experiments with TensorFlow's Rust binding, PyTorch's JavaScript binding, and RNNs in TensorFlow's C# binding.

- **Training set and test set.** We use the provided split of the training set and test set from studied datasets. Before conducting experiments, we perform comprehensive data preprocessing, ensuring that all bindings can work with the same processed data across all experiments.

- **Hyperparameters.** We use the same hyperparameters (e.g., the number of epochs and batch size) and optimizers from prior research [61]. However, TensorFlow's C# binding does not support setting the momentum and weight decay hyperparameters for a stochastic gradient descent (SGD) optimizer. Hence, we

only set the learning rate for the SGD optimizer without enabling momentum and weight decaying when training the LeNet-1, LeNet-5, and VGG-16 models to maintain consistency across all bindings. In addition, to mitigate the risk of default hyperparameters influencing our results, we explicitly defined all configurable parameters and kept them the same across bindings.

- **Random seed.** We fix the value of the random seed across bindings when training the same model to control the randomness.

In addition, we repeat the same training process five times for each binding with different random seeds (that are kept consistent across bindings) to reduce the impact of seed selection on the results.

Running example. We train the LeNet-1 model in TensorFlow's Python, C#, and JavaScript bindings. These bindings all set the same random seed at the start of the training process. To build up the same convolution layers of the model, we use the "Conv2D" interface in Python, "Conv2D" in C#, and the "conv2d" interface in JavaScript. In addition, we use SGD with a learning rate of 0.05 for all three bindings to train the LeNet-1 model.

Step 2 – Record the training correctness and save the model: We record the training accuracy in each epoch for all model training experiments. After the training is completed, we compute the trained model's test accuracy and save the model for later use. Considering the impact of randomness, we repeat the training process 5 times in each training experiment and analyze the distribution of the results to draw conclusions.

Running example. During training the LeNet-1 model in PyTorch's C# binding, we calculate the training accuracy in each epoch and store the value. After finishing the training, we save the trained LeNet-1 model.

Step 3 – Perform inference using the trained models and record the inference correctness: For each model inference experiment, each binding loads

Procedure 1 Measuring Training Time Cost in PyTorch Bindings

```
1: model, optimizer ← initModelAndOptimizer()          ▷ Model and optimizer initialization
2: train_set ← loadDataset()                           ▷ Load pre-processed training set
3: start ← getCurrentTime()                            ▷ Start the timer
4: for epoch ← 1 to epochs do
5:     while not isEndOfDataset(trainSet) do
6:         inputs, labels ← getNextBatch(train_set)    ▷ Batch data loading*1
7:         outputs ← model(inputs)                     ▷ Start forward propagation*2a
8:         loss ← calculateLoss(outputs, labels)       ▷ Loss calculation*2b
9:         loss.backward()                             ▷ Start backward propagation*3a
10:        optimizer.step()                            ▷ Parameter update*3b
11:    end while
12: end for
13: training_time_cost ← getCurrentTime() − start      ▷ Compute elapsed time
14: return training_time_cost
```

*1−3: Subactivities in the training process – forward propagation includes loss calculation and backward propagation includes parameter update.

a pre-trained model via the supported model loading approach(es) (as shown in Table 4.3) and performs inference on the test set on both CPU and GPU. In addition, bindings for the same ML framework perform inference for the same pre-trained model. We select the pre-trained models (which are saved in Step 2) from TensorFlow and PyTorch's default Python bindings since the default bindings tend to have the best support and maintenance [97].

Running example. In TensorFlow's Rust binding, we load the pre-trained LeNet-1 model from TensorFlow's default Python binding via serialization to perform model inference on the test set and record the cross-binding test accuracy.

Step 4 – Measure and record the training time cost: Our primary focus is on measuring the time cost of the entire training process on GPU and recording it, as shown in Procedures 1 and 4. Due to the asynchronous nature of GPU computations (as explained in Section 4.3), we only keep the code directly related to the training process in this step to ensure accurate time measurements, excluding activities like calculating correctness metrics in each epoch (which is included in Steps 1

Procedure 2 Measuring Inference Time Cost in PyTorch Bindings

```
model ← loadSavedModel()                                  ▷ Load trained model
test_set ← loadDataset()                                  ▷ Load pre-processed test set
start ← getCurrentTime()                                  ▷ Start the timer
while not isEndOfDataset(test_set) do
    inputs ← getNextBatch(test_set)                       ▷ Batch data loading*1
    preds ← model(inputs)                                 ▷ Inference forward propagation*2
end while
inference_time_cost ← getCurrentTime() − start            ▷ Compute elapsed time
return inference_time_cost
```

*1−2: Subactivities in the inference process.

Procedure 3 Measuring Time Cost of a Training/Inference Subactivity in PyTorch Bindings

```
start ← getCurrentTime()                                  ▷ Start the timer
runSubactivity()                                          ▷ Execute a subactivity of training/inference
cuda.synchronize()                                        ▷ Wait for the subactivity to finish
time_cost ← getCurrentTime() − start                      ▷ Compute elapsed time
return time_cost
```

and 2). We also do not include the time cost of initialization processes, such as model initialization, optimizer initialization, and initial dataset loading.

Procedure 1 within PyTorch showcases its granular control over the training process. It initiates by setting up the model and optimizer, loading the training dataset, and iterating through the epochs for optimizing the model weights. For each epoch, the process starts with loading a batch of the data. Following this, forward propagation is performed to produce outputs which are used for calculating the loss values. Lastly, backward propagation is executed to calculate the gradients which guide the optimizer for updating the model parameters. In contrast, as demonstrated in Procedure 4, TensorFlow offers less granularity since it encapsulates the entire training process (i.e., batch data loading, forward propagation, and backward propagation) within a single function to optimize performance.

As shown in Procedure 3, the granularity control in PyTorch is particularly helpful in measuring time costs for specific subactivities using the "cuda.synchronize()" func-

Procedure 4 Measuring Training/Inference Time Cost in TensorFlow (TF) Bindings

1: $model \leftarrow$ initModelAndCompile($optimizer, loss\_function$) ▷ Model initialization
2: $train\_set, test\_set \leftarrow$ loadDataset() ▷ Load pre-processed data
3: $start \leftarrow$ getCurrentTime() ▷ Start the timer
4: model.fit($train\_set$, $epochs$)/model.predict($test\_set$) ▷ TF's single training/inference function
5: $time\_cost \leftarrow$ getCurrentTime() $- start$ ▷ Compute elapsed time
6: **return** $time\_cost$

tion to facilitate synchronization between the CPU and GPU. The "cuda.synchronize()" function is only available in the Python and Rust bindings. Procedure 3 starts a timer, runs a subactivity (e.g., forward propagation), waits for the subactivity to finish using "cuda.synchronize()", and then computes the elapsed time.

Running example. We train the LeNet-1 model with PyTorch's Python binding and employ Procedure 1 to record the training time cost. In addition, we rerun the training experiment utilizing Procedure 1 with additional synchronization steps as described in Procedure 3 to capture accurate time costs for individual subactivities.

Step 5 – Measure and record the inference time cost: Similar to Step 4, we measure and record the time cost of the entire inference process on both CPU and GPU following Procedures 2 and 4. For measuring the time costs of inference subactivities (i.e., batch data loading and forward propagation), we rerun the inference experiments employing Procedure 3, but only for PyTorch's Python and Rust bindings on GPU.

Running example. In PyTorch's Python binding, we use Procedure 2 to determine the inference time cost for the pre-trained LeNet-1 model. Furthermore, we rerun the inference experiment with additional steps from Procedure 3 to separately record time costs for batch data loading and forward propagation.

4.4.8 Supported features in studied bindings

Table 4.3 outlines the supported features by each studied binding:

- **Training support:** A lack of training support in certain bindings means de-

velopers might have to use another programming language. This can be inconvenient and result in additional overhead, especially if developers are unfamiliar with the alternative language.

- **Model interface support:** When certain model types are not supported in a binding, developers might still need to switch to another language to train their models.

- **Model loading approaches:** Loading models via serialization provides flexibility as developers don't need to define the model structure. In contrast, loading models via parameters requires the model's structure to be pre-defined. This can lead to challenges, especially when developers try to use pre-trained models.

For our training experiments in Section 4.4.7, certain bindings are exempt due to their limitations: TensorFlow's Rust and PyTorch's JavaScript bindings (which don't support training), TensorFlow's C# binding for RNNs, and all bindings for BERT. We acknowledged the recent inclusion of support for RNNs in TensorFlow's C# binding (aligned with TensorFlow v2.10).[15] However, to maintain consistency in our experimental framework, we focused on TensorFlow version 2.5.0 which is the most commonly supported version of TensorFlow by the studied bindings.

For the inference experiments, all bindings are utilized in our work, with the exception of RNNs in TensorFlow's C# and BERT in C# bindings for both ML frameworks. The reason is that the C# bindings can only load models using parameters and lacks support for RNN and BERT interfaces. Unlike PyTorch's JavaScript binding which despite not supporting CNNs, RNNs, and BERT, does offer loading via serialization without the need for defining model structures.

[15] https://github.com/SciSharp/TensorFlow.NET/issues/640

4.5 Correctness Evaluation

Motivation. Developers can use a binding for an ML framework in their preferred programming language to train a DL model. We want to observe if the DL models trained using a binding for a given ML framework have the same training accuracy as the DL models trained using the ML framework's default Python binding (RQ1). These results can help developers understand if using a binding will achieve the same model accuracy during training and provide the same model performance for the final trained models.

In addition, it is important to ascertain if performing inference for these trained models using different bindings for a given framework will impact the accuracy. Pre-trained models have been widely used by the ML community [65, 169] and bindings can help developers to run inference with pre-trained models in different programming languages. Importantly, in high-stakes domains such as medical diagnosis and autonomous driving, accuracy is particularly important when decisions are made by ML systems [121]. Even a slight drop in accuracy can trigger erroneous decisions with serious implications. Hence, it is vital that bindings have the capability to achieve the same accuracy for pre-trained models as with the binding they were trained with. In RQ2, we investigate the cross-binding test accuracy of pre-trained models using the bindings for TensorFlow and PyTorch to understand whether the pre-trained models perform as we would expect them to.

Together, the bindings' impact on training correctness and inference correctness will enable us to understand the impact on the correctness of the ML software quality.

RQ1: How do the studied bindings impact the training accuracy and test accuracy of the studied DL models?

Approach. We employ both dynamic time warping (DTW) [140] for analyzing training accuracy curves and the Mann-Whitney U test [110] for comparing the performance

Table 4.4: Mean/Max DTW distances of training accuracy curves for bindings in training models with the same random seed. (Highlighted numbers indicate negligible DTW distance. Py: Python; JS: JavaScript; Rs: Rust)

(a) TensorFlow (mean/max DTW distance)

Model	Py-C#	Py-JS	JS-C#
LeNet-1	0.005/0.006	**0.000/0.000**	0.005/0.006
LeNet-5	0.003/0.004	**0.000/0.000**	0.003/0.004
VGG-16	0.018/0.019	0.005/0.006	0.018/0.019
LSTM	-	0.008/0.012	-
GRU	-	0.010/0.012	-

(b) PyTorch (mean/max DTW distance)

Model	Py-C#	Py-Rs	Rs-C#
LeNet-1	**0.000/0.000**	**0.000/0.000**	**0.000/0.000**
LeNet-5	**0.000/0.000**	**0.000/0.000**	**0.000/0.000**
VGG-16	0.007/0.010	0.002/0.003	0.008/0.010
LSTM	0.008/0.009	0.009/0.011	0.010/0.011
GRU	0.010/0.011	0.008/0.009	0.009/0.010

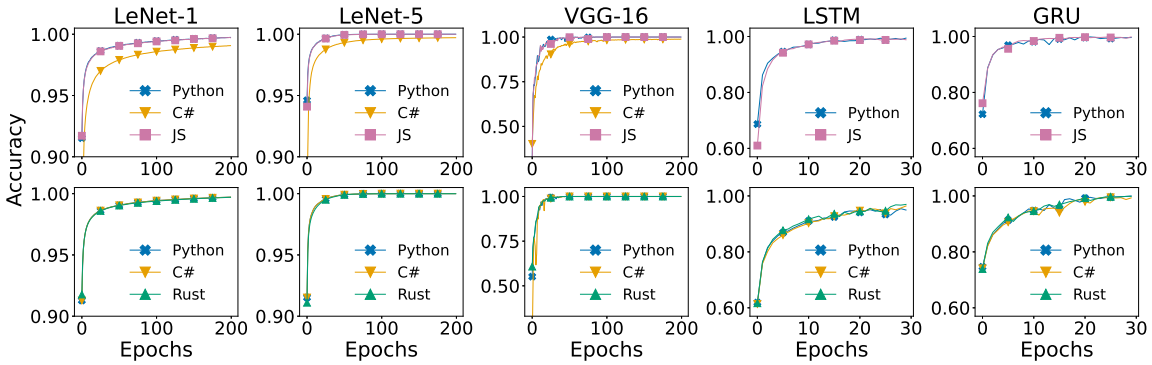


Figure 4.3: Mean training accuracy curves of LeNet-1, LeNet-5, VGG-16, LSTM, and GRU on GPU in bindings for TensorFlow (first row) and PyTorch (second row).

metrics of the final trained models. We chose DTW due to its ability to analyze time-series data, which allows us to investigate whether different bindings follow the same trajectory during training. DTW calculates the distance between the training accuracy curves of the bindings (e.g., between TensorFlow's Python and C# binding) for training the same model. DTW is widely used as a distance measurement for time series data since it can manage time distortion by aligning two time series before computing the distance, which is more accurate than the Euclidean distance [41]. We normalize the calculated DTW distances between 0 to 1 to interpret the results. A normalized DTW distance of 0 means that the difference between the two curves is negligible.

