AN EXPLORATORY STUDY ON THE PLATFORMS OF SHARING REUSABLE MACHINE LEARNING MODELS

MINKE XIU

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
NOVEMBER 2020

Abstract

Recent advances in Artificial Intelligence, especially in Machine Learning (ML), have brought applications previously considered as science fiction (e.g., virtual personal assistants and autonomous cars) into the reach of millions of everyday users. Since modern ML technologies like deep learning require considerable technical expertise and resource to build custom models, reusing existing models trained by experts has become essential. Currently the ML models are shared, distributed, or retailed on multiple ML model platforms which can be divided into two categories based on their usage patterns: (1) ML model stores whose models can be deployed and served with the help of cloud infrastructure, and (2) ML package repositories whose models are free but need to be deployed and used (e.g., embedded into users' applications as a software component) manually.

We conducted an exploratory study on the above two categories of ML model platforms: ML model stores and ML package repositories. We analyzed the struc-

ture and the contents of the ML models platforms, as well as functionalities provided by the package managers. The research subjects were three general purpose ML model stores (AWS marketplace, ModelDepot, and Wolfram neural net repository) and two popular ML package repositories (TensorFlow Hub and PyTorch Hub). When studying the structure of ML model platforms and functionalities of package managers, we compared them against their counterparts from traditional software development: ML model stores vs. mobile app stores (e.g., Google Play and Apple App Store), and ML package repositories vs. programming language package repositories (e.g., npm, PyPI, and CRAN). Through our study, we identified special software engineering practices and challenges for sharing, distributing, and retailing ML models. The implications from this thesis will be helpful for stakeholders to make the ML model platforms better serve the users (i.e., software engineers, data scientists and researchers).

Table of Contents

List of Tables

List of Figures

1 Introduction

The development of artificial intelligence (AI) (especially machine learning (ML)) software is in great demand. ML is making revolutionary changes in many different fields (e.g, healthcare, retail, energy and software) [1]. A recent estimate showed that ML applications have the potential to create between $3.5 and $5.8 trillion in value annually [2].

Unfortunately, the development of ML models, which is the core of AI software, is a non-trivial task. First, the implementation of ML algorithms is difficult as specific skills are required in reading and understanding professional AI literature, and the ML algorithms themselves are complicated. ML frameworks (e.g., TensorFlow [3], PyTorch [4], etc.), however, has greatly lowered the skill requirements for ML model development. With such frameworks, the development of ML models can be as simple as calling appropriate APIs, allowing developers to devote more time and energy to other tasks such as obtaining, pre-processing, labeling and filtering data, adjusting and testing the models' structure, or proposing new ML algorithms.

Secondly, training ML models requires significant computational resources. Models that perform complex tasks like image classification or text embedding need intensive calculation and require a long time to finish training on large-scale datasets [5]. Although expensive equipment such as GPUs can be used to shorten the training duration, they may not always be available to software developers.

As a result, there is such a high demand for shareable and reusable pre-trained ML models. Recently, Gartner has identified that leveraging pre-trained ML models deployed as web services to be one of the top technology trends [6]. Fortunately, there have already been quite a few ML models that are shared, distributed or retailed online via **ML model platforms**, which bridge the gap between **AI experts** (e.g., researchers in ML algorithms, mathematician) and general **users** (e.g., AI software developers, data scientists and researchers).

In this thesis, ML model platforms are divided into the following two categories based on their usage patterns.

- The first category is referred to as *ML model stores*. They provide cloud-based model deployment support. They are comparable to the traditional mobile app stores (e.g., Google Play [7] and Apple's app store [8]) in terms of organizing and retailing products, and deployment facility. ML model stores are introduced by various organizations to facilitate the distribution and retailing of ML models to organizations/developers. Usually, users need to

pay for using and hosting models on the cloud computing resources provided by ML model stores. AWS marketplace [9], ModelDepot [10], and the Wolfram neural net repository [11] are representative examples of such ML model stores.

- The second category is referred to as *ML package repositories*. The models on ML package repositories need to be embedded into consumers' applications like dependencies [12]. These ML packages are generally free to use, but users have to manually deploy and manage these models and other dependencies locally. ML package repositories contain tens to hundreds of pre-trained models that specially bundled up into **ML packages** that are distributed via **ML package managers** (including ML frameworks, ML-related libraries and APIs). For example, the PyTorch Hub repository [13] contains packages that can be accessed by a user via the APIs in PyTorch framework. The most popular examples of such repositories are TFHub [14] and PyTorch Hub [13]. Their distribution practices are similar to programming language-specific software package repositories like npm [15], PyPI [16], and CRAN [17].

Unfortunately, there are no studies reporting the best practices and common pitfalls involving ML model platforms. In particular, what information about models/packages is provided by ML model platforms and how helpful are they to the

users? What ML domains and task types are supported by these ML packages? How are ML models/packages organized and distributed? How are such ML packages/models used? Are there any common practices between these ML model platforms and their counterpart mobile app stores/ software package repositories?

Hence, in this thesis we conducted an exploratory study on three ML model stores (i.e., AWS marketplace (referred to as AWS in later parts of the thesis), ModelDepot and Wolfram Neural Net Repository (referred to as Wolfram in later parts of the thesis)) and two ML package repositories (i.e., TensorFlow Hub (referred to as TFHub in later parts of the thesis) and PyTorch Hub). We not only compared the structure and the information elements (features and policies) among these ML model platforms, but also compared them against their counterparts: mobile app stores (Google Play and Apple's app store) and programming language package repositories (npm, PyPI and CRAN). For ML model stores, we found some special practices like cloud deployment and user instructions, while some information elements (e.g., review policy) are still missing in ML model stores. And we also found few similar offerings between ML model stores. For ML package repositories, our results showed three significant differences between the practices of reusing ML packages and traditional software packages. First, although some of the practices on software package repositories have been adopted by their ML counterparts (e.g., product line architecture, multiple usage contexts), most of the established

SE practices are either not adopted by or in their infant stages on the ML package repositories (e.g., release management, dependency management, security, package management functionalities). Secondly, some practices on the ML package repositories that are not yet adopted by the software package repositories (e.g., quality evaluation of packages). Thirdly, the processes of installing and using ML packages differ from those of software packages. The findings of this research will help software engineers and researchers who are familiar with traditional software engineering practices, but not yet with ML package practices, to have a clear and easier understanding of sharing and using reusable ML models.

1.1 Contributions

The contributions of this thesis are:

- We provided an overview of the current practices on sharing reusable ML packages through a study of the structure and contents of ML model platforms. To the best of our knowledge, this is the first empirical study on ML model platforms.

- By comparing against the sharing mechanism of mobile app stores and software package repositories, we identified a set of unique practices and challenges on distributing, sharing, and using pre-trained ML packages and mod-

els.

- Our comparison between the practices of ML and their counterparts in traditional software engineering presents stakeholders of ML model platforms with opportunities of how to adopt the established practices from traditional software engineering.

1.2 Thesis Organization

The remaining part of this thesis is divided into four chapters. Chapter 2 describes the background and related work of this thesis. Chapter 3 and Chapter 4 present the studies on ML model stores and ML package repositories, respectively. To be noted that when this thesis was under editing, studies in Chapter 3 had been accepted by IEEE Software, and the studies in Chapter 4 had been submitted to Empirical Software Engineering and under review. Chapter 5 concludes the thesis and provides some future work directions.

2 Background and Related Works

This chapter presents the background and related works on ML model platforms.

2.1 Background and Related Work for ML Model Stores

2.1.1 App Stores and Model Stores

Despite their difference in age, app and model stores provide platforms for developers to distribute and retail their products to their intended target audience (end users vs. organizations/developers). App stores have been around for over ten years. Apple's App Store and Google Play, both started in 2008, are currently two of the most popular app stores. Each contains over two million apps. These app stores include mobile apps and software applications for computers (e.g., Mac App Store) and tablets (e.g., Chromebook and iPad). In contrast, the concept of "model store" is relatively new, with ModelDepot starting in 01/2018, Wolfram neural net repository in 06/2018, and AWS marketplace in 11/2018. For brevity, we will call these three model stores as "ModelDepot", "Wolfram", and "AWS".

There are two types of model stores: (1) general purpose, and (2) specialized model stores. General purpose model stores (e.g., AWS) contain all sorts of ML models, whereas specialized model stores (e.g., Nuance AI market [18]) only contain models from certain domains. We focus on the general purpose model stores, since the

2.1.2 Empirical Studies on Mobile App Stores

There is a large corpus of research on empirical studies of the mobile app stores (e.g., information elements [19], user reviews [20], and update frequency [21]) on understanding and improving the quality of the apps. Since the information elements in app stores by now are widely understood, this chapter focuses on an empirical comparison of app stores to the newly introduced model stores.

2.2 Background and Related Works for ML Package Repositories

Package managers are a set of software tools that automate the process of package installation, upgrade, and removal in a consistent manner. Packages are hosted in

Table 2.1: Statistics of software package repositories. "+" indicates a repository containing a small percentage of packages in other programming languages. The software package repository statistics were gathered on April 19, 2020 from `Libraries.io`.

Repository		**Language(s) or Framework**	**Launch Time**	**# Packages**
Software Package Repository	Bower	JavaScript+	Sep 2012	69,678
	Cargo (Crates.io)	Rust+	Jun 2014	40,142
	Clojars	Clojure+	Nov 2009	25,913
	CRAN	R	$\geq$ Aug 1993	17,370
	Go Package Community	Go+	$\geq$ Mar 2008	1,818,628
	Hackage	Haskell+	Jun 2008	14,758
	Hex	Elixir+	Dec 2013	9,911
	Maven	Java+	Sep 2003	185,402
	MELPA (Emacs)	Emacs Lisp+	Oct 2011	5,026
	MetaCPAN (CPAN Search)	Perl+	Nov 2010	37,790
	npm	JavaScript	Sep 2009	1,366,638
	NuGet	C#+	Jan 2011	201,192
	Packagist	PHP+	Apr 2011	328,953
	PyPI	Python	Oct 2008	250,533
	Rubygems	Ruby+	Nov 2003	164,749

Table 2.2: Statistics of ML package repositories, whose statistics were gathered on March 25, 2020.

Repository		**Language(s) or Framework**	**Launch Time**	**# Packages**
ML Package Repository	AIHub TensorFlow Module	TensorFlow (Python)	(March 2019)	322
	DL4J Zoo Models	DL4J (Java)	Jun 2019	16
	MXNet GluonCV Model Zoo	MXNet (Python, Scala, etc.)	$\geq$ Apr 2014	323
	MXNet GluonNLP Model Zoo	MXNet (Python, Scala, etc.)	$\geq$ Apr 2014	42
	PyTorch Hub	PyTorch (Python, C++, Java)	Jun 2019	26
	spaCy Models	spaCy (Python)	$\geq$ Feb 2015	6
	TFHub	TensorFlow (Python, JavaScript, C++, Java, etc.)	March 2018	471
	Torch7 Model Zoo	Torch7 (LuaJIT, C)	Jan 2015	20

and downloaded from package repositories[1,2]. Generally speaking, package managers can be divided into two groups. The first group (e.g., dpkg for Debian, Homebrew for macOS, and Windows Store for Windows) provides compiled (binary) or source code package management for operating system-specific applications, while the second group (e.g., npm for JavaScript and PyPI for Python) provides package management for programming language-specific API-level packages. This research focuses only on API-level software package repositories (referred from this point simply as software package repositories), and contrasts them to ML package repositories.

In order to gain some basic knowledge of the existing software package repositories, Table 2.1 presents the basic statistics of the popular software package repositories (containing more than 4,000 packages) from `Libraries.io` [22], which is a popular index of the most common software package repositories — it monitors the information about the packages within different software package repositories. As shown in the table, we measured the launch time of repositories as the time of the first commit of the GitHub repository that stores the actual source code powering the package sharing websites. There are a few exceptions: the launch times of the Go and CRAN repositories cannot be found, so we noted the time as no earlier

[1]https://en.wikipedia.org/wiki/Package_manager

[2]https://en.wikipedia.org/wiki/List_of_software_package_management_systems#Application-level_package_managers

than ($\geq$) the initial date of their respective programming language (Go and R); NuGet's launch time is gathered from Microsoft's documentation, not GitHub.

Unfortunately, since the concept of ML package repository is relatively new, their information is not tracked yet by `Libraries.io` and we had to manually gather information for ML package repositories. The list of ML package repositories was obtained from the study of Braiek et al. [23]. Their launch times were retrieved from multiple sources (e.g., twitter and blog posts) that announced the launch, the first release, or commit of their respective GitHub repositories. We noted the launch time as no earlier than the release time of the frameworks if no information is found from the previously mentioned sources. The total number of ML packages were counted manually since most of the ML package repositories (except for AIHub TensorFlow module [24] and TFHub [14]) do not provide these statistics. The results are shown in Table 2.2.

Due to their recency and relatively high learning curve (requiring deep ML expertise) and computing resource requirements, there are fewer reusable packages in ML package repositories, compared to software package repositories. We observed an average growth of approximately eight new packages per week in TFHub and two new packages every week in the AIHub repository, indicating that ML package repositories are still an upcoming phenomenon. Despite the relatively slow growth of the number of packages within these ML repositories, we observed a high usage

of TFHub and PyTorch Hub ML packages in open source projects. For example, a preliminary search of PyTorch Hub package loading API keyword[3] shows that PyTorch Hub packages are loaded in over 143K source code files on GitHub. Another search[4] showed that Huggingface Transformers, a package containing most of the complex NLP models, is used within 3.2K source code files on GitHub. These results show that users prefer to reuse such existing models given the difficulty of training custom ML packages. Thus, these limited number of ML packages can power unlimited possibilities in ML software development and ML research.

2.2.1 Software Package Repositories

Exploratory Study: There are a few exploratory studies on software package repositories. Bommarito et al. [25] analyzed the basic data of all of the packages on PyPI at that time, including information elements like packages, releases, dependencies, category classifications, licenses, package imports, authors, maintainers, and organizations. They reported the evolution of the PyPI repository in terms of active packages, new authors and new import statements. They observed highly right-skewed distributions of package release numbers, authors' package and release numbers, package import numbers, size of packages and releases. They also

[3] "`https://github.com/search?q=torch.hub.load%28&type=Code`"

[4] "`https://github.com/search?q=torch.hub.load('huggingface/transformers'%28&type=Code`"

found that most of the packages are contributed by single individuals. Raemaekers et. al. [26] presented a dataset that contains code metrics, dependencies, breaking changes between library versions of more than 148 thousand jar files and a complete call graph of the entire Maven repository.

Dependency Management: Dependency management on software package repositories is an aspect that attracted lots of prior work. Some researchers have studied the impact of dependencies (from software package repositories) on project health. Alqahtani et. al. [27] used a unified ontological representation to establish bi-directional traceability links between security vulnerability databases and software software repositories. It is shown that when packages are shared, knowledge, information and vulnerabilities are also shared. Eghan et. al. [28] took Maven as research subject and found that dependencies on external libraries have an impact on project quality in terms of security vulnerabilities, license violations, and breaking changes.

