Data Version Control (DVC) in Machine Learning Projects

Nhóm MLOps

Ngày 18 tháng 10 năm 2025

Nội dung về Data Version Control (DVC) được chia thành 5 phần chính:

- **Phần 1: Tổng quan về AI, MLOps và Data Versioning**
- **Phần 2: Thách thức trong Quản lý Dữ liệu và Code**
- **Phần 3: Giới thiệu về Data Version Control (DVC)**
- **Phần 4: Case Study: Triển khai DVC cho Dataset MNIST**
- **Phần 5: Tự động hóa Pipelines và Các khái niệm Versioning**

Phần 1: Tổng quan về AI, MLOps và Phiên bản hóa Dữ liệu (Data Versioning)

1.1 Data Versioning là gì ?

Data Versioning (phiên bản hóa dữ liệu) là một hệ thống dùng để quản lý và theo dõi những thay đổi của dữ liệu và mô hình trong suốt vòng đời dự án Machine Learning (ML).
T Nó không chỉ lưu lại các phiên bản khác nhau của bộ dữ liệu (dataset), mà còn quản lý cả các file cấu hình (Config), tham số (parameters) và kết quả đánh giá (Eval result). Về cơ bản, nó hoạt động như "Git cho dữ liệu", giúp kết nối một phiên bản code cụ thể với một phiên bản dữ liệu chính xác mà code đó đã sử dụng để huấn luyện và tạo ra mô hình.

1.2 Vì sao cần Data Versioning?

1.2.1 Góc nhìn nghiệp vụ (Business Perspective)

Từ góc nhìn nghiệp vụ, AI được xem là một chức năng (function) để giải quyết các bài toán cụ thể, ví dụ như phân tích cảm xúc (Sentiment Analysis) hay xây dựng hệ thống gợi ý (Recommendation) trong một hệ thống phần mềm lớn như Thương mại điện tử.

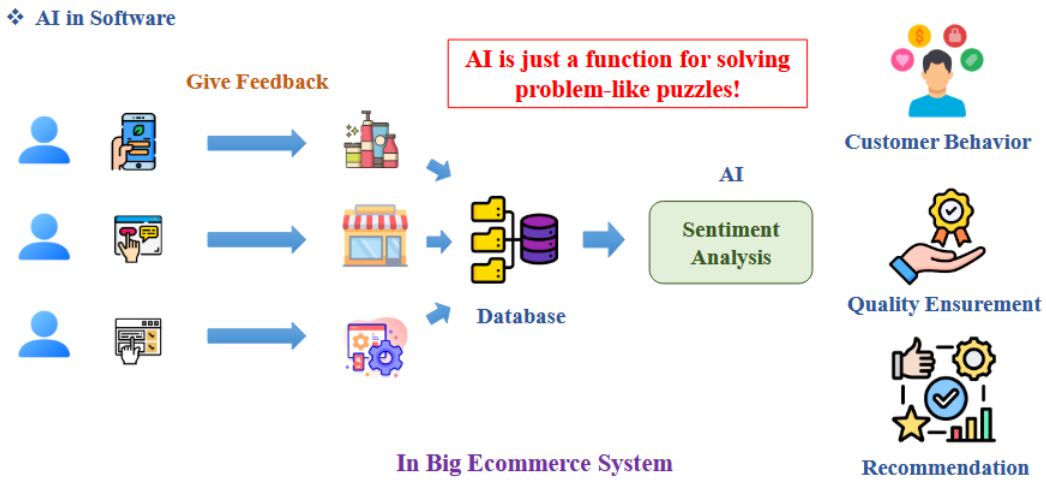


Figure 1: Vai trò của AI trong một hệ thống Thương mại điện tử

Toàn bộ quy trình ML bắt đầu từ **Yêu cầu Nghiệp vụ (Business Require)** và đi qua một vòng đời (life cycle) hoàn chỉnh. Để đảm bảo chất lượng (Quality Ensurement) và giám sát (Monitoring) mô hình AI một cách hiệu quả, doanh nghiệp phải có khả năng trả lời các câu hỏi:

- Mô hình đang chạy trên production được huấn luyện từ dữ liệu nào?
- Khi mô hình dự đoán sai, làm thế nào để tái lập (reproduce) lỗi đó?
- Nếu dữ liệu mới làm giảm hiệu suất, làm sao để quay lại phiên bản mô hình ổn định trước đó?