In addition, we calculate the test accuracy, F1-score, and AUC-ROC for the final trained models to compare their classification performance. For each metric, we perform the Mann-Whitney U test [110] separately at a significance level of $\alpha = 0.05$ to determine if the values obtained from different bindings are significantly different. We computed Cliff's delta d [106] effect size to quantify the difference based on the following thresholds [137]:

$$\text{Effect size} = \begin{cases} negligible, & \text{if } |d| \leq 0.147 \\ small, & \text{if } 0.147 < |d| \leq 0.33 \\ medium, & \text{if } 0.33 < |d| \leq 0.474 \\ large, & \text{if } 0.474 < |d| \leq 1 \end{cases} \tag{4.1}$$

Findings. **Bindings can have different training accuracy curves when training DL models under the same configuration (i.e., model structure, training data, hyperparameters, and random seed).** Table 4.4 reports the mean and maximum DTW distances for the training curves between bindings. Moreover, Figure 4.3 presents the mean training accuracy curves of the models (out of the five training processes) that have the best test accuracy after the last epoch. The figure and table show that bindings can have quite different training accuracy curves according to the DTW distance when using the same training configuration. For example, the distances between the curves of TensorFlow's C# binding and the other two bindings are relatively large for LeNet-1, LeNet-5, and VGG-16 models. Another example is that all PyTorch bindings have a relatively large distance between the curves for the RNN models compared to the distances in the CNN models. One reason could be the differential numerical precision across programming languages. For example, Python supports arbitrary-precision arithmetic, while languages like Rust and C# typically operate with fixed precision. These variations in numerical precision might spawn minor differences in mathematical computation outputs. These minor differences might accumulate over numerous iterations during model training, resulting in variations in the final model accuracy. In contrast, bindings can exhibit nearly the same behaviour for training some DL models; the training accuracy curves of the LeNet models differ negligibly between TensorFlow's Python and JavaScript bindings, as well as between PyTorch's bindings.

Table 4.5: The average test accuracy (Acc), F1-score (F1), and AUC-ROC (AUC) for TensorFlow and PyTorch bindings. (Statistically significant differences between bindings are highlighted in bold. MD: Max Diff; ES: Effect Size)

		TensorFlow						PyTorch				
		LN1	LN5	VGG	LSTM	GRU		LN1	LN5	VGG	LSTM	GRU
Acc	Python	98.8	98.9	84.8	83.7	85.0	Python	98.8	98.9	86.2	86.5	87.9
	C#	98.6	98.9	83.8			C#	98.8	99.0	86.2	87.3	85.5
	JavaScript	98.8	99.0	85.6	84.2	84.7	Rust	98.8	98.9	85.6	87.4	87.0
	MD	**0.2**	0.1	**1.9**	0.6	0.3	MD	0.0	0.1	**0.6**	0.8	**2.5**
	p	**0.01**	0.40	**0.01**	0.10	0.15	p	0.68	0.31	**0.03**	0.10	**0.01**
	ES	**large**	-	**large**	-	-	ES	-	-	**large**	-	**large**
F1	Python	98.8	98.9	84.7	83.5	85.0	Python	98.8	99.0	86.3	86.7	87.9
	C#	98.6	98.9	83.8			C#	98.8	99.0	86.1	87.2	85.1
	JavaScript	98.8	99.0	85.6	83.8	84.7	Rust	98.9	98.9	85.6	87.2	86.9
	MD	**0.2**	0.1	**1.9**	0.3	0.3	MD	0.1	**0.1**	**0.7**	0.5	**2.8**
	p	**0.01**	0.42	**0.01**	0.22	0.15	p	0.06	**0.01**	**0.01**	0.10	**0.01**
	ES	**large**	-	**large**	-	-	ES	-	**large**	**large**	-	**large**
AUC	Python	100.0	100.0	98.2	91.7	92.3	Python	100.0	100.0	98.5	94.1	94.3
	C#	100.0	100.0	97.3			C#	100.0	100.0	98.5	94.6	92.9
	JavaScript	100.0	100.0	98.4	92.3	91.9	Rust	100.0	100.0	98.3	94.5	93.8
	MD	0.0	0.0	**1.1**	**0.6**	**0.5**	MD	0.0	0.0	**0.2**	0.5	**1.5**
	p	0.10	0.84	**0.01**	**0.01**	**0.01**	p	0.55	0.42	**0.01**	0.10	**0.01**
	ES	-	-	**large**	**large**	**large**	ES	-	-	**large**	-	**large**

The trained models produced by certain bindings can perform worse than the models produced by other bindings for the same ML framework. Table 4.5 shows the test accuracy, F1-score, and AUC-ROC for the trained models produced by bindings can be different. For the trained VGG-16 models, the Mann-Whitney U test reveals significant differences between bindings for both frameworks in these metrics with large effect sizes. This pattern is also observed in the trained GRU models in PyTorch's bindings. Specifically, while the test accuracy and F1-score of the trained LeNet-1 models have statistically significant differences between bindings for TensorFlow, the AUC-ROC values of LeNet models in TensorFlow and PyTorch bindings are close (all rounded up to 100 in Table 4.5). Furthermore, we observed some models produced by non-Python bindings have higher values of the metrics than the models produced by the default Python bindings, e.g., the VGG-16 model produced by TensorFlow's JavaScript binding.

Summary of RQ1

TensorFlow and PyTorch bindings can have different training accuracy curves for training the same DL models even when using the same configuration. In addition, the test accuracy of the final trained models can be slightly different. Hence, developers should not assume that all bindings offer the same level of correctness and should verify the model's correctness when utilizing a binding for training.

RQ2: How do the studied bindings impact the cross-binding test accuracy of pre-trained models?

Approach. We conducted inference experiments with all bindings using pre-trained models produced by the default Python bindings for TensorFlow and PyTorch (see Figure 4.4). We loaded the pre-trained models using the supported loading approach(es) and recorded the cross-binding test accuracy on both CPU and GPU for each binding. If the cross-binding test accuracy of a pre-trained model in a binding shows a 0% difference compared to the test accuracy when the model was initially

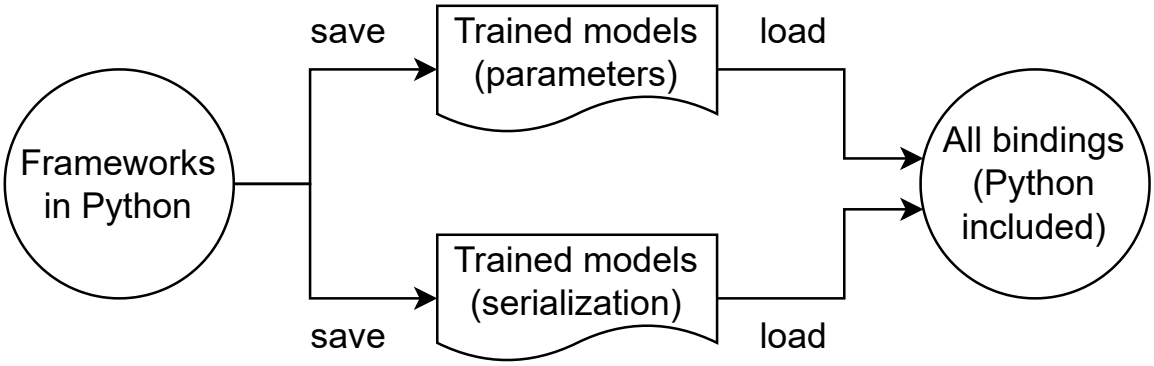


Figure 4.4: All bindings load the trained models that are saved by the default Python bindings for ML frameworks.

trained, we considered the test accuracy "reproduced" by that binding. Any non-zero difference resulted in a "failed" mark. Since some bindings only support one way of loading models (as shown in Table 4.3), we marked the result as "unsupported" if the loading approach is not supported by a binding.

Findings. **The test accuracy of pre-trained models can be reproduced across bindings in different languages for the same ML framework.** Figure 4.5 shows that only PyTorch's C# binding failed to reproduce the test accuracy in the saved VGG-16, LSTM, and GRU models. We noticed that the differences in the test accuracy in these three models are all within 1% and the root cause of the reproduction failure is a bug that results in "eval() and train() methods not being properly propagated to all submodules".[16] This bug prevents setting the model to evaluation mode, hence, the dropout layers of these three models are not disabled which leads to different cross-binding test accuracy. This bug is fixed in version 0.96.0 which does not support PyTorch 1.9.0 but targets version 1.10.0. In other words, the saved models can be reproduced in the newer version of PyTorch's C# binding. For consistency, we still use the 0.93.9 version of this binding for the other experiments.

Bindings can reproduce the test accuracy of pre-trained models via dif-

[16]See https://github.com/dotnet/TorchSharp/pull/501 and https://github.com/dotnet/TorchSharp/issues/500

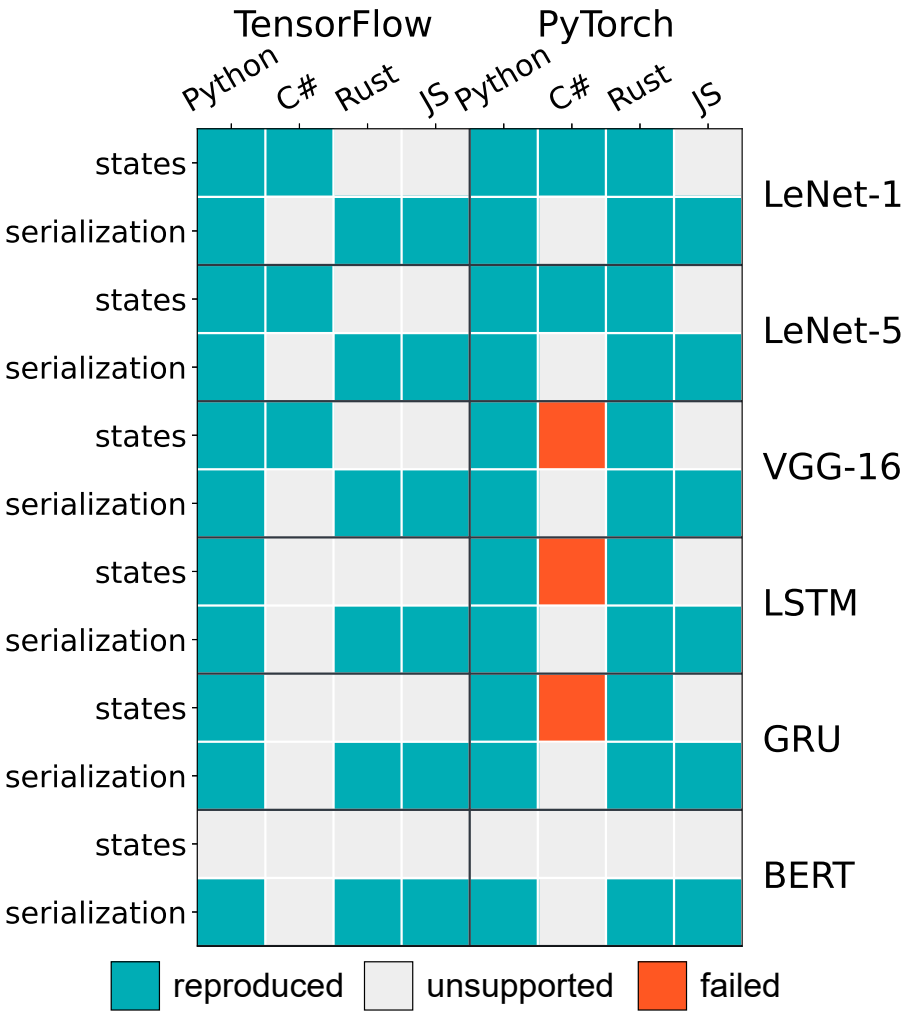


Figure 4.5: Results of reproducing the test accuracy of pre-trained models in TensorFlow and PyTorch bindings on the CPU and GPU (the results are identical). Note: the failed cases in the PyTorch's C# binding were fixed in a newer version of the binding.

