Data versioning in machine-learning architecture

Peter Bartoš, Stanislav Krištof

Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava

November 27, 2024

Abstract

Data versioning plays a crucial role in modern machine learning architecture, ensuring that the complex and ever-evolving datasets that provide the basis for models can be tracked, compared, and managed efficiently. At its core, data versioning refers to the practice of creating unique references for different states of a dataset over time, allowing us to trace changes, restore previous versions, and debug issues. This is vital in machine learning workflows, where even small changes in data can significantly impact model performance.

In this domain, data versioning supports reproducibility by maintaining a consistent link between datasets and the models trained on them. Without version control, it becomes challenging to recreate experiments, leading to inconsistencies in predictions and hindering model audits. Versioning also simplifies collaboration across teams, enabling multiple stakeholders to work on the same data without overwriting each other's progress.

Basic approaches to data versioning systems (DVS) include full duplication of datasets, where copies are saved with each change, and metadata-based versioning, where timestamps indicate the validity of each record. Advanced solutions (like lakeFS and DVC) deal with versioning as a core component of machine learning architecture. They enable storage-efficient data commits, branching, and comparison, similar to how Git handles version control in software development.

Overall, data versioning enhances productivity, reduces errors, and fosters an engineering-driven approach to handling data in machine learning pipelines, ultimately enabling smoother transitions between development stages and more robust model deployment. The objective of the project is to go over these systems and provide detailed overviews of how data versioning has such a crucial role in machine learning architecture.

1 Introduction

According to ACM [6], reproducibility is the ability to obtain measurement with stated precision by different team under the same condition as the initial team. For present-day advances in AI, reproducibility is key. Peng [20] regards reproducibility the ultimate arbiter of scientific claims. A high level of reproducibility enables more transparent acquirement of evidence either for or against certain hypotheses. However, analysis of 602 papers by Pawlik et al.[19] claims that only 7.64% of them were reproducible. More advanced data version control can also lead to better and more complex ablation studies, which greatly increases our understanding of NN. Pawlik et al. [19] argue, that for higher reproducibility dataset should not only contain input data, but also raw data and preparation instructions that convert raw data to input data. Raw data serves to put data into context. We can also generate new input data using discarded data together with the preparation instructions and augmentation techniques.

To understand how data versioning integrates into machine learning architecture, it is essential to examine the components and workflows that define this architecture. Machine learning pipelines rely on iterative experimentation, where data serves as the foundation for every stage, from preprocessing to model training, evaluation, and deployment. At each step, maintaining a consistent and traceable version of datasets and modoels is critical for achieving reproducibility. [5]

2 Version control in Machine Learning

In the process of developing a machine learning model, various artifacts beyond source code are produced. An artifact refers to any file that serves as an input or output of a given process. In traditional software development, inputs such as source code and libraries qualify as artifacts, while outputs like object files and executables are also considered artifacts. For instance, in a build process, these outputs represent the results of transforming inputs through the compilation pipeline. [5, 22]

In the context of machine learning, the most critical artifacts are datasets and models. Datasets act as the inputs to the training process, while models, which encapsulate learned parameters and structures, serve as the outputs of this process. [22]

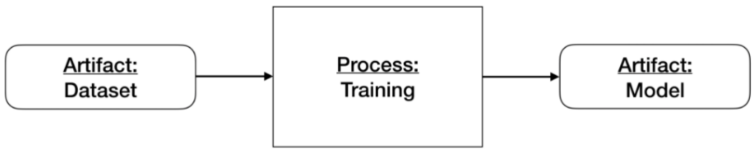

Figure 1: Artifacts in Machine Learning [5]

2.1 Use of Data Versioning in Machine Learning

In machine learning workflows, managing datasets is as important as managing code, especially when it comes to reproducibility and collaboration. A systematic approach to data versioning enables teams to track, store, retrieve, and switch between different versions of datasets seamlessly, ensuring consistency across development and experimentation phases. [5, 22]

The process begins with versioning the dataset. Just as changes in source code are tracked, datasets need to be tracked too. Metadata files can be created to store information about each dataset version without including the actual dataset itself, keeping the repository efficient. These metadata files act as pointers to the specific dataset version associated with a particular stage of the machine learning pipeline. [22, 2]

Next, the dataset itself is stored in a dedicated storage system. This storage can be local or cloud-based, and it holds the actual data files. By separating the data from the metadata stored in the repository, teams can handle large datasets without bloating their version control systems, while still ensuring that every version is accounted for.[2]

Retrieving a specific dataset version is then made possible through the metadata file. The system can fetch the corresponding dataset version from storage based on the metadata, ensuring that the data matches the state of the code and experiments conducted at that time. This eliminates ambiguity and enhances reproducibility in experiments. [2, 1]