Decan et. al. [29] conducted a study about R packages distributed on CRAN and GitHub and found that on GitHub, which is an increasingly used R package distribution platform, packages are subject to inter-repository dependency problems that interfere their automatic installation. Cogo et al. [30] looked at the phenomenon that developers downgrade the dependencies in the npm repository.

They found the reasons behind the occurrence of downgrades, how the versioning of dependencies changed when downgrades occur and how fast downgrade occurs.

Valiev et al. [31] explained that the interdependent network of open source projects is the software repository, the sustainability (maintainability, attractive to new comers, economic value of the project, etc.) of the projects that comprising an repository may be determined by the repository context as well. Through the case study of the PyPI repository, they found that project ties and relative position in dependency network have impact on sustained project activity. Abdalkareem et al. [32] calculated the proportion of trivial packages in npm and PyPI package repositories. They surveyed the developers about the reasons and drawbacks of using trivial packages. They also found that only part of trivial packages are tested, and few studied trivial packages have more than 20 dependencies.

Quality: In addition to dependency management, prior work also studied the quality aspects of various software package repositories. For example, Claes et al. [33] studied the phenomenon that developers copy the code from CRAN packages to their code rather than depend on packages. They learned the characteristics of the evolution of cloned code and the reasons behind the cloning activity. Trockman et al. [34] explained that project maintainers use badges to signal the quality of their projects to contributors and users on social coding platforms. Their investigation

into the badges in npm repository identifed the key quality attributes of interest to project maintainers and how well these qualitity attributes are reflected by badges.

2.2.2 Sharing Reusable ML Packages

Currently there are limited researches in this area. Research mentioning TFHub and PyTorch Hub are mostly using the ML packages from those hubs or contributing new ML packages to them. Touvron et al. [35] proposed an image classification optimization strategy relying on the fine-tuning of PyTorch Hub pre-trained ML packages. Yang et al. [36] proposed a new NLP algorithm and published the pre-trained ML packages on TFHub. Braiek et. al. [23] examined the evolution of ML frameworks and their repository (actors and adoption over time) to understand the role of open-source development in modern ML. They found that ML is between the early adoption and early maturity stage. They also found that companies are the main drivers of open-source ML, with the development teams consisting mostly of engineers and industry scientists. They also identify that the big cloud computing companies introduce a risk of a vendor lock-in for future ML development.

3 An Exploratory Study on Machine Learning Model Stores

In this chapter, we conducted an exploratory study on the current practices of sharing ML models via the ML model store. We focused on the following two research questions.

RQ1: What kind of information elements do model stores provide? This RQ compares (1) the information elements among three model stores; and (2) the information elements between model stores and app stores. By studying (1), we intended to derive a set of software engineering practices (e.g., documentation and delivery) of ML applications. By studying (2), we hoped to identify the commonalities and differences between conventional app stores and ML model stores. The findings of this RQ may be helpful for ML model stores, which is still in their infancy, to learn from established practices and grow faster and stronger in the future.

RQ2: How unique are the models provided by each model store? In

this RQ, we sought to investigate whether different model stores have their unique offerings of ML models. On one hand, the more applications in one store, the more likely that users can find the applications which suits their needs. Hence, app stores periodically report and compare the number of applications in their app stores. On the other hand, application developers hoping to reach more users by porting their applications to different app stores. So the findings of this RQ will be helpful for both users and developers of the models.

For each RQ in this chapter, the first section describes the experiments we did (i.e., how the data is extracted), the second section presents our findings, and the third section discusses the results and their implications.

Here we explain the choice of the research subjects. While the concept of "ML model store" is relatively new (ModelDepot started in January 2018, Wolfram in June 2018, and AWS in November 2018), mobile app stores have been around for more than 10 years. We focused on Apple's App Store (started in July 2008) and Google Play (October 2008), as they are two of the most popular app stores right now, each containing more than two millions apps. Both types of stores provide platforms for developers to distribute and retail their applications.

3.1 RQ1: What kind of information elements do model stores provide?

In this RQ, we compared the information elements between ML model stores and their counterparts mobile app stores. We extracted the information elements from the stores and provide the results in a list. Then through comparison of the information elements, the findings and implications of the commonalities and differences between two types of stores, and the unique software engineering practices and challenges of the ML models were derived.

In this RQ, collectively, we use the term "product" to refer to either an ML model or a mobile app. When referring to products from individual stores, we use the term "models" and "apps", respectively.

3.1.1 Approach

For each of the three considered model stores, we used open coding [37] to label the structure of the webpages used to sell/provide models. Two coders separately split up each page into "information elements", sections that provide a specific functionality geared towards the store's clients. For example, a section can provide a description or the price of a product. We started with the two app stores and tried to rediscover the reported information elements from Jansen et al. [19]. Since two app

stores may use terms differently, we manually merged the corresponding information elements among them. Certain elements from that paper are in considerable detail (e.g., different revenue models), we merged those detailed elements under higher-level elements, where they apply. In the end, all information elements from [19] were found by our study. Furthermore, ten additional store elements were found by us concerning release notes and product permissions, as app stores kept evolving since 2013. A similar process was performed on model stores and new elements unique to the model stores were found. For all stores, we grouped related elements into larger dimensions (e.g., `user feedback`, `usage statistics`, `pricing` under the **Business** dimension). This process was conducted by the me and my supervisor (Zhenming (Jack) Jiang), and later verified by a SE colleague (Bram Adams) to ensure correctness, for details please refer to this paper [38].

In the end, we have identified 26 unique elements across six different dimensions among all the stores as shown in Table 3.1 to Table 3.4. Each row corresponds to one element, while a ✓ indicates the presence of that element in a given store.

Table 3.1: Comparing elements among the five mobile app and model stores. We used the term "product" to refer to both "mobile apps" and "ML models".

Dimension	Element	Model Store (AWS)	Model Store (ModelDepot)	Model Store (Wolfram)	App Store (Apple)	App Store (Google)	Description
Product Information	Owner	✓	✓	✓	✓	✓	Developer information of this product.
	Description	✓	✓	✓	✓	✓	The objectives and the functionalities of this product.
	Demo		✓				A functionality provided for end users, so that they can try before buying/deploying the product.
	Language				✓		Languages used in the user interface of this product.
	Size		✓	✓	✓	✓	Size of the product in disk.
	Version number	✓		✓	✓	✓	The version number of the current release.
	Permission					✓	The list of hardware/software resources needed from a user's device to properly run this product.
	Age rating				✓	✓	Constraints about the user's age.

Table 3.2: Continued Table 3.1.

Dimension	Element	Model Store (AWS)	Model Store (ModelDepot)	Model Store (Wolfram)	App Store (Apple)	App Store (Google)	Description
Technical Documentation	User instruction	✓	✓	✓			Instructions on how to use this product.
	Framework	✓	✓	✓			The underlying development framework for the ML algorithms used in this product.
	ML Algorithms	✓	✓	✓			The types of ML algorithms used in this product.
	Training set	✓	✓	✓			Datasets used for training the underlying ML algorithms.
	Performance	✓	✓	✓			The performance (e.g., precision, recall, and accuracy) of the underlying ML algorithms.
	Origin	✓	✓	✓			Source of where the product originally came from (e.g., academic papers, open source products).
	Release notes	✓			✓	✓	Information regarding the changes in the current version of the product.

Table 3.3: Continued Table 3.2.

Dimension	Element	Model Store (AWS)	Model Store (ModelDepot)	Model Store (Wolfram)	App Store (Apple)	App Store (Google)	Description
Delivery	Deployment instructions	✓	✓	✓			Instructions on how to deploy and configure the product.
	Compatibility				✓	✓	Information on which platforms and versions are compatible with the product.
	Local installation				✓	✓	Automated installation of the product to a user's device.
	Cloud deployment	✓	✓	✓			Automatically deploying the product within the provider's cloud infrastructure.
Business	Pricing	✓	✓	✓	✓	✓	The pricing information about this product.
	User feedback	✓	✓	✓	✓	✓	User feedback (e.g., rating and comments) of this product.
	Usage statistics		✓			✓	Number of downloads for this product.

Table 3.4: Continued Table 3.3.

Dimension	Element	Model Store (AWS)	Model Store (ModelDepot)	Model Store (Wolfram)	App Store (Apple)	App Store (Google)	Description
Product Submission & Store Review	Online submission	✓	✓		✓	✓	Developers can automatically submit new versions of their products online.
	Store review policy				✓	✓	Documentation on policies for developers to follow in order to get approval of the product.
Legal Information	End user license	✓	✓	✓	✓	✓	Regulations on how users can use this product.
	Developer license		✓	✓			Regulations on how developers can further expand, integrate, and distribute a product in an authorized way.

3.1.2 Findings

3.1.2.1 Comparison among Model Stores

Among the total of 26 store elements, 20 exist in one or more model stores and 13 are common among the three studied model stores. Below, we detail our comparison results for each dimension:

- The **Product Information** dimension contains elements describing the characteristics of the model that is being distributed on the model stores. Only two elements (`owner` and `description`) are common among the three model stores. The `owner` element shows the contact information from the developers who submitted a model, while the `description` explains its objectives and functionalities. In addition to the above elements, ModelDepot and Wolfram provide information regarding the models' `size` on disk. ModelDepot also has a unique `demo` element allowing users to try out an ML model inside the browser without installing it. Models in AWS and Wolfram usually include a `version number` for each release, so that their users can easily tell whether they are using the current version of the model.

- The **Technical Documentation** dimension contains the development-specific information related to an ML model. Different from app stores, all three

model stores contain `user instructions`, as the users of ML models generally are organizations/developers. They will likely reuse a model as is in a similar or different product context (transfer learning), re-train a model using the provided training scripts or extend it by adding additional elements to the model. Hence, instead of a purely textual description, the `user instruction` for ML models generally contains programming examples in the form of scripts (e.g., Jupyter notebooks).

Different from AWS, ModelDepot and Wolfram contain many models originating from research prototypes or open source software products published on GitHub or authors' websites. The `origin` information of the models from these two stores is displayed in a dedicated section. So does the information about the `framework` (e.g., TensorFlow) used to train a model and the `ML algorithm` (e.g., Convolutional Neural Network). In ModelDepot, the information regarding the `framework` and the `ML algorithm` is prominently displayed at the top of each ML model's page. In Wolfram, more detailed `framework` and `ML algorithm` information is provided (e.g., number of layers and parameters for neural network architectures). In contrast, only about 4% of AWS models provide the `origin` information, 6% provide `framework` information, and 20% provide `ML algorithm` information.

Among the three model stores, Wolfram and ModelDepot have dedicated ar-

eas to display detailed information about the `training set` used for a model, and its statistical performance on a test dataset. However, usually only a URL is provided, without deeper discussion of the expected data schema. Furthermore, different performance metrics are used for different models, even for products within the same domain. For example, some image classification models used the overall accuracy metric under 10-fold cross validation, whereas others used "top-1"/"top-5" accuracy under 2-fold cross validation. Very few ($\sim 3\%$) AWS models provide performance results and such information is not presented in a structured manner. Whenever a model is updated to a newer version, it is important to document the changes (e.g., feature updates or bug fixes) in a `release notes` document. However, such information is missing or poorly presented in model stores. Although AWS contains release notes, they are usually very brief with only one or two sentences. ModelDepot does not contain `version number` and `release notes`.

- The **Delivery** dimension contains two elements related to the installation and configuration of ML models. Running ML models usually requires specialized hardware (e.g., GPU) or high performance servers. Furthermore, installing and configuring the needed software components for an ML model is a non-trivial task. Hence, all three model stores provide `deployment instructions`. AWS and Wolfram provide dedicated cloud infrastructure

to run all their models, which greatly eases the deployment of these ML models for users. While ModelDepot also provides cloud support, it currently only supports one model.

- The **Business** dimension contains three elements related to the business aspects of the products. All three model stores contain `price` information for their products. This information usually includes the costs of using the store's cloud infrastructure (e.g., VMs and ML APIs). However, the pricing scheme is rather complex and not directly tied to the usage context of end users. For example, AWS charges users on the cloud VM infrastructure and the usage of the model package for training and predicting. Without any performance estimations (e.g., the duration of training/prediction under a particular setup), it is not clear how much one user will be charged for their tasks. The `usage statistics` are missing in AWS and Wolfram. Although ModelDepot provides the number of downloads for each model, it did not provide any information about the types of infrastructure nor the number of API calls for individual products. Such information would be very valuable for software engineers to scale and optimize their ML applications.

- The **Product Submission & Store Review** dimension contains the information related to submission of a model to the store and to review feedback

from the stores. The submission process for AWS and ModelDepot just requires to upload a model online, whereas developers have to contact the store owners of Wolfram in advance to arrange the model submission. None of the three model stores contain any publicly available development policies regarding product reviews and approval.

- The **Legal Information** dimension contains elements related to licensing information of this product. For example, all model stores contain `end user licenses`. The majority of AWS products are developed by commercial companies, whereas the most of the models from Wolfram and ModelDepot are based on research prototypes or open source projects. As such, those models usually adopt open source licences (e.g., Apache or MIT licenses), which allow users to access the models' source code to further modify or extend them.

3.1.2.2 Comparison between Model Stores and App Stores

When comparing the elements between the app and the model stores, we only focused on the elements missing in either all model stores or in both app stores. Under the **Technical Documentation** dimension, there are one unique element in model stores and three unique elements for app stores.

- The `Demo` element only exists in ModelDepot and is missing in all other model

stores and app stores. Compared to mobile apps, which are meant to be downloaded on mobile devices and hence are harder to disable after the expiry of the demo, it is much easier to support model demos as models are meant to be deployed in a container/server.

- Although ML products require access to various computing resources (e.g., images/videos/audio), `permission`, the list of required computing services (e.g., microphone) or data (e.g., calendar), of models are not explicitly documented. The former can be derived through trial-and-error, or by skimming through the annotated scripts. The content of some of the ML products might not be suitable for certain users (a.k.a., `age` or `language`). For example, one model in Wolfram is about determining whether an image contains pornographic content.

Except `release notes`, all other elements under the **Technical Documentation** are missing in both app stores. The difference is mainly due to their different target audience (software engineers vs. end users). Apps are products targeted towards the general population, and hence come with a rich GUI. Furthermore, most of these apps provide in-app tutorials when users initially launch them. In contrast, models generally do not come with a GUI but instead correspond to APIs or components that require programming in order to integrate them into an application.

Instead of detailed developer documentation, models usually provide sample usage in form of annotated scripts.

The elements under the **Delivery** dimension are completely disjoint, with two elements only present in model stores, and two only in app stores. Such differences are mainly because the automated product deployment techniques differ between two types of products: apps installed on the users' devices (app stores) or models on the providers' cloud infrastructure (model stores). The `deployment instructions` and `cloud deployment` information are provided for all model stores, while all app stores check compatibility of apps with the user's device and allow one-click purchase/installation of apps.