Data versioning cung cấp khả năng **truy xuất nguồn gốc (lineage)** này, cho phép theo dõi chính xác code nào, dữ liệu nào, tham số nào đã tạo ra mô hình nào.

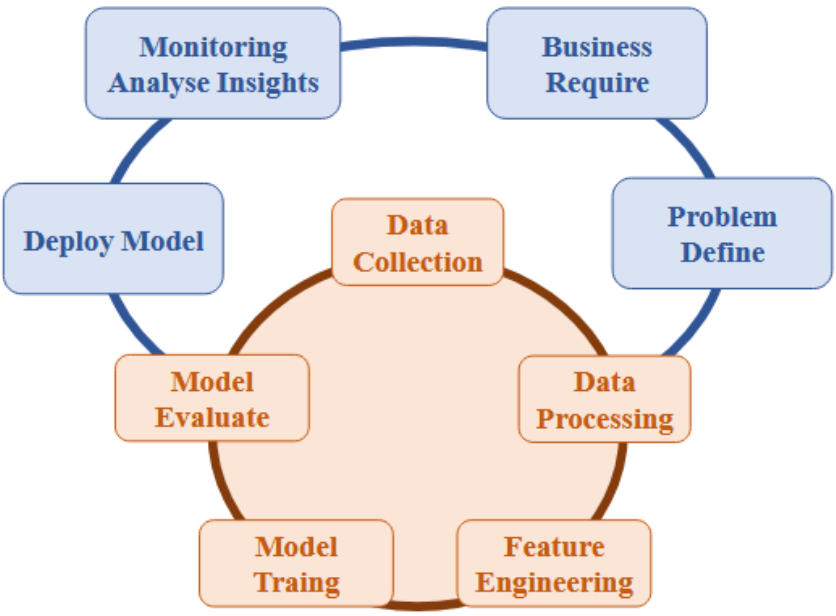


Figure 2: Vòng đời Machine Learning (ML Life Cycle)

1.2.2 Góc nhìn MLOps

MLOps tập trung vào việc tự động hóa và tinh gọn quy trình (pipeline) để chuyển giao mô hình từ môi trường nghiên cứu (Research Environment) sang môi trường triển khai (AI Service). Quy trình này bao gồm các bước lặp đi lặp lại như Xử lý dữ liệu (Data Handling), Huấn luyện mô hình (Model Training), và Đánh giá mô hình (Model Evaluation).