ferent loading approaches and on different types of processing units (i.e., CPU and GPU). As shown in Figure 4.5, PyTorch's Python and Rust bindings and TensorFlow's Python binding support both loading via parameters and serialization, and both loading approaches can reproduce the test accuracy of the pre-trained models. In addition, we noticed that bindings can reproduce the test accuracy of pre-trained models on both CPU and GPU.

Summary of RQ2

TensorFlow and PyTorch bindings can perform inference using pre-trained models and reproduce the same test accuracy as when the models were originally trained. This correctness property holds true whether model inference is performed on CPU or GPU. As a result, developers can leverage the capabilities of pre-trained models while still being able to use the model in their preferred language.

4.6 Time Cost Evaluation

Motivation. In RQ1 and RQ2, we studied the impact of bindings for ML frameworks on correctness, however, the impact of bindings on time cost remains unknown. Given the time-consuming nature of model training and model inference for ML frameworks, it is important to investigate how a binding may impact the time cost. Studies show that runtime efficiency and energy consumption can vary across programming languages [118, 126, 128]. Consequently, these differences may have an impact on the time cost of training and inference when using different bindings.

Thus, in RQ3, we study the time cost of training DL models with bindings in order to offer developers more information about the overhead or advantage in terms of time cost when training with a binding. In RQ4, we study the inference time of pre-trained models in bindings. The time of utilizing bindings in model inference can be a crucial consideration for developers since model inference typically takes place (as a part of the product) in the production environment, which may have limited resources. The findings can help developers decide whether or not to utilize a binding for model inference in their project.

RQ3: How do the studied bindings impact the training time of the studied DL models?

Approach. To study the difference in training time across bindings, we performed the Mann-Whitney U test [110] using the Bonferroni correction [145] to adjust the significance level for multiple comparisons. Specifically, for an initial significance

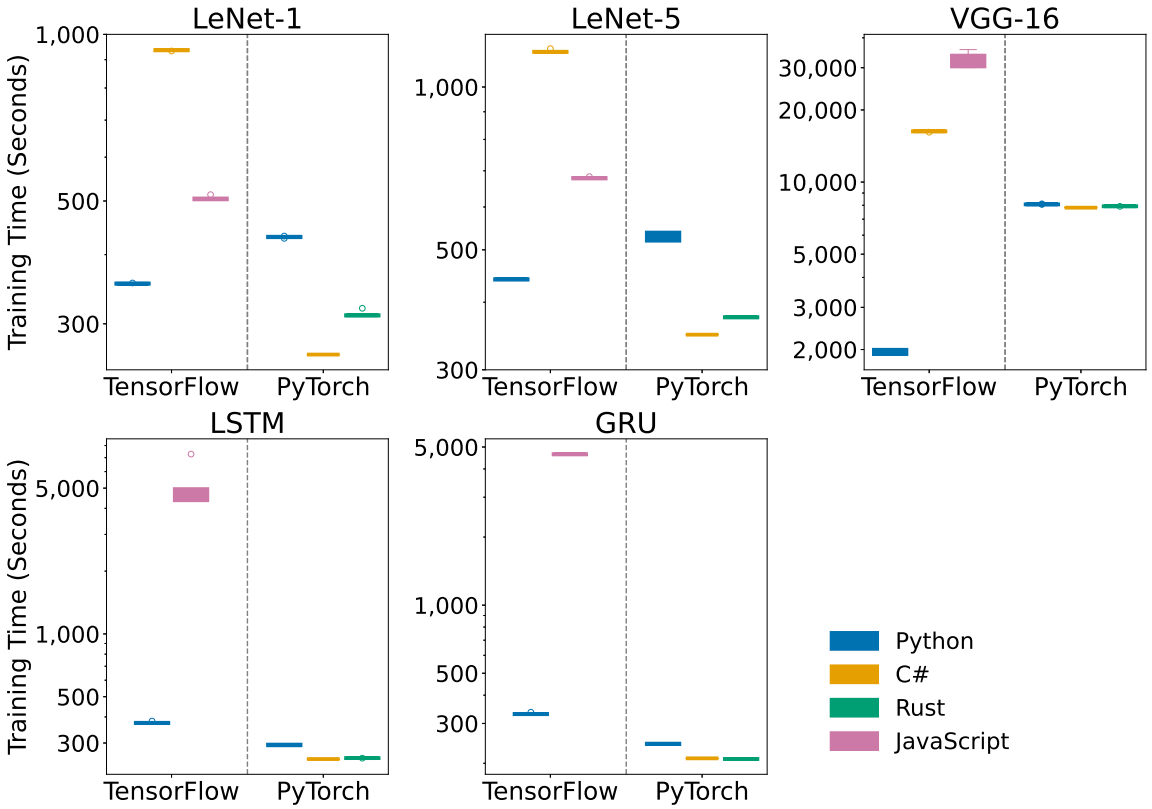


Figure 4.6: Training time distributions when training models in TensorFlow and PyTorch bindings on the GPU.

level of $\alpha = 0.05$, we adjusted the significance level to $\frac{\alpha}{n}$ (where n is the number of comparisons made) to determine whether the distributions of the training times of the default Python bindings and the non-Python bindings, which trained the same model for the same framework, are significantly different. For example, the LeNet-1 model in TensorFlow bindings, we performed Bonferroni-corrected Mann-Whitney U test between the Python and C# bindings and Python and JavaScript bindings with an adjusted significance level of $\frac{\alpha}{2} = 0.025$. We also computed Cliff's delta d [106] effect size to quantify the difference based on Equation 4.1 in Section 4.5.

Findings. **Training times can differ greatly across bindings for the same ML framework.** Figure 4.6 shows the training time distributions on GPU for the studied models across the studied bindings. The Bonferroni-corrected Mann-Whitney U test shows that the training time distributions of the same model are all significantly

Table 4.6: Time costs (in seconds) of the subactivities in the training process using PyTorch's Python and Rust bindings on GPU.

		Load batch data	Forward	Backward	Total
LeNet-1	Python	148.9	76.2	240.2	465.7
	Rust	23.5	69.7	239.2	332.5
LeNet-5	Python	167.4	114.8	293.2	576.4
	Rust	24.1	94.3	278.8	397.4
VGG-16	Python	89.2	7094.0	1557.8	8741.2
	Rust	31.9	6470.4	1469.5	7971.8
LSTM	Python	8.2	95.4	188.8	292.2
	Rust	0.6	83.5	165.3	249.4
GRU	Python	8.5	84.0	151.0	242.8
	Rust	0.6	72.5	130.3	203.5

different between the default Python bindings and the other bindings for the same framework and the effect sizes are all large. In addition, the difference in training time of bindings for the same ML framework can be very large when training certain models. For example, the median training time of TensorFlow's JavaScript binding for the VGG-16 model is 15 times larger than its Python binding (32,783 vs. 1,991 seconds).

PyTorch's default Python binding has the slowest training time for the studied models. Figure 4.6 shows that PyTorch's Python binding is more than two times slower than the other two bindings for training LeNet models. However, we note that the training time difference between PyTorch's Python binding and other bindings for the VGG-16, LSTM, and GRU models is relatively small (less than 15%). In contrast, TensorFlow's default Python binding has the fastest training time in the studied models.

Batch data loading time affects the training cost of PyTorch's Python binding. As shown in Table 4.6, PyTorch's Python binding has a long batch data loading time, which is notably slower (between 4 to 14 times) than the Rust binding for all studied models. Specifically, For LeNet models, the Python binding's batch data loading times account for roughly 30% of the training cost, whereas the Rust binding's batch data loading for the same models consumes less than 10% of the training cost. Furthermore, the Python binding consistently underperforms the Rust binding during both forward and backward propagation phases in the studied models.

The observed variations in batch data loading times between bindings suggest that the native speed of a programming language [118, 126, 128] is an important factor that influences the performance of a binding. However, there could be other factors involved in the implementation of bindings. For example, these factors could include overheads arising from differences in data structure implementations and initialization routines. Additionally, the overhead of the marshalling mechanism [13, 43, 174] implemented to convert data between the binding's programming language and the ML framework could impact efficiency. Finally, the way the binding interacts with the ML framework's lower-level APIs, such as those for memory management and tensor operations, could also play a crucial role in performance differences.

Summary of RQ3

Training times for training the same DL models differ significantly between the default Python bindings and the non-Python bindings for the same ML framework. Surprisingly, non-Python bindings for PyTorch are even faster in training the studied models than the default Python binding. Hence, choosing the right binding can help developers to lower the training time cost for certain models.

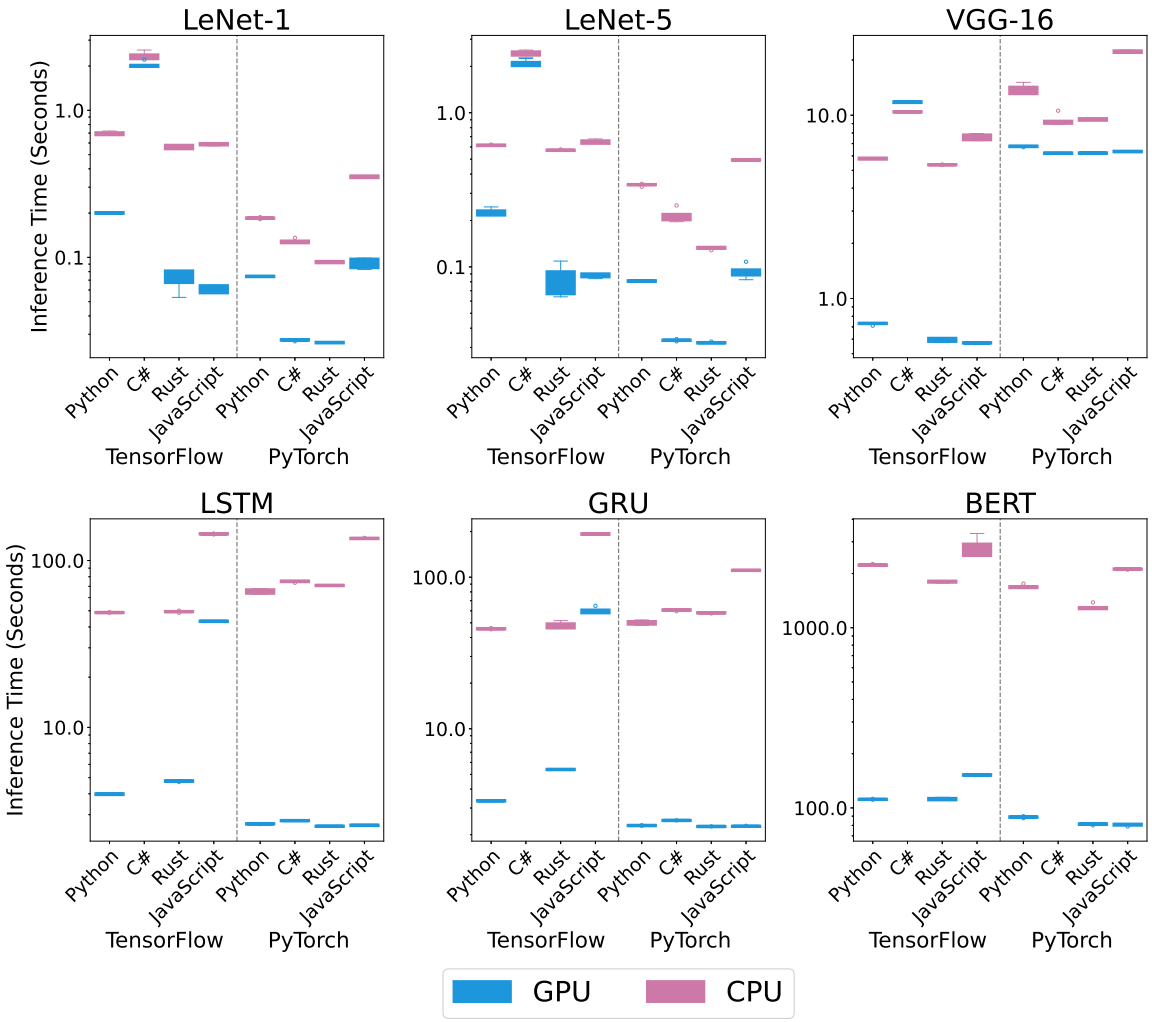


Figure 4.7: Inference time distributions for pre-trained models in TensorFlow (TF) and PyTorch (PT) bindings on the CPU and GPU.

