Can Machine Learning Pipelines Be Better Configured?

Yibo Wang*
Northeastern University
Shenyang, China
yibowangcz@outlook.com

Ying Wang†
Northeastern University, and HKUST
Shenyang, Hong Kong, China
wangying@swc.neu.edu.cn

Tingwei Zhang
Northeastern University
Shenyang, China
592131686@qq.com

Yue Yu
National University of Defense Technology
Changsha, China
yuyue@nudt.edu.cn

Shing-Chi Cheung
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Hai Yu
Northeastern University
Shenyang, China
yuhai@mail.neu.edu.cn

Zhiliang Zhu
National Frontiers Science Center for Industrial Intelligence and Systems Optimization, and Key Laboratory of Data Analytics and Optimization for Smart Industry, Northeastern University
Shenyang, China
ZHUZhiLiang_NEU@163.com

ABSTRACT

A **M**achine **L**earning (ML) pipeline configures the workflow of a learning task using the APIs provided by ML libraries. However, a pipeline's performance can vary significantly across different configurations of ML library versions. Misconfigured pipelines can result in inferior performance, such as inefficient *executions*, *numeric errors* and even *crashes*. A pipeline is subject to misconfiguration if it exhibits significantly inconsistent performance upon changes in the versions of its configured libraries or the combination of these libraries. We refer to such performance inconsistency as a *pipeline configuration (PLC) issue*.

A systematic understanding of PLC issues helps configure effective ML pipelines and identify misconfigured ones. To this end, we conduct the first empirical study of PLC issues' pervasiveness, impact and root causes. To facilitate scalable in-depth analysis, we develop Piecer, an infrastructure that automatically generates a set of pipeline variants by varying different version combinations of ML libraries and detects their performance inconsistencies. We apply Piecer to the 3,380 pipelines that can be deployed out of the 11,363 ML pipelines collected from multiple ML competitions at Kaggle platform. The empirical study results show that 1,092 (32.3%) of the 3,380 pipelines manifest significant performance inconsistencies on at least one variant. We find that 399, 243 and 440 pipelines can achieve better competition scores, execution time and memory usage, respectively, by adopting a different configuration. Based on our findings, we construct a repository containing 164 defective APIs and 106 API combinations from 418 library versions. The defective API repository facilitates future studies of automated detection techniques for PLC issues. Leveraging the repository, we captured PLC issues in 309 real-world ML pipelines.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

Machine Learning Libraries, Empirical Study

ACM Reference Format:
Yibo Wang, Ying Wang, Tingwei Zhang, Yue Yu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023. Can Machine Learning Pipelines Be Better Configured?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616352

1 INTRODUCTION

Machine **L**earning (ML) libraries (e.g., *TensorFlow* and *PyTorch*) have gained much attention in both academia and industry, which are used in a wide range of domains, including natural language processing, image processing, autonomous driving, etc [38, 55, 56, 58, 79]. These libraries provide off-the-shelf ML solutions, optimized numerical computations, efficient data structures, visualization, and other handy utilities, to facilitate application development.

In practice, the calls to ML library APIs are often organized by means of an ML pipeline, which automates the workflow of an ML task in nine stages as shown in Figure 1. To timely incorporate support for new hardware, algorithms and bug fixes, ML libraries keep evolving rapidly. However, different combinations of ML library versions can cause inconsistencies in pipelines' performances, crashes and *NaN* bugs, as demonstrated by the real-life example

*Yibo Wang and Ying Wang made equal contributions to this work.
†Ying Wang is the corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616352

in Section 2.2. In this paper, we refer to the performance inconsistencies of ML pipelines induced by varying ML library versions as *PipeLine Configuration (PLC) issues*.

Despite the widespread use of ML pipelines, there have been no prior systematic studies on PLC issues. We have little knowledge of PLC issues, such as their pervasiveness in real-world ML pipelines, their impact on ML tasks' performances, and their root causes. The deficient knowledge can hinder effective deployment of ML pipelines in various ways: (1) *ML library vendors* lack awareness of the need to test the interactions between specific library versions that could be concurrently deployed by a pipeline. (2) *ML pipeline developers* lack advice on good and bad practices in ML pipeline configuration. (3) *Tool builders* lack scientific criteria to recommend appropriate versions of ML libraries to pipeline developers when maintaining and evolving ML libraries.

To fill the knowledge gap, we conduct a large-scale empirical study to understand the status quo of PLC issues. To facilitate scalable analysis, we develop Piecer, which automatically generates a set of pipeline variants by varying version combinations of ML libraries and compares three aspects of performance inconsistencies: *execution time*, *memory usage*, and *prediction performance* (*precision* and *recall*). To understand the status quo of PLC issues, we leverage Piecer to analyze 11,363 ML pipelines from various competition categories on Kaggle platform. The pipelines utilize 112 popular ML libraries. We investigate the pervasiveness and severity of PLC issues in ML pipelines and analyze their root causes of inducing performance inconsistencies. The main empirical findings are:

(1) We identify 601 library combinations commonly adopted by the 11,363 ML pipelines in our dataset.

(2) Among the 3,380 ML pipelines that we are able to deploy, 1,092 (32.3%) manifest significant performance inconsistencies on at least one variant. We find that 399, 243 and 440 pipelines among the deployable pipelines could achieve higher competition scores, shorter execution time and lower memory usage, respectively, using a different configuration of the library versions available at the time of competition.

(3) By analyzing the historical evolution data of 577 defective APIs that induce PLC issues, we identify four types of issue root causes. Specifically, PLC issues exposed in 164 defective APIs are affected by one library version. PLC issues in 3,720 pipeline variants are caused by combinations of APIs from different library versions.

In summary, this paper makes four major contributions:

- ***An Infrastructure for analyzing ML library version impacts.*** We developed an infrastructure, Piecer, to automatically generate pipeline variants with different ML library version combinations and analyze their performance inconsistencies.
- ***An empirical study on the impacts of various ML library version combinations.*** Leveraging Piecer, we conduct the first empirical study to systematically explore the impacts on different ML library version combinations and categorize the root causes of inducing performance inconsistencies.
- ***A repository of defective APIs for detecting PLC issues.*** Based on our empirical findings, we constructed a repository containing 164 defective APIs and 106 API combinations from 418 ML library versions, which shed light on designing automated approaches to detecting PLC issues. Based on the repository, we captured PLC issues in 309 real-world ML pipelines.

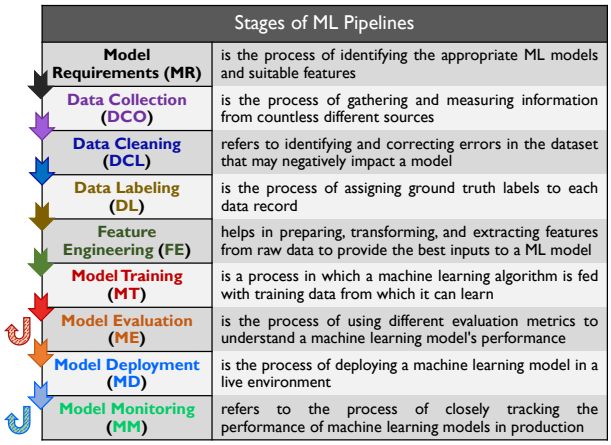

Figure 1: Description of the stages in ML pipeline

2 PRELIMINARIES

2.1 Background

ML Libraries: Machine learning comprises several kinds of learning algorithms such as *supervised learning*, *unsupervised learning*, *deep learning*, and *reinforcement learning*. To provide fast access to these algorithms and facilitate the complicated data analysis process, the open-source community developed two types of ML libraries: ***ML frameworks*** and ***data science libraries***. *ML Frameworks* (e.g., *Scikit-Learn* [9] and *PyTorch* [7]) provide ready-made algorithms and allow developers to easily apply ML-based solutions to applications. *Data science libraries* provide data visualization (e.g., *Matplotlib* [4]), efficient data structures (e.g., *Pandas* [6]), scientific computing functions (e.g., *Numpy* [5]) or other utilities.

ML Pipelines: The term *pipeline* corresponds to the *pipes-and-filter* design pattern that decomposes the software architecture into several stages with processing units (filters) and ordered connections (pipes). By the *ML pipeline*, we are referring to a nine-stage pipeline that goes through **data-oriented** (*collection*, *cleaning and labeling*) and **model-oriented** (*model requirements*, *feature engineering*, *training*, *evaluation*, *deployment*, and *monitoring*) stages.

Figure 1 explains each ML stage. According to the investigation results in studies [49, 85], most performance issues or crashes occurred in the *DCO*, *DCL*, *FE*, *MT*, and *ME* stages of ML pipelines. The five stages are commonly implemented based on various ready-made algorithms encapsulated in ML libraries.

2.2 Motivation

Since ML libraries rapidly evolve to add/remove/change features and fix bugs, various version combinations may dramatically affect pipelines' performance. Figure 2 shows an illustrative example of a PLC issue, which participates in the competition *MNIST* [26] hosted on Kaggle platform. This pipeline aims to identify digits from a dataset of tens of thousands of handwritten images, leveraging two ***TensorFlow*** APIs and eight ***Keras*** APIs to implement *DCL* and *MT* stages, respectively. Interestingly, when using different version combinations of ***TensorFlow*** and ***Keras***, the categorization accuracy of the pipeline varies from 0.559 to 0.997, leading to a rise in competition ranking from $523th$ to $1st$. Even worse, some version combinations cause crashes due to API-breaking changes.

{Keras, Tensorflow} Versions	Score (AUC)	Ranking	Time (ms)	Memory (MB)
{2.7.1, 2.7.0}	Crash	Crash	Crash	Crash
{2.4.3, 2.4.1}	0.768	522	1895.656	1244.446
{2.4.3, 2.3.1}	0.737	521	1882.099	1241.001
{2.4.3, 2.2.0}	0.559	523	1926.980	1248.518
{2.3.1, 2.4.1}	Crash	Crash	Crash	Crash
{2.3.1, 2.3.1}	Crash	Crash	Crash	Crash
{2.3.1, 2.2.0}	0.997	1	1877.330	1199.130
{2.3.1, 2.1.0}	0.997	1	1888.612	1202.602
{2.3.1, 2.0.0}	Crash	Crash	Crash	Crash
{2.3.1, 1.15.2}	0.997	1	1989.861	1194.425
{2.3.1, 1.14.0}	0.997	1	1853.423	1196.269
{2.3.1, 1.13.1}	0.997	1	1901.693	1183.123

Figure 2: An illustrative example of a PLC issue

Similar situations can be found in many issue reports. In the *PLC* issue#20642 [31] of project ***Sklearn***, a developer complained that the execution time of API ***Sklearn***.*cluster.KMeans()* in their ML pipeline varies significantly when using different versions of ***Numpy***. This issue attracted 34 comments from experienced developers to diagnose the root causes and was linked to four similar performance bugs induced by ML library versions (i.e., ***scipy***#15050, ***Sklearn***#21729, ***scipy***#15129, and ***Sklearn***#21808). The above examples motivate us to conduct a large-scale empirical study to investigate the impacts of different combinations of ML library versions on ML pipelines' performances, which can shed light on approaches for detecting or repairing PLC issues.