Lastly, the ability to switch between dataset versions ensures flexibility during development. Just as a developer can revert to an earlier version of their code, they can also revert to or load the dataset version that was used with that code. This synchronization between code and data allows for iterative testing, analysis, and debugging, supporting more rigorous and reproducible experimentation. [2, 1]

By implementing data versioning practices in this way, teams can ensure that their machine learning workflows are robust, efficient, and conducive to collaboration, enabling seamless transitions across different stages of development and research. [5, 2, 1]

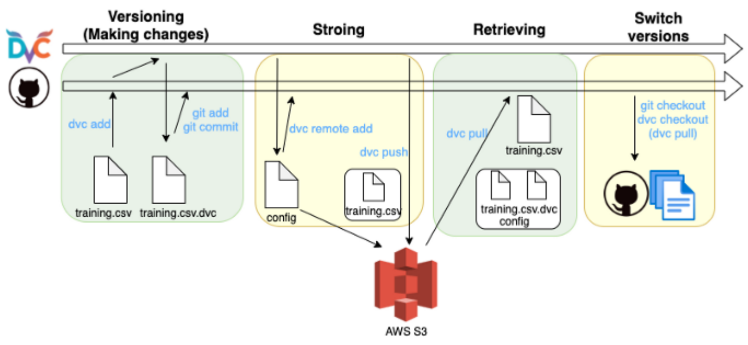

Figure 2: Example of how can be data versioning used in machine learning (DVC and GIT as examples) [2]

2.2 Advantages of Data Versioning in Machine Learning

In machine learning workflows, data versioning is not just about tracking changes; it also facilitates collaboration across multiple teams and ensures consistency in model development processes. By creating a structured system of versioning, practitioners can manage datasets and models at different stages of development, allowing seamless integration, testing, and improvement of machine learning pipelines. [5, 2]

Data versioning enables branching and merging of datasets, similar to how code branches work in software development. This is especially important when multiple teams or practitioners work on different aspects of a project. For instance, different projects or teams may use the same foundational dataset but modify or enhance it according to specific needs. These changes can be organized into separate "streams" or versions, ensuring that each team has a clear record of its modifications while still maintaining a connection to the original data. [5, 2]

Versioning also allows for "harvesting", where changes or improvements made in one branch can be integrated back into the main dataset or shared across other streams. This ensures that advancements in one area can benefit the broader workflow without disrupting other teams' progress. Similarly, the propagation of changes ensures consistency across different projects and keeps all streams aligned with the latest updates. [2, 4]

Moreover, data versioning systems can track multiple hierarchical streams, ranging from enterprise-level datasets to project-specific and even individual practitioners' workstreams. This hierarchical structure ensures flexibility and scalability, allowing datasets to evolve alongside their respective machine learning models. [2, 4]

By implementing a robust data versioning framework, teams can manage data lineage, collaborate effectively, and ensure reproducibility, even in complex machine learning ecosystems. This structured approach enhances the ability to build, test, and deploy models with greater confidence. [2, 4]

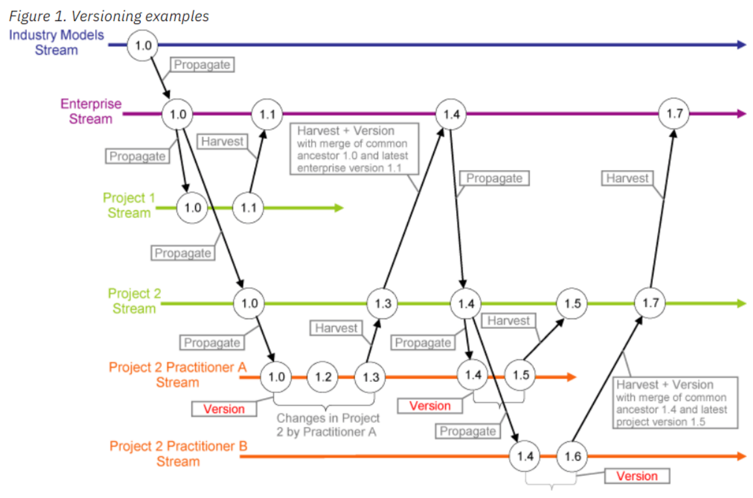
Figure 1. Versioning examples
Industry Models Stream
Enterprise Stream
Project 1 Stream
Project 2 Stream
Project 2 Practitioner A Stream
Project 2 Practitioner B Stream
Propagate
Harvest
Harvest + Version with merge of common ancestor 1.0 and latest enterprise version 1.1
Harvest + Version with merge of common ancestor 1.4 and latest project version 1.5
Version
Changes in Project 2 by Practitioner A
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7