Although all elements under the **Business** dimension exist in both types of stores, the `pricing` information is presented differently. For model stores, the pricing is usually subscription-based or pay-per-use, whereas mobile apps have a wider range of pricing schemes (e.g., entirely free, one-time purchase, and in-app purchase). The `store review policy` under the **Submission & Review** dimension is missing in model stores. As more models are being introduced into the stores, such policies will be needed to protect users and developers. Similar to the **Technical Documentation** dimension above, the `developer license` element is missing under the **Legal Information** dimension for app stores.

3.1.3 Implications

- **Emerging Practices**: Since model stores have been introduced recently, only 65% of the information elements are common across the three model stores. For example, ML model information elements related to technical details like ML algorithms, type of training datasets and cloud deployment, are supported across all three stores. However, some other elements (e.g., demo or release notes) are only present in one or two stores. It would be interesting to study the evolution of the information elements from the model stores, as they are being used by more organizations/developers.

- **Target Audience**: Both model and app stores have several unique elements. For example, model stores contain common usage for each ML model whereas app stores contain age ratings for different apps. This is mainly due to their different target users: app stores for end-users and model stores for organizations/developers.

- **Reviewing Policy**: Some important elements in app stores are currently missing in model stores. In particular, there is no clear policy for submitting and reviewing ML models before they can appear in the model stores. The reviewing of ML models is a very challenging task and requires further research in the following three areas: (1) Requirement specification: Mod-

els used in different context (e.g., health care vs. gaming) need different quality thresholds in order to be usable or safe. For example, how should the safety requirements for radiology-related prediction models be defined?, and (2) Automated monitoring mechanisms: To evaluate the safety and the correctness of different ML models under submission, automated monitoring mechanisms are needed; and (3) Standard quality measurement: common performance measures of ML models are needed as indictors of qulity of service (QoS) to enable users to compare among similar product offerings.

- **Hidden Bias**: Each model in the model store contains three components: the source code, the training dataset, and the trained model(s). However, little information is provided regarding the underlying data distributions and the steps for data pre-processing of the training set. Yet such information is very important in order to identify and remove the hidden bias in models. For example, the deployed ML models can perform poorly, if the images used during training are high resolution images and lower resolution images taken from mobile phones are used in production. Further research is needed to assist organizations/developers to properly identify, report, and remove such bias in model stores in order to yield satisfactory performance of ML applications.

3.2 RQ2: How unique are the models provided by each model store?

In this RQ, we first identified different types of models in model stores, then compared them across model stores.

3.2.1 Approach

In order to obtain information about all models offered by the three studied model stores, we first developed a model store crawler. Since each model store has a different structure (JSON for AWS, HTML sections for Wolfram and ModelDepot) and displays its data differently (typically using Javascript to dynamically reveal information), we had to write a different crawler for each store leveraging headless Chrome to obtain the dynamic store content. Using the manually labeled information elements used in RQ1, we developed parsers to automatically extract the sections of each store. In this RQ, we studied the most recent snapshot obtained using our crawlers at the time of this study (mid March 2019).

3.2.2 Findings

3.2.2.1 Quantitative Analysis

Different model stores have different heuristics to group their models. AWS labels each model using seven criteria (e.g., input and server location) and each model can be under multiple criteria. For example, the input for a computer vision model in AWS can be image(s) or video(s). The model can be deployed in US East or Europe.

After manually studying the grouping criteria of each model store, we decided to group the models based on their input data domain for the following two reasons: (1) it is a common criteria among three stores; and (2) each model can only belong to one input domain. Table 3.5 shows the number of models under each group.

AWS has the largest number of models, followed by Wolfram, and ModelDepot. AWS is the only model store with models in all five groups. Neither ModelDepot nor Wolfram contain any models in `structured data`, while this group contains the majority (45%) of AWS models. The majority of the ModelDepot (75%) and Wolfram (75%) models are focused on images, which is the second largest group in AWS (27%). All three models stores contain only few models in the `audio` and `video` group.

Since RQ1 shows that models in the model stores provide their technical doc-

umentations, we manually went through each model to track their `origin`. As a result, we found that 91% of the Wolfram and 72% of the ModelDepot models refer to 34 and 20 academic papers, respectively. One paper/URL may correspond to multiple ML models even in the same store. For example, we found two different models in ModelDepot using the same implementation of one research prototype, but were trained on two different datasets and used in two different contexts: gender recognition and emotion classification. Similar cases also exist in Wolfram. Very few (4%) models in AWS referenced academic papers, each of which corresponds to different AWS models. Seven models in Wolfram and three in ModelDepot do not contain paper references but GitHub URLs for the model implementation.

3.2.2.2 Similar Offerings of ML Models

We considered two ML models from different model stores as similar offerings (a.k.a., similar models), if they share three information elements in common: ML algorithms, training datasets, and objectives. For example, all three model stores contain an image classification model that uses the same algorithm (ResNet50) and training dataset (Imagenet). Most ML models have such information elements in their individual product webpage. Note that similar ML models may not be exactly identical. For example, although two similar ML models use the same ML algorithms, their underlying implementations can be different. Table 3.5 shows the

Table 3.5: The breakdown of ML models under different model stores. Note that AWS contains 231 URLs, each of which corresponds to one model. Yet there are three models that have two URLs for their two different versions. This brings the number of AWS models to 228. We considered two models from two different stores as similar offers, when the ML algorithms, the training dataset(s), and the objective(s) are the same.

Group		**AWS**	**ModelDepot**	**Wolfram**
Image	Count	61 (27%)	24 (75%)	59 (75%)
	Similar	2 (0.8%)	6 (19%)	7 (9%)
Video	Count	13 (6%)	2 (6%)	0 (0%)
	Similar	-	-	-
Natural language	Count	35 (15%)	5 (16%)	18 (23%)
	Similar	-	1 (3%)	1 (1%)
Audio	Count	12 (5%)	1 (3%)	2 (2%)
	Similar	-	-	-
Structured	Count	107 (47%)	-	-
	Similar	-	-	-
Total	Count	228 (100%)	32 (100%)	79 (100%)
	Similar	2 (0.8%)	7 (22%)	8 (10%)

results. there are only two similar models in AWS to the other two stores, whereas ModelDepot and Wolfram had seven and eight common models. Most of the similar ML models were found under the **image** group.

3.2.3 Implications

- **Product Maturity**: More than 70% of the models from ModelDepot and Wolfram are based on research prototypes. This demonstrates the practical impact of current AI research, which can be converted into production-ready models in a relatively short time-frame. It would to be interesting to track their future development activities of such models to understand the unique challenges and opportunities for maintaining and evolving ML models.

- **Cross-store Support**: The amount of similar models across different model stores is very small. This is mainly due to vendor lock-in. Migrating one model to different stores requires adapting it to different frameworks, like SageMaker for AWS and the Wolfram language for Wolfram. Similar to mobile app stores, cross-platform frameworks for developing and maintaining ML models are needed.

3.3 Threats to Validity

External Validity We studied a variety of models and mobile apps across the three popular model stores and two popular mobile app stores. Among the three studied general purpose model stores, AWS models mainly are developed by commercial companies, whereas the majority of models from Wolfram and ModelDepot are derived from research prototypes. Google Play and Apple's App Store are two of the most popular app stores currently, hosting millions of applications. Despite this variety in models and apps, future work should consider other model and app stores.

Internal Validity Since we did not claim any causality, there are no threats to internal validity.

Construct Validity We used the paper titles, model links, training datasets, and application descriptions as our criteria for deciding whether or not two models are the same, as most of the models from Wolfram and ModelDepot are originated from research prototypes or open source projects. Two models with different usage contexts (e.g., detecting gender vs. detecting emotions) are considered as different applications, even though they could share the same research references. Furthermore, if more ML applications, especially the ones developed by commercial companies, are added into the model stores in the future, these models will contain

sparse references to papers/URLs. Our current approach will not be able to extract the full list of overlapping applications

3.4 Summary

This chapter presented the exploratory study on ML model stores. We first empirically compared the information elements among three model stores and two app stores. Since model stores have been introduced fairly recently, only 65% of the elements are common among model stores. We found some elements (e.g., cloud deployment and user instructions) which are unique to the model stores. Certain elements (e.g., review policy) which are presented in app stores are missing in model stores. Further studies of the models inside the three model stores showed very few offerings of the similar ML models among the model stores, with the majority of the ML models from Wolfram and ModelDepot originating from research prototypes. In the future, better support for effective reviewing of ML models in terms of safety and quality are needed in model stores. Before integrating into ML applications, automated methods are needed to detect, report, and remove hidden bias in pre-trained ML models.

4 Empirical Study on the Software Engineering Practices in Open Source ML Package Repositories

In this chapter, we conducted an exploratory study on the practices of sharing reusable ML models via the ML package (ML packages contain one or multiple ML models One or multiple ML models can be packaged into a package) repositories. We focused on the the following three research questions (RQs).

RQ1 - What types of information are presented on software package repositories and ML package repositories? Given the relatively short existence of ML package repositories, this RQ (Section 4.1) aims to provide us with insights on the structure (e.g., the organization of packages by task types) and practices (e.g., release management) of the ML repositories, as well as to discover any missing or non-formalized information elements (based on the comparison with their counterparts in the software engineering domain). Such discoveries will be

helpful in building a better ML package repository in the future in the sense of providing more information transparency, benefiting more users, especially software engineers without solid background in ML.

RQ2: How are packages organized in ML package repositories? Next, we investigated the organization practices of ML package repositories, in Section 4.2. More specifically, we studied the family phenomenon within these repositories, and its implications on package task type distribution, package similarity, and release management.

RQ3 - What is the process needed in order to use the functionalities from software/ML package repositories? Finally, we studied the functionalities of the package managers (tools and libraries) provided by the ML package repositories and how they are used in Section 4.3. These ML package managers, provide functionalities that allow users to explore, manage and use the ML packages. In this RQ, we aimed to discover the unique practices of the ML package managers by comparing them with the practices of traditional software package managers. The findings and implications of this RQ may point out if there are any practices of traditional software package managers that can be adopted to improve the functionality of ML package managers.

For each RQ, the first section describes our approaches, the second section presents our findings, followed by the third section discussing the results and their

implications. However, before addressing the RQs, we first discuss the five repositories selected from Table 2.1 and Table 2.2 as the subjects of our exploratory study here.

Among the ML package and software package repositories mentioned in Section 2.2, our case study focused on npm, PyPI and CRAN as software package repositories, and TFHub and PyTorch Hub as ML package repositories. The rationales of selecting these repositories are as follows:

- npm, PyPI, and CRAN respectively are the top three "mono-language" software package repositories in terms of the number of packages they host, based on the statistics presented in Table 2.1. They cover JavaScript, Python and R, respectively
- TFHub and PyTorch Hub are the official repositories of TensorFlow and PyTorch, the most popular ML frameworks in academia and industry [39, 40].

4.1 RQ1: What types of information are presented on software package repositories and ML package repositories?

In this RQ, we aimed to understand the types of information presented in the software package and ML package repositories. We focused on the information

elements (IE), each of which describes one aspect of the packages or the repository, e.g., the basic description of the package, the dependencies of the package, hyper-parameter value settings, etc. By comparing the IEs in ML and software package repositories, we learned the structure of the repositories, as well as the missing and new/additional IEs needed by ML packages and ML package repositories. In what follows, we presented the methodology used to achieve this (based on Chapter 3), and a discussion of our findings.

4.1.1 Approach

Here we explained our process of extracting IEs from different repositories. This process is based on the process that has been used in Chapter 3

We referred to several sources to determine the full list of IEs per repository. The fundamental source was the website of the packages. Secondary sources such as the software package description/documentation files[5,6,7], ML package contribution instructions[8,9] and the JSON data structures generated upon loading a package website were also analyzed.

The author of the thesis independently analyzed the information sources in a

[5]https://docs.npmjs.com/files/package.json

[6]https://github.com/pypa/sampleproject/blob/master/setup.py

[7]https://cran.r-project.org/web/packages/policies.html#Source-packages

[8]https://github.com/tensorflow/hub/blob/master/tfhub_dev/README.md

[9]https://github.com/pytorch/hub/blob/master/docs/template.md

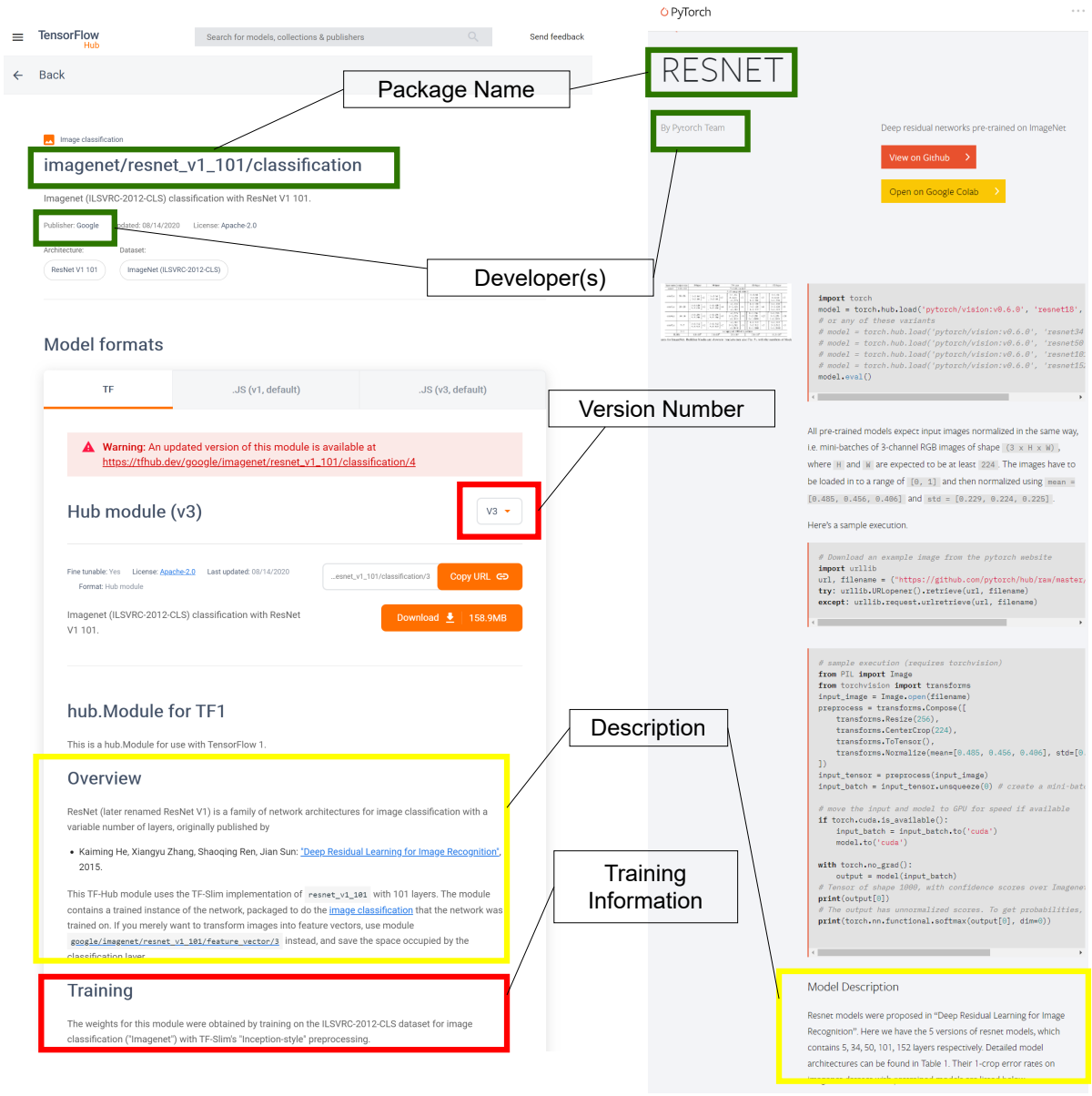


Figure 4.1: An example comparing the IEs in TFHub (left) and PyTorch Hub (right) packages. The example highlights the same IE in both repositories (green), similar IEs with different representations (yellow), and IEs unique to only one repository (red)

first iteration to identify 39 IEs. Following several discussions and 6 more iterations of analysis involving his supervisor and two other SE colleagues (Ellis E. Eghan and Bram Adams), a final set of 33 IEs was agreed upon. The first iteration of analysis by the authors achieved an IRR score (based on the Cohen Kappa coefficient [41]) of 94.87%, and increased to 100% at the last iteration.