RQ4: How do the studied bindings impact the inference time of pre-trained models?

Approach. We followed the same process as shown in Figure 4.4 and investigated the inference time of each model on both CPU and GPU. We performed the Bonferroni-corrected Mann-Whitney U test on the recorded inference time distributions between the default Python bindings and the non-Python bindings, grouped by the same framework, model, and processing unit (CPU or GPU). We also computed Cliff's Delta effect size as described in RQ3.

Findings. **The inference time of the same pre-trained model differs greatly between the default Python bindings and the other bindings for the same ML framework.** Figure 4.7 shows the distributions of the inference time of the pre-trained models in the studied bindings. The results of the Bonferroni-corrected Mann-Whitney U test and Cliff's Delta d show that the Python and non-Python bindings for the same ML framework have significantly different inference times for the same model on the same processing unit (i.e., CPU and GPU) and the effect size is large, except for the TensorFlow bindings for LSTM on CPU and for BERT on GPU where the Python binding has similar inference time costs as the Rust binding. We observed that the default Python bindings for TensorFlow and PyTorch do not always offer the best inference time for all studied pre-trained models, with Rust bindings often outperforming them. On the other hand, TensorFlow's C# binding has the worst performance for the studied models on both CPU and GPU, and PyTorch's JavaScript binding has the worst performance on CPU. Moreover, the performance gap in model inference time can be very large, for example, TensorFlow's Python binding is 17 times as fast as the JavaScript binding for the GRU model on the GPU (3.35 vs. 58.32 seconds).

Inference time differences in PyTorch arise from both batch data loading and forward propagation speed. Table 4.7 shows that the majority of the inference cost is allocated towards forward propagation and the Rust binding outperforms the Python binding in this regard. As we observed the same pattern in RQ3, the Rust binding also demonstrates faster batch data loading times compared to the Python binding across all studied models. Although both bindings leverage PyTorch's computational core, which is written in C/C++ and predominantly runs computations on GPUs, the variations in time costs can be attributed to overheads introduced by the bindings themselves.

Certain bindings on the CPU may have a faster inference time than other bindings on the GPU for the same pre-trained model. Generally,

Table 4.7: Time costs (in seconds) of the subactivities in the inference process using PyTorch's Python and Rust bindings on GPU.

		Load batch data	Forward	Total
LeNet-1	Python	0.06	0.02	0.08
	Rust	0.01	0.02	0.03
LeNet-5	Python	0.06	0.03	0.08
	Rust	0.01	0.02	0.03
VGG-16	Python	0.12	6.74	6.86
	Rust	0.03	6.26	6.29
LSTM	Python	0.15	2.61	2.76
	Rust	0.01	2.57	2.58
GRU	Python	0.15	2.27	2.41
	Rust	0.03	2.25	2.28
BERT	Python	0.13	88.86	88.99
	Rust	0.12	81.82	81.94

inference time for pre-trained models on GPU outperforms CPU in bindings for both studied frameworks (as shown in Figure 4.7). However, we found that for the same framework, one binding that runs inference on CPU can outperform another binding that runs on GPU for the same pre-trained model. For example, the Rust binding for TensorFlow is faster on CPU than the C# binding on GPU for LeNet and VGG-16 models, as well as faster on CPU than the JavaScript binding on GPU for GRU model. Furthermore, we noticed that TensorFlow's C# binding in model inference on CPU is similar to or even faster than on GPU. According to the maintainer of the C# binding, the reason could be that "there is I/O cost underlying"[17] model inference on GPU.

[17]https://github.com/SciSharp/TensorFlow.NET/issues/876

Certain bindings lack support for certain features which leads to a slower inference time. We noticed that TensorFlow's JavaScript binding cannot load a GRU model with "reset_after=True"[18], either by loading parameters or through serialization. However, "reset_after=True" is the default setting in the framework (and other bindings) to enable the "fast cuDNN implementation", which speeds up the inference of the GRU model[19] This unsupported feature can be one of the reasons behind the large increase of GRU inference time in TensorFlow's JavaScript binding (256.5 seconds) compared to the inference time of the default Python binding (3.6 seconds).

Summary of RQ4

TensorFlow and PyTorch bindings have various inference times for the same pre-trained models on CPU and GPU. Remarkably, the inference time of certain models in bindings on the CPU can be faster than other bindings for the same framework on GPU. Therefore, developers can experiment and choose the fastest binding for their usage scenario.

4.7 Implications

4.7.1 Implications for developers

Developers are not limited to writing their projects in Python when using an ML framework. Although Python dominates the development in ML [7, 119], developers can also use bindings in other programming languages. Our results in Section 4.5 shows that non-default bindings for TensorFlow and PyTorch can have the same inference accuracy of a pre-trained model as the default Python binding and sometimes even faster performance. We recommend developers use the binding in their preferred programming language for either model training or inference if supported by the binding. Hence, developers can save time and effort when adopting ML techniques in their projects without having to settle for non-mature ML frameworks that might be available in the language that their current software is programmed in.

[18] https://github.com/tensorflow/tfjs/issues/4621
[19] https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU

For instance, in Integration Scenario 1 of Section 4.2, Anna can use the JavaScript binding to perform inference with pre-trained models provided by the ML team.

Developers can use a binding for an ML framework which has a shorter training time for a certain model and perform inference on the trained model in another binding that has a shorter inference time based on task and requirements. Bindings for an ML framework have various training times and inference times for ML models (Section 4.6). Hence, developers can choose different bindings which are faster for a certain model in training and inference respectively since the accuracy of pre-trained models can be reproduced across bindings for the same framework (Section 4.5). We suggest that developers refer to an existing benchmark like ours or conduct experiments themselves based on our replication package [100]. For example, when using TensorFlow for LeNet models as described in Integration Scenario 3 of Section 4.2, Anna can train the models using the default Python binding for TensorFlow and then run inference for the trained model in the Rust binding with the assistance of a hired expert to save time and computational resources, as this factor is critical in their project requirements.

Developers should perform a sanity check before using a model that was trained by a binding other than the default Python binding. Bindings corresponding to different languages can have different training accuracy curves while training the same model, and the final trained model can behave differently (as discussed in Section 4.5). Since the Python bindings are the default binding for most ML frameworks, these Python bindings have a larger user base and better support than other bindings. We suggest that developers perform a sanity check on the trained model if they are using a binding other than the default Python binding before deploying the models to the production environment.

In resource-limited scenarios (e.g., CPU only), developers may prefer or need to use a non-default binding for model inference. Traditionally, model inference is done using a GPU due to the superior inference time of GPUs [14,

96]. However, GPUs are expensive and not available in all scenarios. We found that the bindings for ML frameworks can be fast for running inference on CPU for some pre-trained models (Section 4.6). Developers can use such bindings if the production environment does not contain a GPU or the computational resource is limited. For example, in Integration Scenario 3 of Section 4.2, if Anna is using PyTorch for LeNet models and there is no GPU available in the production environment, she can use PyTorch's Rust binding on CPU with expert assistance. The inference time of LeNet models in the Rust binding on CPU is faster than the default Python binding both on CPU and GPU. This is particularly beneficial for constrained environments like the Internet-of-Things (IoT) devices (e.g., unmanned aerial vehicles) where resource availability is often limited [3, 48, 80, 139].

4.7.2 Implications for binding owners

Binding owners should include performance benchmarks for their binding. We found that bindings can have very different training and inference times for ML models (Section 4.6), yet this information is not well documented. To address this, we suggest that binding owners introduce performance benchmarks of training and running inference for some frequently used ML models (e.g., VGG models) and record the results in their documentation. This way, developers be aware of the trade-off between choosing a familiar language and the potential impact on time cost for various DL models. For example, the performance benchmarks can help Anna in Integration Scenario 2 of Section 4.2 to make informed decisions when choosing a familiar language for training while considering the potential impact on time cost.

4.7.3 Implications for researchers

Researchers should investigate the impact of ML framework bindings on large-scale models and datasets. Our findings provide a starting point, but further research is needed to fully understand how binding choices influence performance

in large-scale models. While full-parameter fine-tuning can be computationally expensive, parameter-efficient techniques like Low-Rank Adaptation (LoRA) [68] offer a cost-effective alternative. However, LoRA's experimental status in HuggingFace[20] and its lack of binding support highlight a direction for further research. We suggest future research adopt our methodology (see our replication package [100]), starting with representative data subsets and smaller model variants (e.g., the 7 billion parameter variant of Llama 2 [160]). This approach could provide valuable early insights into potential performance variations before committing to full-scale experiments.

Researchers should investigate methods to enhance the interoperability and compatibility of pre-trained models across different bindings for ML frameworks. Our findings demonstrate that pre-trained models can be used across different bindings for the same ML framework with the same level of accuracy (as shown in Section 4.5). However, some models may not be supported or may have a slower inference time when utilizing certain bindings (as discussed in Section 4.6). While developers and binding owners focus on the implementation of bindings, we suggest researchers explore ways to contribute at a higher level: by devising algorithms, methodologies, or protocols to increase the interoperability and compatibility of pre-trained models across different bindings, benefiting a diverse developer base.

Researchers should study the patterns and origins of bugs in bindings for ML frameworks. We found that bugs in bindings for ML frameworks have an impact on the model inference correctness (Section 4.5). While the immediate resolution of bugs in bindings is an engineering concern, a deeper analysis of these issues can provide invaluable insights into software design and testing paradigms for bindings. Although researchers have previously studied bugs in ML frameworks [18, 78, 79], there has been no research specifically on bugs in the bindings for ML frameworks or other libraries. We encourage researchers to systematically analyze the bugs in bindings and provide guidelines for maintainers to avoid introducing such bugs.

[20]https://huggingface.co/docs/diffusers/en/training/lora

4.8 Related work

4.8.1 Impact of ML frameworks on ML software correctness

Researchers have studied the correctness of ML frameworks. However, no one has studied how bindings for those frameworks impact the correctness of the ML software that is created with them. The study by Guo et al. [61] is the closest related to our work. However, even though they included several bindings in their study, their work differs from ours as they focus on the impact on ML software quality of using different ML frameworks and executing ML models on different computing devices (such as PC and various types of mobile devices). In contrast, we run our experiments on the same device but we study the impact of various bindings on ML software quality. Hence, we can reason about the impact of using a binding, while in Guo et al.'s study, the different devices make this impossible.