Figure 3: An example of hierarchical data versioning streams showcasing branching, propagation, and merging across industry, enterprise, project, and individual practitioner levels, enabling collaborative and scalable data management in machine learning workflows [4]

3 Insight into. . .

Types of Data Version Control Systems (DVCS) According to Zolkifli et al. [24] in centralised DVCS, a singular repository is located on the server. These DVCS are suitable for teams with smaller number of members, with the team being located in a single place. Only the last version of a file is retrieved and there is a single point of failure. Examples include Apache Subversion or Perforce Revision Control System. In distributed DVCS, on the other hand, each user has one local repository. Examples include Git, Mercurial, Bazaar or Bitkeeper. This type is suitable regardless of team size. It also enables users being located in different parts of the world. Clients can create their own branches and sync them with the server.

Git As stated by Komsiyski [16], Git works on basis of patch files. As noted by Bryan [7], large and often changing files are not suitable for Git, since they can slow down pushes and pulls. According to Perez et al. [21], binary files in git are stored as a single large entity. Therefore, even small changes lead to new copies in the repository. This also makes it difficult to compare changes in files using diff and may lead to frequent merge conflicts.

Git LFS Git also offers an extension for large files called Git LFS (Large File Storage), which enables more efficient handling of large files [21]. Content of the file is stored in cloud, with the repository containing only pointers to the files, as noted in the documentation [13] (Figure 4). Since Git LFS is a GitHub extension, its advantages are compatibility with Git [14] and file agnosticism (compatibility with all file formats) [17]. Its disadvantages are inefficient storage management - similar to Git, edited files are stored as new files. Therefore, it is not suitable for large frequently-changing files, especially if they are compressed (such as file formats frequently used in computer vision, e.g. jpg, png) [15]. In addition, data in Git LFS do not stay in place. It also does not scale as well and its data retrieval is slow. As such, this approach is suitable mostly for game developers and not for for ML and data science purposes.

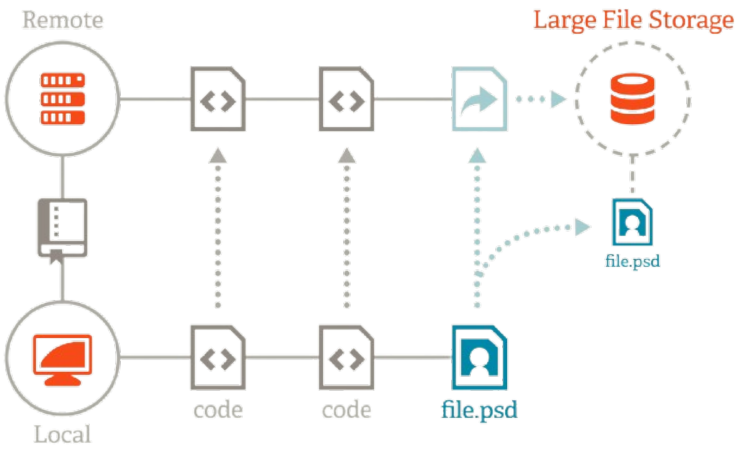

Figure 4: Software architecture of Git LFS [18]

Dolt Dolt is a version-controlled SQL database and as such it may serve as an example of a centralised DVS. Analogous to Git, one can track schemas and changes in the database, create and merge branches [8]. Under the hood, Dolt uses Prolly trees - a data structure related to both B-trees [23], commonly used in RDBMS and Merkle trees (used by distributed version control systems such as Git or Mercurial)[17]. This data structure provides both fast performance (including diffs and merges) and efficient storage management (each portion of data shared between trees is shared only once). However, like other RDBMS, while Dolt may be the ideal solution for structured data, it is not suitable for unstructed data.

DVC by Iterative Another solution, suitable for ML learning may be DVC [3]. DVC achieves faster data retrieval by its data staying in place. In addition, DVC also uses a caching layer (Figure **??**), which allows faster data retrieval for multiple members of team [17]. DVC supports both structured and unstructed data. It however does not support RDBMS. Since it caters to data scientists, it offers several features which greatly lead to higher reproducibility, such as defining data pipelines [9], visualisation of pipelines [12], experiment tracking [11] and hydra compatibility [10]. It also does not scale as well, mostly due to the same reasons as git.