3 EMPIRICAL STUDY

Our study aims to answer the following three research questions:

- **RQ1 (Common ML Library Combinations):** ***What combinations of ML libraries do developers commonly use?*** To answer RQ1, we collect 11,363 published ML pipelines and analyze their library usage to identify common library combinations.
- **RQ2 (Impacts of Different ML Library Version Combinations):** ***Do different version combinations of ML libraries affect pipelines' performances?*** To answer RQ2, for each ML pipeline, we generate a series of variants with different ML library version combinations to inspect their performance inconsistencies. In particular, we define the generation rules of ML library version combinations, to systematically explore their impacts on pipelines' performances.
- **RQ3 (Root Causes of Performance Inconsistencies):** ***What are the root causes of pipelines' performance inconsistencies when adopting different version combinations of ML libraries?*** To answer RQ3, we consider the pipeline variants that induce (1) significant performance inconsistencies, (2) crashes and (3) *NaN* bugs as subjects, to analyze the corresponding root causes and triggering conditions.

3.1 Piecer

To scale up the impact analysis of possible ML library version combinations on pipelines' performances, we develop an infrastructure, Piecer (PIpEline Configuration ExploRe), to automate such a process. Figure 2 shows an overall architecture of Piecer. It mainly consists of three components:

Component 1: Constructing ML Library Version Pool. For a given ML pipeline, Piecer constructs a dependency pool $Dp = \{V_{Lib_1}, V_{Lib_2}, \cdots, V_{Lib_n}\}$ to collect all the installable version candidates $V_{Lib_i} = \{v_1, v_2, \cdots, v_m\}$ of referenced ML libraries Lib_i. To achieve this, it first parses the pipeline's dependency management script (*Requirement.txt*, *Setup.py* or *METADATA*) to identify all the versions of direct dependencies that satisfy the specified version constraints. For each version of direct dependency, it iteratively collects the version constraints of the required transitive dependencies. If a library Lib_i corresponds to multiple version constraints, Piecer considers their intersection as installable version candidates V_{Lib_i}. In the cases where a library Lib_i is specified with incompatible version constraints, we remove the versions of direct dependencies that induce such dependency conflicts from Dp.

Component 2: Generating Pipeline Variants with Different ML Library Version Combinations. To explore the impacts of library versions on a pipeline's performance, Piecer generates a set of pipeline variants with different version combinations of ML libraries selected from our dependency pool Dp. For a given ML pipeline, it considers the newest versions of referenced ML libraries as ***a version combination baseline.*** By defining a series of comparison rules (see definitions in Section 3.5), Piecer generates a set of ML library version combinations by varying the versions of concerned libraries and keeping the remaining library versions be consistent with the baseline.

According to the dependency resolution rules of Python build tool *pip*, it installs the newest library versions from PyPI central repository that satisfies the corresponding version constraint. To enable the installation of an expected version combination of ML libraries, Piecer customizes the dependency resolution rules of *pip* to select our specified library version candidates (satisfying the corresponding version constraints) from our dependency pool Dp.

Component 3: Analyzing Performance Inconsistencies. For each ML pipeline, Piecer runs the baseline and all the variants to check their performance inconsistencies across various library version combinations. It focuses on three aspects of performance inconsistencies: *execution time*, *memory usage*, and *evaluation metrics* (e.g., *Precision* and *Recall*). Specifically, Piecer performs two tasks:

- *Measuring performance inconsistencies:* For each library API referenced by the pipeline, it leverages measurement functions to capture the performance inconsistencies between the variant and baseline. For example, Piecer adopts functions ***timeit***.*default_timer()*, ***tracemalloc***.*get_traced_memory()* and ***pynvml***.*nvml DeviceGet-MemoryInfo()* to measure the execution time, and CPU and GPU memory usage of the changed APIs, respectively. The evaluation metrics can be implemented with the aid of functions ***sklearn***.*metrics.accuracy_score()* and ***sklearn***.*metrics.recall_score()*, etc. For the pipelines involving deep learning models, Piecer retrains the model to capture the impacts of library version changes precisely. To reduce the interference of random factors, it runs each pipeline five times and averages the outcomes.
- *Recording the problematic library version combinations:* Compared with the baseline, Piecer records the library version combinations (with the concerned APIs) that (1) induce significant performance inconsistencies of pipelines (see definitions in Section 3.5); (2) cause program crashes or *NaN* bugs, for further analysis.

3.2 Data Collection

We collected pipelines from Kaggle [2], the most popular crowdsourced platform for ML competitions. We select Kaggle pipelines

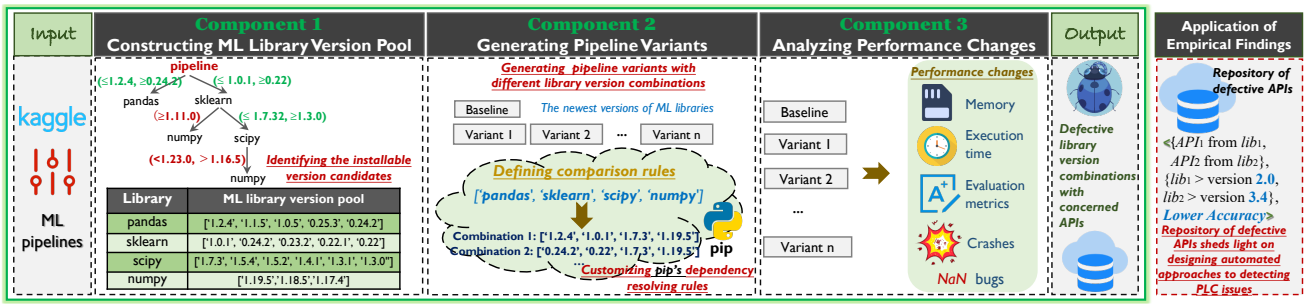

Figure 3: The overall architecture of PIECER (Repository of defective APIs is described in Section 4)

as subjects to conduct our empirical study for four reasons: (1) all the ML pipelines in a competition correspond to the unified evaluation metrics for ranking, which enable us to compare the outcomes across different combinations of library versions; (2) each pipeline provides its corresponding source code and dataset, which facilitate our reproduction; (3) competitions in KAGGLE platform cover a wide range of domains and techniques, which guarantee the diversity of ML tasks; (4) the small size pipelines on KAGGLE have few interference factors (e.g., environmental dependencies), which enables our analysis for the root causes of PLC issues. We did not select open-source large-scale ML pipelines as subjects, because they rarely provide (1) detailed training steps in documentation, (2) complete datasets, and (3) all the required dependencies. Therefore, we face challenges on deploying the complex pipelines. However, the common ML library version combinations identified in Kaggle pipelines are also adopted by large-scale ML pipelines (see statistics in Section 4). As such, the empirical findings identified in Kaggle pipelines can scale to more complex ML-based projects.

We collected the ML pipelines from KAGGLE competitions that satisfy three criteria. First, the competitions were hosted in the last three years (from Jan 1, 2019 to Jul 31, 2021). The search returned 168 competitions that involve 14,772 pipelines. Second, we only kept the competitions that contain more than 100 competing pipelines (i.e., popularity). Third, the pipelines should depend on more than three ML libraries, including both ML frameworks and data science libraries (i.e., complexity). With the above process, we finally obtained 11,363 pipelines from 42 competitions.

In total, our collected pipelines depend on 112 ML libraries, including 23 ML frameworks and 89 data science libraries. Among them, the top five most frequently used libraries are *Pandas* (13,054), *Sklearn* (7,821), *Numpy* (6,967), *Tensorflow* (3,279) and *PyTorch* (2,836). Figure 4 shows the demographics of the 112 ML libraries. On average, each ML library has 29,156±338 downstream projects, 4,094,010±3,285 monthly downloads, 7,460±121 stars, and 603±45 issues and 4,358±198 commits in their GitHub repositories. According to a ranked list of awesome ML libraries provided by a popular GitHub project (updated weekly) [8], the 112 libraries cover 25 categories (including image & video processing, etc.). The above statistics demonstrate that the ML libraries used in KAGGLE pipelines are also widely used in large amounts of open-source ML pipelines, indicating their representativeness.

3.3 RQ1: Common ML Library Combinations

Study Methodology. A combination of ML libraries are typically used in certain stages of ML pipeline workflows [38]. To granularly identify common library combinations, we analyze the library usage in pipeline workflow stages. Specifically, we perform two tasks in

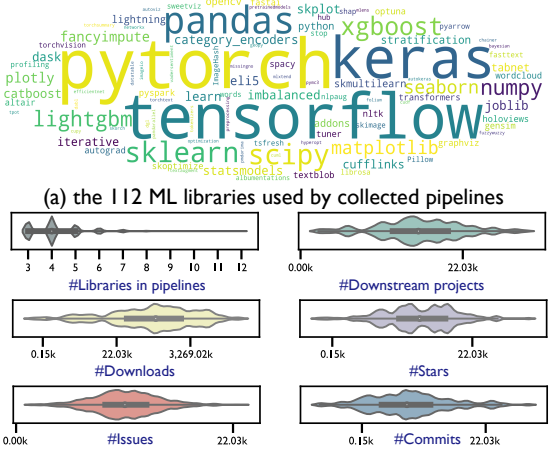

Figure 4: Description of the stages in ML pipeline

Table 1: Occurrences of ML library combinations in the pipelines

Occurrences	*DCO*	*DCL*	*FE*	*MT*	*ME*	*Data Oriented Stages*	*Model Oriented Stages*
[1,10)	125	80	386	425	23	2,124	641
[10,50)	34	13	66	60	5	161	73
[50,100)	10	6	9	8	0	18	16
[100,500)	2	11	16	14	2	11	21
[500, ∞)	4	1	2	5	1	2	4
Avg	59	90	21	20	275	5	14
#Common	16	13	55	56	2	372	87

† *Data Oriented Stages* = *DCO* + *DCL* + *FE*; *Model Oriented Stages* = *MT* + *ME*.
Avg denotes the average occurrences of ML library combinations in each stage;
#Common denotes the number of common library combinations;
We consider a library combination is commonly adopted by ML pipelines if its occurrences in each stage is above ***Avg***.