Several others have focused on comparing the accuracy of the same model across ML frameworks. Chirodea et al. [20] compared a CNN model that was built with TensorFlow and PyTorch and found that these two frameworks have similar training curves but the final trained model has a lower accuracy in PyTorch. Gevorkyan et al. [55] gave an overview of five ML frameworks and compared the accuracy of training a neural network for the MNIST dataset. They reported that the final trained model has a lower accuracy in TensorFlow than in other frameworks. Moreover, Elshawi et al. [44] conducted training experiments for six ML frameworks by using the default configuration and reported that certain frameworks have better performance than the other frameworks on the same model (e.g., Chainer on the LSTM model).

4.8.2 Impact of ML frameworks on ML software time cost

Many studies have compared the time cost across ML frameworks. In a comparison of the training and inference time for a CNN architecture using PyTorch and TensorFlow, Chirodea et al. [20] found that PyTorch is faster in both model training and

inference than TensorFlow. However, Gevorkyan et al. [55] showed that PyTorch has the worst training time for neural networks among five studied ML frameworks. In our work, we compared the training and inference time across bindings for the same ML frameworks.

Several studies have focused on the time cost of ML frameworks on different hardware devices. Buber and Diri [14] compared the running time of DL models on CPU and GPU and found that GPU is faster. Jain et al. [76] focused on the performance of training DNN models on CPU with TensorFlow and PyTorch. They show that multi-processing provides better training performance when using a single-node CPU. For mobile and embedded devices, Luo et al. [107] introduced a benchmark suite to evaluate the inference time cost based on six different neural networks.

4.8.3 Impact of ML frameworks on ML software reproducibility

Reproducibility has become a challenge in ML research [60, 112, 159]. Liu et al. [103] surveyed 141 published ML papers and conducted experiments for four ML models. The results showed that most studies do not provide a replication package and the models are highly sensitive to the size of test data. In addition, Isdahl and Gundersen [74] introduced a framework to evaluate the support of reproducing experiments in ML platforms and found that the platforms which have the most users have a relatively lower score in reproducibility. In this chapter, we studied the reproducibility of pre-trained models across different bindings for the same ML framework.

To improve the reproducibility of ML models, many researchers have conducted studies to understand and resolve non-deterministic factors in ML software. Pham et al. [127] studied nondeterminism-introducing-factors in ML frameworks (e.g., weight initialization and parallel processes) and found that these factors can cause a 10% accuracy difference in ML models. To improve the reproducibility of ML models, Chen et al. [17] suggested using patching to minimize nondeterminism in hardware

and proposed a record-and-reply approach to eliminate randomness in software. In addition, they provided guidelines for producing a reproducible ML model. Nagarajan et al. [116] studied deterministic implementation for deep reinforcement learning and proposed a deterministic implementation of deep Q-learning by identifying and controlling five common sources of nondeterminism.

4.8.4 Empirical Studies of ML Frameworks

Many empirical studies of ML frameworks exist that study software quality aspects such as software bugs [18, 78, 79], technical debt [104, 143], and programming issues [70, 75, 180]. However, no prior work has investigated the impact of bindings for ML frameworks on the ML software quality.

Many studies have focused on the bugs of ML frameworks. Jia et al. [78, 79] investigated TensorFlow's GitHub repository and identified six symptoms and eleven root causes of bugs in TensorFlow. In addition, they found that most bugs are related to interfaces and algorithms. Chen et al. [18] studied bugs from four ML frameworks and investigated the testing techniques in these frameworks. They showed that the most common root cause of the bugs is the incorrect implementation of algorithms, and the current testing techniques have a low percentage of test coverage.

ML software has ML-specific technical debts such as unstable data dependence, hidden feedback loop, and model configuration debts [143]. This technical debt can hurt the maintainability of ML systems and introduce extra costs. Liu et al. [104] analyzed self-admitted technical debt in 7 DL frameworks and concluded that technical debt is common in DL frameworks, although application developers are often unaware of its presence.

Researchers have also aimed to understand the ML frameworks from a developer perspective to study the programming issues when using an ML framework. They typically researched the questions and answers (Q&As) of developers about ML frameworks on Stack Overflow (SO). Zhang et al. [180] investigated Q&As which are related

to TensorFlow, PyTorch and Deeplearning4j on SO and reported that model migration is one of the most frequently asked questions. Humbatova et al. [70] studied Q&As of these three ML frameworks on SO as well and included interviews with developers and researchers to build a taxonomy of faults in ML systems. Islam et al. [75] mined Q&As about ten ML frameworks on SO and reported that developers need both static and dynamic analysis tools to help fix errors.

4.8.5 FFIs and Bindings in Software Engineering

FFIs and language bindings are instrumental in software engineering, serving as bridges that enable different programming languages to collaborate seamlessly. These bridges often enable developers to develop applications in their language of choice while simultaneously using mature libraries that are developed in another language. The existing body of work predominantly proposes approaches to design and improve such bindings and FFIs within one specific language. For instance, Yallop et al. [174] conducted experiments to create bindings for using the *ctypes* library in OCaml. Their study differentiated the performance of dynamic and static bindings, revealing that static bindings could be between 10 to 65 times faster than their dynamic counterparts. This finding aligns with our investigation into the time costs associated with diverse ML software bindings.

Researchers also proposed several approaches to FFIs. For instance, Bruni et al. [13] introduced an FFI approach called NativeBoost. This approach requires minimal virtual machine modifications and generates native code directly at the language level. They compared the time cost of different FFIs and the results show that NativeBoost is competitive. Ekblad et al. [43] presented an FFI tailored for web-targeting Haskell dialects, emphasizing simplicity and automated marshalling. The authors compare their FFI with the vanilla FFI, which is based on C calling conventions, and show that their FFI has some advantages in terms of simplicity and expressiveness, safety, without introducing excessive performance (i.e., time cost) overhead.

In addition, Ravitch et al. [134] automated the generation of library bindings using static analysis, aiming to simplify the often laborious manual creation process. Their method not only refined the automated binding generation but also unveiled type bugs in manually created bindings, highlighting potential threats to software correctness. Meanwhile, Grimmer [59] explored high-performance language interoperability in multi-language runtimes. Their approach leveraged just-in-time (JIT) compilers to optimize across language borders, enhancing the efficiency of cross-language operations.

To the best of our knowledge, our study is the first to systematically investigate the impact of using different language bindings on ML software quality. While Ravitch et al. [134] touched upon type correctness in bindings, the unique challenges posed by the inherently non-deterministic nature of ML software remain under-explored. Our work stands out as we specifically evaluate the impact of bindings on the correctness of ML software for model training and inference across different languages. In addition, The computationally intensive nature of ML software introduces unique challenges when assessing time costs, especially when relying on GPUs. While time cost is a widely used metric in the domain of FFIs and bindings, existing works do not explore its significance within the context of ML frameworks. Our research actively fills this void, presenting a comprehensive analysis of time costs associated with different bindings in ML software on CPUs and GPUs.

4.9 Threats to Validity

4.9.1 Construct validity

We use the accuracy metric to assess the correctness of TensorFlow and PyTorch bindings on model training and inference since it is a widely used metric among researchers and developers [20, 44, 55, 61, 107]. However, other metrics may also be used to assess correctness and use of other metrics could potentially change our

results. For evaluating the time cost of bindings on model training, we ran training experiments on the GPU since training DL models on CPU is time-consuming and developers usually train DL models on GPU. The results might be different from those obtained by measuring the time cost on CPU.

4.9.2 Internal validity

When implementing the studied models in TensorFlow and PyTorch bindings, we used the same/similar interfaces to ensure that the structures of these models are consistent across bindings. However, bindings might have different implementations for these interfaces (or have hidden bugs) that result in different structures in the built models. We saved the built models in bindings (via parameters or serialization) and loaded them back into the default Python bindings for TensorFlow and PyTorch to examine whether the structures were the same. The verification results confirm that the produced models in bindings have the same structures.

TensorFlow's JavaScript binding does not support training and inference for GRU with "reset_after=True". Hence, we set "reset_after=False" in the training experiment of TensorFlow's JavaScript binding for GRU and performed inference with a GRU model that was trained with "reset_after=False" in the default Python binding. This setup differs from other bindings, although it has no effect on the model's structure. We compared the results from the JavaScript binding to the results in the Python binding using "reset_after=False", and our findings still hold. Future studies should investigate how one can automatically confirm that the configurations of the bindings are exactly the same.

4.9.3 External validity

We focused on TensorFlow and PyTorch bindings in our work and the results of our study might not apply directly to other ML frameworks. One reason could be that other ML frameworks could have a different implementation and do not provide GPU

support. Furthermore, the findings of our investigation may not be able to generalize to other models and datasets. Future studies should leverage our methodology to analyze bindings for other ML frameworks using different models and datasets.

Our analysis focused on small to medium-sized models that are widely adopted in real-world applications. However, the implications for large-scale models, particularly frontier ML models with billions or trillions of parameters, require further investigation. Future research should build on our work to examine how the observed differences might persist or change at this extreme scale.

4.10 Conclusion

In this chapter, we investigate the impact on ML software quality (correctness and time cost) of using bindings for ML frameworks for DL model training and inference. We conducted model training and model inference experiments on three CNN-based models and two RNN-based models in TensorFlow and PyTorch bindings written in four different programming languages. The most important findings of our study are:

- When training models, bindings for ML frameworks can have various training accuracy curves and slightly different test accuracy values for the trained models.
- Bindings have different training times for the same model, and the default Python bindings for ML frameworks may not have the fastest training time.
- Bindings for ML frameworks have the capabilities to reproduce the accuracy of pre-trained models for inference.
- Bindings for ML frameworks have different inference times for the same pre-trained model and certain models in bindings on the CPU can outperform other bindings on the GPU.

Our findings show that developers can utilize a binding to speed up the training time for an ML model. For pre-trained models, developers can perform inference in their favoured programming language without sacrificing accuracy, or they can choose a binding that has better inference time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The rapid growth of ML has driven developers from many domains to integrate ML capabilities into their applications across different programming languages. However, the complexity of most ML libraries necessitates extensive expertise for development and maintenance, making recoding the libraries from scratch in different programming languages infeasible. Bindings have emerged as a practical solution, allowing developers to reuse the functionality of existing ML libraries across different programming languages without recoding. In this thesis, we conducted three studies to investigate the maintenance quality and software quality of bindings for ML libraries in software package ecosystems.

In the first research study (Chapter 2), we introduced an approach for automatically identifying bindings and their associated host library names. This enabled a comprehensive analysis of bindings for ML libraries within software ecosystems, revealing challenges such as technical lag and inadequate release coverage. In the second research study (Chapter 3), we examined how ML library bindings utilize release-level deprecation mechanisms compared to general packages in software package ecosystems. The study provided insights into the rationale behind deprecation practices within these bindings and proposed enhancements for deprecation mechanisms in software ecosystems. In our final research study (Chapter 4), we assessed

the impact of bindings on the software quality (specifically correctness and time efficiency) of TensorFlow and PyTorch. These experiments highlighted the potential of utilizing bindings to optimize ML systems and emphasized the importance of thorough evaluation for these bindings.

We outline the methodologies employed, major findings, and significant contributions from each study as follows:

- In Chapter 2, we introduced `BindFind` for identifying bindings and extracting host library names within software package ecosystems. Using `BindFind`, we found that bindings are prevalent across ecosystems, with most being maintained by the community rather than the official library organizations. However, we discovered that bindings often cover only a limited range of their host library's releases and experience significant delays in supporting new releases, leading to widespread technical lag. Our findings revealed the availability and quality of these bindings across different ecosystems. Our findings suggest that while bindings are prevalent and mostly community-maintained, they often fail to promptly support new releases of their host libraries, highlighting a crucial gap in their lifecycle management.

- In Chapter 3, we investigated the prevalence and rationale behind release-level deprecation in bindings for ML libraries, in comparison to general packages in the Cargo and npm ecosystems. We found that bindings in Cargo have a higher deprecation rate compared to general packages, while the deprecation rate in npm is similar. The primary reasons for deprecation include package removal or replacement and defects. We also identified the issue of implicitly deprecated releases in Cargo due to the propagation of deprecation through the dependency network. These findings advocate for enhancements in package management systems, including the introduction of a package-level deprecation mechanism and mechanisms for specifying reasons for deprecation, to improve

transparency and governance.