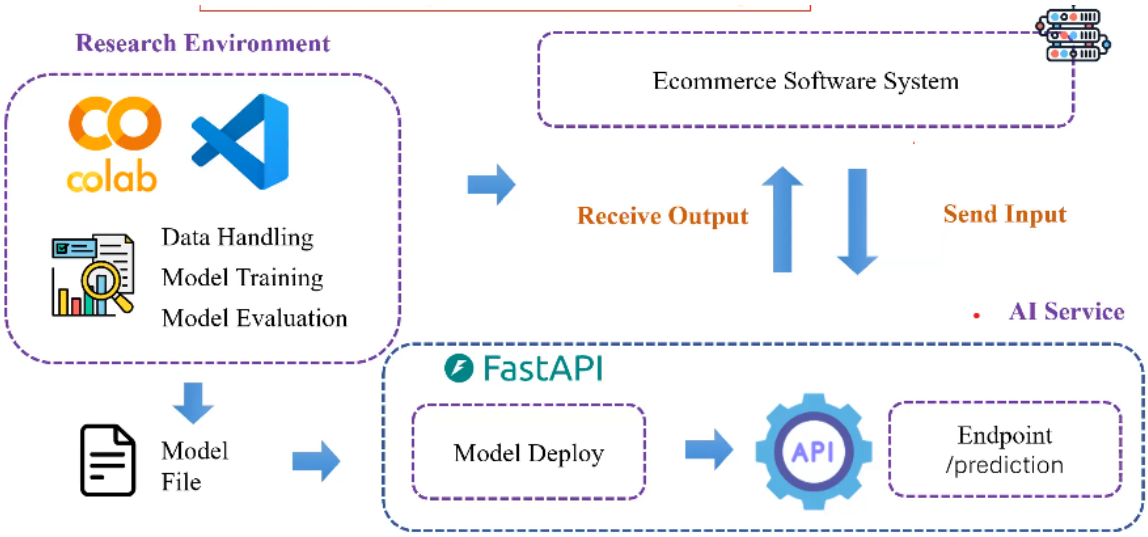


Figure 3: Quy trình MLOps cơ bản

Data versioning là thành phần quan trọng trả lời câu hỏi "Ở đâu và tại sao chúng ta cần phiên bản hóa dữ liệu ?". Nó giúp tự động hóa việc quản lý các "đầu ra" (artifacts) của mỗi bước (như dữ liệu đã xử lý, file mô hình, file chỉ số) và đảm bảo tính nhất quán giữa các môi trường, đặc biệt là khi làm việc nhóm.

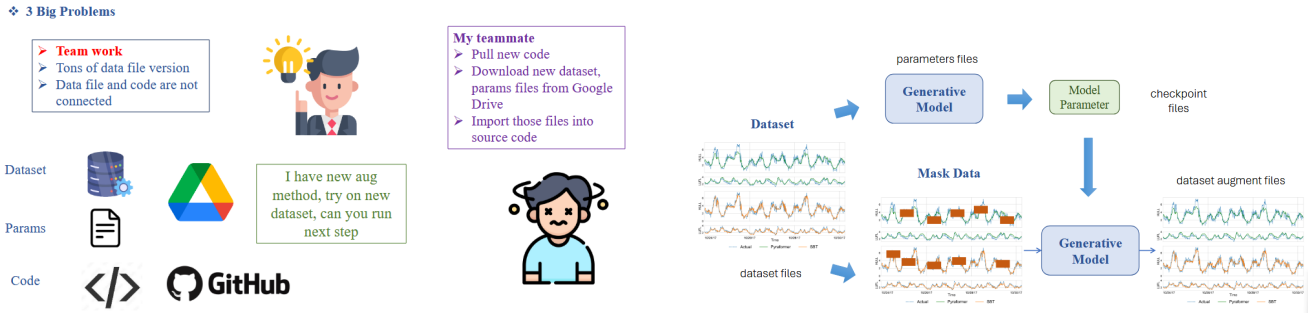
Phần 2: Thách thức trong Quản lý Dữ liệu và Code

Khi thực nghiệm trên 1 tập dữ liệu lớn, các vấn đề như các thành viên trong nhóm "mất kết nối / ngừng hiểu" với dữ liệu, code, tham số và phiên bản mô hình là điều dễ thấy trong 1 nhóm. Nhất là đối với dữ liệu dạng TimeSeries vừa lớn mà vừa phải cập nhật liên tục. Để nhìn rõ hơn về vấn đề này, mình sẽ đi qua 2 ví dụ khi CÓ và KHÔNG CÓ data versioning pipeline.

2.1 Ví dụ thực tế: Thử nghiệm mô hình Time Series

2.1.1 Khi KHÔNG có Data Versioning (Cách làm thủ công)

Giả sử nhóm của bạn đang phát triển một mô hình dự báo chuỗi thời gian (time-series) phức tạp. Quy trình này tạo ra vô số file: ‘dataset files‘ (dữ liệu gốc), ‘parameters files‘ (tham số), ‘checkpoint files‘ (điểm kiểm tra mô hình), và ‘dataset augment files‘ (dữ liệu tăng cường).