It is normal for repositories to present the same IE differently, especially by referring to the same IE by different terms. For example, Figure 4.1 shows how IEs are presented on sample packages on the TFHub and PyTorch Hub repositories. In our approach, we manually unified the IEs that are essentially the same. Furthermore, we grouped related IEs into **dimensions**. We referred to the previous work of Bommarito et al [25] to verify the sanity of IEs. In their work, the authors used several IEs extracted from PyPI to study the repository's evolution and the distributions of important statistics (package release numbers, authors' package numbers, package import numbers, sizes, etc.). The IEs (packages, releases, dependencies, category classifications, licenses, package imports, authors, maintainers, and organizations) investigated by Bommarito et al. can all be extracted from our analyzed repositories, except for package imports, which is an internal property of packages and not related to this RQ.

The final list of extracted IEs are presented in Table 4.1 to Table 4.4. Each

row is an IE, and the ✓ represents its presence in the associated repository[10]. For example, in the **Package Information** dimension, the **Demo** row shows that npm, TFHub and PyTorch Hub have this IE while PyPI and CRAN do not. IE dimensions are ordered alphabetically, and the IEs in every dimension are implicitly grouped as follows. The first group contains IEs that belong to all five repositories. The second group contains IEs that exist only in software package repositories (at least one). The third group contains IEs that exist only in ML package repositories (at least one). The fourth group contains other IEs in the dimension that cannot be classified into the aforementioned groups, i.e., IEs that belong to some of the ML repositories and some of the software repositories. Within each implicit group, the IEs are ordered alphabetically.

4.1.2 Findings

4.1.2.1 Comparison between Software Package Repositories and ML Package Repositories

We analyzed the common and unique IEs between the two types of repositories: software package and ML package repositories. It should be noted that if we say a type of repository has an IE, it means that this IE appears in at least one of the

[10]The IEs are extracted from software package and ML package repository pages at the end of March, 2020.

Table 4.1: Information elements among the three software package repositories and two ML package repositories. Information elements in bold are unique to either software package repositories (at least one) only or ML package repositories (at least one) only.

Dimension	Element	SW Pkg Repo (npm)	SW Pkg Repo PyPI	SW Pkg Repo CRAN)	ML Pkg Repo (TFHub	ML Pkg Repo PyTorch Hub)	Description
Delivery	Dependencies	✓	✓	✓	✓	✓	The packages this package depends on. Without them, this package cannot function normally.
	Running Environment	✓	✓	✓	✓	✓	Information related to usage context. E.g., multiple formats, hardware and software environment requirements.
	Dependents	✓		✓			The packages that depend on this package. Without this package, those packages cannot function normally.
	Downloadable Provided		✓	✓	✓		The package can be downloaded as files or a zipped file other than using deployment, installation or initialization command
Legal Information	License	✓	✓	✓	✓		Regulations on how users can use this package.
	Copyright			✓			Information claiming the owner of the copyright (not necessarily the author).

Table 4.2: Continued Table 4.1

Dimension	Element	SW Pkg Repo (npm)	SW Pkg Repo PyPI	SW Pkg Repo CRAN)	ML Pkg Repo (TFHub	ML Pkg Repo PyTorch Hub)	Description
Package Information	Description	✓	✓	✓	✓	✓	The objectives, functionalities and other basic information of this package.
	Developer(s)	✓	✓	✓	✓	✓	Owner, publisher or collaborators of this package.
	Extra Information	✓	✓	✓	✓	✓	Other information related to this package, e.g., homepage links, more resources, academic references.
	Indexing Keywords	✓	✓	✓	✓	✓	Keywords of this package that can be used to search similar packages. Classify and select packages according to keywords.
	Package Name	✓	✓	✓	✓	✓	Name of the package.
	# Downloads	✓					Number of downloads of this package.
	GitHub Statistics	✓	✓				E.g., # PRs, # Issues, # Stars, # Forks.
	Demo	✓			✓	✓	A functionality through which users can try before downloading/deploying the package.
	Published Time	✓	✓	✓	✓		The publish date of this version.
	Size	✓			✓		Package's disk size.
	Version Alert	✓	✓		✓		Information indicating that this package is either not the latest version or not suitable for use.
	Version Number	✓	✓	✓	✓		The version number of the current release.

Table 4.3: Continued Table 4.2

Dimension	Element	SW Pkg Repo (npm)	SW Pkg Repo PyPI	SW Pkg Repo CRAN)	ML Pkg Repo (TFHub	ML Pkg Repo PyTorch Hub)	Description
Package Submission & Review	Developer Contribution	✓	✓	✓	✓	✓	General developers have access to submit and publish packages.
	Review Mechanism			✓	✓	✓	Review process for submitted package. E.g., policies, pull request review.
Security	**Vulnerability Report**	✓					Report a security vulnerability for this package.
Software Development	Issue Tracking Information	✓	✓	✓	✓	✓	Information for reporting issues. E.g., GitHub repository issue link.
	Source Code Repo	✓	✓	✓	✓	✓	The GitHub or other repository for this package.

Table 4.4: Continued Table 4.3

Dimension	Element	SW Pkg Repo (npm)	SW Pkg Repo (PyPI)	SW Pkg Repo (CRAN)	ML Pkg Repo (TFHub)	ML Pkg Repo (PyTorch Hub)	Description
Technical Documentation	User Instruction	✓	✓	✓	✓	✓	Instructions on how to use this package. E.g., example code snippets.
	Package Component Information	✓	✓	✓			Information about the package's components. E.g., source code file location, data file location.
	Algorithm				✓	✓	Information describing the algorithm of this package. E.g., neural network architecture.
	Data Description				✓	✓	Information about the dataset. E.g., dataset name, IO data shape, data pre-procession.
	Package Quality Evaluation				✓	✓	Information about how good the package is.
	Training Information				✓		Details about training this package. E.g., the training checkpoint file used, hyper-parameter settings.
	Pre-defined Interfaces	✓	✓		✓	✓	E.g., the entrypoint(s) of the program, the main file/ function of a package, a special function related to a command.
	Package Domain	✓	✓		✓	✓	The application domain or task type of the package, e.g., `image classification` for ML packages, `front-end` for software packages.
	Release History	✓	✓	✓	✓		Accessible old versions of this package.
	Release Notes			✓	✓		Information explaining the changes that have been made for each release.

repositories of this type. For example, since npm has `Demo`, we say software package repositories have this IE. By comparing between the two types of repositories, we identified what meaningful IEs are missing in ML package repositories.

In the **Delivery** dimension, both types of repositories have four IEs each, with three IEs (`Dependencies, Running Environment, Downloadable Provided`) in common.

While software package repositories usually provide a special area for dependencies on the package's webpage, dependencies in ML package repositories are often in free-text within the package description. Furthermore, ML packages' dependencies are usually Python packages that can be installed from PyPI, e.g., dependency software packages of ML packages typically are related to data processing like `opencv-python` [11], `tensorflow-text` [12], rather than other ML packages.

Although the `Dependents` IE is unique to software package repositories, we found evidence of its possible inclusion in ML packagage repositories given the observed dependencies between ML packages such as `llr-pretrain-adv-latents` and `llr-pretrain-adv-linear`. The output of the former is the input of the latter, and the output of the latter ML package can be used as the basis for classification.

[11] https://pypi.org/project/opencv-python/

[12] https://pypi.org/project/tensorflow-text/

In other words, to complete the classification task, two ML packages must be used in combination. PyTorch Hub also has similar examples. The output of ML package `Tacotron2` is used as the input of ML package `WaveGlow`. The two ML packages can be used together to complete the text to speech task. Because the existence of such examples, it is worthy to consider adding a special dependents/dependencies IE to the ML package page to illustrate the interrelationship between ML packages.

In the **Legal Information** dimension, `License` is the only common IE between both types of repositories. Packages on both type of the repositories are licensed. Software package developers need to explicitly specify a license in the description file of the software package. In ML package repositories, however, packages bear the default license unless specially declared.

If the `Copyright` belongs to people other than the author, the copyright holder is also needed to be specified. This IE is only found in the CRAN software package repository.

In **Package Information**, Software package repositories have 12 IEs, including all ten IEs that ML package repositories have.

The `Developer`s of software packages tend to be individuals while the ML package owners are usually organizations. Generally, software packages are less likely to be connected with an organization. Previous studies show that 75% of npm pack-

ages are published by individual developers [42], and only about 5% of PyPI package authors are organizations [25]. Conversely, we observe that both the TFHub and PyTorch Hub repositories each contain only three ML packages published by individual developers.

There is no formal versioning mechanism in ML package repositories. Software package repositories mostly adopt the semantic versioning format, e.g., `x.y.z` where `x` means a major change, `y` means a minor change and `z` means a patch [30]. In TFHub, the version number is simply represented by integers like version `1`, `2` and `3`. However, there is no versioning mechanism in PyTorch Hub.

`# Downloads` and `GitHub Statistics` are indicators of the popularity of packages, which are completely missing in ML package repositories. Such an IE can reflect the wide usage of package and its huge possibility of satisfying the need of most developers. This will be helpful for ML package users, especially for people without solid ML expertise (like general software engineers), but might not be of much use to experienced data scientists and ML researchers.

All the IEs in the **Package Submission & Review** dimension are found in both types of repositories. However, different mechanisms are used in both types of repositories — software package repositories have special command line tools and review processes while ML package repositories' contribution is based on GitHub

pull request. For example, npm and PyPI users use command line tools to submit their locally developed projects to the package repositories. Successfully submitted packages do not undergo any further review and are made immediately available for other users (except for CRAN, which has specific contribution policies[13]). In contrast, the contribution of ML package repositories is based on GitHub pull requests. Everyone can contribute their ML packages to TFHub and PyTorch Hub by creating a pull-request. Pull request-based submission mechanism can be a good point for ML packages, since pull requests usually go through a specific review process before they are merged into code base and afterwards being available to users.

In the **Security** dimension, only the npm package repository provides the `Vulnerability Report`. Though ML packages may also suffer from vulnerability issues, no such IE is currently provided in ML package repositories. According to Wang et al. [43], attackers can use the publicly available knowledge (algorithms, dataset, architecture, etc.) of pre-trained models to create vulnerabilities to undermine the performance of dependent models (models built based on existing pre-trained models). In their work, Wang et al. demonstrated how vulnerabilities can be created for models pre-trained on the ImageNet [44] dataset using the VGG [45] and ResNet [46] algorithms. Due to the existence of similar packages/models on

[13]https://cran.r-project.org/web/packages/policies.html

TFHub and PyTorch Hub, it is extremely important for ML repositories to provide users with vulnerability detection and reporting functionalities.

In the **Software Development** dimension. ML package repositories and software package repositories share all of the IEs. Software package repositories provide the GitHub Issue link of a package for users to report issues. This practice is different from the `Issue Tracking Information` of ML package repositories (described further in Subsection 4.1.2.2). All of the software package repositories, just like PyTorch Hub, provide the `Source Code Repository` GitHub links of software packages/ ML packages in a fixed area on the package's page.

In **Technical Documentation**, software package repositories have six IEs while ML package repositories have nine. Among those IEs, `User Instruction, Pre-defined Interfaces, Package Domain, Release History, Release Notes` are common in both types of repositories.

The `Release Notes` are organized differently in the different types of repositories. For example, CRAN's release note can either be a GitHub commit message or an HTML page containing all the correlated information. In TFHub, the release notes will be directly presented at the end of the description and they generally follow a certain format.

ML package repositories have four unique IEs. The IEs `Algorithm, Data Description, Package Quality Evaluation, Training Information` are all ML related and need a certain amount of ML expertise. Such IEs provide transparency to ML package users. Unlike ML packages, most software packages are not developed based on a single particular algorithm or dataset. Thus, their implementation details are rarely a concern for users; users just access the functionality of the package through its provided APIs.

While software packages are considered of high technical quality based on the proportion of passed test cases, ML packages' technical quality is determined by statistical evaluation criteria. This indicator is not in a formalized structure across ML package repositories (e.g., there is no fixed area on an ML package's site to show its performance) and it requires the users to have a relatively good understanding of the related ML task types (e.g., ML packages in image classification task type use top-1 or top-5 accuracy to evaluate their performance, ML packages in object detection task type use intersection over union). In our previous study on model stores in Chapter 3, we found that limited information in terms of IEs like source code and training dataset (like how dataset are pre-processed) may introduce the hidden bias to the model re-usage. Thus, providing more ML implementation-related information and making such information more detailed and clearly organized will help users better understand use and modification of the ML packages easier.

Software package repositories do not directly provide package quality evaluation information. The quality of a software package is reflected through its usage statistics. So software package repositories may need to add a quality evaluation IE to show the test coverage results or CI results (usually captured in the external websites of software packages) within the repository for users.

4.1.2.2 Comparison among ML Package Repositories

There are 28 IEs (belonging to six dimensions) in at least one of the ML package repositories. Among them, 16 are common in both of the ML package repositories. We presented a detailed analysis of the results of each dimension below. By doing these comparisons between ML package repositories, we identified the common IEs of both ML package repositories and what unique IEs of one repository should also be possessed by another one.

In the **Delivery** dimension, ML package repositories have two IEs in common (`Dependencies` and `Running Environment`) while the `Downloadable Provided` IE is unique to TFHub.

In TFHub, the running environment IE describes the format(s) of the package and version of TensorFlow required to load the package. TFHub packages can con-

tain models of different formats e.g., general package formats (hub.Module, TF2 SavedModel), format for TensorFlow on JavaScript, and format for deployment on edge device (computational equipment that have limited computational resource, like mobile phone). The format of a packages/model determines its usage context (more details are in Section 4.3.2.3). PyTorch Hub packages do not provide the packages that are suitable for multiple usage context. However, they use the running environment IE to indicate if an ML package needs accelerator support like GPU and CUDA [47](NVIDIA parallel computing architecture for GPU); having an accelerator can make a big difference in ML efficiency.

