An Exploratory Study of Dataset and Model Management in Open Source Machine Learning Applications

Tajkia Rahman Toma
tajkiatoma@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

Cor-Paul Bezemer
bezemer@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

Datasets and models are two key artifacts in machine learning (ML) applications. Although there exist tools to support dataset and model developers in managing ML artifacts, little is known about how these datasets and models are integrated into ML applications. In this paper, we study how datasets and models in ML applications are managed. In particular, we focus on how these artifacts are stored and versioned alongside the applications. After analyzing 93 repositories, we identified the most common storage location to store datasets and models is the file system, which causes availability issues. Notably, large data and model files, exceeding approximately 60 MB, are stored exclusively in remote storage and downloaded as needed. Most of the datasets and models lack proper integration with the version control system, posing potential traceability and reproducibility issues. Additionally, although datasets and models are likely to evolve during the application development, they are rarely updated in application repositories.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → *Software development process management*; • **Information systems** → **Version management**.

KEYWORDS

Machine Learning (ML) artifact management, ML application development, SE4AI

ACM Reference Format:
Tajkia Rahman Toma and Cor-Paul Bezemer. 2024. An Exploratory Study of Dataset and Model Management in Open Source Machine Learning Applications. In *Proceedings of 3rd International Conference on AI Engineering — Software Engineering for AI (CAIN 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, machine learning (ML) has become an integral part of solving various real-world problems [21]. A software application that uses an ML model to solve a problem is referred to as an ML application. Other than traditional software artifacts, an ML application relies on two additional types of artifacts: dataset and model. Like the traditional artifacts, the datasets and models evolve over time during the development of the application [2, 3]. The dataset can evolve for many reasons like updating the data cleaning algorithm and the addition of new data etc. As the data evolves, the model must be updated too. In parallel, the application also evolves with new features and updates of old features. Therefore, as with traditional software artifacts, datasets and models require meticulous management. The management of software artifacts consists of the systematic process of creating/collecting, storing, organizing, versioning, distributing and deploying assets connected to a software project. However, the distinctive aspects of ML applications, such as data-centricity and model customization and reuse, make dataset and model management uniquely challenging [2]. Also, the collaboration among the application developers, model developers and data engineers in the project is a challenge [8, 13, 16]. Systematic management of the datasets and models can facilitate a better collaboration between these three groups.

There are many systems and platforms to support dataset and model developers in managing the ML artifacts which are often referred to as ML Artifact Management Systems (ML AMSs) [24]. Many recent studies have compared these ML AMSs from different perspectives [9–11, 15, 20, 23–26, 28]. Also, researchers are focusing on how these systems and platforms are being used in open-source ML applications [4, 17]. To the best of our knowledge, we are the first to analyze how ML application developers manage the datasets and models in their application repositories, regardless of whether they use an ML AMS.

Although dataset and model developers can manage ML artifacts using ML AMSs, application developers widely use traditional version control systems like Git to manage their application artifacts' history. Therefore, they need to integrate the datasets and models into their systems to keep working in their current working environment. However, little is known about how the datasets and models are integrated into an ML application and how they are managed. Therefore, in this paper, we study two important aspects of dataset and model management in ML applications: (1) storage and (2) versioning. A proper storage location ensures the availability of the artifacts while versioning facilitates traceability and reproducibility, and helps maintain the overall artifact quality.

Our goal is to explore the current practices of storing and versioning datasets and models within ML applications. We manually analyze the code of 93 randomly selected GitHub ML application repositories and the 321 data files and 354 model files that the applications use. We study the following research questions where RQ1 and RQ2 address the storage aspect and RQ3 covers the versioning aspect of management:

CAIN 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

- **RQ1: How are the datasets stored in the studied ML applications?** The datasets' storage location is important in the process of maintaining and versioning them. We found 6 types of storage locations for data files, with the file system being used the most commonly, by 39% of the repositories. As, in many of the repositories, the data files are stored in the file system, it is the responsibility of the application developers to manage their data files' availability.
- **RQ2: How are the models stored in the studied ML applications?** Like the datasets, the models' storage locations play an important role in maintaining their availability. We found 4 types of storage locations for model files and most (51%) repositories loaded models from the file system.
- **RQ3: How are the datasets and models versioned in the application repositories?** As applications evolve, models and datasets change over time. Our study shows that the application repositories often lack version information for datasets and models. Most of the data and model files that are versioned in the version control system (VCS) are rarely modified during application development. If there were any changes to the datasets or models, these are not reflected in the application repositories.

Our study gives an overview of current practices for managing datasets and models in open-source ML applications. We observe that there is a lack of dataset and model management in terms of versioning in most of the studied applications, often because a storage location was chosen that makes versioning difficult. The results of our study suggest that it could be beneficial for open-source ML application developers to have an easy integration of ML AMS in their development ecosystem. Our replication package including all data, scripts and documents available in a public repository[1]. The data was collected between February and April 2023.

The rest of the paper is structured as follows: Section 2 discusses how we collected and processed our data and the steps of our manual analysis. Sections 3, 4 and 5 discuss the results of our analysis for RQ1, RQ2 and RQ3 respectively. Section 6 discusses the overall implications of our study. Section 7 discusses the threats to the validity of this work and Section 8 summarizes the related work. Section 9 concludes our work.

2 STUDY SETUP

Our goal is to explore the current practices of dataset and model management in ML applications from the storage location and version management points of view. Figure 1 outlines the steps of our exploratory study. We collected and processed the data in four steps:

- **Step 1. Data collection:** Collect a list of ML application repositories using data from GitHub and Libraries.io
- **Step 2. Data filtering:** Filter the collected list to ensure its quality
- **Step 3. Data processing:** Prepare the data for manual analysis
- **Step 4. Manual analysis:** Perform a detailed manual analysis of the code segments responsible for loading datasets, and loading and training models

[1]https://github.com/asgaardlab/dataset-and-model-management

The following sections provide an in-depth explanation of each step.

2.1 Data collection

We first made an initial list of ML applications. We executed the following steps to get the list.

Select top three ML libraries. As our study involves the manual analysis of source code, to facilitate the analysis process, we focused on ML applications that use at least one of the following popular ML libraries: TensorFlow [1], PyTorch [18], and Scikit-learn [19]. Gonzalez et al. [7] identified these libraries as some of the most popular in the field of ML development. During the library selection, we re-validated their popularity by checking the number of stars on GitHub.