(a) Vấn đề khi làm việc nhóm thủ công

(b) Pipeline sinh ra nhiều phiên bản file

Figure 4: Sự hỗn loạn khi quản lý thủ công các file dữ liệu và thử nghiệm

Khi một thành viên trong nhóm ("My teammate") muốn chạy thử nghiệm mới, quy trình của họ rất thủ công và làm chậm quá trình phát triển:

1. **Pull new code**: Tải code mới nhất từ GitHub.
2. **Download new dataset/params**: Vì Github không cho lưu file lớn hơn 100mb, hóm truy cập Google Drive, tìm đúng file dữ liệu, file tham số mới nhất và tải về thủ công.
3. **Import files**: Sao chép các file này vào đúng thư mục trong source code.

Điều này dẫn đến 3 vấn đề lớn: (1) Có quá nhiều phiên bản file dữ liệu, tham số của các mô hình khác nhau. (2) Code và dữ liệu của mỗi người không được đồng bộ, và (3) Gây cản trở khi làm việc nhóm. Mọi người sẽ bối rối tự hỏi "file `cnn_ettm1_mix1_mask025.pkl` này được huấn luyện từ dữ liệu, mô hình, thời điểm nào ?".

$\longrightarrow$ Cần 1 quy trình (pipeline) thống nhất để kiểm soát các phiên bản mô hình, dữ liệu và code để cả team có thể hoạt động 1 cách đồng bộ mà không phải thông báo, gọi Google Meet để trao đổi nhiều chỉ để xác nhận những vấn đề nhỏ nhặt có tính lặp lại được nêu trên.

2.1.2 Khi CÓ Data Versioning (với DVC)

Khi sử dụng một công cụ như DVC, quy trình trên được đơn giản hóa triệt để.

1. **git pull**: Thành viên nhóm tải code mới. Lần này, code mới bao gồm các file `.dvc` (siêu dữ liệu) nhỏ nhẹ trỏ đến dữ liệu.
2. **dvc pull**: Họ chỉ cần chạy lệnh này. DVC sẽ đọc các file `.dvc`, tự động tìm và tải về đúng phiên bản dữ liệu, tham số, và mô hình tương ứng với phiên bản code đó từ bộ lưu trữ (ví dụ: S3, Google Drive, hoặc SSH).

Bằng cách này, DVC giải quyết vấn đề "disconnected" bằng cách liên kết code và dữ liệu một cách chặt chẽ thông qua các `commit`. Mọi thử nghiệm đều được ghi lại, nhất quán và có thể **tái lập (reproducible)** một cách chính xác bất cứ lúc nào.

Phần 3: Giới thiệu về Data Version Control (DVC)

DVC (Data Version Control) là một công cụ mã nguồn mở được thiết kế để quản lý dữ liệu và các dự án Machine Learning, hoạt động song song để bổ trợ cho Git [Ite25].

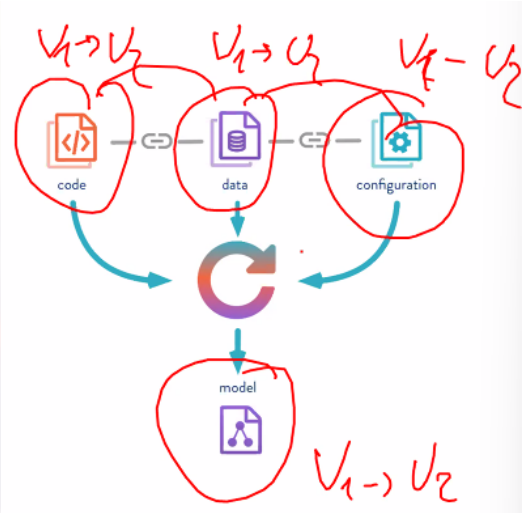


Figure 5: DvC đồng bộ bộ hóa code, dữ liệu, cấu hình và mô hình cho mỗi phiên bản, mỗi khi có sự thay đổi sau Huấn Luyện

3.1 Workflow Quản lý Phiên bản Dữ liệu

Một workflow phiên bản hóa dữ liệu (Data Versioning) là một vòng lặp quản lý liên tục. Quy trình này bắt đầu từ **Model Training** (Huấn luyện Mô hình), sử dụng một bộ `Data Config` (cấu hình dữ liệu) và `Model (params)` (tham số mô hình) cụ thể. Toàn bộ tài sản này (dữ liệu, tham số, kết quả đánh giá) được hệ thống **Data & Model Management** theo dõi và lưu trữ.