the investigation: (1) Leveraging JEDI [3], a static analysis tool for Python, we located 8,853 unique ML library APIs invoked by our collected 11,363 pipelines. (2) Two authors of this paper assigned label of workflow stages for each ML library API. The labels were assigned based on the code comments in ML pipelines for stating the API usage, or the description of the package (or sub-package), in which the API was declared. For example, we considered API ***xgboost***.*score()* worked in the *ME* stage, since its declaration was accompanied by a explanatory code comment "*#Evaluate model performance*" in the ML pipeline. We did not assign labels for the APIs that (*i*) did not have a description in its package or (*ii*) the two authors could not reach a consensus on the descriptions. We employed *Cohen's Kappa* [73] to measure the consistencies between the two authors. If the *Kappa* value was less than 0.9, the authors needed to discuss to resolve disagreements. This iterative process would stop once the *Kappa* value was equal to or greater than 0.9, indicating substantial agreement. Eventually, we obtained 7,983

Table 2: Statistics of top 5 common ML library combinations in single-stage

Rank	*Data COllection*		*Data CLeaning*		*Feature Engineering*		*Model Training*		*Model Evaluation*	
	Combinations	*Occurrences*	*Combinations*	*Occurrences*	*Combinations*	*Occurrences*	*Combinations*	*Occurrences*	*Combinations*	*Occurrences*
1	Pandas	3,858	Numpy	1,745	Sklearn	1,295	Sklearn	1,503	Sklearn	5,940
2	Numpy, Pandas, PyTorch	807	PyTorch	428	Numpy	701	Tensorflow	1,422	Tensorflow	284
3	Pandas, PyTorch	553	Nltk	358	Keras	385	PyTorch	993	Sklearn, Tensorflow	258
4	Nltk, Pandas	521	Tensorflow	332	Tensorflow	359	Keras	851	Keras, Sklearn	12
5	OpenCV, Numpy, Pandas, PyTorch	487	Pandas	277	Pandas	357	Lightgbm	519	Pytorch-lightning, Sklearn	11

† *The top common ML library combinations in multi-stages are provided on our website* **http://piecer-plc.github.io**

labeled ML library APIs. The labelling process took two months for the two authors who have over two years ML development experience to analyze and perform cross validation. We released the dataset on the website for public scrutiny.

Results. By inspecting the 7,983 labeled ML library APIs, we observed that the pipelines from Kaggle platform only involved five stages of workflows: *DCO*, *DCL*, *FE*, *MT*, *ME*. As aforementioned, recent empirical studies [49, 85] revealed that the majority of performance issues or crashes occurred in the five stages. In our study, we focus on these core stages of pipeline workflows, to identify their common combinations of library usage.

Table 1 summarizes the occurrences of ML library combinations in the single-stage/multi-stages of pipeline workflows. We identified 176, 112 and 32 library combinations in the *DCO*, *DCL* and *ME* stages, respectively. Since the open-source community provides various solutions to help developers extract critical features from raw data and automatically train their ML models, 480 and 513 library combinations were identified in the *FE* and *MT* stages of our collected pipelines. Our results reveal that 16/176, 13/112, 55/480, 56/513 and 2/32 of library combinations were commonly adopted by the majority of ML pipelines to implement *DCO*, *DCL*, *FE*, *MT* and *ME* stages, respectively (occurrences $\geq$ *Avg*). The combinations may include both ML frameworks and data science libraries. Table 2 illustrates the top 5 common library combinations in single-stage. For example, 3,858 pipelines use *Pandas* in *DCO* stage, while 807 pipelines combines *Numpy*, *Pandas* with *PyTorch* for data collection.

As shown in Table 1, during the synthesis of the data oriented stages (*DCO*, *DCL* and *FE*), we identified the use of 2,316 library combinations. 372 out of the 2,316 combinations occurred in more than five pipelines. We also identified 755 ways of implementing model oriented stages (*MT* and *ME*) using combinations of libraries. 87 of them were commonly used in our collected pipelines. The top common ML library combinations used in multi-stages are provided on our website. We observed that 1,134 pipelines adopts a combination of *Pandas*, *Numpy* and *Sklearn* for data oriented stages. 725 pipelines combine *TensorFlow* with *Sklearn* for modeling oriented stages.

> ***Answer to RQ1:*** *We identified 16, 13, 55, 56 and 2 common ML library combinations in the DCO, DCL, FE, MT and ME stages, respectively. In the data oriented stages (DCO, DCL and FE) and model oriented stages (MT and ME), we identified 372 and 87 common ML library combinations, respectively. Such common ML library combinations are adopted by the majority of ML pipelines.*

3.4 RQ2: Impacts of Different ML Library Version Combinations

Study Methodology. We present the study methodology of RQ2 in three aspects: *Subject selection*, *Generation rules of library version combinations* and *Experiment setup*.

Subject selection. Since each pipeline corresponds to a series of variants, the comparison experiment is time-consuming and would take lots of machine resources. To guarantee the feasibility of our RQ2 study, we set three filter conditions to select subjects from our collected 11,363 pipelines:

- 1,490 pipelines that do not use common ML library combinations in the single- or multi-stages are filtered out.
- 4,299 pipelines that involve the private datasets (not published on Kaggle) are filtered out.
- 157 pipelines that use the private libraries (could not be found from PyPI or GitHub repositories) are filtered out.
- We restrict the dataset size of pipelines to 100GB. 2,037 pipelines are filtered out according to this condition.

Eventually, we obtained 3,380 pipelines, involving 13 ML frameworks and 77 data science libraries. The 90 ML libraries cover 80.4% of popular ones we presented in Figure 4.

Generation rules of library version combinations. We deployed *Component 1* of Piecer to take each pipeline as an input to construct a dependency pool $Dp = \{V_{Lib_1}, V_{Lib_2}, \cdots, V_{Lib_n}\}$, by collecting all the installable version candidates of the referenced ML libraries. To reduce the computation complexity, in our study, we only considered popular versions of each ML library in Dp that are widely used by downstream users (the popularity of library versions are provided on *Libraries.io* website). Leveraging *Component 2* of Piecer, we generate a set of pipeline variants with different ML library version combinations. To this end, we define two generation rules of ML library version combinations, to systematically explore their impacts on pipelines' performances:

- ***Rule 1****: Varying version combinations of ML libraries in the single-stage:* For a given ML pipeline, Piecer iteratively varies the version combinations of ML libraries used in each workflow stage (*DCO*, *DCL*, *FE*, *MT* and *ME*), while keeps the remaining libraries to be the newest versions in our dependency pool Dp. Let $L = \{Lib_1, Lib_2, \cdots, Lib_m\}$ be a ML library combination (direct dependencies) used in a certain workflow stage. Each library $Lib_i \in L$ corresponds to a set of version candidates $V_{Lib_i} = \{v_1, v_2, \cdots, v_t\}$ in our dependency pool Dp. Piecer determines a set of version combinations of ML libraries used in a certain workflow stage as $LV = V_{Lib_1} \times V_{Lib_2} \cdots \times V_{Lib_m}$, where $Lib_i \in L$ and the symbol "$\times$" denotes *Cartesian product*.
- ***Rule 2****: Varying version combinations of ML libraries in the multi-stages:* Piecer iteratively varies the version combinations of ML libraries used in data-oriented stages (*DCO*, *DCL* and *FE*) and model-oriented stages (*MT* and *ME*), while keeps the remaining libraries to be the newest versions in our dependency pool Dp. Let $LV(DCO)$, $LV(DCL)$, $LV(FE)$, $LV(MT)$ and $LV(ME)$ be the sets of installable version combinations of ML libraries used in *DCO*, *DCL*, *FE*, *MT* and *ME* workflow stages, respectively

Table 3: Statistics of pipeline variants generated by PIECER

#Pipeline	#Variant	#Variants generated by *Rule 1*					#Variants generated by *Rule 2*		
		DCO	*DCL*	*FE*	*MT*	*ME*	*Data oriented*	*Model oriented*	*Overall architecture*
3,380	74,373	357	447	867	4,333	530	12,741	21,359	33,639

† *We filtered out the overlapping pipeline variants generated by Rules 1 and 2*

(generated based on *Rule 1*). PIECER determines the ML library version combinations in the multi-stages as follows: (1) *Data-oriented stages*: $LV(Data) = LV(DCO) \times LV(DCL) \times LV(FE)$. (2) *Model-oriented stages*: $LV(Model) = LV(MT) \times LV(ME)$. (3) *Overall architecture*: $LV(Overall) = LV(Data) \times LV(Model)$.

Table 3 shows the statistics of pipeline variants. In total, PIECER generated 74,373 variants for the 3,380 pipelines, based on the generation rules of library version combinations. On average, each pipeline corresponds to 22±8 variants.

Experiment setup. For a ML pipeline, PIECER considers the newest versions of referenced ML libraries as a version combination baseline. Using *Component 3*, we run the baseline and all its corresponding variants, to capture the performance inconsistencies. Based on confidence interval analysis [57], we consider the library version combinations induce performance inconsistencies, if:

$$\exists Metric_b \mid Metric_v \notin [\bar{x} - z\frac{s}{\sqrt{n}}, \bar{x} + z\frac{s}{\sqrt{n}}] \quad (1)$$

where $Metric_b$ and $Metric_v$ denote the *execution time*, *memory usage* or *competition scores* of baseline and a pipeline variant, respectively; $\bar{x}$ and s are the average value and standard deviation of $Metrics_v$, respectively; n denotes the total number of pipeline variants. According to the parameter setting and z table provided in approach [80], we calculate the 99.9% confidence interval of the mean and the z value is taken as 3.29053. Note that, *competition scores* (e.g., *Precision* and *Recall*) are the metrics provided by KAGGLE for evaluating the pipelines' outcomes.

We use *Cohen*'s d [35] to measure how far the performance inconsistent outcomes exceed the confidence interval. According to formula (2), we divide the outcomes into two groups: performance inconsistency group and normal group. *Cohen*'s d is used to describe the standardized mean difference between two group means.

$$d = \frac{\bar{x_1} - \bar{x_2}}{\sqrt{\frac{(n_1-1)s_1^2+(n_2-1)s_2^2)}{n_1+n_2-2}}} \quad (2)$$

In this formula, $\bar{x_1}$ and $\bar{x_2}$ are the mean values of performance inconsistency group and normal group, respectively; n_i is the sample size (i.e., number of outcomes in each group), and s_i is the sample variance of each group. According to approach [66], *Cohen*'s d interprets effect sizes in the meta-analysis: small effect = 0.2 - 0.5; medium effect = 0.5 - 0.8; large effect = 0.8 and higher. In our study, we consider the outcomes with large effect size as the ***significant performance inconsistencies*** induced by pipeline variants.

Our study is conducted on three machines with the same configuration: Intel Xeon Silver 4110 machine with 128GB RAM, Ubuntu 16.04.7 LTS, and two NVIDIA Tesla V100 GPUs. The comparison experiments took about eight months for three authors of this paper to deploy and run the 74,373 variants of 3,380 pipelines.