Only TFHub provides links for directly downloading the package files in multiple formats.

Legal Information contains a single IE related to licensing. According to the TFHub contribution tutorial, if no license is specified, the default license for an ML package will be Apache 2.0 [48]. No information was found about the licenses used by PyTorch Hub packages.

Package Information. ML package repositories have ten IEs in this dimension, with `Description`, `Developer(s)`, `Extra Information`, `Indexing Keywords`, `Package Name`, `Demo` are common between the two repositories. The `Description`

are actually in freestyle and may contain other IEs. ML packages generally will provide academic papers and GitHub links as `Extra Information`. These academic papers are usually the original sources of the algorithms used by the ML packages. So this IE can be helpful for users like data scientists and researchers who may benefit from dig into the basic principle of the algorithms. As for the `Indexing Keywords`, they can be task types (text embedding, image classification, etc.), datasets (ImageNet, etc.), algorithms (CNN, Transformer, etc.) and some other ML attributes that help users narrow down the search scope. The `Names` of TFHub ML packages are more complicated than PyTorch Hub ML packages, because the former contain not only the key algorithm names but also the names of dataset or configuration values used during training. Their `Demo` is supported by Google Colab, an online Python notebook environment. These example notebooks usually contain complete use cases of this ML package.

There are four other IEs unique to TFHub. Due to Pytorch Hub's lack of a versioning mechanism, the `Version Alert, Version Number`, and `Publication Time` are missing. Also, there is no `Size` information in PyTorch Hub. We discuss the implication of this lack of a formalized versioning mechanism in Section 4.1.3.

Both IEs in the **Software Development** dimension are found in the two ML package repositories.

Regarding the `Issue Tracking Information` IE, PyTorch Hub provides a link to its GitHub issue page while TFHub provides a form for users to submit any kind of feedback directly from the repository. In both ML package repositories, users can also report bugs identified in either the frameworks or ML packages on the `tensorflow_hub` library's GitHub repository[14] and PyTorch Hub's GitHub repository[15].

Though both repositories also provide links to the `Source Code Repository`, most packages in TFHub include these links as freestyle text in the package description (not in a dedicated area like in PyTorch Hub), making this information difficult to identify.

The **Technical Documentation** dimension has six common IEs (`User Instruction`, `Algorithm`, `Data Description`, `Package Quality Evaluation`, `Training Information`, `Package Domain`) in both ML package repositories. However, these common IEs are usually not organized as independent IEs or formally presented; they are a part of the ML package description.

The `Package Domain` can usually be the indexing keywords of the ML packages. It contains ML application domains (computer vision, natural language processing,

[14]https://github.com/tensorflow/hub

[15]https://github.com/pytorch/hub

etc.) and ML task type (image classification, text embedding etc.). Note that ML package repositories do not have `Package Quality Evaluation` in all of their ML packages. Only 17 (out of 26) and 35 (out of 383) packages in the PyTorch Hub and TFHub repositories, respectively, provide their quality evaluation result. In TFHub this IE is also a part of the general package description rather than independent IE.

In addition to the shared IEs, both ML package repositories have unique IE(s). Because of the lack of versioning mechanism, PyTorch Hub does not have `Release History` and `Release Note`. As for `Training Information`, TFHub ML packages may provide more detailed algorithm information like the model optimizer arguments[16]. PyTorch Hub usually does not provide such details, but it has a unique area in a ML package page for listing the `Pre-defined Interfaces` which are the entrypoints. Entrypoint is a mechanism in PyTorch Hub to manage variant models within a single ML package. Users specify an entrypoint when loading a PyTorch package to get the needed variant of a model. On the other hand, TFHub ML packages do not have entrypoints but rather utilize a signature mechanism. The signature mechanism is used by TFHub packages to organize combinations of input and output tensors (basic data structure in ML). These two mechanisms will be explained in detail in Section 4.3.

[16]https://tfhub.dev/deepmind/spiral/default-fluid-gansn-celebahq64-gen-19steps/1

Only 173 TFHub packages (out of 383) introduced signatures (e.g., telling users about what this signature can do, what hyper-parameter it needs) in their description section. Though the signatures are not listed formally, the TFHub users can get a full list of the signature they supports by calling an API of the packages. In PyTorch Hub, only 10 (out of 26) packages provide a list of their entrypoints within their package pages.

4.1.3 Implications

Dependency Management. ML package repositories currently assume that, unlike software packages with several dependencies and dependents, ML packages rarely depend on each other but rather depend on existing Python packages and the core ML frameworks. However, our analysis was able to identify dependencies between ML packages (as discussed in Section 4.1.2.1).

Hidden dependencies are a major risk for developers. Dependency-related information helps developers to make a better estimation of the effort needed to upgrade to a given software/ML package. For example, users may be concerned of the risk of introducing bugs or breaking changes during an upgrade, as well as the extra work needed to make their current dependencies compatible with the new dependencies introduced by the included software/ML package. Thus, dependency-related IEs

may have an impact on when and whether developers decide to upgrade the ML package.

Release Information. TFHub implements a basic incremental versioning mechanism (version numbers and release notes) while PyTorch Hub has no mechanism in place. Versioning helps users learn whether the ML packages have been updated and whether it is worthy to upgrade to a new release. ML package repositories can adopt a practice similar to the semantic versioning of software package repositories to indicate the severity of changes and backward-compatibility of APIs (more details on this discussion in versioning is provided in Section 4.2). Given the numerous points of change in ML packages (e.g., algorithm, training dataset, configurations, etc.), a consensus among ML package stakeholders would have to be reached on the definition of a major and minor changes, as well as patches.

Popularity Indicator. Experienced users of ML packages can tell which ML package is better by looking at the performance evaluation result, while users without solid ML expertise (like general software engineers or researchers not in ML area), however, may refer to information such as the popularity, reviews and the quality of technical functionalities when deciding on a ML package or software package to use. Intuitively, such users may choose the packages with the most downloads

(popularity) [49] or the most positive reviews from other users (common in some traditional software engineering repositories like mobile app stores) [50].

Such indicators make it easier for users who do not know how to differentiate between algorithm performance and datasets to know which ML package is the most popular or has a good reputation. So the quality indicator information elements are highly recommended to be provided by ML package repositories.

Security. TensorFlow, PyTorch and most of the other popular ML related libraries are mainly Python-based and can be installed through PyPI. However, since PyPI does not provide any submission review and security vulnerability report mechanism, this increases the quality and security risks of ML packages. These security risks are hard to discover. For example, an ML engineer may expend much effort to locate a bug in the model's source code, but the bug may actually originate in an imported Python library installed from PyPI.

Although the bugs may originate from the external Python dependencies, there are several vulnerability analysis tools (e.g., WhiteSource [51], snyk [52]) that ML developers need to include in their workflow to identify the propagation of vulnerabilities from dependent packages into the ML application.

Technical Documentation. There is no formalized (or unified) structure or or-

ganization of the technical information within ML package repositories. ML packages have some unique technical documentation like algorithms, dataset description, training details, ML package tune-ability and ML package quality evaluation. Given this lack of formalism, users of ML packages need to read the documentation or description of packages in order to extract such information. These technical documentation vary in style and form, making it difficult for users to understand and compare ML packages. It would be beneficial if packages within a ML package repository are required to follow a documentation standard that ensures the same structure of information elements across different packages.

4.2 RQ2: How are packages organized in ML package repositories?

In Section 4.1, we identified and compare the IEs presented in software and ML package repositories, while this RQ is specific to the domain of machine learning (packages). In this RQ, we went one step further by investigating some of the major IEs (e.g., task types, algorithms, datasets) and the inter-relationship (e.g., similarities, distributions in different task types) of ML packages within the ML package repositories. In particular, the analysis performed in this RQ is centered

around a unique phenomenon in ML package repositories: ML package/model families, which are groups of packages/models that are similar with each other in terms of task types, algorithms and datasets. Thus, through the study of this RQ, we provided details on the organization practices within such ML package repositories and provide the users with information about what kind of packages to expect in each ML repository.

4.2.1 Approach

Using web crawlers and custom scripts (see our replication package [53]), we extracted the IEs from the JSON data structures (generated by each ML package's individual page) and webpages of each package in the TFHub and PyTorch Hub repository.

It should be noted that PyTorch Hub provides models in PyTorch's general formats (.pt or .pth files) only. So in order to perform a fair comparison, we only considered the two general formats of TFHub, i.e., hub.Module and TF2 SavedModel. For example, we did not consider TFHub packages that are not provided in either hub.Module or TF2 SavedModel formats (e.g., `mobilenet_v2_1.0_224_quantized` [17]). However, if a package provides extra formats in addition to the two general ones

[17] https://tfhub.dev/tensorflow/coral-model/mobilenet_v2_1.0_224_quantized/1/default/1

(e.g. `imagenet/mobilenet_v2_075_96/feature_vector` [18]), we only took those two general formats into account. Thus, the analysis in this RQ was performed on 383 TFHub packages (741 versions) and 26 PyTorch Hub packages (including 132 models). The snapshots for both repositories were taken in the middle of March 2020.

For each ML package, we extracted information about the task type, algorithm, and training dataset. For TFHub, the values and contents of IEs in our research scope were extracted automatically from the JSON data structure. Although PyTorch Hub does not provide such a data structure, we manually extracted the needed information from the PyTorch Hub packages due to their limited number.

4.2.2 Findings

4.2.2.1 Task Type of ML Packages/Models

Each published ML package in the studied repositories is trained with a specific algorithm on a specific dataset to help developers with a particular ML task (e.g., image classification). TFHub and PyTorch Hub define different type task classifications. In our research, we adopt TFHub's classification due to its clarity, and manually apply this classification on PyTorch Hub's packages. Task types unique to PyTorch Hub are added to the task type set. It should be noted that we further

[18] https://tfhub.dev/google/imagenet/mobilenet_v2_075_96/feature_vector/4

Table 4.5: Task types and ML package/model distribution on TFHub and PyTorch Hub

Application Domain	**Input**	**ML Task Type**	**# ML Packages**		**Task Type Description**
			TFHub (%)	**PyTorch Hub (%)**	
Audio Processing	Audio	Embedding	3 (0.8%)	0 (0.0%)	Changing the audio into a mathematical vector.
		Pitch Extraction	1 (0.3%)	0 (0.0%)	Recognize the dominant pitch in sung audio.
	Mel Spectrogram	Audio Generative	0 (0.0%)	1 (3.8%)	Synthesizes audio taking Mel Spectrogram(an acoustic time-frequency representation of a sound [54]) as input.

categorized similar task types under new created task types. The result of this process are the task types in Table 4.5 to Table 4.7, among which two are in audio processing, six are in computer vision and two are in natural language processing.

As shown in Table 4.5 to Table 4.7, image feature vector ML models take up the largest proportion (around 29%) in TFHub; the second and third largest task types in TFHub are text embedding (around 26%) and image classification (around 25%). In PyTorch Hub, the top three largest task type groups are text embedding (around 52%), image classification (around 39%) and image generator (around 3.8%).

Both repositories have ML models of the image classification, image generator, object detection, image segmentation and text embedding task types. TFHub has more ML models in all five types than PyTorch Hub. Only TFHub has ML models

Table 4.6: Continued Table 4.5

Application Domain	**Input**	**ML Task Type**	**# ML Packages**		**Task Type Description**
			TFHub (%)	**PyTorch Hub** (%)	
Computer Vision	Image	Augmentation	6 (1.6%)	0 (0.0%)	Augment the images. (like rotation, shearing)
		Classification	94 (24.5%)	15 (57.7%)	Classify the images according to their contents.
		Feature Vector	111 (29.0%)	0 (0.0%)	Extract image features.
		Generator	42 (11.0%)	2 (7.7%)	Generate images. (e.g., synthesize a photo, picture style transfer, enhance resolution)
		Object Detection	4 (1.0%)	1 (3.8%)	Find the objects in an image.
		Segmentation	10 (2.6%)	3 (11.5%)	Divide the different regions of a image.
		Other	1 (0.3%)	0 (0.0%)	-
	Video	Classification	2 (0.5%)	0 (0.0%)	Classify the videos according to their contents.
		Generator	5 (1.3%)	0 (0.0%)	Generate videos.
		Text	2 (0.5%)	0 (0.0%)	Extract video features.

Table 4.7: Continued Table 4.6

Application Domain	**Input**	**ML Task Type**	**# ML Packages**		**Task Type Description**
			TFHub (%)	**PyTorch Hub (%)**	
Natural Language Processing	Text	Question Answering	3 (0.8%)	0 (0.0%)	Answer questions in natural language
		Embedding	99 (25.8%)	3 (11.5%)	Changing the text (word, phrase, document) into a mathematical vector.
		Text to Mel Spectrogram	0 (0.0%)	1 (3.8%)	Generates Mel Spectrogram with natural language text

of audio embedding, audio pitch extraction, image augmentation, image feature vector, text question answering and video processing types. At the same time, users have to go to PyTorch Hub for ML models of audio generative (with Mel Spectrogram), text to Mel Spectrogram, and image semantic segmentation task types.

4.2.2.2 ML package Organization Practices: Family Phenomenon

ML packages are not organized in the same fashion in the TFHub and PyTorch Hub repositories. Although each ML package (in either TFHub or PyTorch Hub) has its individual page, we observed possible similarities among ML packages in terms of algorithm, training dataset and task type. Thus, in this section, we performed an in-depth analysis of the organization of these two ML package repositories through

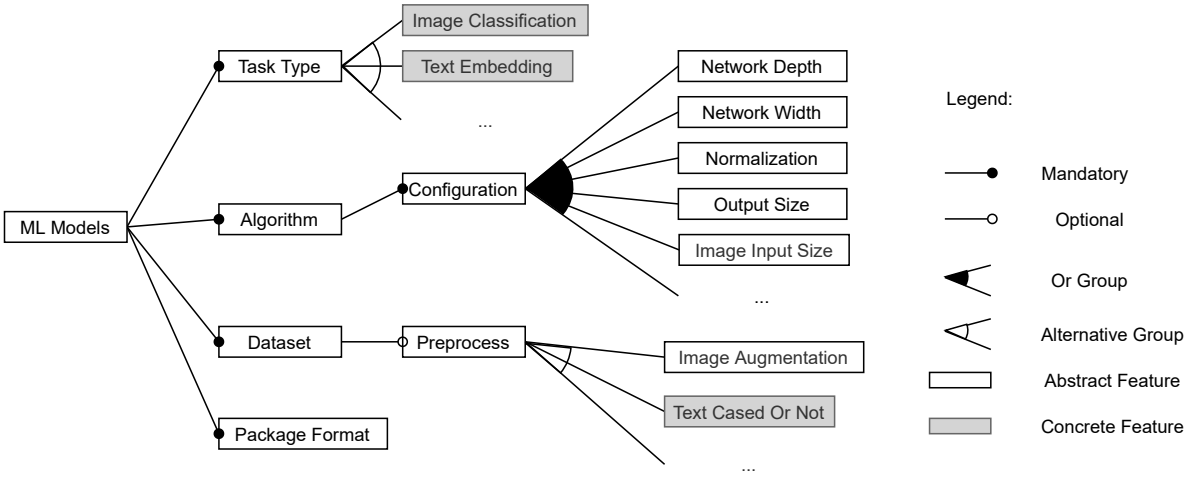


Figure 4.2: The feature diagram of ML models

a study of the family phenomenon (introduced at the beginning of this RQ).