Khi có dữ liệu mới (`New Data`) hoặc thay đổi cấu hình (`Diff Config`), quy trình **Continuos Training** (Huấn luyện Liên tục) sẽ được kích hoạt để tạo ra mô hình mới (`New Model`) và kết quả đánh giá mới (`New Eval result`). Phiên bản mới này lại tiếp tục được hệ thống quản lý, hoặc sẵn sàng cho việc **Model Deployment** (Triển khai).

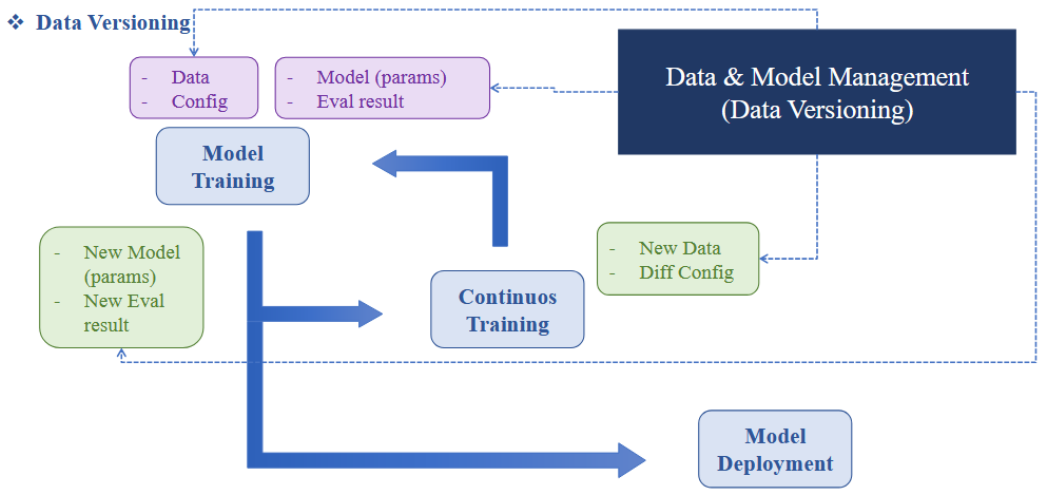


Figure 6: Workflow quản lý dữ liệu và mô hình trong MLOps

3.2 So sánh DVC và Giới thiệu chi tiết

Trên thị trường có nhiều công cụ để phiên bản hóa dữ liệu như Delta Lake, Pachyderm, và DVC. Tuy nhiên, DVC trở nên rất phổ biến vì các lý do chính:

- **Mã nguồn mở**: Sử dụng giấy phép Apache 2.0.
- **Độc lập với định dạng**: **Data Format Agnostic**, nghĩa là nó có thể quản lý bất kỳ loại file nào (model, .csv, .parquet, hình ảnh...).
- **Độc lập với lưu trữ**: **Cloud/Storage Agnostic**, hỗ trợ hầu hết các nền tảng lưu trữ phổ biến như S3, GCP, Azure, SSH.
- **Dễ sử dụng**: **Simple to Use**, vì có các lệnh tương tự Git.

	Open Source	Data Format Agnostic	Cloud/Storage Agnostic* (Supports most common cloud and storage types)	Simple to Use	Easy Support for BIG Data
DVC	✅ (Apache 2.0)	✅	✅	✅	❌
DELTA LAKE	✅ (Apache 2.0)	❌	✅	❌	✅
Git Large File Storage	✅ (MIT)	✅	❌	✅	❌
Pachyderm	✅⚠ (Non-standard license)	✅	✅	❌	✅
DOLT	✅ (Apache 2.0)	❌	❌	❌	❌
lakeFS	✅ (Apache 2.0)	✅	❌	❌	✅

DAGsHub

Figure 7: So sánh các công cụ Data Version Control phổ biến

Về cơ bản, DVC là một nền tảng khoa học dữ liệu cho phép bạn liên kết `code`, `data` (dữ liệu), và `configuration` (cấu hình) để tạo ra các `model` (mô hình) có thể tái lập (reproducible).

