Data versioning in machine-learning architecture

Peter Bartoš, Stanislav Krištof

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

November 8, 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 [2], 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 [16] 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.[15] 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. [15] 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.

Outlook The rest of this report is structured as follows. Section 2 provides an insight into... Section 3 explains in more detail some important approaches to... Section 4 brings the initial steps to... (Characterize each section by a sentence.) Section 8 concludes the paper and indicates some directions for further work.

2 Insight into...

Types of Data Version Control Systems (DVCS) According to Zolkifli et al. [19] 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 [12], Git works on basis of patch files. As noted by Bryan [3], large and often changing files are not suitable for Git, since they can slow down pushes and pulls. According to Perez et al. [17], 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 [17]. Content of the file is stored in cloud, with the repository containing only pointers to the files, as noted in the documentation [9] (Figure 1). Since Git LFS is a GitHub extension, its advantages are compatibility with Git [10] and file agnosticism (compatibility with all file formats) [13]. 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) [11]. 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.

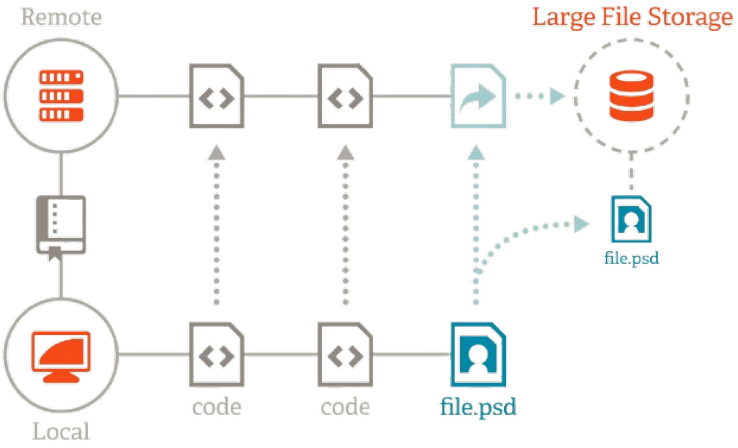

Figure 1: Software architecture of Git LFS [14]

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 [4]. Under the hood, Dolt uses Prolly trees - a data structure related to both B-trees [18], commonly used in RDBMS and Merkle trees (used by distributed version control systems such as Git or Mercurial)[13]. 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 [1]. 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 [13]. 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 [5], visualisation of pipelines [8], experiment tracking [7] and hydra compatibility [6]. It also does not scale as well, mostly due to the same reasons as git.

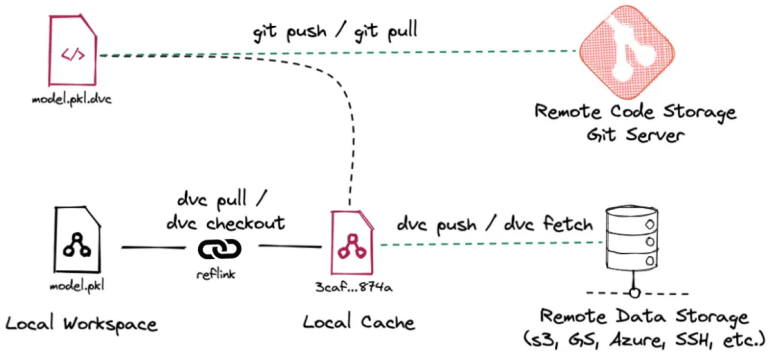

Figure 2: Software architecture of DVC [13]

Present your insight into the state of the art. Favor comparison and critique over description. Avoid lengthy descriptions with lots of quoted material.

Use your own title for this section..

You may structure your sections (see below). If you use subsections, use at least two, i.e., don't put only one subsection.

It might be a good idea to explain the structure of the rest of the section. Sometimes, an explicit way of doing at just as is demonstrated at the end of the introduction (see Section 1).

2.1 Some Aspects

...

2.2 Other Aspects

...

3 Important Approaches to...

You may need one more section to treat the state of the art. Everything said for the previous section, holds for this one, too.

4 Initial Steps to...

Describe your own approach. Of course, use your own title for this section.

Put your diagrams in figures as so-called floating object. Refer to them using their numbers. E.g., "in Figure 3 we can see..."

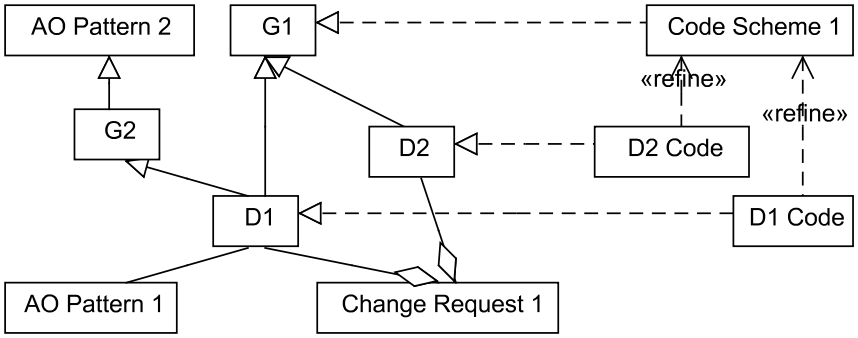

Figure 3: Generally applicable and domain specific changes.

You may include code snippets to explain what you've done:

```
public class SMTPServerM extends SMTPServer {
   . . .
}
. . .
public aspect SMTPServerBackupA {
   public pointcut SMTPServerConstructor(URL url, String user, String password):
      call(SMTPServer.new(..)) && args (url, user, password);
   SMTPServer around(URL url, String user, String password):
      SMTPServerConstructor(url, user, password) {
      return getSMTPServerBackup(proceed(url, user, password));
   }
   private SMTPServer getSMTPServerBackup(SMTPServer obj) {
      if (obj.isConnected()) {
         return obj;
      }
      else {
         return new SMTPServerM(obj.getUrl(), obj.getUser(),
            obj.getPassword());
      }
   }
}
```

If you need to display more code, use appendices referring the reader to them, e.g., "see Appendix A for a more detailed example."

5 Further Steps to...

You may need several sections to describe your approach.

6 Evaluation

You may describe your evaluation efforts in a separate, often generically entitled section.

6.1 Essential Evaluation

Use your own title here.

6.2 Threats to Validity

. . .

7 Related Work

Compare your achievements to related ones achieved by others.

8 Conclusions and Further Work

Emphasize the main results.

Indicate what can be done next.

The concluding section is typically not decomposed into subsections. Simply use several paragraphs to present conclusions, and then use at least one paragraph to indicate further/future work.

References

[1] Dvc documentation. 2024.

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

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

A Some Appendix

B Yet Another Appendix