4.2.2.2.1 Definition

ML packages differ from each other in terms of task types, algorithms, datasets and package formats, as illustrated in Figure 4.2. The former three are ML-related information elements and the last one depends on the package's implementation (e.g., different frameworks provide different model formats).

ML packages usually contain one or multiple ML models; a TFHub package always contains a single model while a PyTorch Hub package may contain several models through the entrypoint mechanism (generally speaking, one entrypoint maps to one model). From the perspective of models, we found that some of them are the

same in terms of the task type, algorithm and training dataset, but differ from each other due to different configurations of the algorithm or different pre-processing of the dataset. This phenomenon inspires us to group such models as families.

In the context of our research, **family members have the same task type, algorithm and dataset**, but may differ in configurations, output sizes and data pre-processing. Configurations (e.g., network depth, network width, normalization, etc.) are the most common differences among family members. Such configurations can cause the ML models to be of different sizes in terms of FLOPs (floating point operations, a metric for the complexity of the ML model) and number of parameters; thus, having an impact on the performance and deployment of ML models. Generally speaking, the larger a ML model's size, the better its performance (like classification accuracy). However, large ML models require more computational resources and are not suitable for usage contexts like mobile phones. Another difference observed in family members is the output size. For image generation ML models, the output size is the size of generated images, or for text embedding ML models, the length of embedding vectors. Different data pre-processing steps such as text case normalization (e.g., lower and upper case and accent markers are kept or removed uniformly) are used among family members, especially in NLP ML models.

The family phenomenon in ML package repositories is comparable to the prod-

uct line architecture in traditional software engineering, which makes it easier to create closely related but varying versions of the same product [55]. It should however be noted that there is no direct mapping between model families and signatures/entrypoints. TFHub and PyTorch Hub packages are required to have signatures and entrypoints, respectively. However, their implementation mostly depends on the developer of the package. For example, a developer can create multiple entrypoints within the same package but none would use the same algorithm, dataset or task type. Thus, a package can contain models belonging to different sets of families (e.g., the `Semi-Supervised and Semi-Weakly Supervised ImageNet Models` package contains 12 models that form four families).

Analyzing the differences between such family members provides additional understanding about ML model management and presentation; thus, it is essential that any ML package analysis considers this concept. We provided details on these different organization practices in the subsequent sections. In order to have a uniform analysis and comparison across the two ML package repositories, the subjects of the research in the subsequent subsections are rather the models within ML packages.

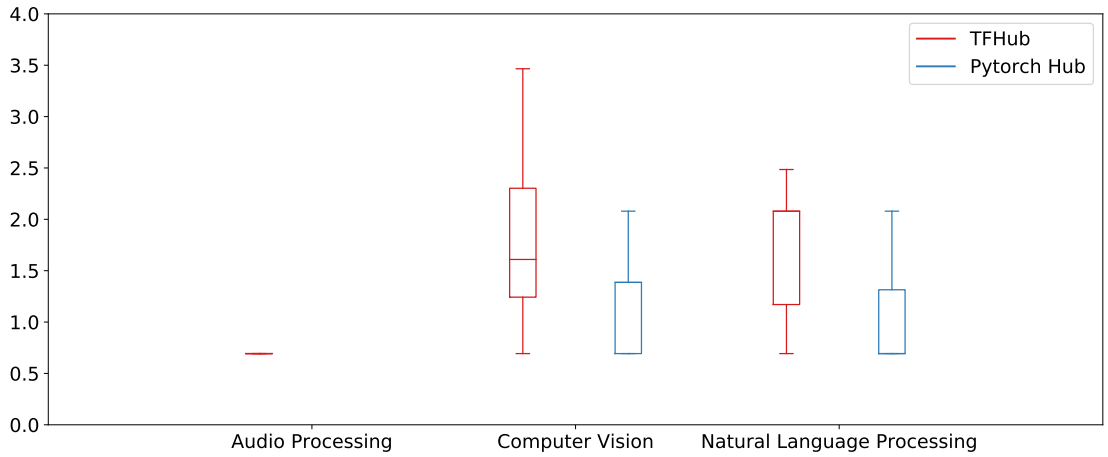


Figure 4.3: Distribution of the number of family members in ML models of the studied ML package repositories based on application domain (ln-scaled). The total number of families of TFHub and PyTorch Hub are 43 and 28 respectively.

4.2.2.2.2 Family Grouping Result and Analysis

There are 43 and 28 families in the TFHub and PyTorch Hub repositories, respectively. MobileNet V1 and MobileNet V2, both trained on ImageNet, are the two largest families in TFHub with 32 and 23 members, respectively. TFHub model families have a median of five family members, with most families having two members. In PyTorch Hub, VGG Nets and BERT families have the largest number of members (eight). The median number of family members is 2.5 and most families have two members.

Table 4.8: Statistics of number of families and median number of family members (only task types with at least a single family in either repository are shown)

Application Domain	**Input**	**ML Task Type**	**# Families**		**Median # Family Members**	
			TFHub	**PyTorch Hub**	**TFHub**	**PyTorch Hub**
Audio Processing	Audio	Embedding	1	-	2	-
Computer Vision	Image	Classification	7	13	4	4
		Feature Vector	12	-	2	-
		Generator	7	1	5	2
		Object Detection	-	1	-	2
		Segmentation	1	-	10	-
	Video	Text	1	-	2	-
Natural Language Processing	Text	Question Answering	1	-	3	-
		Embedding	13	13	8	2

Figure 4.3 shows the distribution of the number of family members in TFHub and PyTorch Hub based on application domain. We observed the family phenomenon in only nine task types across both TFHub and PyTorch Hub (see Table 4.8). Six of these tasks belong to the computer vision domain (`Image Classification`, `Image Feature Vector`, `Image Generator`, `Video Text`, `Image Segmentation` and `Object Detection`), one (`Audio Embedding`) belongs to audio processing, and two (`Question Answering` and `Text Embedding`) belong to the natural language processing domain. Audio embedding, image feature vector, image segmentation, video text and text question answering are only discovered in TFHub, while object detection family only exists in PyTorch Hub. Some statistics about the task types are in Table 4.8.

In both ML package repositories, the text embedding type has the largest number of families. TFHub has more families than PyTorch Hub in the image generator task type, while PyTorch Hub has more families of image classification task type.

TFHub and PyTorch Hub families use 27 and 20 algorithms, respectively, with only two algorithms in common: `ResNet-V1` (image classification) and `BERT` (text embedding). On TFHub, text embedding algorithm `NNLM` has the largest number of ML models (58) while the text embedding algorithm `BERT` has the largest number of ML models (23) on PyTorch Hub.

A great diversity of datasets is also used to train the ML models within the

Table 4.9: Similar ML models in the studied ML package repositories (* The ML model on TFHub is trained on ImageNet 2012, the PyTorch Hub model is trained on ImageNet 2014)

Application Domain	**Input**	**ML Task Type**	**Algorithm**	**Dataset**	**# Models**	
					TFHub	**PyTorch Hub**
Computer Vision	Image	Classification	Inception V1 (GoogleNet)	ImageNet*	1	1
			Inception V3	ImageNet	2	1
			ResNet V1	ImageNet	4	5
			MobileNet V2	ImageNet	23	1

identified families. TFHub and PyTorch Hub ML models use 23 and 17 different datasets, respectively. Among these datasets, only three of them are common across the two repositories: `ImageNet` (image classification, image generator), `CelebA HQ` (image generation) and `Wikipedia & BookCorpus` (text embedding). In both of the TFHub and PyTorch Hub, ImageNet is used in most of the families (20 on TFHub and 12 on PyTorch Hub).

4.2.2.2.3 Similar Models Across ML Model Repositories

As previously mentioned in Section 4.2.2.1, some task types are unique to a single ML package repository. To some extent, this finding reflects the difference in terms of the contents of ML repositories. Having introduced the family concept, this

section studies the similarity of the contents within the two ML package repositories. Table 4.9 presents the number of similar ML models across the two studied ML package repositories.

There are actually only a few ML models that overlap, see Table 4.9. For example, in TFHub, there are three ResNet V1 [46] image classification ML models trained on ImageNet, their names are `imagenet/resnet_v1_50/classification`, `imagenet/resnet_v1_101/classification`, `imagenet/resnet_v1_152/classification` and `resnet_50/classification`. While in PyTorch Hub package `ResNet`, there are five models `resnet18, resnet34`, `resnet50`, `resnet101`, and `resnet152` corresponding to the TFHub models.

4.2.2.2.4 Release Management of ML models

As previously discussed in RQ1, ML package repositories do not have any formalized release management or versioning mechanisms. We also found that ML package repositories do not have a well-defined release management practice in terms of algorithm upgrades. In TFHub's ML package organization practice, a change to the algorithm leads to the creation of a new package, rather than an upgraded package. For example, when the algorithm used by a package is changed from `MobileNet V1` [56] to `MobileNet V2` [57], a new package page is built for the upgraded ML package. Furthermore, except for the algorithm, the two packages are the same

in terms of the other two family deciding criteria (dataset and task type). We observed several algorithm changes in TFHub such as ResNet V1 [46] to V2 [58], and BERT [59] to ALBERT [60].

Though PyTorch Hub does not support an explicit versioning mechanism, there are some cases of algorithm upgrades; an upgrade of algorithms usually leads to different entrypoints (different models) in the same package rather than different versions. Examples of observed algorithm upgrades are BERT to distillBERT [61] and RoBERTa [62], and GPT [63] to GPT-2 [64].

4.2.3 Implications

Release Management. Currently, the upgrade of ML package's algorithm is not a versioned change. It is worthwhile for ML package repositories to consider new versions of an algorithm as an upgrade. There are two benefits: (1) Users are better aware of how many versions of an algorithm to choose from. It is a good practice to provide ML package users with information transparency. For example, without our research, TFHub users may not easily know that there are two versions of MobileNet algorithm, two versions of ResNet algorithm and three versions of Inception algorithm. If the number of ML packages keeps growing, this transparency will be more helpful for users. (2) This practice helps users better understand a group of algorithms and help them narrow down the search

scope. For example, although all MobileNet ML packages are suitable for mobile platform deployment, MobileNet V2 being better than V1 guides users to choose ML packages directly from V2 ones, rather than trying out from V1 ones.

Compared to traditional software package evolution, the evolution of an ML package can involve any of these things: (1) algorithm update, (2) update to non-algorithm related code (e.g., command line arguments, tuning, etc.), (3) changes to data, and (4) changes to input/output tensors. Among them, (1) and (3) are ML package specific, whereas (2) is common for all software projects. (4) can be applicable under the model family phenomenon. Furthermore, some of these are orthogonal for ML packages. For example, one product may change the algorithm but keep the data as it is, or update both the algorithm and the data. Some products may have multiple variants trained on different data. The aforementioned issues pose several research opportunities in the ML domain. Thus, there is the need for ML researchers to identify existing release management approaches in the traditional software engineering and product line domains that can be adopted and extended.

Impedance mismatch between model families and software package distribution/versioning. There are many families in the same task type, and there are many members in a family. Though this provides users with a great diversity of

ML models, the similarity between families and members makes choosing right ML model be difficult and confusing for users without solid ML expertise (like general software engineer and non-ML researchers). In the worst case, users may need to try the different model families, and probably each ML model within a family, to identify trade-offs between performance and computational resource consumption.

Based on our findings, we observe that the unit of shipping models is not that straightforward and formalized: should a package contain a whole family or a subset of members (based on entrypoints), or even multiple families? In addition, the family phenomenon may introduce some other software engineering challenges. For example, how are models in a family upgraded? How are changes within these families managed as the models evolve over time? Given the inherent similarities with the family phenomenon, ML researchers should seek to adopt the advanced practices of the traditional software product line architecture domain [65].

4.3 RQ3: What is the process needed in order to use the functionalities from software/ML package repositories?

Having identified the information within the studied ML package repositories (Section 4.1), as well as their organization practices (Section 4.2), this RQ investigates the processes needed to reuse these shared ML packages. First, we examined the

basic functionalities and the package usage practices supported by the ML package managers (e.g., `tensorflow_hub` library [66] for TFHub, PyTorch Hub API in PyTorch library). The functionalities and practices were compared against those of software package managers, whenever applicable, to understand the commonalities and uniqueness between them. Next, we studied the different usage contexts supported by ML frameworks from their documentation.

4.3.1 Approach

Package managers provide some basic functionalities such as installing, upgrading and removing packages within a programming language-specific development environment [67]. Due to the longer existence and wider usage of software package repositories, we regarded the software package managers and their APIs as a baseline, and we attempted to identify such similar functionalities for ML libraries.

First, we looked for documentation and online tutorials about the three functionalities (installation, upgrade, removal) of both ML package repositories and software package repositories. If no corresponding materials of a functionality were found for a given repository, we regarded this functionality to be unsupported. Table 4.10 summarizes the basic functionalities, the supported package formats and primary usage mechanism of the studied package repositories. Next, we also identified the different supported usage contexts and the steps needed to use the packages

of the ML and software package repositories.

4.3.2 Findings

4.3.2.1 Basic functionalities of ML and Software package managers (Installation, Upgrade, Removal)

Though users have to follow similar steps before using packages within software and ML package repositories, there are, however, a few significant differences in how these steps are implemented for software and ML packages. It generally takes 2 steps to use any software or ML package: (1) installing/upgrading packages, and (2) invocation of the functionalities from a loaded/imported package.

Software packages are mostly installed via terminal commands provided by their package managers (see Figure 4.4(a)). However, there is no clear division between installation and loading in ML package repositories. ML packages require a runtime load step that downloads the ML artifacts if not yet in cache, and selects the right model for further use. For example, figure 4.4(b) demonstrates how to load a ML package from TFHub. As shown in the figure, TensorFlow's `tensorflow_hub` library, which is the library that mainly supports the ML package management functionalities and ML package usage, provides an initialization API called `hub.load()`. This API takes in an argument for the location of the ML package; this can be ei-

Table 4.10: Basic functionalities and usage information of software and ML package repositories

Repositories		**Package Manage Functionalities**			**Supported Format(s) of Packages**	**Usage Mechanism**
		Install	**Upgrade**	**Remove**		
Software	npm	✓	✓	✓	Various (code file)	API
	PyPI	✓	✓	✓	Standard (.tar.gz, .whl)	API
	CRAN	✓	✓	✓	Standard (zipped package)	API
ML	TFHub	✓			Various (hub.Module, TF2 SavedModel, other formats)	Signature
	PyTorch Hub	✓			Mostly Standard (GitHub repository + .pt/.pth/unknown format)	Entrypoint

ther a link to TFHub pages, a link to some specified online zip file or a path to local package. Given the location, the library downloads the package (if not already in the cache), loads it, and makes it ready for use. A similar practice is used to load PyTorch Hub packages, as shown in Figure 4.4(c).