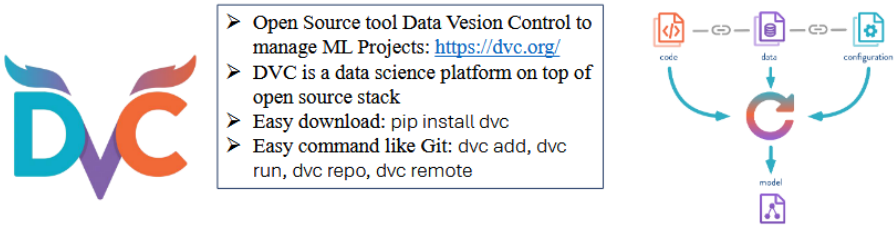


Figure 8: DVC liên kết Code, Data, và Configuration để quản lý Model

3.2.1 DVC khác Git như thế nào?

Điểm khác biệt cốt lõi là: **Git quản lý Code** (mã nguồn), trong khi **DVC quản lý Data** (dữ liệu).

Git không được thiết kế để xử lý các file lớn (ví dụ: mô hình 500MB). Khi bạn dùng DVC, quy trình làm việc (Local/Remote) sẽ được tách biệt rõ ràng:

- **Git (Code)**: Bạn dùng `git push/pull` để đồng bộ code (ví dụ: `train.py`) và các file `.dvc` siêu dữ liệu (chỉ nặng vài KB) lên máy chủ Git (như GitHub, GitLab).
- **DVC (Data)**: Bạn dùng `dvc push/pull` để đồng bộ các file dữ liệu lớn thực tế (ví dụ: `model.pkl` 500MB) lên một máy chủ lưu trữ (Storage) riêng biệt (như S3, Azure, Google Cloud, SSH).

file `model.pkl.dvc` (1KB) mà Git theo dõi chỉ là một "con trỏ" (pointer) trỏ đến file `model.pkl` (500MB) thực sự được DVC quản lý. Điều này giữ cho kho Git của bạn luôn nhỏ gọn.

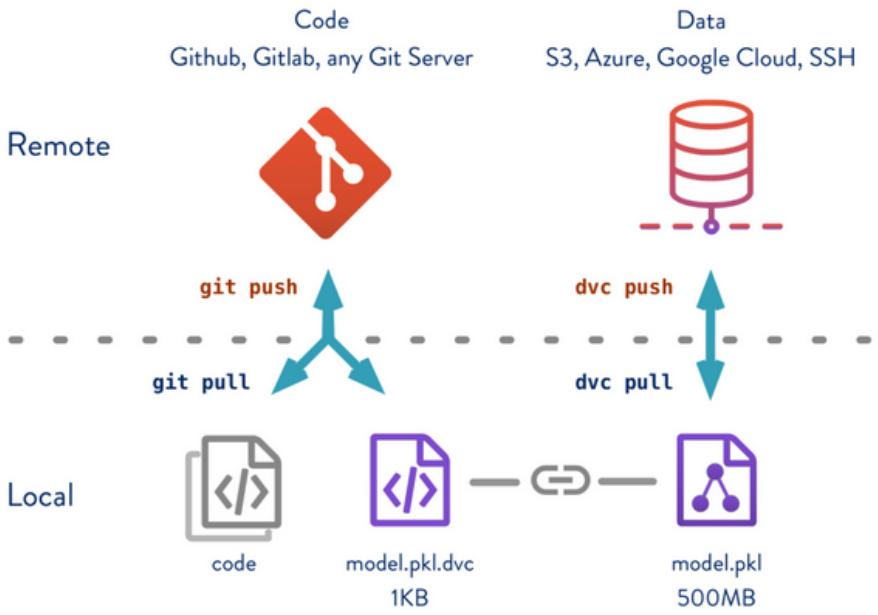


Figure 9: DVC và Git hoạt động song song: Git quản lý Code, DVC quản lý Data

DvC có độ tương đồng 1-1 so với Git về câu lệnh (ví dụ: `dvc add`, `dvc push`, `dvc pull`), giúp bất kì ai quen thuộc với version control của git hiểu ngay khi sử dụng.