List ML repositories. We consider a repository as ML repository if it depends on an ML library. We listed all the ML repositories by collecting the dependent repositories of the aforementioned ML libraries from their dependency graph in GitHub. We kept the repositories with 10 or more stars on GitHub to ensure their quality [6]. At the end of this step, we found 38,368 dependents of the 3 libraries.

Remove libraries from ML repositories. Since our focus is on observing how datasets and models are versioned in ML applications, we excluded libraries from the list of ML repositories as these may not include a dataset or model themselves. We got the list of dependent libraries of the three selected libraries from Libraries.io [12] API[2]. After excluding the libraries, we are left with 34,742 ML applications.

2.2 Data filtering

We want to focus on repositories that are well-maintained and not only intended for learning purposes. Additionally, we exclude the repositories that are using old versions of the libraries to narrow down the scope of our analysis and make it more manageable. We retrieved the details for every repository in the list from GitHub using the GitHub REST API[3]. We then filtered the list of application repositories as follows:

- **By number of commits**: To make sure we focus on well-maintained repositories, we included only repositories with at least 100 commits (resulting in 9,937 repositories).
- **By last commit date:** We further refined the repository list by selecting active repositories, that had their most recent commit made within the last six months (resulting in 6,429 repositories).
- **By repository purpose:** To filter out repositories used for learning, like tutorials or courses, we examined each repository's name, description, and associated topics. We began with Gonzalez et al.'s [7] terms and progressively expanded the list until it became a broad set of terms and topics to represent learning resources. If a repository's name or description included any of these terms, it was considered a learning repository. Likewise, if the repository's associated

[2]https://libraries.io/api
[3]https://docs.github.com/en/rest

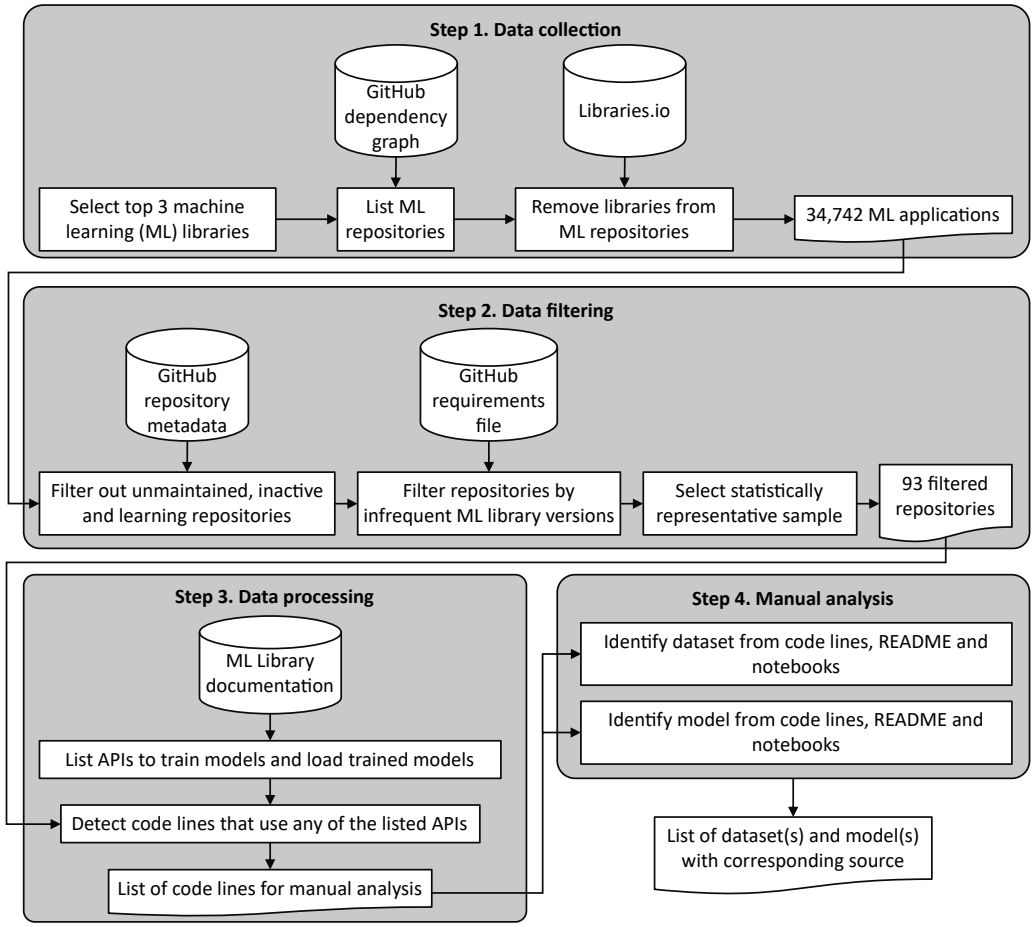


Figure 1: Overview of the study setup

topics matched any of our listed topics, it was marked as a learning repository (resulting in 4,023 repositories).

- **By library version:** We focused on the most popular versions of the covered ML libraries to keep the manual analysis manageable. We examined the dependencies of the repositories to find out the popular library versions. For Python repositories, one common method of managing dependencies is by listing them, along with their versions, in a *requirements* file, usually named `requirements.txt`[4]. We retrieved these requirements files using the GitHub Search API, querying for files with the name `requirements.txt`. Then, we parsed the requirements files and resolved the specifications following the rules outlined in PEP 440[5] to determine the exact library versions used in the repositories. The resolved versions distribution among the repositories in Table 1 shows that while newer versions of the studied ML libraries are popular, older versions are still utilized in many repositories. However, very old versions, such as TensorFlow's 0.∗ and PyTorch's 0.∗, are rarely used. Thus, we selected TensorFlow 2.∗ and 1.∗, PyTorch 2.∗ and 1.∗, and Scikit-learn 1.∗ and 0.∗ for their APIs in our analysis. We also filtered out the application repositories that did not use these versions (resulting in 2,862 repositories).

To identify the dataset and model files from the repositories, we conducted a manual analysis. To complete our manual analysis in a manageable time, we selected a subset of repositories with a 95% confidence level and a 10% error margin that statistically represents all the repositories (resulting in 93 repositories).

2.3 Data processing for manual analysis

To facilitate our analysis, we automated the search for the relevant code segments in the application repositories. In this section, we explain how we automated the identification of code responsible for model training with datasets and loading trained models in these repositories.