Results. Among the 3,380 pipelines, 1,092 (32.3%) manifest significant performance inconsistencies on at least one variant. 42,488 and 536 variants (involving 2,323 pipelines) induce program crashes and *NaN* bugs, respectively.

Figure 5 and Table 4 summarize the performance inconsistencies of the 1,092 pipelines. We can observe that:

Figure 5: Performance inconsistencies: *Baseline v.s. Variants*

Table 4: Performance inconsistencies: *Baseline v.s. Variants*

Competition Score	Execution Time	Memory Usage	#Pipeline	#Variants
⇑	⇑	⇑	141	596
⇑	⇑	⇓	97	318
⇑	⇓	⇑	158	742
⇑	⇓	⇓	125	471
⇑	⇓	-	143	404
⇑	⇑	-	128	367
⇓	⇑	⇑	144	570
⇓	⇑	⇓	115	329
⇓	⇓	⇑	184	960
⇓	⇓	⇓	151	526
⇓	⇓	-	154	406
⇓	⇑	-	125	346
⇑	-	-	249	587
⇓	-	-	275	614
-	⇑	-	465	1,535
-	⇓	-	597	1,566
-	-	⇑	644	2,488
-	-	⇓	652	2,422

"⇑/⇓" *denotes that at least one pipeline variant achieves* better/worse *performance than the baseline; "-" denotes unchanged performances.*

Table 5: Impacts of library versions on competition ranking

Competitions	Rise /Fall	#pipeline variants that affect competition rankings									
		R: 0%–10%	*R: 10%–20%*	*R: 20%–30%*	*R: 30%–40%*	*R: 40%–50%*	*R: 50%–60%*	*R: 60%–70%*	*R: 70%–80%*	*R: 80%–90%*	*R: 90%–100%*
Competition 1 [14]	↑	219	5	2	3	17	2	0	0	0	0
	↓	262	0	7	124	0	32	22	0	0	0
Competition 2 [16]	↑	184	67	29	15	3	1	0	1	0	0
	↓	120	38	20	30	8	22	16	5	0	0
Competition 3 [17]	↑	142	38	12	0	0	0	0	0	0	0
	↓	77	21	18	1	2	0	0	0	0	0
Competition 4 [18]	↑	705	17	0	0	0	8	0	2	6	0
	↓	677	35	9	2	0	7	12	0	0	0
Competition 5 [19]	↑	1,144	62	4	13	10	12	35	18	1	0
	↓	1,230	218	81	59	87	110	145	238	45	0
Competition 6 [20]	↑	126	0	0	0	0	0	0	0	0	0
	↓	52	0	71	0	0	0	0	0	0	0
Competition 7 [21]	↑	188	1	0	1	0	0	0	0	11	0
	↓	348	14	10	16	0	1	0	0	0	0
Competition 8 [22]	↑	54	23	18	11	12	0	0	0	0	0
	↓	59	17	5	1	0	4	3	1	0	0
Competition 9 [23]	↑	24	2	2	0	0	0	0	0	0	0
	↓	14	2	2	0	24	0	0	3	0	0
Competition 10 [15]	↑	4	2	3	0	0	0	0	55	0	0
	↓	0	9	0	0	0	0	0	0	0	0

$R = (Rank_b - Rank_v) / Rank_b$, *where* $Rank_b$ *and* $Rank_v$ *are competition rankings of the baseline and variant, respectively.*

- For 399, 243 and 440 pipelines, changing the version combinations of ML libraries from the baselines (i.e., the newest versions) could achieve better performances in *competition scores*, *execution time* and *memory usage*, respectively. The variants of 109

Table 6: Statistics of the ML libraries whose varied versions significantly affect pipelines' performances

Metrics	Type	ML Libraries × #pipelines	#Problematic APIs				
			DCO	DCL	FE	MT	ME
Competition Score	Baseline better	**Pandas** × 1, **Opencv-python** × 1, **Nltk** × 1, **Numpy** × 1, **Torchvision** × 1, **Autokeras** × 1 **Catboost** × 2, **Sklearn** × 39, **Keras** × 52,**Tensorflow** × 72	6	3	20	109	6
	Baseline worse	**Pandas** × 1, **Transformers** × 1, **Opencv-python** × 2, **Numpy** × 2, **Sklearn** × 9, **Keras** × 12, **Tensorflow** × 24	5	4	11	61	4
Execution Time	Baseline better	**Scipy** × 3, **Sklearn** × 135, **Opencv-python** × 18, **Nltk** × 31, **Category_encoders** × 13, **Numpy** × 12 **Pandas** × 10, **Gensim** × 3, **Transformers** × 11, **Catboost** × 40, **Xgboost** × 63, **TensorFlow** × 191 **Lightgbm** × 58, **Keras** × 146, **Spacy** × 22	15	22	80	348	28
	Baseline worse	**Autokeras** × 1, **Torchvision** × 3, **Numpy** × 7, **Opencv-python** × 7, **Catboost** × 9, **Nltk** × 12 **Pandas** × 18, **Keras** × 682, **Tensorflow** × 789, **Sklearn** × 795	5	14	42	159	14
Memory Usage	Baseline better	**Optuna** × 1, **Spacy** × 1, **Catboost** × 1, **Transformers** × 2,**Numpy** × 2, **Nltk** × 2, **Pytorch** × 3 **Lightgbm** × 3, **Pandas** × 4, **Keras** × 4, **Xgboost** × 5, **Sklearn** × 11,**Tensorflow** × 14	4	9	16	92	6
	Baseline worse	**Keras** × 1, **Numpy** × 1, **Xgboost** × 1, **Pandas** × 2, **Sklearn** × 3	3	4	3	7	5

#pipeline denotes the number of pipelines that adopt the ML libraries.

pipeline manifested over 50% of performance differences compared with the baselines. For example, *CFEC#1254* [11] adopts the APIs of library *Sklearn* ***sklearn***.*neighbors.KNeighborsClassifier()* and ***sklearn***.*naive_bayes.GaussianNB()* in *MT* workflow stage. Combining older version *Sklearn* 0.19.2 with other ML libraries' newest versions, the pipeline achieves significant improvement in *execution time* and *memory usage*, compared with the baseline (22.6s & 409.8MB ***v.s.*** 451.2s & 2,337.9MB).

- For 252, 340 and 209 pipelines, the baselines outperform all the other ML library version combinations in *competition scores*, *execution time* and *memory usage*, respectively. In 496 pipelines, the newest versions of ML libraries manifested over 50% of performance differences, compared with the average outcomes of variants. For example, *TPS-#839* [32] uses the API of library *Catboost* ***Catboost***.*CatBoostRegressor()* in *MT* workflow stage. The baseline significantly outperforms the other pipeline variants in *execution time* (14.0s ***v.s.*** 141.6s).
- As shown in Table 4, compared with baselines, the variants of 1,057 pipelines achieved gain on certain performance metrics, with a drop of other ones. The variants of 141 and 151 pipelines manifest an overall rise and fall in performance metrics, respectively. The results indicate that a significant proportion of performance changes are the issues deserved further investigations, rather than trade-offs between different measurements.

In general, ML pipelines with different library combinations more easily induce significant inconsistencies of *execution time* and *memory usage*. 60.0% of 653 pipelines still manifest the inconsistencies of *competition scores* on the variants. Table 5 illustrates ten competitions whose rankings (determined by competition scores) can be affected by changing the ML library versions in pipelines. Interestingly, we observed that the rankings in *Competitions 5* [19], *9* [23] and *10* [15] even changed dramatically on the pipeline variants. For example, among the 4,668 variants generated by Piecer for the 249 pipelines in *Competitions 5* [19], 1,299 and 2,213 variants can cause the rise and fall in ranking, respectively. In particular, MNIST#85 [27] moved up from 201*th* to 5*th* place in the ranking, after changing the versions of libraries keras 2.7.0 and tensorflow 2.7.0 to keras 2.3.1 and tensorflow 2.1.1, respectively.

Piecer exposed PLC issues in 37 ML libraries, involving 890 APIs. Table 6 shows statistics of the ML libraries whose versions significantly affect pipelines' performances. We can observe that:

- The defective ML libraries include both ML frameworks and data science libraries. For each ML library, a proportion of APIs perform best in the newest versions, while the other APIs achieve the optimal performances in their old versions. For example, *TensorFlow* provides 41, 125 and 65 APIs whose newest versions negatively affect the pipelines' *competition scores*, *execution time* and *memory usage*, respectively. Besides, the three metrics reward the 200 APIs of *TensorFlow* in the newest versions.
- Among the five workflow stages, *MT* involves the most library APIs (479 APIs from 22 libraries) that can induce PLC issues into 892 ML pipelines.

Answer to RQ2: *Among the 3,380 deployable ML pipelines, 1,092 (32.3%) manifest significant performance inconsistencies on at least one variant. For 399, 243 and 440 pipelines, changing the version combinations of ML libraries from the baselines (i.e., the newest versions) could achieve better performances in competition scores, execution time and memory usage, respectively. For 252, 340 and 209 pipelines, the baselines outperform all the other ML library version combinations in competition scores, execution time and memory usage, respectively.*
Implication: *Due to the version constraints between dependencies, or the limitations of the installation environment, various library version combinations are likely to be adopted in ML pipelines. Understanding their impacts on pipeline performances helps developers avoid PLC issues.*

3.5 RQ3: Root Causes of Performance Inconsistencies

Study Methodology. To further analyze the root causes of PLC issues, we randomly sampled: (a) 294 out of the 1,092 pipelines that induced significant performance inconsistencies after varying library versions, (b) 329 out of the 2,276 pipelines causing crashes, and (c) 42 out of the 47 pipelines inducing *NaN* bugs. This sampling approach [53] guarantees that the root causes distilled from the sampled pipelines can be generalized to the whole dataset with a 95% confidence level and a 5% confidence interval. Specifically, we performed four tasks as follows.

- For each pipeline variant, we identified a collection of ML library APIs $A = \{a_1, a_2, \cdots, a_n\}$ that induced PLC issues, using *Component 3* of Piecer. Let $a_1.Libv$ be the ML library version that contains API $a_i \in A$. In total, we obtained 577 APIs from 452 library versions in the 19,048 pipeline variants.
- We performed an in-depth analysis on defective APIs by inspecting their *historical evolution data* across the ML library versions, including release notes, issues, code commits and code comments (on GitHub repositories). To understand whether the PLC issue of an API $a_i \in A$ was affected by a combination of library versions, we divided the APIs into two groups:
  - ***Group 1***: We consider a PLC issue is ***induced by the individual library***, if it satisfies: (1) the issue is exposed in API $a_i \in A$; (2) among the library versions used by the pipeline variant, only $a_i.Libv$ is inconsistent with that in baseline. 295 APIs $a_i \in A$ inducing PLC issues in 9,055 pipeline variants fall into *Group 1*. We considered the PLC issues were induced by the evolution of library $a_i.Libv$, and focused on its historical data for inspection.