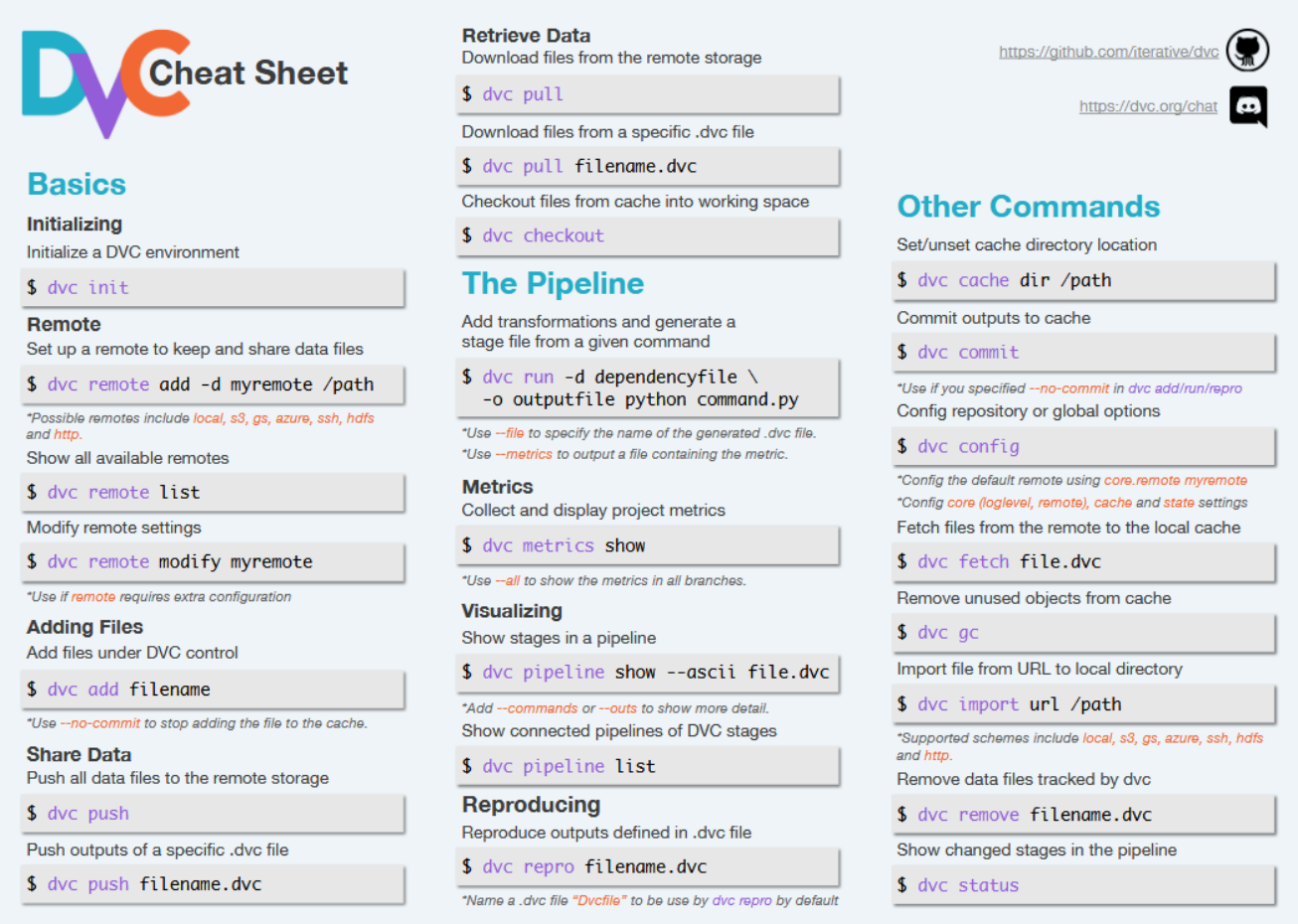


Figure 10: DVC Cheat Sheet cho thấy sự tương đồng với các lệnh Git

3.3 Quy trình DVC Pipeline hoàn chỉnh

Một DVC pipeline hoàn chỉnh cho phép bạn tự động hóa toàn bộ quy trình ML và đảm bảo tính tái lập (reproducibility). Bộ não hay còn gọi là cấu hình (config) của pipeline là file `dvc.yaml`.