List the APIs of the libraries. In an ML application, a developer can either train a model using a dataset or load a trained model either from a saved model or the saved weights of the model. We

[4]https://pip.pypa.io/en/stable/reference/requirements-file-format/
[5]https://peps.python.org/pep-0440/

Table 1: Package version distribution among the application repositories. The versions in bold are included in our study

Libraries	Library Versions	Included in # of Repositories
TensorFlow	**2.***	1,049
	1.*	342
	0.*	18
PyTorch	**2.***	977
	1.*	1,270
	0.*	50
	Unresolved	2
Scikit-learn	**1.***	1,774
	0.*	1,486
All Repositories		4,023

searched the documentation[6][7][8][9][10][11] of every studied ML library version to identify the APIs that should be used to train or load a model. Table 2 shows the identified APIs.

Detect candidate code lines for manual analysis. We then generated the AST of the `.py` files in the repositories to help us identify where the methods were called in the application and mark the identified line as a candidate code line. The code lines are considered candidates because the exact module of the functions cannot always be determined due to Python's dynamic nature. We verify that the candidate code lines indeed originate from one of the studied ML libraries during our manual analysis.

2.4 Manual Analysis

We manually analyzed the candidate code lines to extract the datasets and models as follows:

- **Identify dataset(s)** From the parameters of the function calls and instance calls of the training APIs, we identified the dataset(s) used for training the model and traced the path(s) of the dataset(s) file(s) from the training code segment. We call these files *data files*. A dataset can consist of multiple data files. For datasets that contain image files, we considered the directory that contains the image files as a data file as it contains all the data points.
- **Identify model(s)** We identified the trained models mentioned as function parameters in the model loading APIs and followed the path(s) in the model loading code segment. If we found the file within the repository, we noted the path. Otherwise, if the model is downloaded from a remote system and then used, we document the download URL(s) as the path(s) of the model.

[6]https://www.tensorflow.org/api_docs/python/tf
[7]https://www.tensorflow.org/versions
[8]https://pytorch.org/docs/stable/
[9]https://pytorch.org/docs/versions.html
[10]https://scikit-learn.org/stable/
[11]https://scikit-learn.org/dev/versions.html

We also analyzed the Jupyter notebooks in 40 repositories where model training or loading may occur. To identify these code segments, we manually reviewed the notebook code. We then extracted and documented the path of the dataset and model files from these code segments. In addition, we examined the repository's `README` file to gather information about the paths of the dataset and model files. It is worth noting that a repository can train multiple models using the same data file, leading to several code segments in the repository for the data file. Similarly, a single trained model can be loaded multiple times in a repository for different purposes, resulting in multiple code segments for the same model file. Thus, at the end of the manual analysis, we found 321 data files in 572 model training code segments and 354 model files in 543 model loading code segments.

3 RQ1: HOW ARE THE DATASETS STORED IN THE STUDIED ML APPLICATIONS?

Motivation: To train a model for an ML application, developers need a dataset. The storage of the dataset plays a key role in ensuring its availability for model training. In this research question, we investigate dataset storage practices in ML applications and identify how dataset size is correlated with storage decisions.

Approach: We have 321 data files from 93 repositories, which means, some repositories have more than one data file. Therefore, we first analyzed the distribution of the files in the repositories to understand how the data files are scattered across the repositories. Then, we determined the data files' storage locations from the identified data file paths as follows:

- **Database:** The dataset is loaded from a database
- **File system:** The dataset is loaded from the developers' local file system and is not included in the application repository
- **Library:** The dataset is loaded using a library API
- **Remote storage:** The dataset is loaded from remote storage through URLs
- **Repository:** The dataset is loaded from the developers' local file system and is included in the application repository
- **Runtime memory:** The dataset is loaded from program-generated fixed or random values
- **Unknown source:** The dataset path is unknown

To understand how frequently the developers use each storage to store the data files, we counted the number of repositories employing each storage location. As one repository can use multiple storage locations, the total is not equal to the number of studied repositories. We also discuss the dataset availability in different storage types. Additionally, we gathered the file sizes of the data files to investigate whether dataset size correlates with the choice of any specific storage location.

Findings: **A median repository has one data file.** Figure 2 shows that the number of data files in the repositories varies from 0 to 39. 46 of the 93 repositories contain more than 1 data file, and in 15 of these repositories, all the data files are stored in the same storage type.

The file system is the most popular choice among application developers for storing data files.. From Figure 3, we see that 36 (39%) of the 93 repositories use the file system to store data files, making it the most commonly used storage location. The next

Table 2: Methods to train and load models using the selected library versions

Libraries	Purposes	Methods
TensorFlow	Train Load	`basic_train_loop()`, `fit()`, `fit_generator()`, `train_on_batch()` `load()`, `load_model()`, `load_v2()`, `load_weights()`
PyTorch	Train Load	Create instance of `nn.Module` or `nn.Sequence` or their subtypes and call the instance `load()`, `load_state_dict()`
Scikit-learn	Train Load	`fit()`, `fit_predict()`, `fit_transform()`, `partial_fit()` `load()`, `loads()`

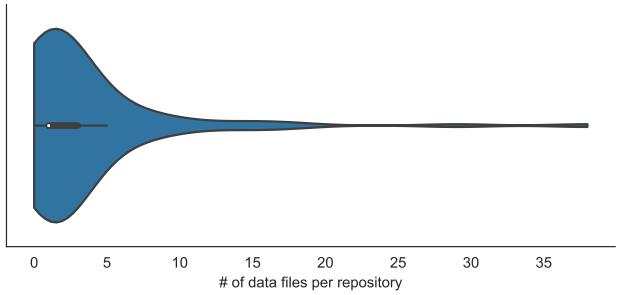


Figure 2: Distribution of data files per repository

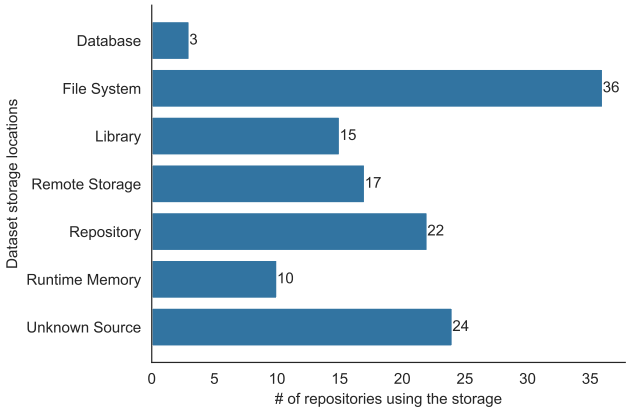


Figure 3: Comparison of usage of the storage locations for data files per repository. Note that a repository may use multiple storage locations for different data files.

most commonly used storage location by the repositories is the repository itself (24%). Repositories also employ remote storage (18%), libraries (16%), runtime memory (11%), and databases (3%) for storing data files. We could not trace the data files' path for 24 repositories. The path cannot be traced for several reasons. For example, the dataset passed as a method parameter where the method has not been called from anywhere in the code, or the path is set at runtime through a command line argument or configuration file, or the code was too tangled to identify the data file path.