Additionally, there is no real support for package upgrades in ML package repositories. Unlike software packagers that provide users with package upgrade commands, users of ML packages have to manually specify the version of a package to load at runtime. This practice is further exacerbated due to the lack of formal package versioning mechanisms. For example, users of TFHub packages need to specify the version number as part of the url (e.g., the "`../4`" at the end of `https://tfhub.dev/google/imagenet/mobilenet_v1_050_160/classification/4` shows the version). Given that PyTorch Hub has no versioning support for packages, users must decide at runtime whether to download an updated PyTorch package from its GitHub repository[19] or use an existing cached copy. As a result, a different GitHub snapshot of a model can be loaded each time, introducing severe inconsistency problems to users.

Also, we could not find any similar functionality for the removal of packages provided by ML libraries. The package initialization API of PyTorch has an argu-

[19]The source code of PyTorch Hub packages are stored on GitHub. The links to model files (`.pt`, `.pth` files) will be stored in the GitHub code but the model files themselves may be stored on some other places rather than GitHub.

ment that if set, would always download a new package even if there is a cached one. But this is not a real removal functionality as users need to manually remove the cached models eventually.

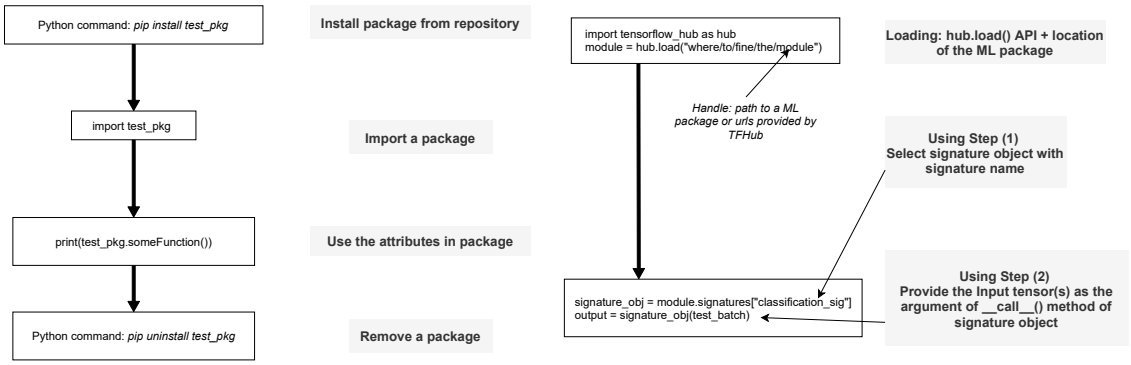
4.3.2.2 Package Usage

Despite the close similarities of how ML and software packages are installed or loaded, the process of actually using these packages is different. Once software packages are installed, the users only need to import the package and utilize the various APIs within their system. ML packages, on the other hand, require an additional step after they are loaded before they can be used. This difference is brought by the signature mechanism of TFHub packages and entrypoint mechanism of PyTorch Hub packages. An in-depth discussion of how ML packages are used under these two mechanisms is provided below.

4.3.2.2.1 TFHub Signature Mechanism

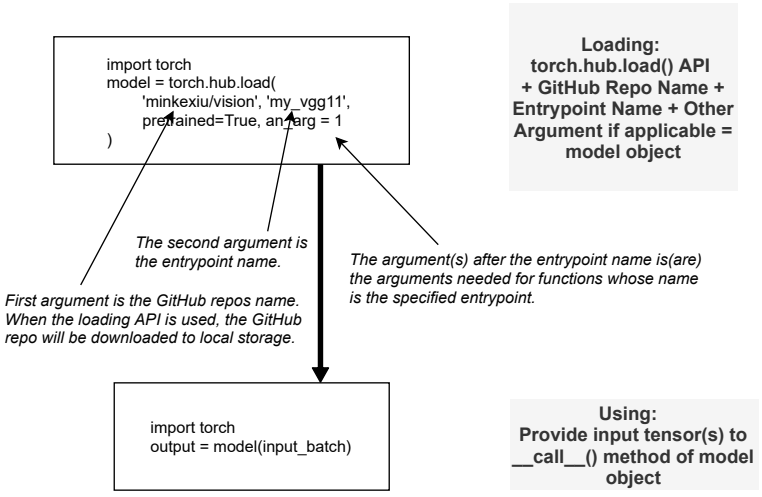
A signature is a particular combination of input and output data-structures (also called tensors [68]) used by a ML model. Given that some ML packages can be used for more than one task, the signature mechanism is used to allow users to express the task to perform. For example, `imagenet/mobilenet_v1_050_160/classification`[20]

[20]https://tfhub.dev/google/imagenet/mobilenet_v1_050_160/classification/4



(a) PyPI

(b) TFHub



(c) PyTorch Hub

Figure 4.4: The process of loading and using packages

is a package which allows users to perform either image classification or feature extraction in a set of images. Thus, a user performing an image classification task with the `imagenet/mobilenet_v1_050_160/classification` package must provide the input tensor in the expected shape before the correct output tensor will be returned.

4.3.2.2.2 PyTorch Hub Entrypoint Mechanism

When loading a PyTorch package for use, the `torch.hub.load()` API requires, in addition to the GitHub repository containing the necessary code of this package, the name of an entrypoint name and any additional arguments needed by the provided entrypoint. An entrypoint is essentially a Python function defined in the package's source code that implements a particular configuration of an algorithm. It should be noted that how the function actually works is totally decided by developers, including the argument settings, implementation logic, and where to find the pre-trained model files.

We observe that currently the PyTorch Hub entrypoint mechanism is not as formalized as the TFHub signature mechanism. Several packages on PyTorch Hub do not provide a complete entrypoint list (an example of how entrypoints are displayed for a package is shown in Figure 4.5); one has to manually search within the source code repository[21]. Secondly, package developers may implement the en-

[21] https://pytorch.org/hub/pytorch_fairseq_translation/

Model structure	Top-1 error	Top-5 error
vgg11	30.98	11.37
vgg11_bn	26.70	8.58
vgg13	30.07	10.75
vgg13_bn	28.45	9.63
vgg16	28.41	9.62
vgg16_bn	26.63	8.50
vgg19	27.62	9.12
vgg19_bn	25.76	8.15

Figure 4.5: An example of the list of entrypoints of a PyTorch Hub package with their respective quality measures

trypoint differently given the lack of formalism or best practices. Such differences in entrypoint definitions makes the loading process of a package difficult for users. The aforementioned two issues show the need for the package developers to prepare sufficient documentation that explains the full functionality and loading process of their packages, rather than expecting general users to read and understand the source code. In comparison, TFHub packages do a better job by providing APIs to inform the users of the signatures within packages.

4.3.2.2.3 Signature/Entrypoint mechanism vs. Model families

We observed that the variation within ML packages can be viewed along different dimensions: model families (see RQ2) and signatures/entrypoints. There may be a number of models (in a the same family) that use the same algorithms and datasets but different training hyper-parameter configurations or computational features (e.g., with or without batch-normalization hyper-parameters). We could also have different variations of an ML package defined through entrypoints.

However, we observe that although the number of entrypoints in a PyTorch Hub package indicates the number of different models in the package, not all entrypoints in a package belong to the same family; a PyTorch Hub package can have multiple families, each consisting of a subset of its entrypoints. In TFHub, each package consists of exactly one model which can have multiple signatures, another form of ML-specific variation. Thus, there seems to be no relation between signatures and model families.

Consequently, the development process of ML packages can be considered to be analogous to the concept of product line architectures. Such product line architecture-like practices of ML packages bring some benefits. First, it makes the organization of different variants easier. For example, without the signature mechanism, the multiple signatures have to be independent packages, causing re-

dundancy and the synchronization of their changing and maintenance will take a lot of efforts. Secondly, such mechanism help the users to easily compare the functionalities of related packages.

4.3.2.3 Usage Contexts Supported by ML Libraries

TensorFlow and PyTorch provide support for using packages in different contexts. Usage contexts are the specific deployment platforms or software environments upon which ML packages are loaded and called. Each usage context prioritizes certain attributes of an ML package such as size, performance and portability. As such, ML packages and models used in different contexts are usually in different formats. It should be noted that in this subsection, our analysis is focused at the level of the models within the ML packages, rather than the packages themselves.

Table 4.11 shows the various usage contexts available to TensorFlow and PyTorch packages. In the table, each row represents a group of relevant deployment scenarios. The first column shows the usage contexts supported by TensorFlow and PyTorch. It should be noted that `edge device` and `remote device` have some overlap, e.g., TensorFlow models may be run on Python environments on both Raspberry Pi (as a kind of edge device) and web servers (as a kind of remote device). The reason for splitting these two rows is to emphasize their most representative characteristics, i.e., computational resource limitation for edge devices and

Table 4.11: TensorFlow and PyTorch package/model usage contexts

Deployment Target	**Typical Model Format**		**Call (Serve) Locally or Remotely**	
	TensorFlow	**PyTorch**	**TensorFlow**	**PyTorch**
Python	Checkpoint, SavedModel, Frozen GraphDef, hub.Module	(All the formats)	Locally	Locally
Edge Device	FlatBuffers (iOS, Android, RPi) (Swift, Objective-C)	.pt File (iOS, Android)	Locally	Locally
JavaScript	TensorFlow.js Model (Browsers, node.js)	-	Locally	-
Remote Device	SavedModel	.pt, .pth File	Remotely	Remotely (Flask Needed)
Other Languages	SavedModel, .pb File (C++, Java, Go, etc.)	TorchScript (C++, Java)	Locally	Locally
Other ML Frameworks	-	ONNX (Caffe2, MXNet, CNTK, etc.)	-	Locally

Table 4.12: Advantages and disadvantages of different formats of packages/models

ML Framework	**Package/ Model Format**	**Advantages**	**Disdvantages**
TensorFlow	checkpoint	Suitable for recording training process.	Not suitable for deployment.
	SavedModel	Standard model saving format. Suitable for deployment.	May not support specialized usage context.
	Frozen GraphDef	Size saving. Suitable for inference-only usage.	Parameters in it cannot be changed.
	hub.Module	Specified for model sharing on TFHub.	Out of date in TensorFlow 2.x era.
	TensorFlow Lite Model	Can be optimized in size.	Optimization may reduce performance.
	TensorFlow.js Model	Can be used in JavaScript environment.	May have less support than traditional TensorFlow.
PyTorch	.pt, .pth file	Standard model saving format.	May not support specialized usage context.
	TorchScript	Can be used in multiple languages.	Inappropriate for internal model deployments.
	ONNX	Can be used by different ML frameworks.	May not support specialized usage context.

different calling mechanisms for remote server. The second and third columns represent the typically used model formats within the given deployment environment (based on the ML package repository). The fourth and fifth column explain how the models are called (or served). If the loading and the whole usage process happen on the same machine, it is considered "local". However, if the package is loaded on another machine and the usage process need remote communication between different machine, it is considered "remote". For example, a user can use HTTP requests to obtain inference results from an image classification package deployed on a remote server. Both ML frameworks have five deployment scenario groups. In general, TensorFlow supports more usage contexts, which can be attributed to TensorFlow's longer existence.

Unsurprisingly, TensorFlow and PyTorch have the best support for integration directly into a Python code base. It is a general practice that a model is developed on Python and deployed in other scenarios. For TensorFlow, there are four common saved model formats: checkpoint[22], SavedModel[23], Frozen GraphDef[24] and hub.Module[25]. Checkpoint is suitable for temporarily save the training process, generally only contains parameter values but not calculations. So this character

[22]https://www.tensorflow.org/guide/checkpoint

[23]https://www.tensorflow.org/guide/saved_model

[24]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/tools/freeze_graph.py

[25]https://www.tensorflow.org/hub/api_docs/python/hub/Module

makes it not very suitable for sharing because the code of algorithm should be separately provided. SavedModel can save both parameters and calculations, it can be used off the shelf without any algorithm code. This character makes it suitable for deployment and sharing. In TensorFlow 2.x it is also the general format for saved model. Frozen GraphDef is extremely lightweight and suitable for deployment and doing inference. But its parameter values cannot be changed so models in this format cannot be further trained or fine-tuned. hub.Module is specially invented format for model sharing on TFHub and it's being replaced in TensorFlow 2.x API era. As for PyTorch, the common saved model formats under Python environment are `.pt` and `.pth`. But shared models in PyTorch Hub package may not be very suitable for deployment because the parameter values (in `.pt` or `.pth` files) and algorithms (in code base) are separately stored. A summary of the advantages and disadvantages of each model format is provided in Table 4.12.

TensorFlow and PyTorch models also can be deployed on edge devices, like mobile platforms, which suffer from limited computational and storage resources. TensorFlow provides a set of tools in TensorFlow Lite [69] for deployment on edge device. Normal TensorFlow Python models can be converted into a TensorFlow Lite format model named FlatBuffers[26]. In addition, both TensorFlow and PyTorch provide some quantization methods, like saving the values in lower precision, that

[26]https://www.tensorflow.org/lite/convert/index

make models smaller and minimize the degradation of performance.

TensorFlow and PyTorch models can be deployed for use in other languages. For TensorFlow, the SavedModel can be loaded by the TensorFlow API in C++, Java, Go, Swift, etc. PyTorch only supports C++ and Java in its documentation currently. PyTorch uses `TorchScript` to represent a model that is independent from Python and can be loaded and executed by PyTorch API in C++.

TensorFlow and PyTorch models can be served remotely and called through a REST API for inference. For TensorFlow, a served model will be in SavedModel format. The containerized serving process for TensorFlow models is introduced by TensorFlow's documentation in detail. In addition to REST API calling, served TensorFlow models can also be called through gRPC [70], a high performance RPC framework. For PyTorch, the models needs to be deployed and served with the help of Flask [71], a third-party lightweight web framework in Python.

TensorFlow and PyTorch each has some unique usage contexts as well. TensorFlow models can be deployed on JavaScript environments like browsers and Node.js. TensorFlow in JavaScript is called TensorFlow.js [72]. The TensorFlow.js models can be converted from normal Python-based models. PyTorch models can be exported into ONNX format [73] and loaded by any ONNX-supporting frameworks, like Caffe2, MXNet, CNTK etc..

From the Table 4.11, we found that the usage contexts are more complex in ML

packages than in software packages. Software packages can be deployed successfully within any usage context supports the languages and they do not have much variation in formats. For example, a Python package can be deployed on any platforms or environments that support Python, and the Python package will always be in `.whl` or `.tar.gz` formats[27]. However, this is not the case for ML packages/models. When a user wants to use a model in different contexts, the model formats will vary and the variation may lead to different loading and usage practices. For example, different APIs are needed to load and use TensorFlow.js and SavedModel model formats, or a SavedModel may need to be converted to a TensorFlow.js format before it is used in a browser. Though package/model formats can be converted between each other, considering the most suitable format in advance can at least save the conversion effort.