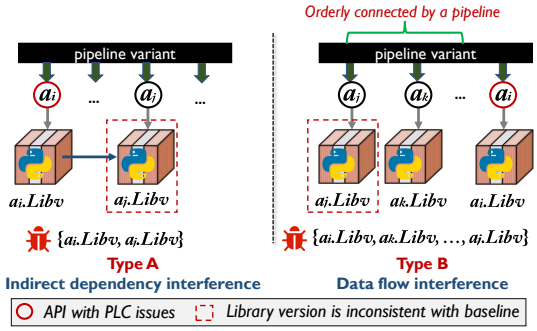

Figure 6: Usage patterns of defective library version combinations

- ***Group 2***: As shown in Figure 6, we divided the PLC issues ***induced by combinations of library versions*** into two types:
  Type A. Suppose a pipeline variant directly uses APIs a_i and a_j from different ML libraries, and a_i transitively invokes the APIs from library $a_j.Libv$. We consider a PLC issue is induced by ***indirect dependency interference***, if it satisfies: (1) the issue is exposed in API $a_i \in A$; (2) among the library versions used by the pipeline variant, only $a_j.Libv$ is inconsistent with that in baseline. We then consider $\{a_i.Libv, a_j.Libv\}$ is a defective library version combination. For the 98 defective APIs in 3,799 pipeline variants falling into *Type A*, we inspected their historical evolution data.
  Type B. Type B. Suppose APIs a_j, a_k, $\cdots$, a_u and a_i from different libraries are orderly connected by a pipeline, and the outcomes of APIs a_j, a_k, $\cdots$, a_u can affect the inputs of a_i. We consider a PLC issue is induced by ***data flow interference***, if it satisfies: (1) the issue is exposed in API $a_i \in A$; (2) among the library versions used by the pipeline variant, only $a_j.Libv$ is inconsistent with that in baseline; (3) a_i does not invoke APIs from $a_j.Libv$. We then consider $\{a_j.Libv, a_k.Libv, \cdots, a_u.Libv, a_i.Libv\}$ is a defective library version combination. We inspected the historical evolution data of defective API a_j and analyzed the impacts of defective library version combination.
- We followed an *open coding procedure* [36], a widely-used approach for qualitative research, to distill and categorize the root causes of performance inconsistencies. Initially, two authors of this paper, who had over two years ML development experience, independently analyzed the release notes, issues, code commits and code comments of 577 defective APIs across the library versions in variant and baseline. After the first round of analysis and labeling, the two authors gathered to compare and discuss their results, in order to adjust the taxonomy, with the help of a third author to resolve conflicts. In this manner, we constructed the pilot taxonomy. This led to a more clear-cut labeling strategy. Next, the first two authors continued to label the root causes of the remaining 306 APIs' performance inconsistencies and iteratively refined the results. The conflicts of labeling were further discussed during meetings and resolved by the third author. We adjusted the pilot taxonomy and obtained the final results.
- For each defective API, we identified all the pipeline variants that use it from the same library version, to further inspect whether they have similar performance inconsistencies. Such inspections helped us understand the triggering conditions of PLC issues.

Results. For the 577 defective APIs, we identified the root causes of PLC issues by analyzing related library release notes, issues and code commits. We identified four common root causes that can arise in two scenarios: *performance impacts induced by individual library* and *induced by library version combinations*.

Performance impacts induced by individual ML library. PLC issues of 164 defective APIs were affected by one library version. The issues share four common root causes below.

- ***API Optimization*** (96/164 = 58.5%). For 96 defective APIs, we found descriptions in the historical evolution data for declaring the optimization for *hardware support*, *data processing*, and *calculations*, etc. For example, in NLPDT#plyger [29], the memory usage of API ***Transformers***.*trainer()* is changed from 2MB to 91MB, when downgrading library ***Transformers*** to 4.11.3 or lower versions. In commit log *7a0adbb* [13], developers stated that they added support for gradient checkpointing in BERT since ***Transformers*** 4.12.0 for optimizing memory performance. The API optimization results in the improvement of execution time (36/96), memory usage (32/96) and competition scores (28/96). On average, the library versions containing non-optimized APIs still have 45,647 monthly downloads in PyPI.
- ***Default Hyperparameter Changes*** (16/164 = 9.8%). For 16 defective APIs, developers added, deleted, or changed the values of default hyperparameters for training ML models, leading to performance inconsistencies across different library versions. The above clues can be located in the historical evolution data of libraries. For example, the release note [30] of library ***Sklearn*** 0.22.0 states that developers change a default hyperparameter *'gamma'* in API ***Sklearn***.*svm.SVC()*. We observed that the accuracy of MNIST#85 [27] rose from 0.143 to 0.986 after upgrading ***Sklearn*** to 0.22.0 or higher versions.
  The symptoms of changing default hyperparameters are manifested as the improvement of execution time (5/16), memory usage (7/16), competition scores (2/16) or cause *NaN* bugs (2/16). For each API with changed default hyperparameters, all the pipeline variants using it could reproduce the symptoms of performance changes, in the cases that developers did not customize the arguments. However, such library versions with poor performances still have 905,851 monthly downloads on average.
- ***Technical Debts*** (5/164 = 3%). For 5 defective APIs, we found declarations in their code comments stating that there are technical debts in the old library versions. Developers either considered them as experimental APIs or incomplete implementations. For example, in the code comments of API ***Sklearn***.*ensemble.HistGradientBoostingClassifier()*, developers explained that they improved this experimental API since ***Sklearn*** 0.24.2. We observed that the memory usage of CFECII#385 [12] reduced from 105MB to 55MB, after upgrading ***Sklearn*** to 0.24.2 or higher versions.
  The symptoms of technical debts are manifested in the bad performance of execution time (2/5), memory usage (2/5) or competition scores (1/5) . By inspection, for each API with technical debts, all the pipeline variants adopting it could manifest the similar performance changes. This means that such PLC issues can be easily triggered if any ML pipeline uses our identified defective APIs. On average, the above library versions with technical debts have 589,3827 monthly downloads.
- ***API Breaking Changes*** (47/164 = 28.7%). For 47 defective APIs, developers introduced breaking changes (i.e., *API addition*, *removal* and *signature changes*) during library evolution. They

induced 41,204 crashes and 492 *NaN* bugs in the pipeline variants. Digging out the above API breaking changes in the popular library versions can help developers avoid compatibility issues.

Performance impacts induced by ML library version combinations. According to the definition of *Types A* and *B* issues in Group 2, *for the defective library version combinations, the root causes of problems may arise from one ML library, but the PLC issues are actually exposed in the APIs from other libraries, due to the interactions between libraries in the pipelines.* By inspecting historical evolution data of the concerned defective APIs, we observed that the root causes can be *API Optimization*, *Default Hyperparameter Changes*, *Technical Debts* or *API Breaking Changes*. In total, we identified PLC issues in 3,720 pipeline variants that are affected by 106 defective API combinations from different ML library versions.

- *Type A: Indirect Dependency Interference (41/106 = 38.7%).* We located 41 *Type A* library version combinations, involving 57 APIs from 10 ML libraries. 2,511 pipeline variants adopting the defective library version combinations manifested the poor performance of execution time (15/41), memory usage (11/41), competition scores (5/41) or induced crashes (7/41) and *NaN* bugs (3/41). For example, pipeline CFEC#1211 [10] directly depends on ML libraries {***Sklearn*** 1.0.1, ***Pandas*** 0.25.0} in the *MT* stage. The pipeline invokes API ***Sklearn***.*logisticRegression()*, which transitively depends on API ***Pandas***.*rolling()* from library ***Pandas***. We observed the memory usage of ***Sklearn***.*logisticRegression()* changed from 593MB to 1,329MB after downgrading ***Pandas*** to version 0.24.2. After investigation, we found an issue ***Pandas***#25893 [25] that API ***Pandas***.*rolling()* suffered from memory bugs in ***Pandas*** 0.24.2. The combination of ***Sklearn*** and ***Pandas*** is used by 19,774 pipelines on Kaggle and 1,616 of them invoke ***Sklearn***.*linear_model.logisticRegression()*. Such defective versions are liable to induce PLC issues.
- *Type B: Data Flow Interference (65/106 = 61.3%).* We located 96 *Type B* library version combinations from 4,972 pipeline variants, involving 121 APIs from 15 ML libraries. The above defective library version combinations caused inferior performance of execution time (24/96), memory usage (16/96), competition scores (19/96) or induced crashes (3/96) and *NaN* bugs (3/96). For example, pipeline NLPDT#785 [28] invokes APIs ***Nltk***.*stem.porter.PorterStemmer()*, ***Sklearn***.*extraction. text.TfidfVectorizer()* and ***Catboost***.*CatBoost Classifier()* from libraries {***Nltk*** 3.6, ***Sklearn*** 1.0.1, ***Catboost*** 1.0.3}. The outcomes of former two APIs can affect the inputs of ***Catboost***.*CatBoostClassifier()* via data flow interactions in the pipeline, while the former two APIs are not invoked by the latter one. After downgrading ***Nltk*** to version 3.5, we observed the execution time of ***Catboost***.*CatBoostClassifier()* varied from 16s to 36s. The combination of ***Nltk***, ***Sklearn*** and ***Catboost*** is adopted by 310 pipelines on Kaggle, 42% of them invoked the concerned three APIs. Such PLC issues can be easily triggered since the buggy version ***Nltk*** 3.5 has 831,749 monthly downloads.

Answer to RQ3: *By analyzing the historical evolution data of 577 defective APIs, we identified four types of essential root causes of inducing PLC issues. PLC issues exposed in 164 defective APIs are affected by one library version. PLC issues in 3,720 pipeline variants are caused by combinations of APIs from different library versions.*
Implication: *Future research may focus on designing approaches to detecting and repairing PLC issues by automatically mining and monitoring the evolution of defective ML library APIs.*

4 APPLICATION OF EMPIRICAL FINDINGS

Repository of Defective APIs. Let a 3-tuple $D =< A, LV, P >$ be a defective API (combination) whose invocation(s) potentially trigger PLC issues, where A denotes the signature(s) of API (combination); LV denotes the defective library version range; P denotes the aspects of performance changes (e.g., execution time, memory usage and *Precision*). For example, $< \{\textbf{Nltk}.stem.porter.PorterStemmer(), \textbf{Catboost}.CatBoostClassifier()\}, \{\textbf{Catboost} = 1.0.3, \textbf{Nltk} = 3.5\}, Executiontime >$ denotes an API combination of **Nltk**.*stem. porter.PorterStemmer*() and **Catboost**.*CatBoostClassifier*() in libraries *Catboost* 1.0.3 and *Nltk* 3.5, respectively, which achieves worse performance in execution time. Based on our empirical findings in RQs1-3, we construct a repository containing 164 defective APIs and 109 API combinations, involving 19 ML libraries. Leveraging the defective API repository, we can capture the PLC issues in real-world ML pipelines and ML libraries.