File system, remote storage and repository data files are stored in file format, which comprise 242 out of 321 total data files. **We found a wide range of file extensions used to store these data files.** Fifty-three files are stored in a compressed format using zip, gz, or tgz extensions. Other than that, `csv` (39), `fasta` (37) and `jpg/jpeg` (17) are the top 3 most used file extensions. Some file extensions are rarely used, for example, `all-data`, `c2v`, `file`, `mna_jl`, `pgm`, `pickle`, `spickle`, `tfrecord`, `utf8`, `wav` and `xlsx` are found only once. The remaining 79 of the 321 data files are not in a file format where 11% of the total are loaded through the third-party libraries' API, 5% are generated programmatically in runtime memory and 1% are loaded from databases.

The data files available through URLs are dynamic by nature, that is, they can be updated without any notice by the data engineer. Additionally, we **encountered several issues with URL-based data access**:

- **Continuity:** Six URLs point to personal storage locations like Dropbox and Google Drive. Additionally, 28 points to university domains. These storage locations may not guarantee continuous availability.
- **Error:** We could not download data from 7 URLs due to errors. For 6 URLs, we cannot find the files (HTTP 404) and for 1 URL, we do not have permission to access the resource (HTTP 403).
- **Unspecified data file:** For 15 data files, we cannot determine the data file from the URL. The reasons one or more of the following:
  - Ten URLs lead to web pages, such as https://motchallenge.net/, rather than directly to data files or folders. There is no instruction on data selection or retrieval on the web page or the corresponding repository, making it challenging to identify the needed resources.
  - Ten URLs have multiple versions available to download which leads to confusion of which dataset to download. Additionally, two URLs provide real-time data such as https://systems.jhu.edu/research/public-health/ncov. This real-time data keeps changing over time at regular intervals. Thus the dataset changes with regular intervals without any versioning of the data.

Not all the data files are available for model training. Most notably, data files that are stored in the file system are not available by nature as they are in the application developers' local file system. Also, the 7 data files fetched through URLs which had errors are unavailable as well. Moreover, among the 3 databases used for model

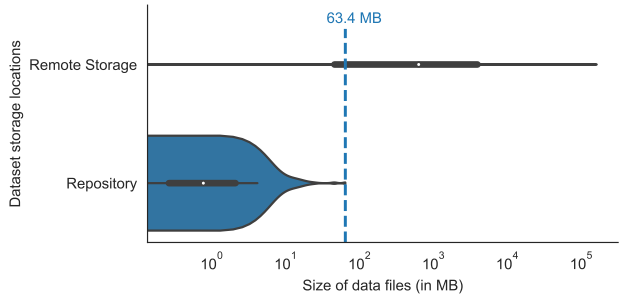


Figure 4: Comparison of the available data files' sizes in different storage locations

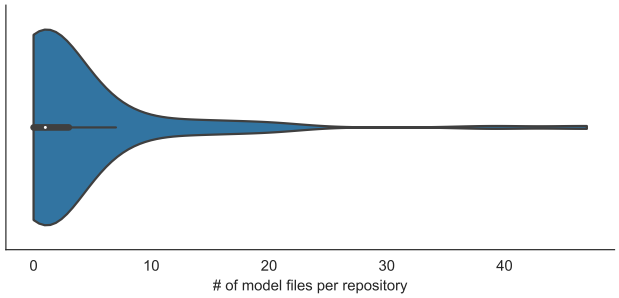


Figure 5: Distribution of model files per repository

training, one is an SQLite file which is stored in the file system of which the file is unavailable. The other two databases are MySQL and PostgreSQL. None of the datasets are available as the repositories do not have any instructions on how to get the databases. In contrast, datasets loaded from libraries, repositories and runtime memory are available for model training. It is worth noting that datasets may be unavailable because of being proprietary, which one repository explicitly mentioned.

From the available datasets, data loaded from libraries and runtime memories are loaded directly to the memory without storing them. Thus we have the size of the datasets loaded from remote storage and repositories. **Developers use remote storage for larger files, while smaller ones are stored inside the repository.** Figure 4 shows that the median file size in the repositories is 0.76 MB, while in remote storage, it is 623.5 MB. Additionally, files larger than 63.4 MB are stored in remote storage only. 44 data files from remote storage and 4 data files from the file system are stored as compressed files, which can be attributed to the reduction of the files' original sizes.

> File systems, utilized by 39% of repositories, represent the most commonly used storage location, although it causes unavailability and can eventually lead to reproducibility issues. Additionally, larger data files are stored only in remote storage and downloaded when needed.

4 RQ2: HOW ARE THE MODELS STORED IN THE STUDIED ML APPLICATIONS?

Motivation: The storage of an ML model plays an important role in maintaining its availability. In this research question, we investigate the storage practice of ML models in application repositories by the application developers. Additionally, we explore whether the storage location is correlated with the size of the model files.

Approach: We have 354 model files from 93 repositories. We first analyzed the distribution of the files across the repositories. Then we interpreted the storage location of a model file by examining its path following the same process we followed for data files. The only exception is, for the model files having unknown paths, we deduced their location to the *File System* as the model loading APIs are meant for loading local models. Next, we analyzed how frequently the developers used each storage type to store the model files in the same way we did for data files.