- In Chapter 4, we conducted training and inference experiments to assess the correctness and time cost of using bindings for TensorFlow and PyTorch across four programming languages. Our findings indicated that models trained with one binding could be utilized for inference with another without compromising accuracy. Furthermore, non-default bindings can outperform the default Python bindings in certain tasks while maintaining accuracy. We also found significant differences in inference times across bindings, with some CPU inferences being faster than GPU inferences in other bindings. These results illustrate the benefits of selecting appropriate bindings based on specific performance requirements (e.g., computational speed and hardware specification), thereby aiding developers in making informed choices to maximize efficiency in ML projects.

5.2 Future Work

This thesis provided a comprehensive analysis of bindings for ML libraries in software package ecosystems, offering valuable insights and suggesting practical approaches. Nonetheless, there remain several promising avenues for future research:

- **Explore bindings across software domains using `BindFind`.** Chapter 2 demonstrated the effectiveness of `BindFind` in identifying bindings. Future research should leverage `BindFind` in different domains such as web development frameworks. This expansion could unveil how bindings enhance the interoperability of software libraries across different domains. Our replication package [98] provides a dataset of 250,668 bindings and their host names, which serves as a foundation for such investigation.

- **Investigate the differences between officially-maintained and community-maintained bindings.** Chapter 2 highlighted differences in coverage, delays,

and technical lags between community-maintained and officially-maintained bindings. Future research should explore the underlying reasons behind these differences and examine strategies to improve communication between developers of community-maintained bindings and the host library.

- **Design automatic matching tools for ecosystem binding releases.** In Chapter 2, we found that identifying which host library release is supported by an ecosystem binding is a complex and tedious task. Future work should develop automatic version matching tools for bindings, enabling developers to find a suitable release without manually examining all the related files of a binding.

- **Design automatic semantic versioning guarantee checkers.** Chapter 3 indicated that *Breaking SemVer* is the most common rationale behind yanking for general packages, highlighting the difficulty for package owners to determine whether an update adheres to the guarantee. Future work should develop tools to analyze code for automatically verifying semantic versioning guarantees before publishing updates. Such tools could also facilitate investigating how packages in different software ecosystems adhere to the semantic versioning guarantee.

- **Evaluate the impact of ML library bindings on large-scale models and datasets.** While the findings of Chapter 4 provided a starting point, further research is needed to fully understand how binding choices influence performance in large-scale models. We suggest future studies adopt our methodology (available in our replication package [100]), starting with smaller datasets and model variants to obtain preliminary insights before scaling up the experiments.

- **Explore the interoperability and compatibility of pre-trained models across different bindings for ML libraries.** Although Chapter 4 demon-

strated that pre-trained models maintain accuracy across various bindings, issues such as limited support and reduced inference speeds persist. Future research should explore new algorithms, methodologies, or protocols that could improve the interoperability and compatibility of pre-trained models across different bindings, thus broadening their utility for developers.

- **Investigate the patterns and origins of bugs in bindings for ML libraries.** In Chapter 4, we found that bugs in bindings for ML libraries impact model inference correctness. Future research should conduct a systematic investigation into the nature and origins of these bugs, potentially establishing guidelines for maintainers of the bindings.

Bibliography

[1] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, USENIX Association, 2016, pp. 265–283, ISBN: 9781931971331. DOI: 10.5555/3026877.3026899.

[2] S. S. Abdullahi *et al.*, "Deep sequence models for text classification tasks," in *International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 2021, pp. 1–6. DOI: 10.1109/ICECCE52056.2021.9514261.

[3] A. Albanese, M. Nardello, and D. Brunelli, "Low-power deep learning edge computing platform for resource constrained lightweight compact uavs," *Sustainable Computing: Informatics and Systems*, vol. 34, p. 100 725, 2022, ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2022.100725.

[4] B. van Amsterdam, M. J. Clarkson, and D. Stoyanov, "Gesture recognition in robotic surgery: A review," *IEEE Transactions on Biomedical Engineering*, vol. 68, no. 6, pp. 2021–2035, 2021. DOI: 10.1109/TBME.2021.3054828.

[5] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2019, pp. 1–12. DOI: 10.1109/ESEM.2019.8870181.

[6] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of Apache," in *2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13, IEEE Computer Society, 2013, pp. 280–289. DOI: 10.1109/ICSM.2013.39.

[7] H. Ben Braiek, F. Khomh, and B. Adams, "The Open-Closed Principle of Modern Machine Learning Frameworks," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Association for Computing Machinery, 2018, pp. 353–363, ISBN: 9781450357166. DOI: 10.1145/3196398.3196445.

[8] K. Blincoe, F. Harrison, and D. Damian, "Ecosystems in GitHub and a method for ecosystem identification using reference coupling," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, IEEE Press, 2015, pp. 202–207, ISBN: 9780769555942. DOI: 10.5555/2820518.2820544.

[9] K. Blincoe, F. Harrison, N. Kaur, and D. Damian, "Reference coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems," *Information and Software Technology*, vol. 110, pp. 174–189, 2019. DOI: 10.1016/j.infsof.2019.03.005.

[10] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Association for Computing Machinery, 2016, pp. 109–120, ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950325.

[11] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, 2021. DOI: 10.1145/3447245.

[12] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344. DOI: 10.1109/ICSME.2016.31.

[13] C. Bruni, S. Ducasse, I. Stasenko, and L. Fabresse, "Language-side foreign function interfaces with nativeboost," in *International Workshop on Smalltalk Technologies*, 2013.

[14] E. Buber and B. Diri, "Performance analysis and CPU vs GPU comparison for deep learning," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[15] N. Cerpa and J. M. Verner, "Why did your project fail?" *Commun. ACM*, vol. 52, no. 12, pp. 130–134, 2009, ISSN: 0001-0782. DOI: 10.1145/1610252.1610286.

[16] S. Chacon and B. Straub, *Pro Git*, 2nd. Apress, 2014, ISBN: 1484200772. DOI: 10.1007/978-1-4842-0076-6.

[17] B. Chen, M. Wen, Y. Shi, D. Lin, G. K. Rajbahadur, and Z. M. J. Jiang, "Towards training reproducible deep learning models," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Association for Computing Machinery, 2022, pp. 2202–2214, ISBN: 9781450392211. DOI: 10.1145/3510003.3510163.

[18] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, 2023, ISSN: 1049-331X. DOI: 10.1145/3587155. [Online]. Available: https://doi.org/10.1145/3587155.

[19] S. Chiba, "Foreign language interfaces by code migration," in *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2019, Association for Computing Machinery, 2019, pp. 1–13, ISBN: 978-1-4503-6980-0. DOI: 10.1145/3357765.3359521.

[20] M. C. Chirodea, O. C. Novac, C. M. Novac, N. Bizon, M. Oproescu, and C. E. Gordan, "Comparison of Tensorflow and PyTorch in convolutional neural network-based applications," in *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2021, pp. 1–6. DOI: 10.1109/ECAI52376.2021.9515098.

[21] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, Association for Computational Linguistics, 2014, pp. 103–111. DOI: 10.3115/v1/W14-4012.

[22] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Association for Computing Machinery, 2022, pp. 946–957, ISBN: 9781450392211. DOI: 10.1145/3510003.3510042.

[23] M. Claes, T. Mens, and P. Grosjean, "maintaineR: A web-based dashboard for maintainers of CRAN packages," in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 597–600. DOI: 10.1109/ICSME.2014.104.

[24] M. Claes, T. Mens, and P. Grosjean, "On the maintainability of CRAN packages," in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 308–312. DOI: 10.1109/CSMR-WCRE.2014.6747183.

[25] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ACM, 2017, pp. 186–196, ISBN: 9781450351058. DOI: 10.1145/3106237.3106246.

[26] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, "Identifying unmaintained projects in GitHub," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18, ACM, 2018, ISBN: 9781450358231. DOI: 10.1145/3239235.3240501.

[27] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2457–2470, 2021. DOI: 10.1109/TSE.2019.2952130.

[28] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08, Association for Computing Machinery, 2008, pp. 160–167, ISBN: 9781605582054. DOI: 10.1145/1390156.1390177.

[29] E. Constantinou, A. Decan, and T. Mens, "Breaking the borders: An investigation of cross-ecosystem software packages," in *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop, Delft, the Netherlands, December 10th - to - 11th, 2018*, G. Gousios and J. Hejderup, Eds., ser. CEUR Workshop Proceedings, vol. 2361, CEUR-WS.org, 2018, pp. 1–5.

[30] E. Constantinou and T. Mens, "An empirical comparison of developer retention in the RubyGems and npm software ecosystems," *Innovations in Systems and Software Engineering*, vol. 13, no. 2, pp. 101–115, 2017. DOI: 10.1007/s11334-017-0303-4.

[31] E. Constantinou and T. Mens, "Socio-technical evolution of the Ruby ecosystem in GitHub," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 34–44. DOI: 10.1109/SANER.2017.7884607.

[32] D. R. Cox and A. Stuart, "Some quick sign tests for trend in location and dispersion," *Biometrika*, vol. 42, no. 1/2, pp. 80–95, 1955.

[33] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: An analysis of inter-repository package dependency problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 493–504. DOI: 10.1109/SANER.2016.12.

[34] A. Decan and T. Mens, "How Magic Is Zero? An Empirical Analysis of Initial Development Releases in Three Software Package Distributions," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20, Association for Computing Machinery, 2020, pp. 695–702, ISBN: 978-1-4503-7963-2. DOI: 10.1145/3387940.3392205.

[35] A. Decan and T. Mens, *Lost in zero space – an empirical comparison of 0.y.z releases in software package distributions*, 2021. arXiv: 2101.00836 [cs.SE].

[36] A. Decan and T. Mens, "What Do Package Dependencies Tell Us About Semantic Versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2021, ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2918315.

[37] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Association for Computing Machinery, 2018, pp. 181–191, ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196401.

[38] A. Decan, T. Mens, and P. Grosjean, "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Softw. Engg.*, vol. 24, no. 1, pp. 381–416, 2019, ISSN: 1382-3256. DOI: 10.1007/s10664-017-9589-y.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423.

[40] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, 2021, ISSN: 1049-331X. DOI: 10.1145/3453478.

[41] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: Experimental comparison of representations and distance measures," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1542–1552, 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454226.

[42] D. Eddelbuettel and J. J. Balamuta, *Rcppmlpack*, Accessed: 13 March 2024, 2020. [Online]. Available: https://github.com/rcppmlpack/RcppMLPACK1.

[43] A. Ekblad, "Foreign exchange at low, low rates a lightweight ffi for web-targeting haskell dialects," in *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, ser. IFL '15, Association for Computing Machinery, 2015, ISBN: 9781450342735. DOI: 10.1145/2897336.2897338.

[44] R. Elshawi, A. Wahab, A. Barnawi, and S. Sakr, "DLBench: A comprehensive experimental evaluation of deep learning frameworks," en, *Cluster Computing*, vol. 24, no. 3, pp. 2017–2038, 2021, ISSN: 1573-7543. DOI: 10.1007/s10586-021-03240-4.

[45] A. Esteva *et al.*, "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, pp. 24–29, 2019.

[46] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, ACM, 2020, pp. 246–257, ISBN: 9781450371216. DOI: 10.1145/3377811.3380413.

[47] H. Fang, H. Lamba, J. Herbsleb, and B. Vasilescu, ""this is damn slick!": Estimating the impact of tweets on open source project popularity and new contributors," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Association for Computing Machinery, 2022, pp. 2116–2129, ISBN: 9781450392211. DOI: 10.1145/3510003.3510121.

[48] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough, "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[49] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones, "H/Direct: A binary foreign language interface for Haskell," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98, Association for Computing Machinery, 1998, 153–162, ISBN: 1581130244. DOI: 10.1145/289423.289437.

[50] O. Franco-Bedoya, D. Ameller, D. Costal, and X. Franch, "Open source software ecosystems: A systematic mapping," *Information and Software Technology*, vol. 91, pp. 160–185, 2017, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.07.007.