Our study aims to answer the following two research questions:

- **RQ4:** ***Do real-world ML complex pipelines use the identified defective library APIs?***
- **RQ5:** ***Can the repository of defective APIs help capture real PLC issues?***

Experiment Setup. To answer RQ4, we collected large-scale representative ML pipelines from GitHub and *Hugging Face* [1] platforms released by leading IT companies (including *Microsoft*, *Google*, *Tencent* and *Alibaba*) as subjects. The large-scale industrial pipelines typically handle real-world challenging tasks, which are typically more complex than the specific competition tasks implemented by Kaggle pipelines. We selected *Hugging Face* platform because it is the most popular AI model repository, where users can share pre-trained models, datasets, and demos of ML projects [72, 76]. To achieve this, we formulated search keywords as {***topic***: machine-learning, ***topic***: deep-learning} × {***org***: *Microsoft*, ***org***: *Google*, ***org***: *Tencent*, ***org***: *Aalibaba*}. The search returns 124 and 34 ML pipelines from GitHub and *Hugging Face*, repectively. Table 7 shows the statistics of collected large-scale pipelines. On average, they achieve 1,660 stars on GitHub and 262,248 downloads on *Hugging Face* (popular), have 53,583 KLOC (large in scale), depend on 5 ML libraries and 136 ML library APIs (complex). As we discussed in Section 3.2, we face challenges on deploying large-scale pipelines, due to lacking detailed training steps in documentation, complete datasets, or required dependencies. For the collected ML pipelines, we investigate whether they use defective APIs in the problematic library version combinations identified in our repository.

To answer RQ5, we collected 1,064 pipelines as subjects from the 17 competitions hosted between 1 August 2021 and 1 June 2022 on Kaggle platform. Eventually, 497 out of 1,064 pipelines have been successfully deployed. Among them, 303 and 194 pipelines invoke 80 (80/164=48.8%) defective APIs and 69 (69/106=65.1%) API combinations in our repository, respectively. For each pipeline using defective APIs, we checked if the performance inconsistencies, crashes and *NaN* bugs can be exposed on the corresponding problematic library versions. Since Kaggle platform has no issue trackers, we posted the validated results (i.e., performance changes across various library versions) as comments on the pipelines' websites to warn pipeline developers/users against PLC issues. For the 43 defective APIs invoked by 17 pipelines that achieve worse performances on ML libraries' newest versions, we also reported the PLC issues to their issue trackers to ask library developers for validation.

Table 7: Statistics of collected real-world large-scale pipelines

	Stars/ Downloads			*KLOC*			*# ML Libraries*			*# ML Library APIs*		
	Min	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
GitHub Pipelines	9	20,249	1,660	31	2,543,862	59,513	1	18	4	2	1,510	110
Hugging Face Pipelines	107	6,057,167	262,248	691	447,470	47,653	1	29	6	1	1,499	161

Microsoft: *86 pipelines*, Google: *48 pipelines*, Tencent: *5 pipelines*, Alibaba: *19 pipelines*

Table 8: Experiment results of RQ4

	M_1	M_2	M_3	M_4
GitHub Pipelines	101	55	18	123
Hugging Face Pipelines	31	14	14	73
Sum	142	69	18	184

M_1: *#Pipelines using defective ML library version combinations in our repository;*
M_2: *#Pipelines using the defective APIs (combinations) in our repository;*
M_3: *#Defective library version combinations used by the pipelines;*
M_4: *#Defective APIs (combinations) used by the pipelines*

{Sklearn, xgboost} Versions	Score	Time (ms)	Memory (MB)	{Sklearn, xgboost} Versions	Score	Time (ms)	Memory (MB)
{1.0.1, 1.5.1}	0.622	3755.792	2330.385	{0.22.1, 1.4.2}	0.650	4207.324	2354.084
{1.0.1, 1.2.1}	0.653	3807.698	2649.305	{0.22.1, 1.1.1}	0.652	7974.568	2648.538
{1.0.1, 1.1.1}	0.652	8151.661	2649.224	{0.22.1, 1.0.2}	0.656	7451.785	2647.906
{1.0.1, 1.0.2}	0.656	7783.346	2649.181	{0.22.1, 0.90}	0.654	4743.717	2329.738
{1.0.1, 0.90}	0.654	4634.113	2649.073	{0.22.0, 1.5.1}	0.622	3752.547	2357.178
{0.24.2, 1.5.1}	0.621	3763.312	2330.292	{0.22.0, 1.4.2}	0.650	4168.140	2354.079
{0.24.2, 1.1.1}	0.652	7926.313	2651.979	{0.22.0, 1.0.2}	0.657	7664.165	2647.902
{0.24.2, 1.0.2}	0.655	7645.457	2651.979	{0.22.0, 0.90}	0.654	4683.703	2330.310
{0.22.1, 1.5.1}	0.622	3772.575	2357.182	{0.20.3, 1.0.2}	0.652	406.435	2649.010

Figure 7: A PLC issue GoogleBrain-VPP#1686 [24] reported by us

Results of RQ4. Table 8 reports the experiment results of RQ4. Among the collected 158 large-scale ML pipelines, 142 (89.9%) of them depend on the identified defective ML library version combinations in our repository, and 69 of them invoke the corresponding defective APIs (combinations). In total, the collected large-scale ML pipelines involve 18 defective library version combinations (75%) and 184 defective APIs (combinations) (67.4%) in our repository. For example, API *cv2.imread*() from library *OpenCV* is used by 19 ML pipelines in *DCO* stage, which may induce significant performance inconsistencies in execution time. The results indicate that our empirical findings identified in Kaggle pipelines can scale to more complex industrial pipelines.

Results of RQ5. Table 9 reports the statistics of our experiment results. In total, we exposed PLC issues in 309 out of 497 real-world pipelines. 208 (67.3%) of them were induced by one ML library version, while the remaining 101 (32.7%) of pipelines were caused by the combinations of ML library versions. The PLC issues involve 58 defective APIs and 42 API combinations. Specifically, 251, 50 and 9 pipelines manifested in performance inconsistencies, crashes and *NaN* bugs, respectively. We can observe that 188 (37.8%) pipelines invoking the defective API (combination) did not reveal PLC issues. The conditions of triggering such issues may be affected by many factors, such as hyperparameters and dataset size, which deserve further investigations in the future research.

Encouragingly, twenty pipeline developers shown their great interests in our detailed diagnosis info of PLC issues generated by Piecer. For example, as shown in Figure 7, Piecer's testing results pointed out that the execution time of pipeline GoogleBrain-VPP#1686 [24] changed dramatically across different version combinations of ***Sklearn*** and ***Xgboost***. Developers confirmed the issue and left a comment: "*Thanks for this valuable information. I will surely try and compare different versions.*"

For the 43 defective APIs that achieve worse performance on ML libraries' newest versions, we filed them into 17 issue reports on GitHub. 7 (41.2%) of them had been quickly confirmed by developers and were being fixed based on our testing results. 6 issues (35.3%) were acknowledged by the developers to be worthy of further investigation. Developers considered Xgboost#8033 were not bugs. They had to improve some aspects of performance at the cost of other metrics. The remaining reports are pending probably because of inactive project maintenance.

For example, we reported a PLC issue to library ***Catboost*** whose API ***Catboost***.*CatBoostClassifier()* in the latest version nearly consumes five times memory, compared with that in versions $\leq$ 0.10.3. Developers quickly confirmed this issue and commented that "*I could reproduce your results and saw a jump in RAM consumption from 34 MB to 150 MB.*"

The experiment results demonstrate that our defective API repository shed light on designing automated approaches to detecting PLC issues on real-world pipelines and ML libraries.

5 IMPLICATIONS

For ML pipeline developers. Our empirical study reveals the pervasiveness and seriousness of PLC issues in ML pipelines. Developers can utilize Piecer or our provided defective API repository to select version combinations of ML libraries when optimizing the performance of their pipelines.

For ML library vendors. Our empirical findings show that an ML library may achieve worse performance on its latest version when working with other ML libraries in the pipelines. ML library developers should consider common library combinations and perform integration testing before releasing new versions.

For SE researchers. For SE researchers, future research can focus on designing promising techniques to detect and repair PLC issues.

For tool builders. Tool builders can leverage our defective API repository and Piecer to recommend appropriate version combinations of ML libraries to pipeline developers when maintaining/evolving ML libraries.

We hope that this paper can inspire a symbiotic ecosystem where researchers, tool builders, and library vendors work together to assist developers combat PLC issues.

6 THREATS TO VALIDITY

Internal Validity. Our empirical study involved manual effort, which might introduce subjectivity and bias. To mitigate this threat, we followed an open coding scheme where two authors independently checked and cross-validated all the results. We employed *Cohen's Kappa* [73] to measure the consistencies between the two authors. If the *Kappa* value was less than 0.9, the authors needed to discuss to resolve disagreements. To enhance transparency and accountability, we made our dataset publicly accessible for scrutiny.

External Validity. The external validity concerns about the generality of our results. To mitigate such a threat, we collected 11,363 high-quality ML pipelines from Kaggle platform as subjects to conduct our empirical study. The pipelines depend on 112 popular ML libraries, including 23 ML frameworks and 89 data science libraries. Besides, we also collected large-scale representative ML pipelines from GitHub and *Hugging Face* [1] platforms released by leading IT companies as subjects, to investigate whether they use defective APIs in the problematic library version combinations identified in our repository. Our empirical findings obtained based on such a representative dataset can be generalized to more ML pipelines.