We also classify the trained models based on their source to understand their availability. Developers can either use a self-trained model or a pre-trained model in their applications. Thus, we classified trained models into three following categories:

- **Self-trained models:** The training code of the models is found in the same application repository
- **Pre-trained models:** The models are trained and provided by others
- **Unknown trainer:** The models with unknown paths

For both the self-trained and pre-trained models, we got the model files from the model loading code segments. To mark a model as a self-trained model, we searched for its information in the repository's `README` file and within the application's codebase. If we found relevant details in the `README` file or located the code responsible for training and saving the model in the application repository, we classified it as a self-trained model. Otherwise, the model was considered a pre-trained model. Then, we compared the number of model files in each storage type for both the self-trained models and the pre-trained models to understand if the model storage location choice differs for the model categories and discuss the availability of the self-trained and pre-trained model files.

To investigate whether the sizes of the model files are correlated with where developers choose to store them, we plotted the storage location types against the file sizes of the model files.

Findings: A median repository has one model file. Figure 5 shows that the number of model files varies from 0 to 47 in the repositories. 41 repositories have more than 1 model file, and in 23 of these repositories, all the model files are stored in the same storage type.

The file system is the most popular choice among application developers for storing model files. From Figure 3, we see that 47 (51%) of the 93 repositories use the file system to store data files, making it the most commonly used storage type. Repositories also employ remote storage (18%), repositories (16%) and runtime memory (1%) for storing data files. However, we did not find any model file loaded from the database like we found for data files. Also, we do not have any models from libraries like we have datasets from libraries because we did not analyze the model loading using third-party libraries except the three selected libraries in our analysis.

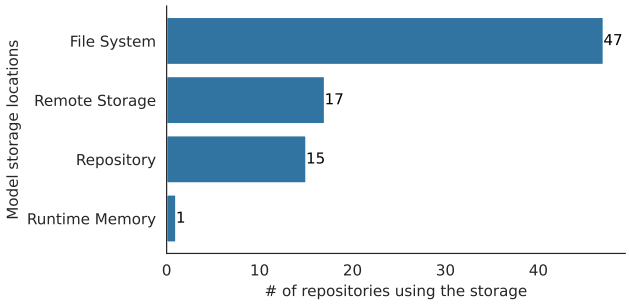
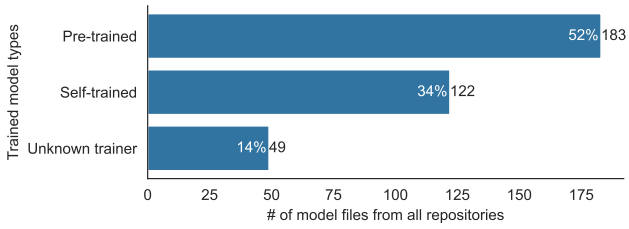
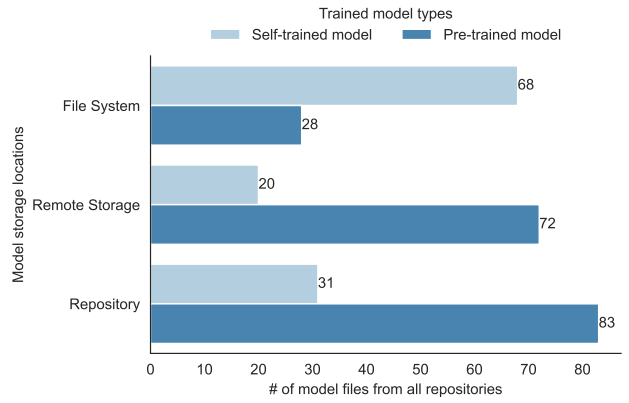


Figure 6: Comparison of usage of the storage locations for model files per repository. Note that a repository may use multiple storage locations for different model files.



(a) Type of trained model



(b) Model storage locations per type of trained model

Figure 7: Distribution of types of trained models and the distribution of the model storage locations across these types

The studied repositories referred more to pre-trained than self-trained models. From Figure 7a, we observe that among the 354 model files, 52% of the models are pre-trained, whereas, 34% of the models are self-trained. For 14% of the model files, the model's path remained unknown. This occurred when paths were set dynamically during code execution, were passed as method parameters without any method calls, or when the configuration file was missing from where the path had been set, or the code was too complicated to understand.

Self-trained models are commonly stored in the file system and pre-trained models are commonly stored in the application's repository. Figure 7b shows that self-trained model files are most commonly stored in file system (68), repository (31), and remote storage (20). The model files that are stored in the repository and the remote storage are available, however, the 68 self-trained model files that are stored in the file system, can be available after training the model from the available training code. On the other hand, for pre-trained model files, the order of popularity of the storage is repository (83), remote storage (72), and file system (28). These model files are considered pre-trained as neither the codebase for training the model nor any information of the model in the `README` was available. Like the self-trained model files, files that are stored in the repository and the remote storage are available, however, the 28 pre-trained model files that are stored in the file system, are not available in any way.

From the 354 model files, The path was unknown for 20 model files and 40 model files were found from the repositories' `README` file. The remaining **201 models were loaded from weights and 93 were loaded by restoring the saved model.** However, it is not always possible to understand if the loaded file is a weight file or a saved model file from the file's extension only. We had to understand this by analyzing the model loading code segments. Also, some of the files are used as checkpoints to resume training. Table 3 gives us an overview of how the models are used after loading them in 543 model loading code segments. We observed that 11% of the model loading code segments are used to resume training. However, it is not possible to understand from the file extension whether it is a checkpoint file. Thus, the flexibility of using any file extension to save files for different purposes like weight files and checkpoint files, requires additional instruction with the file on how to use them. From our manual analysis, we also found that developers sometimes load model files to transform the model. Table 3 shows that 4% of the model loading code segments load model files either to reduce the model file size or export the model to a different format for deployment. This gives us an overall impression that developers might sometimes encounter challenges in storing model files because of their data structure.

We encountered similar issues with URLs for model files as we did for data files. We found 8 model files in personal storage locations and 25 model files in university storage. We could not access 3 model files due to HTTP 404 errors. We also found 3 URLs referring to pages having multiple versions of the models.

Like the datasets, we have the size of the models loaded from remote storage and repositories. **Developers store larger models in remote storage locations and download the model when needed.** From Figure 8, we can see that although small model files are stored more in repositories, those larger than 65.72 MB are only stored remotely. The size of a median file in remote storage is 53.0 MB whereas the size of a median file stored inside a repository is 4.22 MB.