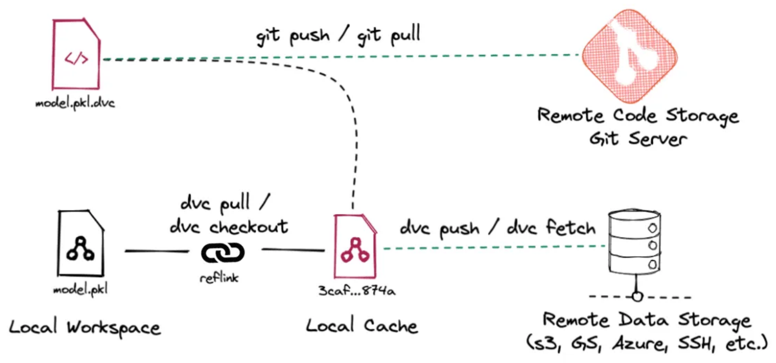

Figure 5: Software architecture of DVC [17]

4 Conclusions and Further Work

Data versioning has proven to be an indispensable component of modern machine learning workflows, addressing critical challenges such as reproducibility, collaboration, and scalability. As demonstrated, implementing data versioning systems (DVS) enables teams to track and manage datasets efficiently, ensuring consistency and traceability throughout the machine learning pipeline. The discussion on tools like Git LFS, Dolt, and DVC highlights the varying approaches to tackling version control, emphasizing the trade-offs between structured and unstructured data management, performance, and scalability.

Despite these advancements, challenges remain. Centralized systems often struggle with scalability and single points of failure, while distributed systems require efficient handling of large, frequently changing datasets. Tools like Git LFS, although widely used, are limited in their capacity to manage large binary files effectively. Similarly, while Dolt is highly efficient for structured data, it does not address unstructured data needs. DVC offers significant advantages for data scientists but lacks support for relational database management systems (RDBMS).

Future research and development in data versioning would focus on developing systems that can efficiently handle large, frequently updated datasets, especially those used in domains like computer vision, genomics, and IoT. There is also a need to address the designing of hybrid systems that can seamlessly manage both relational and non-relational data to address the diverse needs of modern machine learning workflows. Also leveraging AI to automate dataset versioning is also a must. This would identify anomalies, and suggest optimal data branches or merges based on usage patterns. For final touches there needs to be a better performance optimization to ensure faster data retrieval, reduced storage overheads, and better handling of binary files through innovations in data structures and caching mechanisms.

By addressing these challenges, data versioning systems can continue to evolve, further cementing their role as a foundational element of machine learning architecture. In doing so, they will not only improve the efficiency and reliability of machine learning pipelines but also unlock new possibilities for reproducibility and collaboration in the broader AI and data science community.

References

[1] Mlops: Data versioning with dvc — part i. 2021.

[2] How data versioning can be used in machine learning. 2022.

[3] Dvc documentation. 2024.

[4] In-depth guide to data versioning: Benefits and formats. 2024.

[5] Intro to mlops: Data and model versioning. 2024.

[6] ACM. Artifact review and badging. August 24, 2020.

[7] J. Bryan. Excuse me, do you have a moment to talk about version control? *The American Statistician*, 72(1):20–27, 2018.

[8] Dolt. *Data and Model Quality Control.* Accessed on 2024-11-6.

[9] DVC. *Defining Pipelines.* Accessed on 2024-10-12.

[10] DVC. *Hydra Composition.* Accessed on 2024-10-12.

[11] DVC. *Reviewing and Comparing Experiments.* Accessed on 2024-10-12.

[12] DVC. *Running pipelines.* Accessed on 2024-10-12.

[13] GitHub. *About Git Large File Storage.* Accessed on 2024-11-05.

[14] GitHub. *Collaboration with Git Large File Storage.* Accessed on 2024-11-05.

[15] GitHub. When to use git lfs. Accessed on 2024-10-13.

[16] V. Komsiyski. Binary differencing for media files. 2013.

[17] E. Orr. Data versioning as your 'get out of jail' card - dvc vs. git-lfs vs. dolt vs. lakefs. 2024. Accessed on 2024-10-12.

[18] A. Pasha. Understanding git large file storage (lfs): Efficiently managing large files in git repositories. 2023. Accessed on 2024-11-08.

[19] M. Pawlik, T. Hütter, D. Kocher, W. Mann, and N. Augsten. A link is not enough–reproducibility of data. *Datenbank-Spektrum*, 19:107–115, 2019.

[20] R. D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[21] Y. Perez-Riverol, L. Gatto, R. Wang, T. Sachsenberg, J. Uszkoreit, F. d. V. Leprevost, C. Fufezan, T. Ternent, S. J. Eglen, D. S. Katz, et al. Ten simple rules for taking advantage of git and github, 2016.

[22] M. R. Pulicharla. Data versioning and its impact on machine learning models. *Journal of Science & Technology*, 5(1):22–37, 2024.

[23] T. Sehn. Prolly trees. 2024. Accessed on 2024-11-08.

[24] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018.