[51] J. Frankle, G. K. Dziugaite, D. Roy, and M. Carbin, "Linear mode connectivity and the lottery ticket hypothesis," in *Proceedings of the 37th International Conference on Machine Learning*, vol. 119, PMLR, 2020, pp. 3259–3269.

[52] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 4, 18:1–18:63, 2008, ISSN: 0164-0925. DOI: 10.1145/1377492.1377493.

[53] K. Gao, R. He, B. Xie, and M. Zhou, "Characterizing deep learning package supply chains in pypi: Domains, clusters, and disengagement," *ACM Trans. Softw. Eng. Methodol.*, 2024, ISSN: 1049-331X. DOI: 10.1145/3640336.

[54] D. M. German, B. Adams, and A. E. Hassan, "The Evolution of the R Software Ecosystem," in *17th European Conference on Software Maintenance and Reengineering*, ser. CSMR, IEEE Press, 2013, pp. 243–252. DOI: 10.1109/CSMR.2013.33.

[55] M. N. Gevorkyan, A. V. Demidova, T. S. Demidova, and A. A. Sobolev, "Review and comparative analysis of machine learning libraries for machine learning," *Discrete and Continuous Models and Applied Computational Science*, vol. 27, no. 4, pp. 305–315, 2019, ISSN: 2658-7149. DOI: 10.22363/2658-4670-2019-27-4-305-315.

[56] S. Godbole and S. Sarawagi, "Discriminative methods for multi-labeled classification," in *Pacific-Asia conference on knowledge discovery and data mining*, Springer, 2004, pp. 22–30.

[57] D. Gonzalez, T. Zimmermann, and N. Nagappan, "The state of the ML-universe: 10 years of artificial intelligence & machine learning software development on GitHub," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 431–442.

[58] M. Grichi, E. E. Eghan, and B. Adams, "On the impact of multi-language development in machine learning frameworks," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 546–556.

[59] M. Grimmer, "High-performance language interoperability in multi-language runtimes," in *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '14, Association for Computing Machinery, 2014, pp. 17–19, ISBN: 9781450332088. DOI: 10.1145/2660252.2660256.

[60] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018, ISSN: 2374-3468.

[61] 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)*, ser. ASE '19, IEEE Press, 2019, pp. 810–822, ISBN: 9781728125084. DOI: 10.1109/ASE.2019.00080.

[62] A. Gupta, A. Anpalagan, L. Guan, and A. S. Khwaja, "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues," *Array*, vol. 10, p. 100 057, 2021, ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2021.100057.

[63] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, "An Empirical Study of the Dependency Networks of Deep Learning Libraries," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ISSN: 2576-3148, IEEE Press, 2020, pp. 868–878. DOI: 10.1109/ICSME46990.2020.00116.

[64] J. Han, S. Deng, X. Xia, D. Wang, and J. Yin, "Characterization and prediction of popular projects on GitHub," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 21–26. DOI: 10.1109/COMPSAC.2019.00013.

[65] X. Han *et al.*, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.

[66] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[67] O. Hourrane, N. Idrissi, and E. H. Benlahmar, "An empirical study of deep neural networks models for sentiment classification on movie reviews," in *1st International Conference on Smart Systems and Data Science (ICSSD)*, 2019, pp. 1–6. DOI: 10.1109/ICSSD47982.2019.9003171.

[68] E. J. Hu *et al.*, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022.

[69] Q. Hu *et al.*, "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Transactions on Software Engineering and Methodology*, 2022, ISSN: 1049-331X. DOI: 10.1145/3511598.

[70] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Association for Computing Machinery, 2020, pp. 1110–1121, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380395.

[71] G. Iaffaldano, I. Steinmacher, F. Calefato, M. Gerosa, and F. Lanubile, "Why do developers take breaks from contributing to OSS projects? a preliminary analysis," in *Proceedings of the 2nd International Workshop on Software Health*, IEEE Press, 2019, pp. 9–16. DOI: 10.1109/SoHeal.2019.00009.

[72] S. K. Imminni *et al.*, "SPYSE: A semantic search engine for Python packages and modules," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, ACM, 2016, pp. 625–628, ISBN: 9781450342056. DOI: 10.1145/2889160.2889174.

[73] International Organization for Standardization, *ISO/IEC TR 10182:2016, Information technology - Programming languages, their environments and system software interfaces - Guidelines for language bindings*, International Standard, 2016. [Online]. Available: https://www.iso.org/standard/67465.html.

[74] R. Isdahl and O. E. Gundersen, "Out-of-the-box reproducibility: A survey of machine learning platforms," in *15th International Conference on eScience (eScience)*, 2019, pp. 86–95. DOI: 10.1109/eScience.2019.00017.

[75] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, *What Do Developers Ask About ML Libraries? A Large-scale Study Using Stack Overflow*, 2019. arXiv: 1906.11940 [cs.SE].

[76] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, "Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891042.

[77] S. Jansen, "Measuring the health of open source software ecosystems: Beyond the scope of project health," *Information and Software Technology*, vol. 56, no. 11, pp. 1508–1519, 2014, Special issue on Software Ecosystems, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.04.006.

[78] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside TensorFlow," in *Database Systems for Advanced Applications*, Y. Nah, B. Cui, S.-W. Lee, J. X. Yu, Y.-S. Moon, and S. E. Whang, Eds., Springer International Publishing, 2020, pp. 604–620, ISBN: 978-3-030-59410-7.

[79] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110 935, 2021, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110935.

[80] M. Jouhari *et al.*, "Distributed cnn inference on resource-constrained uavs for surveillance systems: Design and optimization," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1227–1242, 2022. DOI: 10.1109/JIOT.2021.3079164.

[81] K. Kannee, R. Kula, S. Wattanakriengkrai, and K. Matsumoto, "Intertwining communities: Exploring libraries that cross software ecosystems," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, IEEE Computer Society, 2023, pp. 518–522. DOI: 10.1109/MSR59073.2023.00077.

[82] J. Katz, *Libraries.io Open Source Repository and Dependency Metadata*, version 1.6.0, 2020. DOI: 10.5281/zenodo.3626071. [Online]. Available: https://doi.org/10.5281/zenodo.3626071.

[83] Keras. "About Keras." (2021), [Online]. Available: https://keras.io/about/.

[84] S. S. Khanal, P. Prasad, A. Alsadoon, and A. Maag, "A systematic review: Machine learning based recommendation systems for e-learning," *Education and Information Technologies*, vol. 25, no. 4, pp. 2635–2664, 2020.

[85] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is It All Lost? A Study of Inactive Open Source Projects," in *9th Open Source Software (OSS)*, ser. Open Source Software: Quality Verification, vol. AICT-404, Springer, 2013, pp. 61–79. DOI: 10.1007/978-3-642-38928-3_5.

[86] S. Kiliçarslan and M. Celik, "RSigELU: A nonlinear activation function for deep neural networks," *Expert Systems with Applications*, vol. 174, p. 114 805, 2021, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2021.114805.

[87] H. Ko, S. Lee, Y. Park, and A. Choi, "A survey of recommendation systems: Recommendation models, techniques, and application fields," *Electronics*, vol. 11, no. 1, p. 141, 2022.

[88] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images (Technical Report)," 2012.

[89] R. G. Kula and G. Robles, "The life and death of software ecosystems," in *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability: Communications of NII Shonan Meetings*. Springer Singapore, 2019, pp. 97–105, ISBN: 978-981-13-7099-1. DOI: 10.1007/978-981-13-7099-1_6.

[90] R. G. Kula, C. D. Roover, D. M. Germán, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *25th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER 2018, IEEE Computer Society, 2018, pp. 288–299. DOI: 10.1109/SANER.2018.8330217.

[91] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite bert for self-supervised learning of language representations," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[92] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[93] Y. LeCun, C. Cortes, and C. J. Burges. "The MNIST database of handwritten digits." (1998), [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[94] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: Synthesizing dynamic bug detectors for foreign language interfaces," *SIGPLAN Not.*, vol. 45, no. 6, pp. 36–49, 2010, ISSN: 0362-1340. DOI: 10.1145/1809028.1806601.

[95] H. Lei, S. Zhang, J. Wang, G. Xiao, Y. Liu, and Y. Sui, "Why do deep learning projects differ in compatible framework versions? an exploratory study," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 509–520. DOI: 10.1109/ISSRE59848.2023.00076.

[96] F. Li, Y. Ye, Z. Tian, and X. Zhang, "CPU versus GPU: Which can perform matrix computation faster—performance comparison for basic linear algebra subprograms," *Neural Computing and Applications*, vol. 31, no. 8, pp. 4353–4365, 2019, ISSN: 1433-3058. DOI: 10.1007/s00521-018-3354-z.

[97] H. Li and C.-P. Bezemer, "Studying popular open source machine learning libraries and their cross-ecosystem bindings," *arXiv preprint arXiv:2201.07201*, 2022. DOI: 10.48550/ARXIV.2201.07201.

[98] H. Li and C.-P. Bezemer, *The replication package of our study*, Accessed: 13 March 2024, 2024. [Online]. Available: https://github.com/asgaardlab/MLBindings.

[99] H. Li, F. R. Cogo, and C.-P. Bezemer, "An empirical study of yanked releases in the rust package registry," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 437–449, 2023. DOI: 10.1109/TSE.2022.3152148.

[100] H. Li, G. K. Rajbahadur, and C.-P. Bezemer. "The replication package of our study on bindings for TensorFlow and PyTorch." (2022), [Online]. Available: https://github.com/anonymous-git/dl_with_different_languages.

[101] X. Li, B. Karimi, and P. Li, "On distributed adaptive optimization with gradient compression," in *International Conference on Learning Representations*, 2022.

[102] E. Lin, Q. Chen, and X. Qi, "Deep reinforcement learning for imbalanced classification," *Applied Intelligence*, vol. 50, no. 8, pp. 2488–2502, 2020, ISSN: 1573-7497. DOI: 10.1007/s10489-020-01637-z.

[103] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the reproducibility and replicability of deep learning in software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, 2021, ISSN: 1049-331X. DOI: 10.1145/3477535.

[104] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, ser. ICSE-SEIS '20, Association for Computing Machinery, 2020, pp. 1–10. DOI: 10.1145/3377815.3381377.

[105] Y. Liu *et al.*, "RoBERTa: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[106] J. D. Long, D. Feng, and N. Cliff, "Ordinal analysis of behavioral data," en, in *Handbook of Psychology*, I. B. Weiner, Ed. American Cancer Society, 2003, ch. 25, pp. 635–661, ISBN: 9780471264385. DOI: 10.1002/0471264385.wei0225. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471264385.wei0225.

[107] C. Luo, X. He, J. Zhan, L. Wang, W. Gao, and J. Dai, "Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices," *arXiv preprint arXiv:2005.05085*, 2020.

[108] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, ser. HLT '11, Association for Computational Linguistics, 2011, pp. 142–150.

[109] K. Manikas, "Revisiting software ecosystems research: A longitudinal literature study," *Journal of Systems and Software*, vol. 117, pp. 84–103, 2016, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.02.003.

[110] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 –60, 1947. DOI: 10.1214/aoms/1177730491.

[111] J. Maqsood, I. Eshraghi, and S. S. Ali, "Success or failure identification for GitHub's open source projects," in *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences*, ACM, 2017, pp. 145–150, ISBN: 9781450348348. DOI: 10.1145/3034950.3034957.

[112] M. B. A. McDermott, S. Wang, N. Marinsek, R. Ranganath, L. Foschini, and M. Ghassemi, "Reproducibility in machine learning for health research: Still a ways to go," *Science Translational Medicine*, vol. 13, no. 586, eabb1655, 2021. DOI: 10.1126/scitranslmed.abb1655.

[113] K. Meerbergen, K. Fresl, and T. Knapen, "C++ bindings to external software libraries with examples from BLAS, LAPACK, UMFPACK, and MUMPS," *ACM Trans. Math. Softw.*, vol. 36, no. 4, 2009, ISSN: 0098-3500. DOI: 10.1145/1555386.1555391.