Table 3: Model load purposes in model loading code segments

Purpose	Description	# of code segments	%
Generate data	Infer or generate data for model's performance evaluation or for problem-solving	377	69%
Resume training	Resume/extend an early training or fine-tune a trained model	60	11%
Transform model	Export model file to different format for deployment or reduce model file size	21	4%
Unknown purpose	Could not find the reason for loading the model	85	16%
Total		**543**	**100%**

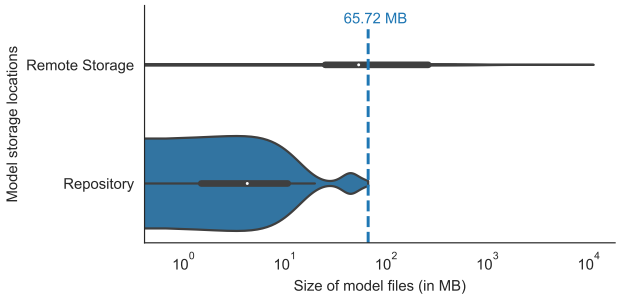


Figure 8: Comparison of the model file sizes in different storage locations

Similar to the datasets, models are typically stored in the file system, and larger ones are typically hosted in remote systems and downloaded when required. The studied repositories used pre-trained models more than self-trained models. 68 self-trained model files stored in the file system are available after training the model from the available training code and 28 pre-trained model files stored in the file system are not available in any way.

5 RQ3: HOW ARE THE DATASETS AND MODELS VERSIONED IN THE APPLICATION REPOSITORIES?

Motivation: The models and datasets change as the ML applications they are used in evolve. Therefore, the changes in the models and datasets should be managed in the application repositories like other software artifacts. In this research question, we want to understand how these changes are managed in the application repositories.

Approach: The data and model files are stored in different locations. We first discuss the versioning of the datasets and models from all storage locations based on the available data and model files. For the data and model files that are stored in the file system without versioning information, we checked if the files were intentionally moved out from the version control systems (VCSs) of the repositories. Using the `git check-ignore` command[12], we checked if the files were ignored from the VCS by any of the `.gitignore` files.

[12]https://git-scm.com/docs/git-check-ignore

As developers use traditional VCSs for software artifact versioning, we examine how data and model files are versioned in the VCS. For the data and model files that are versioned in repositories through VCS, we counted the commits for the files in the applications' history. For each repository, we listed the data and model files from the repository and listed the commits from the application repository history that changed the files. Then we counted such commits for the data files and the model files in the repository separately, and plotted the number of data file-changing commits and model file-changing commits in the repositories to see how often they change in the application repositories.

Findings: **Data and model files are most often missing version information in the application repositories.** Among the databases used for model training, the SQLite file is not versioned in the VCS. The other two MySQL and PostgreSQL databases are server-based databases, so cannot be versioned in a VCS. The data we obtained through library APIs, does not have associated version information. Data and model files fetched from personal remote storage or university domains through URLs are changeable without even versioning them. Also, multiple versions are available in remote storage for 10 data URLs and 3 model URLs, however, there is no information in the application repository on which version of data or model to use. In contrast, the data and model loaded into runtime memory are linked to version information due to the versioning of the data generation code.

The data and model files that are stored in the file system, are not traceable from the application repositories. **Data and model files from the file system are sometimes intentionally not versioned in the VCS.** 21 data files and 22 model files were ignored in 17 repositories through the `.gitignore` file. Among the 22 model files, 14 model files are self-trained and 8 are pre-trained. If these ignored data and pre-trained model files update, this will lead to a traceability challenge for data and model files in the application repository.

On the other hand, the files that are stored in the repositories are by nature versioned through the VCSs. However, data and model files are not often stored in repositories. Among all the data files, Figure 9 shows that only 26% of the data files are versioned in the VCS of the application repositories whereas among all the model files, Figure 10 show that only 32% of the model files are versioned in the VCS of the application repositories. Additionally, Figure 7b shows that self-trained models are less likely to be versioned in the VCS than pre-trained ones. Among the self-trained model files, 74% of them are not versioned in VCS. Developers may assume that they can retrain the model using the available code

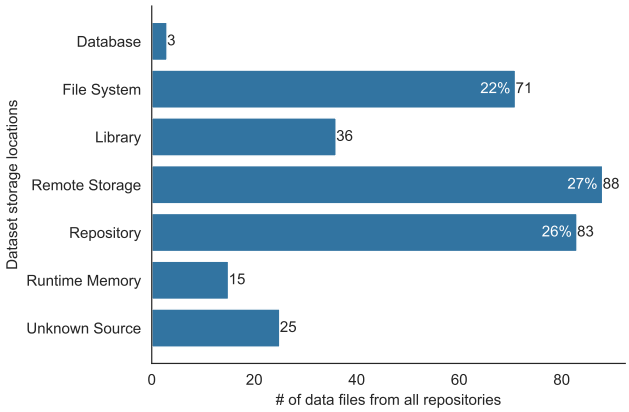


Figure 9: Comparison of the number of data files in different storage locations

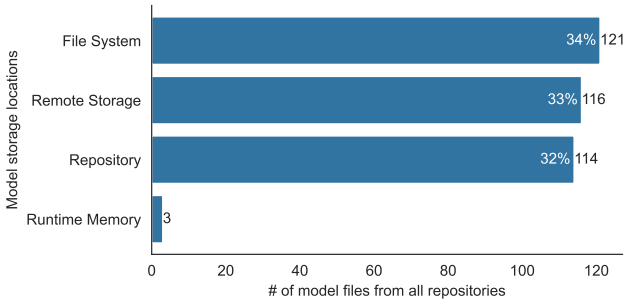


Figure 10: Comparison of the number of model files in different model storage locations

and dataset in an older revision, however, retraining a large model can be time-consuming and inefficient when reproducing issues. Also, the dataset also needs to be versioned with proper updates in the application history. Thus, the data and model files that are not versioned can lead to reproducibility issues in older revisions for bug fixes.

Most of the datasets and models stored in repositories are rarely updated during the development of the application. Models are updated more than the datasets in the repositories. According to Figure 11, although the number of repositories having versioned datasets is higher than the number of repositories having versioned models, the number of commits for the models in a repository is more than the number of commits for the datasets in the repositories. Repositories have a median of 1.5 commits for the datasets in the repository and 2 commits for the models in the repository. This indicates that, in an average repository, a data file is updated once or never after its initial commit. Similarly, an average repository updates the model files just once after their initial commit. Remarkably, one repository stands out with 22 commits for 46 saved model files in VCS.