Table 9: Results of RQ5: PLC issues identified in real-world pipelines and ML libraries

PLC Issues Identified in 497 Real Pipelines (*#pipeline ID*)		
Performance Impacts Induced by One ML Library Version		#36502↓↓↓, #36523↑↑↑, #36544↓↓, #36567↓↑, #36575↓↑, #36577↑↓↑, #36586↓↓, #36590↓, #36594↑, #36600↑↓↑, #36616↑↓↓, #36618↓↓, #36625↑↑↓, #36630↓↑↓, #11094↓↓↓, #11119↑↓, #11167↑↑↓, #31772↓↓↓, #31773↓↓, #31776↑↓↑, #31784↑↑, #31785↓↓↑, #31824↓↓↓, #31856↑↑, #31870↓↓↑, #31883↓, #31902↑↓, #37112↓↑↓, #37117↓↑↑, #37124↓↑↑, #37258↓↓, #37259↓, #37261↓↑↑, #37297↓↓↓, #37298↑↓, #37317↓↓↓, #37321↓↓↑, #37335↑, #37403↓, #37410↓↓↑, #37426↓↓↓, #36151↓↑↑, #36153↓↓↑, #36169↓↓↓, #36173↓↑↓, #36199↓↓, #36209↓↓, #36237↓↑↑, #36326↓↓, #36333↑, #36335↓, #36471↑↑↑, #36476↓↓, #36488↓↑, #36501↑↑, #36634↓↑↓, #36635↑↑, #36637↑, #36639↑↑, #36641↓↑, #36642↑↓↑, #36647↓↑↑, #36649↓, #36690↑↑↑, #36734↓↑↑, #36916↓↓, #36938↑↓↑, #37107↑↑, #37111↑↑, #33431↑↓, #33445↓↑, #33468↓↑↑, #33472↑↓, #33487↓↓, #33514↓↑↓, #33600↓↓↑, #33602↓, #33603↓↑↑, #33620↓, #33659↑↓↓, #33668↑↓↑, #33704↑↓, #33705↓↓, #33125↑↓, #33129↓↑, #33147↓, #33171↓↑↓, #33176↓↓↑, #33185↓↑, #33194↓↓, #33253↓↓, #33275↓, #33288↓↑, #33325↓↑↓, #33372↓↓, #33377↑↓, #33409↑↓↑, #34197↓↓, #34198↓↓↓, #34207↓↓, #34215↓↑↓, #34220↑↓↓, #34229↑↓, #34241↓↓, #34243↑↓, #34243↓, #34248↓, #34327↓↓↓, #34328↓, #34331↑, #34333↓↑↑, #31920↓↓↑, #31950↓↑, #31975↓↓↑, #32053↑↑, #32292↑, #32347↓, #32427↑↑↓, #32449↑↑↓, #32532↑, #32564↓, #32569↑↑↑, #32574↓, #32835↓↑, #33123↓↓, #33797↓↓↑, #33801↓↑, #33807↓↓, #33823↑↓↓, #33941↑↓, #33948↑↓↑, #33981↑↑↓, #34150↓↑, #34154↓↑, #34159↑↓↓, #34185↑↑↓, #34186↓↓↓, #34187↓↓↓, #34334↑, #34367↓↑↑, #34388↓, #34442↓, #34463↓↓↑, #34474↑↓↓, #34601↑↑, #34658↓↑↑, #34789↓↓, #35760↑↓, #35776↓, #36124↓↑, #36128↓↓, #36146↓↓, #37445↓↓↓, #37652↓↑↓, #37731↓↑, #38020↑↑↑, #38107↑↓↑, #38150↓, #31915↓↑, #33423↓, #33789↓↑↓, #34767, #34923, #36172, #36252, #36187, #36214, #33335, #11180, #31900, #31947, #31997, #32009, #32253, #32518, #32560, #32566, #32607, #32923, #32928, #33138, #33193, #33335, #33346, #33721, #33724, #33346, #33742, #33754, #33770, #33776, #33790, #33796, #33811, #33812, #33877, #33878, #33913, #33886, #34068, #34116, #34332, #34336, #34352, #37763
Performance Impacts Induced by Combinations of ML Library Versions	***Type A***	#11087↓↓↓, #11166↓↓, #11193↓↓↑, #11195↑↓↑, #11224↓↓, #31751↑↑, #31753↓↑↑, #31770↑↑, #32490↑↑, #33247↑↓↑, #33274↑↑↑, #33302↓↑, #33303↓↓↑, #33329↑↑, #33341↓↓, #33342↓↓↑, #33394↓↑↑, #33395↓↓↑, #33398↑↓↓, #33437↑↓↑, #33439↓↓↑, #33539↓↓↑, #33744↑↑↓, #33805↓↓↓, #33837↑↓↑, #33839↓↓, #33853↑↓↓, #33959↑↓↓, #33963↓↓↓, #33994↓↑, #34177↓↑↑, #34226↓↑↑, #34712↓↓↓, #35699↓↑↑, #36157↑↓↑, #36174↑↓↑, #36188↓↑↓, #36198↓↑↓, #36211↑↓↓, #36265↑↓, #36306↓↑↑, #36316↓↓↓, #36563↑↓↑, #36572↑↑, #36582↓, #36728↑↑↓, #36729↓↓↓, #37061↓↓, #37359↓↑↑, #37448↑↓↓, #37758↓↑, #37832↑↑, #37913↓↑, #36267↓↑↑, #33939↓↓↑, #36267↓↑↑, #36487, #36557, #36624, #36650, #36662, #34249, #34300, #36552
	Type B	#32522↓, #33734↓↑, #33735↓, #33775↓↓↑, #33794↓↓↓, #33850↓↓↑, #33857↓↓, #33896↓, #33910↓↑↓, #33935↓, #34344↑↓↑, #34374↓, #34637↑↓, #34646↑↓↓, #34650↑↓, #34693↑↓, #34699↓, #34702↑↓, #36239↑, #36264↑, #36284↓↓, #36574↓↓, #36587↑↓, #36664↑ #36848↓ #37343↑↑ #37360↑↓ #37543↓↓↓ #37552↓↑ #37768↑↓ #37776↑↑ #38067↓↓ #38126↓↓ #38162↑ #38230↑↓ #36695, #36879, #36907, #34299, #34322, #36621

18 PLC Issues Repored to ML libraries (*Report ID, #Problematic APIs*)
[*Catboost*#2142, 2 APIs]♠,[*ImbalancedLearn*#923, 2 APIs]♠, [*Keras*#17217, 8 APIs]♠, [*OpenCV*#22472, 2 APIs]♠, [*Optuna*#3976, 2 APIs]♠,[*Sklearn*#23427, 3 APIs]♠,[*Xgboost*#8033, 2 APIs]♠, [*CategoryEncoders*#362, 2 APIs]♣,[*CategoryEncoders*#364, 2 APIs]♣,[*LightGBM*#5336, 3 APIs]♣,[*LightGBM*#5475, 3 APIs]♣,[*Sklearn*#24138, 1 APIs]♣, [*Transformers*#18950, 3 APIs]♣, [*Tensorflow*#56600, 4 APIs]♣,[*Tensorflow*#56958, 7 APIs]♣, [*Shap*#2644, 1 APIs]♠, [*Shap*#2644, 1 APIs]♠

PLC issues were manifested in performance changes (rise↑ / fall↓) of three aspects: Competition scores, Execution time and Memory usage, and Crashes/ NaN bugs;
♠ *Confirmed issues under fixing;* ♣ *Acknowledged by developers (Developers considered the issues were worthy of further investigation);* ♠ *Not bugs;* ♣ *Not bugs in the latest versions;* ♠ *Pending Issues.*

7 RELATED WORK

Machine Learning Libraries. There have been a lot of work focusing on the performance enhancement and quality assurance of ML frameworks. Zhang et al. [85] conducted an empirical study on the common bugs of ML applications using *TensorFlow*. Nargiz et al. [47] and Islam et al. [49] further conducted studies on *Keras*, *TensorFlow*, *PyTorch*, *Caffe* and *Theano* with a taxonomy of bugs. Han et al. [43] performed an exploratory study on the dependency networks of deep learning libraries. They studied the application domains, dependency degrees, version update behaviors of ML applications that depend on *Tensorflow*, *PyTorch*, and *Theano*. Dilhata et al. [38] conducted a quantitative and qualitative empirical study to investigate the ML library usage and evolution. They only focused on the ML frameworks and explored how developers used these libraries (e.g., common library combinations) and how the usage evolved over the projects' lifetime. Wang et al. [74], Guo et al. [42] and Phamet et al. [63] proposed novel mutation testing/differential techniques to detect behavioral inconsistencies across multiple ML frameworks. Georgiou et al. [40] presented an in-depth empirical analysis to investigate and compare the energy consumption and run-time performance of DL frameworks (*PyTorch* and *TensorFlow*).

Our study distinguishes from the existing work in three aspects: (1) we focus on various types of common ML libraries including both ML frameworks and data science libraries; (2) we conduct the first empirical study to explore the impacts of different library version combinations on ML pipelines' performances; (3) with the aid of our identified defective API repository, we detect many *PLC* issues in real-world ML applications.

Machine Learning Applications. Several SE researchers presented comprehensive empirical studies on real bugs and technical challenges of ML applications. Islam, et al. [50], Guo et al. [41], Han et al. [44], Zhang et al. [84] and Chen et al. [34] collected a set of high-quality DL questions&answers on Stack Overflow and summarized the main challenges in developing and deploying ML applications, respectively. Approaches [54, 83] presented a root cause taxonomy of deep learning specific failures to facilitate future software development. Shen et al. [67] studied the bugs rising in the popular deep learning compilers, and provided a series of valuable guidelines for deep learning compiler bug detection and debugging. Phamet et al. [64] studied the variance of DL systems and the awareness of this variance among researchers and practitioners. Existing and new testing techniques [33, 37, 39, 45, 45, 48, 51, 52, 59, 60, 62, 65, 69–71, 75, 77, 78, 81, 82, 86] were also proposed for and adapted to machine learning applications to expose real bug. The works by Ma et al. [46, 61] and Shen et al. [68] proposed mutation operators specific to deep learning applications. Humbatova et al. [48] followed a systematic process to extract mutation operators from existing bug taxonomies to simulate the effects of real DL bugs. In this paper, we capture the performance inconsistencies of ML pipelines using differential testing across various library version combinations.

8 CONCLUSION AND FUTURE WORK

In this paper, we empirically studied 11,363 ML pipelines from diverse competitions on Kaggle to explore the impacts of different ML library version combinations on their performances. Our study reveals the pervasiveness and severity of *PLC* issues in ML pipelines. Our findings can motivate the establishment of a symbiotic ecosystem where researchers, tool builders, and library vendors work together to assist developers in combating *PLC* issues.

9 DATA AVAILABILITY

We provide a reproduction package at ***http://piecer-plc.github.io*** to facilitate future research. The package includes (1) a dataset containing 11,363 ML pipelines and 7,983 library APIs, (2) an available tool Piecer, (2) a defective API repository for detecting PLC issues, and (4) a list of PLC issues captured by us from real-world pipelines and ML libraries, along with our issue reproducing results.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers for their constructive comments. The work is supported by the National Natural Science Foundation of China (Grant Nos. 62141210, 61932021, 61902056), the Hong Kong RGC/GRF grant 16205722, MSRA grant, ITF grant (MHP/055/19, PiH/255/21), Research Grants Council (RGC) Research Impact Fund under Grant R5034-18, the Fundamental Research Funds for the Central Universities (Grant No. N2217005), Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01), and 111 Project (B16009).