4.3.3 Implications

Usage Context. Unlike software packages that have similar usage steps, irrespective of the OS or hardware of a system, this is not the case for ML packages. There are many usage contexts for ML packages and models. The considerations for ML packages in different usage contexts are more than those for software packages. The loading and usage processes may differ according to the usage context (e.g., differ-

[27]https://packaging.python.org/tutorials/packaging-projects/#generating-distribution-archives

ent sets of APIs, different formats of saved models). This characteristic is another reason for the users (like software engineers and data scientists) to consider the usage context earlier and make suitable trade-offs.

Pre-defined interfaces for ML packages (signatures and entrypoints). ML libraries use special mechanisms for loading and using a package: `signatures` for TFHub packages and `entrypoints` for PyTorch Hub packages. These mechanisms are different from software packages that can be simply imported and then used. Those mechanisms adopt the product line architecture from traditional software engineering and leverage some of its benefits. Though these mechanisms provide the needed flexibility for creating and using variants of an ML package, they also introduce a steep learning curve for new ML developers, especially traditional software engineers.

Package Management Functionalities. These two aspect of the ML package repositories are still a work in progress. Currently, ML package repositories only support package installation. The upgrade, and removal functionalities are either missing or have limited support. In the future, TensorFlow and PyTorch maintainers can think about developing the actual ML package upgrade and removal functionalities.

Package Documentation. There is limited documentation on how users (especially with limited ML experience) can integrate pre-trained models into their applications. Given the relative complexity of ML packages, in comparison to software packages, the documentation of ML packages are expected to be very detailed. Loading a package requires a lot of information in terms of versions, signatures and entrypoints. Such details need to be in well-described. For example, the documentation of PyTorch Hub packages' entrypoints are not complete (5 out of the 15 packages with multiple entrypoints do not provide a full list of their entrypoints). Contributors of the PyTorch Hub and TFHub packages need to treat the documentation highly to serve the users better. Researchers can also propose code summary techniques for generating detailed and formalized documentation, and tools to ensure that strict naming conventions for parameters can are adhered to.

4.4 Threats to Validity

4.4.1 Construct Validity

Although the information elements in RQ1 were extracted from multiple resources (like original webpages, definition files, JSON data structure), they may be incom-

plete. To mitigate this threat, the thesis author, his supervisor and two other SE colleagues (Ellis E. Eghan and Bram Adams) performed independent analysis of the studied package repositories to verify the list of identified IEs. We also verified the information elements with previous work [25].

4.4.2 Internal Validity

As discussed in the research questions, the release management processes for ML packages are not well-defined. In particular, some of the newer versions of ML packages are either shown within the same product pages (e.g., V1 to V4 of the same ML package are organized in the same page[28]) or on separate product pages (e.g., two versions of a ML package trained on different algorithms are displayed in different ML package pages[29,30], packages in the same algorithm but trained on different datasets and saved into different formats are also in separate pages[31,32]). However, in software package repositories, new versions of a package will be generally organized in the same product page (e.g., all the versions of the TensorFlow, as Python package, are organized on the same page). When counting the number of ML packages, we actually counted the number of separate ML package pages.

[28]https://tfhub.dev/google/imagenet/mobilenet_v1_100_128/feature_vector/4

[29]https://tfhub.dev/google/imagenet/mobilenet_v1_100_128/feature_vector/4

[30]https://tfhub.dev/google/imagenet/mobilenet_v2_100_128/feature_vector/4

[31]https://tfhub.dev/google/bert_multi_cased_L-12_H-768_A-12/1

[32]https://tfhub.dev/tensorflow/bert_multi_cased_L-12_H-768_A-12/1

These different criteria of counting the number of ML packages and packages may threaten the results of our repository comparisons.

4.4.3 External Validity

ML package repositories are relatively new and they are rapidly changing. For example, we noticed during our analysis that TFHub's information elements and JSON data structure changes over time (e.g., new information elements are added and the order of IE was changed). Hence, although our current findings are useful for practitioners and SE researchers, they may be out-dated in a few years. It would be worthwhile to replicate this study after a period of time to analyze the evolution of software practices (e.g., versioning, usage, package organization) within the ML package repositories.

4.5 Summary

This chapter is an exploratory study on ML package repositories: TFHub and PyTorch Hub. First, we compared the information elements between ML package repositories and software package repositories, and then between two ML package repositories. We discovered some concerns of ML package repositories in terms of dependency management, release information, popularity indicator security and technical documentation information transparency. The second research question

is about the contents (packages and models within) of ML package repositories. We looked into how the packages are classified and organized, the similarities between packages and models across different ML package repositories, and the release management practices of ML packages. The third research question is closely related to the basic functionalities (loading and using a package) and usage contexts the ML libraries support. We discovered that the ML package upgrade and removal functionalities are still missing in ML libraries. Also the signature and entrypoint mechanisms provided by TensorFlow and PyTorch, as well as the family phenomenon of ML packages may bring about package evolution and usage challenges.

5 Conclusions and Future Work

In this thesis, we looked into two kinds of ML model platforms: ML model stores and ML package repositories. ML model stores provide commercialized cloud-based deployment support for models. Though ML package repositories provide free models, users need to manually manage them. We compared ML model platforms with their counterparts in traditional software engineering: ML model stores vs. mobile app stores, ML package repositories for deep learning frameworks vs. software package repositories for programming languages.

We found that both kinds of ML model platforms are in their infancy and have rooms for improvement. First, some features in traditional software engineering (e.g., review policy) and conventional app store or package repositories (e.g., user reviews and usage statistics) are missing in ML model platforms. Secondly, some package management features are missing in certain ML model platforms. For example, existing ML package managers do not fully support package upgrade and removal. There are some other essential features missing or not in a complete

form: version control, signature and entrypoint mechanisms that bear the stamp of product line architecture, cross-platform support, security concerns of ML models/packages, dependency management, etc. Thirdly, ML models and their platforms have some unique features (e.g., descriptions of the dataset, the model evaluation approaches and the results) requiring new software engineering practices/processes. In the future, we will look into the evolution of ML models and their associated sharing pratices. In addition, we will investigate software engineering practices for developing and maintaining ML models.

Bibliography

[1] 11 Industries Being Disrupted By AI. https://www.cmswire.com/information-management/11-industries-being-disrupted-by-ai/. Last accessed: 05/27/2020.

[2] Michael Chui, James Manyika, Mehdi Miremadi, Nicolaus Henke, Rita Chung, Pieter Nel, and Sankalp Malhotra. Notes from the AI Frontier Insights from Hundreds of Use Cases. https://www.mckinsey.com/featured-insights/artificial-intelligence/notes-from-the-ai-frontier-applications-and-value-of-deep-learning. Last accessed: 03/25/2019.

[3] TensorFlow. https://www.tensorflow.org. Last accessed: 11/03/2019.

[4] PyTorch. https://pytorch.org. Last accessed: 11/03/2019.

[5] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018.

[6] Gartner Identifies the Top 10 Strategic Technology Trends for 2019. https://www.gartner.com/en/newsroom/press-releases/2018-10-15-gartner-identifies-the-top-10-strategic-technology-trends-for-2019. Last accessed: 12/01/2019.

[7] Google Play. https://play.google.com/store?hl=en. Last accessed: 24/03/2019.

[8] App Store - Apple. https://www.apple.com/ios/app-store/. Last accessed: 24/03/2019.

[9] AWS Marketplace: Machine Learning & Artificial Intelligence . https://aws.amazon.com/marketplace/solutions/machinelearning/. Last accessed: 03/30/2019.

[10] ModelDepot - Open, Transparent Machine Learning for Engineers . https://modeldepot.io/. Last accessed: 03/30/2019.

[11] Wolfram Neural Net Repository of Neural Network Models . https://resources.wolframcloud.com/NeuralNetRepository/. Last accessed: 03/30/2019.

[12] Continuous Delivery for Machine Learning. https://martinfowler.com/articles/cd4ml.html. Last accessed: 10/07/2020.

[13] For Researchers — PyTorch. https://pytorch.org/hub/research-models. Last accessed: 11/03/2019.

[14] TensorFlow Hub (module repository). https://tfhub.dev/. Last accessed: 11/24/2019.

[15] npm. https://www.npmjs.com. Last accessed: 03/17/2019.

[16] PyPI · The Python Package Index. https://pypi.org. Last accessed: 03/17/2019.

[17] The Comprehensive R Archive Network. https://cran.r-project.org. Last accessed: 03/17/2019.

[18] Nuance AI Marketplace for developers . https://aimarketplace.portal.azure-api.net/. Last accessed: 03/30/2019.

[19] Slinger Jansen and Ewoud Bloemendal. Defining app stores: The role of curated marketplaces in software ecosystems. In International Conference of Software Business, pages 195–206. Springer, 2013.

[20] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? IEEE Software, 32(3):70–77, 2015.

[21] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Empirical Software Engineering, 21(3):1346–1370, 2016.

[22] Libraries.io - The Open Source Discovery Service. https://libraries.io/. Last accessed: 25/03/2019.

[23] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. The Open-closed Principle of Modern Machine Learning Frameworks. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR), 2018.

[24] AI Hub - TensorFlow module. https://aihub.cloud.google.com/s?\category=tensorflow-module. Last accessed: 19/03/2019.

[25] Ethan Bommarito and Michael Bommarito. An empirical analysis of the python package index (pypi). arXiv preprint arXiv:1907.11073, 2019.

[26] Steven Raemaekers, Arie Van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In 2013 10th Working Conference on Mining Software Repositories (MSR), pages 221–224. IEEE, 2013.

[27] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. Tracing known security vulnerabilities in software repositories–a semantic web enabled modeling approach. Science of Computer Programming, 121:153–175, 2016.

[28] Ellis E Eghan, Sultan S Alqahtani, Christopher Forbes, and Juergen Rilling. Api trustworthiness: an ontological approach for software library adoption. Software Quality Journal, 27(3):969–1014, 2019.

[29] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe 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 (SANER), volume 1, pages 493–504. IEEE, 2016.

[30] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. IEEE Transactions on Software Engineering, 2019.

[31] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 644–655, 2018.

[32] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. Empirical Software Engineering, 25(2):1168–1204, 2020.

[33] Maëlick Claes, Tom Mens, Narjisse Tabout, and Philippe Grosjean. An empirical study of identical function clones in cran. In 2015 IEEE 9th International Workshop on Software Clones (IWSC), pages 19–25. IEEE, 2015.

[34] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In Proceedings of the 40th International Conference on Software Engineering, pages 511–522, 2018.

[35] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. In Advances in Neural Information Processing Systems, pages 8250–8260, 2019.

[36] Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, et al. Multilingual universal sentence encoder for semantic retrieval. arXiv preprint arXiv:1907.04307, 2019.

[37] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016.

[38] Minke Xiu, Zhen Ming Jack Jiang, and Bram Adams. An exploratory study on machine-learning model stores. IEEE Software, 2020.

[39] One simple graphis Researchers love PyTorch and TensorFlow - O'Reilly. https://www.oreilly.com/content/one-simple-graphic-researchers-love-pytorch-and-tensorflow/. Last accessed: 31/03/2019.

[40] The Deep Learning Framework Backed By Facebook Is Getting Industry's Attention. https://www.forbes.com/sites/janakirammsv/2019/02/11/the-deep-learning-framework-backed-by-facebook-is-getting-industrys-attention/#7b62d01f3314. Last accessed: 07/17/2020.

[41] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. biometrics, pages 159–174, 1977.

[42] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. Extracting insights from the topology of the javascript package ecosystem. In 2017 24th Asia-Pacific Software Engineering Conference (APSEC), pages 298–307. IEEE, 2017.

[43] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. With great training comes great vulnerability: Practical attacks against transfer learning. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1281–1297, 2018.

[44] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pages 248–255. Ieee, 2009.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[47] What is CUDA — NVIDIA Official Blog. https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/. Last accessed: 05/28/2020.

[48] Apache License, Version 2.0 — Open Source Initiative. https://opensource.org/licenses/Apache-2.0. Last accessed: 03/16/2019.

[49] Don't believe the download numbers when evaluating open source projects. https://blog.tidelift.com/dont-believe-the-download-numbers-when-evaluating-open-source-projects. Last accessed: 22/07/2019.

[50] Rajesh Vasa, Leonard Hoon, Kon Mouzakis, and Akihiro Noguchi. A preliminary analysis of mobile app user reviews. In Proceedings of the 24th Australian Computer-Human Interaction Conference, pages 241–244, 2012.

[51] Whitesource Renovate - Automated Dependency Updates. https://renovate.whitesourcesoftware.com/. Last accessed: 19/09/2020.

[52] Snyk — Developer Security — Develop Fast. Stay Secure. https://snyk.io/. Last accessed: 19/09/2020.

[53] XMK233/EMSE2020_MLPackage: Replication package for EMSE 2020 ML package paper. https://github.com/XMK233/EMSE2020_MLPackage. Last accessed: 09/20/2020.

[54] MelSpectrogram. https://www.fon.hum.uva.nl/praat/manual/MelSpectrogram.html#:~{}:text=MelSpectrogram&text=One%20of%20the%20types%20of,on%20a%20Mel%20frequency%20scale). Last accessed: 07/17/2020.

[55] Paul Clements. Software product lines: A new paradigm for the new century. Crosstalk, 12(2):20–22, 1999.

[56] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

[57] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4510–4520, 2018.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In European conference on computer vision, pages 630–645. Springer, 2016.

[59] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[60] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942, 2019.

[61] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108, 2019.

[62] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692, 2019.

[63] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding with unsupervised learning. Technical report, OpenAI, 2018.

[64] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. OpenAI Blog, 1(8):9, 2019.

[65] Goetz Botterweck and Andreas Pleuss. Evolution of software product lines. In Evolving Software Systems, pages 265–295. Springer, 2014.

[66] TensorFlow Hub (Library). https://www.tensorflow.org/hub. Last accessed: 05/27/2020.

[67] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in foss distributions: Details and challenges. In Proceedings of the 1st international workshop on hot topics in software upgrades, pages 1–5, 2008.

[68] Introduction to Tensors — TensorFlow Core. https://www.tensorflow.org/guide/tensor. Last accessed: 11/08/2020.

[69] TensorFlow Lite guide — TensorFlow. https://www.tensorflow.org/lite/guide. Last accessed: 11/21/2019.

[70] gRPC. https://www.grpc.io/. Last accessed: 11/21/2019.

[71] Welcome to Flask — Flask Documentation (1.1.x). http://flask.palletsprojects.com/en/1.1.x/. Last accessed: 11/21/2019.

[72] TensorFlow.js. https://www.tensorflow.org/js. Last accessed: 11/21/2019.

[73] ONNX. https://onnx.ai/. Last accessed: 11/21/2019.