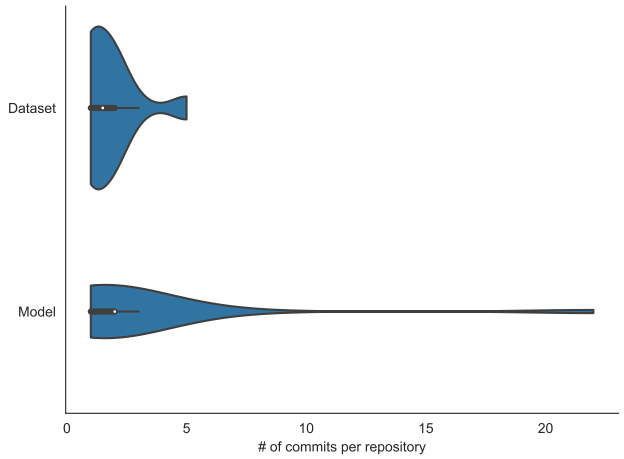


Figure 11: Commit frequency of the data and model files per repositories

The application repositories often lack version information for datasets and models. Most of the data and model files that are versioned in VCS are rarely modified during application development, with model files undergoing more updates than data files. This may lead to traceability and reproducibility issues for developers while fixing bugs in older application versions.

6 IMPLICATIONS

We found that developers typically handle datasets and models similarly in terms of storage location. They mostly store the data and model files on the file system, often not ensuring their availability. As expected, developers exclusively use remote storage to store large files. They may choose remote storage for large files to save space in their local file system, securely store the files for future use, or share the files with others. However, storing files in remote storage sometimes hinders the files' availability. To resolve the issue, developers can utilize Git Large File Storage (Git LFS) to store large data and model files to make them available with versioning within the repository, which also serves all the aforementioned purposes.

Although traditional VCSs are not well-suited for managing the large and diverse assets used in machine learning model development [22], our study shows that datasets and models are sometimes versioned in VCS alongside other software artifacts. However, they are not updated very often (if at all). Also, the datasets and models that are loaded from remote storage or libraries are not often labelled with version information. Therefore, application developers need guidelines and tool support for integrating datasets and models into their systems while preserving their version information and keeping track of their further updates in the application repositories.

Many systems and platforms support data analysts and model developers in creating datasets and models, as well as managing

machine learning artifacts. However, model developers often do not utilize these ML AMSs to manage the datasets and models. A study by Barrak et al. [4] identified only 391 GitHub repositories that use DVC[14], one of the most popular ML AMSs integrated with Git. In our study, we did not find any application documentation that mentioned any use of the ML AMSs. Even if model developers used such tools to manage dataset and model evolution, the updates of the dataset and model files are not reflected in the VCS of the application repositories.

7 THREATS TO VALIDITY

Internal Validity: We filter out the learning repositories by their topic, name and description (Section 2.2). However, this filtering will not work for the repositories with empty topics and descriptions, and unembedded names. After randomly selecting 93 repositories, we read the documentation of the repositories. We found one repository used for course management. We replaced the repository with a new randomly selected repository.

To select repositories by library version (Section 2.2), we parsed the dependencies of the repositories from their `requirements` files only. However, repositories can use other files for their dependency management. From the 4,023 repositories, we found `requirements` files in 3,352 repositories. As we did a manual analysis, we ignored these 671 repositories in further processing.

To properly understand the purpose of the code during the manual analysis (Section 2.4), we read the code comments, analyze the identifier names, and navigate all traceable execution paths of the code. However, because of the dynamicity of Python, we still may have missed some execution paths or misunderstood some parts of the code. Also, it is possible that some of the code we included in our result is code that is not actually used in practice by the project.

Recently, third-party model providers (such as Hugging Face) have become popular for managing (very large) pre-trained models. In our study, we focused on models that could be loaded into the application specifically using TensorFlow, PyTorch and Scikit-learn, and we did not specifically cover other third-party providers. Future studies should expand our study to include such providers.

External Validity: One of the external threats to our results is generalization. Our findings apply to open source repositories from GitHub only. Hence, future studies are necessary to investigate if our results hold for other hosting platforms like Bitbucket, GitLab.

8 RELATED WORK

We group the related work based on their purpose and present them in the following two sections.

8.1 ML application management studies

Calefato et al. [5] studied the deployment aspect of ML applications' management. They gathered an initial understanding of how MLOps solutions are used to automate task executions to build and deploy ML projects. They analyzed 397 GitHub Actions workflows from 155 GitHub repositories and 38 CML workflows from 29 GitHub repositories. The authors aimed to identify the current state of MLOps workflows in the GitHub repositories and noted that the adoption of MLOps workflows in open-source GitHub projects is currently somewhat limited.

Barrak et al. [4] and Njomou et al. [17] studied how the ML AMSs are used in open-source ML applications. Barrak et al. [4] empirically studied the prevalence of ML pipelines in open-source projects, as well as the amount of maintenance effort involved. They aimed to address this goal through a high-level empirical study of 391 GitHub projects using the DVC[14] versioning tool, followed by a more detailed analysis of the 25 most active projects. The authors found a non-negligible maintenance overhead for developers and data scientists working on ML applications due to the tight coupling between ML-related artifacts and other software artifacts, such as build files and infrastructure-as-code files.

Njomou et al. [17] discussed the main challenges that developers face when adopting and/or migrating existing ML projects to Machine Learning Life Cycle Management (MLLCM) platforms. They migrated the platform of 13 ML projects on GitHub to DVC[14] and MLFlow[29], logged the challenges they faced at each stage of the migration, and recommended potential solutions to overcome these challenges. They also propose some steps that developers can take when building ML projects in order to ease any future migration to MLLCM platforms.

8.2 Improved Versioning Support for ML Applications

Tran [27] claimed that tools with versioning support for machine learning assets are more tailored toward the data scientists' perspective and less towards software engineers' perspective. The tools are not connected to established version control systems, which would require software engineers to adapt to a completely different kind of tooling that they are not used to for the management of machine learning assets. Therefore, the author developed a query language that is able to manage machine learning assets with improved versioning support that is more focused on software engineers. This addresses the need for a new tool with improved versioning support for machine-learning-based systems dedicated to software engineers which has been indicated in several studies. The author also presented a design science approach to creating this tool that is accessible to both software engineers and data scientists.