REFERENCES

[1] 2021. Hugging Face. https://huggingface.co/models. Accessed: 2021-08-01.
[2] 2021. Kaggle. https://www.kaggle.com/. Accessed: 2021-08-01.
[3] 2021. Kaggle. https://jedi.readthedocs.io//. Accessed: 2021-08-01.
[4] 2021. Matplotlib. https://matplotlib.org/. Accessed: 2021-08-01.
[5] 2021. Numpy. https://numpy.org/. Accessed: 2021-08-01.
[6] 2021. Pandas. https://pandas.pydata.org/. Accessed: 2021-08-01.
[7] 2021. Pytorch. https://pytorch.org/. Accessed: 2021-08-01.
[8] 2021. A ranked list of awesome ML librarie. https://github.com/ml-tooling/best-of-ml-python. Accessed: 2021-08-01.
[9] 2021. Scikit-Learn. https://scikit-learn.org/stable/. Accessed: 2021-08-01.
[10] 2022. CFEC#1211. https://www.kaggle.com/code/nandhuelan/catembed/notebook. Accessed: 2022-07-01.
[11] 2022. CFEC#1254. https://www.kaggle.com/dskagglemt/categorical-feature-encoding-challenge. Accessed: 2022-07-01.
[12] 2022. CFECII#385. https://www.kaggle.com/nandhuelan/let-s-tickle-the-cat-meow. Accessed: 2022-07-01.
[13] 2022. Commit log 7a0adbb. https://github.com/huggingface/transformers/pull/4659/commits/7a0adbb56ae719a784f781bb2d80edf856e71916. Accessed: 2022-07-01.
[14] 2022. Competition 1. https://www.kaggle.com/c/lish-moa. Accessed: 2022-07-01.
[15] 2022. Competition 10. https://www.kaggle.com/c/champs-scalar-coupling. Accessed: 2022-07-01.
[16] 2022. Competition 2. https://www.kaggle.com/c/aerial-cactus-identification. Accessed: 2022-07-01.
[17] 2022. Competition 3. https://www.kaggle.com/c/tabular-playground-series-may-2021. Accessed: 2022-07-01.
[18] 2022. Competition 4. https://www.kaggle.com/competitions/nlp-getting-started. Accessed: 2022-07-01.
[19] 2022. Competition 5. https://www.kaggle.com/c/Kannada-MNIST. Accessed: 2022-07-01.
[20] 2022. Competition 6. https://www.kaggle.com/competitions/cat-in-the-dat. Accessed: 2022-07-01.
[21] 2022. Competition 7. https://www.kaggle.com/competitions/tabular-playground-series-apr-2021. Accessed: 2022-07-01.
[22] 2022. Competition 8. https://www.kaggle.com/c/plant-pathology-2020-fgvc7. Accessed: 2022-07-01.
[23] 2022. Competition 9. https://www.kaggle.com/c/optiver-realized-volatility-prediction. Accessed: 2022-07-01.
[24] 2022. GoogleBrain-VPP#1686. https://www.kaggle.com/code/ranjeetshrivastav/ventilator-pressure-prediction-xgboost/comments. Accessed: 2022-07-01.
[25] 2022. Issue Pandas#25893. https://www.kaggle.com/dskagglemt/categorical-feature-encoding-challenge. Accessed: 2022-07-01.
[26] 2022. MNIST. https://www.kaggle.com/c/Kannada-MNIST. Accessed: 2022-07-01.
[27] 2022. MNIST#85. https://www.kaggle.com/joesmithkaggle/the-simple-kernel-for-kannada-mnist. Accessed: 2022-07-01.
[28] 2022. NLPDT#785. https://www.kaggle.com/code/onuraydere/decisiontreeandbernoullinb. Accessed: 2022-07-01.
[29] 2022. NLPDT#plyger. https://www.kaggle.com/code/plyger/notebook268f8a3109/notebook. Accessed: 2022-07-01.
[30] 2022. Release note of Sklearn 0.22.0. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. Accessed: 2022-07-01.
[31] 2022. Sklearn#20642. https://github.com/scikit-learn/scikit-learn/issues/20642. Accessed: 2022-07-01.
[32] 2022. TPS-Jul#839. https://www.kaggle.com/code/tps-july-2021-simple-fast-code. Accessed: 2022-07-01.
[33] Houssem Ben Braiek and Foutse Khomh. 2019. Deepevolution: A search-based testing approach for deep neural networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 454–458. https://doi.org/10.1109/ICSME.2019.00078
[34] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762. https://doi.org/10.1145/3368089.3409759
[35] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge. https://doi.org/10.4324/9780203771587
[36] John W. Creswell. 2013. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches (3rd Edition)*. https://doi.org/10.1177/1524839915580941
[37] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56. https://doi.org/10.1145/3540250.3549085
[38] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–42. https://doi.org/10.1145/3453478
[39] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 477–487. https://doi.org/10.1145/3338906.3338954
[40] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. 2022. Green ai: Do deep learning frameworks have different costs?. In *Proceedings of the 44th International Conference on Software Engineering*. 1082–1094. https://doi.org/10.1145/3510003.3510221
[41] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 810–822. https://doi.org/10.1109/ASE.2019.00080
[42] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498. https://doi.org/10.1145/3324884.3416571
[43] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. 2020. An empirical study of the dependency networks of deep learning libraries. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 868–878. https://doi.org/10.1109/ICSME46990.2020.00116
[44] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering* 25 (2020), 2694–2747. https://doi.org/10.1007/s10664-020-09819-6
[45] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Lei Ma, Mike Papadakis, and Yves Le Traon. 2022. Efficient Testing of Deep Neural Networks via Decision Boundary Analysis. *arXiv preprint arXiv:2207.10942* (2022). https://doi.org/10.48550/arXiv.2207.10942
[46] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1158–1161. https://doi.org/10.1109/ASE.2019.00126
[47] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121. https://doi.org/10.1145/3377811.3380395
[48] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 67–78. https://doi.org/10.1145/3460319.3464825
[49] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520. https://doi.org/10.1145/3338906.3338955
[50] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1135–1146. https://doi.org/10.1145/3377811.3380378
[51] Hong Jin Kang, Pattarakrit Rattanukul, Stefanus Agus Haryono, Truong Giang Nguyen, Chaiyong Ragkhitwetsagul, Corina Pasareanu, and David Lo. 2022. SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries. *arXiv preprint arXiv:2212.04038* (2022). https://doi.org/10.48550/arXiv.2212.04038
[52] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108
[53] Robert V Krejcie and Daryle W Morgan. 1970. Determining sample size for research activities. *Educational and psychological measurement* 30, 3 (1970), 607–610. https://doi.org/10.1177/001316447003000308
[54] Yunkai Liang, Yun Lin, Xuezhi Song, Jun Sun, Zhiyong Feng, and Jin Song Dong. 2022. gDefects4DL: a dataset of general real-world deep learning program defects. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 90–94. https://doi.org/10.1145/3510454.3516826
[55] Ben Liblit, Linghui Luo, Alejandro Molina Ramirez, Rajdeep Mukherjee, Zachary Patterson, Goran Piskachev, Martin Schäf, Omer Tripp, and Willem Visser. 2023. Shifting left for early detection of machine-learning bugs. (2023). https://doi.org/10.1007/978-3-031-27481-7_33
[56] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the reproducibility and replicability of deep learning in software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–46. https://doi.org/10.1145/3477535
[57] Geoffrey R Loftus and Michael EJ Masson. 1994. Using confidence intervals in within-subject designs. *Psychonomic bulletin & review* 1, 4, 476–490. https://doi.org/10.3758/BF03210951
[58] Qinghua Lu, Liming Zhu, Xiwei Xu, Jon Whittle, and Zhenchang Xing. 2022. Towards a roadmap on software engineering for responsible AI. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*.

101–112. https://doi.org/10.1145/3522664.3528607
[59] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–618. https://doi.org/10.1109/SANER.2019.8668044
[60] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131. https://doi.org/10.1145/3238147.3238202
[61] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111. https://doi.org/10.1109/ISSRE.2018.00021
[62] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18. https://doi.org/10.1145/3132747.3132785
[63] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107
[64] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 771–783. https://doi.org/10.1145/3324884.3416545
[65] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering* 25 (2020), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0
[66] Shlomo S Sawilowsky. 2009. New effect size rules of thumb. *Journal of modern applied statistical methods* 8, 2 (2009), 26. https://doi.org/10.22237/jmasm/1257035100
[67] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980. https://doi.org/10.1145/3468264.3468591
[68] Weijun Shen, Jun Wan, and Zhenyu Chen. 2018. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 108–115. https://doi.org/10.1109/QRS-C.2018.00032
[69] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 109–119. https://doi.org/10.1145/3238147.3238172
[70] Florian Tambon, Foutse Khomh, and Giuliano Antoniol. 2023. A probabilistic framework for mutation testing in deep neural networks. *Information and Software Technology* 155 (2023), 107129. https://doi.org/10.1016/j.infsof.2022.107129
[71] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314. https://doi.org/10.1145/3180155.3180220
[72] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers*. " O'Reilly Media, Inc.".
[73] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.
[74] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799. https://doi.org/10.1145/3368089.3409761
[75] Moshi Wei, Yuchao Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2022. Cocofuzzing: Testing neural code models with coverage-guided fuzzing. *IEEE Transactions on Reliability* (2022). https://doi.org/10.1109/TR.2022.3208239
[76] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6
[77] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157. https://doi.org/10.1145/3293882.3330579
[78] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. Diffchaser: Detecting disagreements for deep neural networks. International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2019/800
[79] Minke Xiu, Zhen Ming Jack Jiang, and Bram Adams. 2020. An exploratory study of machine learning model stores. *IEEE Software* 38, 1 (2020), 114–122. https://doi.org/10.1109/MS.2020.2975159
[80] Chunyong Yin, Bo Li, and Zhichao Yin. 2020. A distributed sensing data anomaly detection scheme. *Computers & Security* 97 (2020), 101960. https://doi.org/10.1016/j.cose.2020.101960
[81] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2019.2962027
[82] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 132–142. https://doi.org/10.1145/3238147.3238187
[83] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1159–1170. https://doi.org/10.1145/3377811.3380362
[84] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020
[85] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140. https://doi.org/10.1145/3213846.3213866
[86] Jianyi Zhou, Feng Li, Jinhao Dong, Hongyu Zhang, and Dan Hao. 2020. Cost-effective testing of a deep learning model through input reduction. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 289–300. https://doi.org/10.1109/ISSRE5003.2020.00035

Received 2023-03-02; accepted 2023-07-27