[114] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, "Studying evolving software ecosystems based on ecological models," in *Evolving Software Systems*. Springer Berlin Heidelberg, 2014, pp. 297–326, ISBN: 978-3-642-45398-4. DOI: 10.1007/978-3-642-45398-4_10.

[115] M. M. Morovati, A. Nikanjam, F. Khomh, and Z. M. J. Jiang, "Bugs in machine learning-based systems: A faultload benchmark," *Empirical Softw. Engg.*, vol. 28, no. 3, 2023, ISSN: 1382-3256. DOI: 10.1007/s10664-023-10291-1.

[116] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic implementations for reproducibility in deep reinforcement learning," in *AAAI 2019 Workshop on Reproducible AI*, 2019. DOI: 10.48550/ARXIV.1809.05676.

[117] S. Nakata, M. Sugaya, and K. Kuramitsu, "Fault model of foreign function interface across different domains," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, ISSN: 2325-6664, 2011, pp. 248–253. DOI: 10.1109/DSNW.2011.5958850.

[118] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 778–788. DOI: 10.1109/ICSE.2015.90.

[119] G. Nguyen *et al.*, "Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: A survey," *Artif. Intell. Rev.*, vol. 52, no. 1, pp. 77–124, 2019, ISSN: 0269-2821. DOI: 10.1007/s10462-018-09679-z.

[120] J. Ni, Y. Chen, Y. Chen, J. Zhu, D. Ali, and W. Cao, "A survey on theories and applications for self-driving cars based on deep learning methods," *Applied Sciences*, vol. 10, no. 8, 2020, ISSN: 2076-3417. DOI: 10.3390/app10082749.

[121] A.-M. Nussberger, L. Luo, L. E. Celis, and M. J. Crockett, "Public attitudes value interpretability but prioritize accuracy in artificial intelligence," *Nature communications*, vol. 13, no. 1, p. 5821, 2022.

[122] OpenCV, *OpenCV-Python is now an official OpenCV project*, Accessed: 13 March 2024, 2021. [Online]. Available: https://opencv.org/blog/opencv-python-is-now-an-official-opencv-project.

[123] K. Osman and O. Baysal, "Health is wealth: Evaluating the health of the bitcoin ecosystem in GitHub," in *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, 2021, pp. 1–8. DOI: 10.1109/SoHeal52568.2021.00007.

[124] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, ser. NeurIPS, Curran Associates, Inc., 2019, pp. 8024–8035.

[125] F. Pedregosa *et al.*, "Scikit-Learn: Machine learning in Python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011, ISSN: 1532-4435.

[126] R. Pereira *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Association for Computing Machinery, 2017, pp. 256–267, ISBN: 9781450355254. DOI: 10.1145/3136014.3136031.

[127] H. V. Pham *et al.*, "Problems and opportunities in training deep learning software systems: An analysis of variance," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20, Association for Computing Machinery, 2020, pp. 771–783, ISBN: 9781450367684. DOI: 10.1145/3324884.3416545.

[128] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000. DOI: 10.1109/2.876288.

[129] PyTorch, *Pytorch design philosophy*, Accessed: 13 March 2024, 2022. [Online]. Available: https://pytorch.org/docs/stable/community/design.html.

[130] PyTorch, *PyTorch Mobile, end-to-end workflow from training to deployment for ios and android mobile devices*, Accessed: 13 March 2024, 2021. [Online]. Available: https://pytorch.org/mobile/ios.

[131] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for SQuAD," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, I. Gurevych and Y. Miyao, Eds., Association for Computational Linguistics, 2018, pp. 784–789. DOI: 10.18653/v1/P18-2124.

[132] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, and X. Carreras, Eds., Association for Computational Linguistics, 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264.

[133] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, 2020.

[134] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit, "Automatic generation of library bindings using static analysis," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Association for Computing Machinery, 2009, pp. 352–362, ISBN: 9781605583921. DOI: 10.1145/1542476.1542516.

[135] J. Reppy and C. Song, "Application-specific foreign-interface generation," in *Proceedings of the 5th international conference on Generative programming and component engineering*, ser. GPCE '06, Association for Computing Machinery, 2006, pp. 49–58, ISBN: 978-1-59593-237-2. DOI: 10.1145/1173706.1173714.

[136] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? the case of a Smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, ACM, 2012, ISBN: 9781450316149. DOI: 10.1145/2393596.2393662.

[137] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*, Citeseer, 2006, pp. 1–51.

[138] F. Roseiro Côgo, G. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, 2021.

[139] A. S., S. Sinha, and S. K.G., "Optimization of convolutional neural networks on resource constrained devices," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 19–24. DOI: 10.1109/ISVLSI.2019.00013.

[140] S. Salvador and P. Chan, "FastDTW: Toward accurate dynamic time warping in linear time and space," *Intelligent Data Analysis*, vol. 11, no. 5, pp. 561–580, 2007.

[141] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of bert: Smaller, faster, cheaper and lighter," in *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing (EMC2) co-located with the Thirty-third Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[142] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 561–571. DOI: 10.1145/3180155.3180170.

[143] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., ser. NIPS'15, vol. 28, MIT Press, 2015, pp. 2503–2511.

[144] A. Serebrenik and T. Mens, "Challenges in software ecosystems research," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ser. ECSAW '15, ACM, 2015, ISBN: 9781450333931. DOI: 10.1145/2797433.2797475.

[145] J. P. Shaffer, "Multiple hypothesis testing," *Annual review of psychology*, vol. 46, no. 1, pp. 561–584, 1995.

[146] K. Shen, Y. Zhang, L. Bao, Z. Wan, Z. Li, and M. Wu, "Patchmatch: A tool for locating patches of open source project vulnerabilities," in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '23, IEEE Press, 2023, pp. 175–179, ISBN: 9798350322637. DOI: 10.1109/ICSE-Companion58688.2023.00049.

[147] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" In *IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 691–701.

[148] A. A. Shvets, A. Rakhlin, A. A. Kalinin, and V. I. Iglovikov, "Automatic instrument segmentation in robot-assisted surgery using deep learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 624–628. DOI: 10.1109/ICMLA.2018.00100.

[149] J. Siebert *et al.*, "Construction of a quality model for machine learning systems," en, *Software Quality Journal*, vol. 30, no. 2, pp. 307–335, 2022, ISSN: 1573-1367. DOI: 10.1007/s11219-021-09557-y.

[150] S. da Silva Amorim, J. D. McGregor, E. S. de Almeida, and C. von Flach G. Chavez, "Software ecosystems architectural health: Challenges x practices," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW '16, Association for Computing Machinery, 2016, ISBN: 9781450347815. DOI: 10.1145/2993412.3011881.

[151] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[152] D. Smilkov *et al.*, "TensorFlow.js: Machine learning for the web and beyond," in *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, 2019, pp. 309–321.

[153] C. R. de Souza, F. Figueira Filho, M. Miranda, R. P. Ferreira, C. Treude, and L. Singer, "The social side of software platform ecosystems," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, Association for Computing Machinery, 2016, pp. 3204–3214, ISBN: 9781450333627. DOI: 10.1145/2858036.2858431.

[154] StackOverflow, *Stack Overflow Annual Developer Survey 2023*, Accessed: 13 March 2024, 2023. [Online]. Available: https://survey.stackoverflow.co/2023.

[155] C. Steglich, A. Majdenbaum, S. Marczak, and R. Santos, "A study on organizational it security in mobile software ecosystems literature," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 234–241. DOI: 10.1109/ICSA-C50368.2020.00047.

[156] C. Steglich *et al.*, "How do business factors affect developers in mobile software ecosystems?" In *XVI Brazilian Symposium on Information Systems*, ser. SBSI'20, Association for Computing Machinery, 2020, ISBN: 9781450388733. DOI: 10.1145/3411564.3411571.

[157] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 228–237. DOI: 10.1109/APSEC51365.2020.00031.

[158] B. Sullivan and A. Kaszynski, "PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)," *Journal of Open Source Software*, vol. 4, no. 37, p. 1450, 2019. DOI: 10.21105/joss.01450.

[159] R. Tatman, J. Vanderplas, and S. Dane, "A practical taxonomy of reproducibility for machine learning research," in *Reproducibility in Machine Learning Workshop at ICML 2018*, 2018.

[160] H. Touvron *et al.*, *Llama 2: Open foundation and fine-tuned chat models*, 2023. arXiv: 2307.09288 [cs.CL].

[161] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2018, pp. 644–655, ISBN: 9781450355735. DOI: 10.1145/3236024.3236062.

[162] R. Štrobl and Z. Troníček, "Migration from deprecated API in Java," in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '13, ACM, 2013, pp. 85–86, ISBN: 9781450319959. DOI: 10.1145/2508075.2508093.

[163] B. Wang *et al.*, "On position embeddings in BERT," in *International Conference on Learning Representations*, 2021.

[164] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2020, pp. 233–244, ISBN: 9781450370431.

[165] K. Wang, Y. Dou, T. Sun, P. Qiao, and D. Wen, "An automatic learning rate decay strategy for stochastic gradient descent optimization methods in neural networks," *International Journal of Intelligent Systems*, 2022. DOI: https://doi.org/10.1002/int.22883.

[166] Y. Wang *et al.*, "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, ACM, 2020, pp. 125–135, ISBN: 9781450371216. DOI: 10.1145/3377811.3380426.

[167] M. Wei, Y. Huang, J. Wang, J. Shin, N. S. Harzevili, and S. Wang, "API recommendation for machine learning libraries: How far are we?" In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Association for Computing Machinery, 2022, pp. 370–381, ISBN: 9781450394130. DOI: 10.1145/3540250.3549124.

[168] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Association for Computing Machinery, 2016, pp. 351–361, ISBN: 9781450341868. DOI: 10.1145/2901739.2901743.

[169] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Association for Computational Linguistics, 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6.

[170] T. Wolter, A. Barcomb, D. Riehle, and N. Harutyunyan, "Open source license inconsistencies on GitHub," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, 2023, ISSN: 1049-331X. DOI: 10.1145/3571852.

[171] Y. Xi, L. Shen, Y. Gui, and W. Zhao, "Migrating deprecated API to documented replacement: Patterns and tool," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, ACM, 2019, ISBN: 9781450377010. DOI: 10.1145/3361242.3361246.

[172] X. Xia, S. Zhao, X. Zhang, Z. Lou, W. Wang, and F. Bi, "Understanding the archived projects on GitHub," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 13–24. DOI: 10.1109/SANER56733.2023.00012.

[173] D. Xu, S. Zhang, H. Zhang, and D. P. Mandic, "Convergence of the RMSProp deep learning method with penalty for nonconvex optimization," *Neural Networks*, vol. 139, pp. 17–23, 2021, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2021.02.011.

[174] J. Yallop, D. Sheets, and A. Madhavapeddy, "A modular foreign function interface," *Science of Computer Programming*, vol. 164, pp. 82–97, 2018, Special issue of selected papers from FLOPS 2016. DOI: https://doi.org/10.1016/j.scico.2017.04.002.

[175] Z. Yan, J. Zhou, and W.-F. Wong, "Near lossless transfer learning for spiking neural networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, pp. 10 577–10 584, 2021.

[176] C. Yang, B. Xu, J. Liu, and D. Lo, "Techsumbot: A stack overflow answer summarization tool for technical query," in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '23, IEEE Press, 2023, pp. 132–135, ISBN: 9798350322637. DOI: 10.1109/ICSE-Companion58688.2023.00040.

[177] J. Yasmin, Y. Tian, and J. Yang, "A first look at the deprecation of RESTful APIs: An empirical study," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 151–161. DOI: 10.1109/ICSME46990.2020.00024.

[178] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 589–593. DOI: 10.1109/SANER.2019.8667997.

[179] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories — and its application to npm," *Journal of Software: Evolution and Process*, vol. 31, no. 8, e2157, 2019. DOI: https://doi.org/10.1002/smr.2157.

[180] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An Empirical Study of Common Challenges in Developing Deep Learning Applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 104–115. DOI: 10.1109/ISSRE.2019.00020.

[181] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An Empirical Study on TensorFlow Program Bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Association for Computing Machinery, 2018, pp. 129–140, ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213866.