9 CONCLUSION

ML applications rely on two key components: datasets and models, which evolve during application development. Hence, these components require management like traditional software artifacts. While several ML Artifact Management Systems exist to support the management of ML artifacts during model development, the management of the datasets and models by the ML application developers in the application repositories remains unexplored. This paper focuses on the storage and versioning aspects of dataset and model management within ML applications. A manual analysis of 93 GitHub repositories reveals varying storage and versioning practices. Many repositories store data files on the file system and load models from the file system. However, the versioning of data and model files in application repositories is limited. This study highlights the need for more specialized software engineering tools and practices to support developers in integrating datasets and models into their applications while preserving version information and tracking updates in application repositories.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Montreal, QC, Canada, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[3] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. 2018. Software Engineering Challenges of Deep Learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Prague, Czech Republic, 50–59. https://doi.org/10.1109/SEAA.2018.00018

[4] Amine Barrak, Ellis E. Eghan, and Bram Adams. 2021. On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 422–433. https://doi.org/10.1109/SANER50967.2021.00046

[5] Fabio Calefato, Filippo Lanubile, and Luigi Quaranta. 2022. A Preliminary Investigation of MLOps Practices in GitHub. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '22)*. Association for Computing Machinery, Helsinki, Finland, 283–288. https://doi.org/10.1145/3544902.3546636

[6] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564. https://doi.org/10.1109/MSR52588.2021.00074

[7] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. The State of the ML-Universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 431–442. https://doi.org/10.1145/3379597.3387473

[8] Marc Hesenius, Nils Schwenzfeier, Ole Meyer, Wilhelm Koop, and Volker Gruhn. 2019. Towards a Software Engineering Process for Developing Data-Driven Applications. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, Montreal, QC, Canada, 35–41. https://doi.org/10.1109/RAISE.2019.00014

[9] Nipuni Hewage and Dulani Meedeniya. 2022. Machine Learning Operations: A Survey on MLOps Tool Support. *arXiv preprint* (2022). https://doi.org/10.48550/ARXIV.2202.10169 arXiv:2202.10169 [cs.SE]

[10] Samuel Idowu, Daniel Strüber, and Thorsten Berger. 2021. Asset Management in Machine Learning: A Survey. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Madrid, ES, 51–60. https://doi.org/10.1109/ICSE-SEIP52600.2021.00014

[11] Samuel Idowu, Daniel Strüber, and Thorsten Berger. 2022. Asset Management in Machine Learning: State-of-Research and State-of-Practice. *Comput. Surveys* 55, 7, Article 144 (dec 2022), 35 pages. https://doi.org/10.1145/3543847

[12] Jeremy Katz. 2020. Libraries.io Open Source Repository and Dependency Metadata. https://doi.org/10.5281/ZENODO.808272

[13] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2884781.2884783

[14] Ruslan Kuprieiev, Dmitry Petrov, Paweł Redzyński, Casper Da Costa-Luis, Saugat Pachhai, Alexander Schepanovski, Ivan Shcheklein, Peter Rowlands, Jorge Orpinel, Fábio Santos, Aman Sharma, , Zhanibek, Dani Hodovic, Karajan1001, Andrew Grigorev, , Earl, Nabanita Dash, Nik123, George Vyshnya, Maykulkarni, Xliiv, Max Hora, , Vera, Sanidhya Mangal, Wojciech Baranowski, Clemens Wolff, Alex Maslakov, Alex Khamutov, Kurian Benoy, and Ophir Yoktan. 2020. DVC: Data Version Control - Git for Data & Models. https://doi.org/10.5281/ZENODO.4278171

[15] Sergio Moreschi, Gilberto Recupito, Valentina Lenarduzzi, Fabio Palomba, David Hastbacka, and Davide Taibi. 2023. Toward End-to-End MLOps Tools Map: A Preliminary Study based on a Multivocal Literature Review. *arXiv preprint* (2023). arXiv:2304.03254 [cs.SE]

[16] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3510003.3510209

[17] Aquilas Tchanjou Njomou, Marios Fokaefs, Dimitry Fumtim Silatchom Kamga, and Bram Adams. 2022. On the Challenges of Migrating to Machine Learning Life Cycle Management Platforms. In *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering* (Toronto, Canada) *(CASCON '22)*. IBM Corp., USA, 42–51.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).

[19] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[20] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. 2021. Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences* 11, 19 (2021), 8861. https://doi.org/10.3390/app11198861

[21] Iqbal H Sarker. 2021. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN computer science* 2, 3 (2021), 160. https://doi.org/10.1007/s42979-021-00592-x

[22] Danilo Sato, Arif Wider, and Christoph Windheuser. 2019. Continuous Delivery for Machine Learning. https://martinfowler.com/articles/cd4ml.html. [Accessed 17-11-2023].

[23] Marius Schlegel and Kai-Uwe Sattler. 2022. CORNUCOPIA: Tool Support for Selecting Machine Learning Lifecycle Artifact Management Systems. In *Proceedings of the 18th International Conference on Web Information Systems and Technologies - WEBIST*. SCITEPRESS - Science and Technology Publications, Valletta, Malta, 444–450. https://doi.org/10.5220/0011591700003318

[24] Marius Schlegel and Kai-Uwe Sattler. 2023. Management of Machine Learning Lifecycle Artifacts: A Survey. *ACM SIGMOD Record* 51, 4 (jan 2023), 18–35. https://doi.org/10.1145/3582302.3582306

[25] Monika Steidl, Michael Felderer, and Rudolf Ramler. 2023. The pipeline for the continuous development of artificial intelligence models—Current state of research and practice. *Journal of Systems and Software* 199 (2023), 111615. https://doi.org/10.1016/j.jss.2023.111615

[26] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A. Papakostas. 2022. MLOps - Definitions, Tools and Challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, Las Vegas, NV, USA, 0453–0460. https://doi.org/10.1109/CCWC54503.2022.9720902

[27] Erik Tran. 2022. *Development of a Query Language for Improved Versioning Support for Machine-Learning-Based Systems*. Master's thesis. University of Gothenburg.

[28] Carl Vågfelt Nihlmar. 2023. *Tools evolving AI systems via experiment management: A survey of machine learning practitioners*. Bachelor's thesis. University of Gothenburg.

[29] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.