3.3.1 Định nghĩa Pipeline (dvc.yaml)

file `dvc.yaml` là nơi bạn định nghĩa tất cả các **stages** (giai đoạn) của quy trình ML. Mỗi stage giống như một bước trong "công thức" của bạn. Một stage thường bao gồm:

- `cmd`: Lệnh thực thi (ví dụ: `python train.py`).
- `deps`: Các file phụ thuộc (dependencies) là đầu vào của stage, ví dụ như code (‘train.py‘) hoặc dữ liệu thô (‘data/raw.csv‘).
- `params`: Các tham số (parameters), thường được định nghĩa trong file `params.yaml` (ví dụ: learning rate, số epochs).
- `outs`: Các file đầu ra (outputs) mà stage này tạo ra, ví dụ như mô hình đã huấn luyện (`model.pkl`) hoặc dữ liệu đã xử lý.

3.3.2 Thực thi Pipeline (dvc repro)

Khi bạn chạy lệnh `dvc repro`, DVC sẽ thực hiện một việc rất thông minh: Nó kiểm tra file `dvc.yaml` và so sánh **hash** (một chuỗi định danh duy nhất) của các file `deps` và `params` hiện tại với thông tin được lưu trong file `dvc.lock`.

- file `dvc.lock` lưu lại "dấu vân tay" (hash) của các file `deps`, `params`, và `outs` từ lần chạy thành công trước đó.
- Nếu hash của bất kỳ file `deps` (ví dụ: bạn sửa code `train.py`) hoặc bất kỳ `params` nào thay đổi, DVC sẽ nhận ra stage đó là "lỗi thời" (outdated) và **chỉ chạy lại stage đó** cùng các stage phụ thuộc vào nó.
- Nếu không có gì thay đổi, DVC sẽ không chạy gì cả, giúp tiết kiệm thời gian tính toán.

Đây chính là cách DVC đảm bảo rằng mô hình của bạn luôn nhất quán với code và dữ liệu đã tạo ra nó.

3.4 Các khái niệm DVC cốt lõi cần nhớ

Đây là những ghi chú quan trọng và các khái niệm cần nhắc lại (từ các câu hỏi quiz buổi Thứ 5 Module 5 Week 1) trước khi đi vào thực hành.

3.4.1 Quy trình làm việc chuẩn: DVC trước, Git sau

Đây là quy tắc quan trọng nhất. Khi bạn có một file dữ liệu lớn (ví dụ: `data.zip`):

1. **'dvc add data.zip'**: Lệnh này bảo DVC bắt đầu theo dõi `data.zip`. DVC sao chép file này vào kho lưu trữ cục bộ (`.dvc/cache`) và tạo ra một file siêu dữ liệu (pointer) nhỏ tên là `data.zip.dvc`.
2. **'git add data.zip.dvc'**: Bạn dùng Git để theo dõi file pointer `.dvc` (chỉ vài KB), **KHÔNG BAO GIỜ** 'git add' file `data.zip` gốc.
3. **'git commit -m "Track new data"'**: Lưu lại trạng thái của file pointer.

Nếu bạn 'git add' file lớn trước, kho Git của bạn sẽ bị phình to và DVC sẽ không thể quản lý file đó.

3.4.2 git push vs. dvc push

Đây là điểm khác biệt cốt lõi (như trong hình `git_v_dvc.png`):

- **'git push'**: Chỉ đẩy code ('.py') và các file siêu dữ liệu/con trỏ ('.dvc', 'dvc.yaml') lên máy chủ Git (như GitHub).
- **'dvc push'**: Đẩy các file dữ liệu lớn thực tế (mà DVC đang theo dõi) từ cache cục bộ (`.dvc/cache`) lên kho lưu trữ từ xa (Remote Storage) như AWS S3, GCS, hoặc SSH.

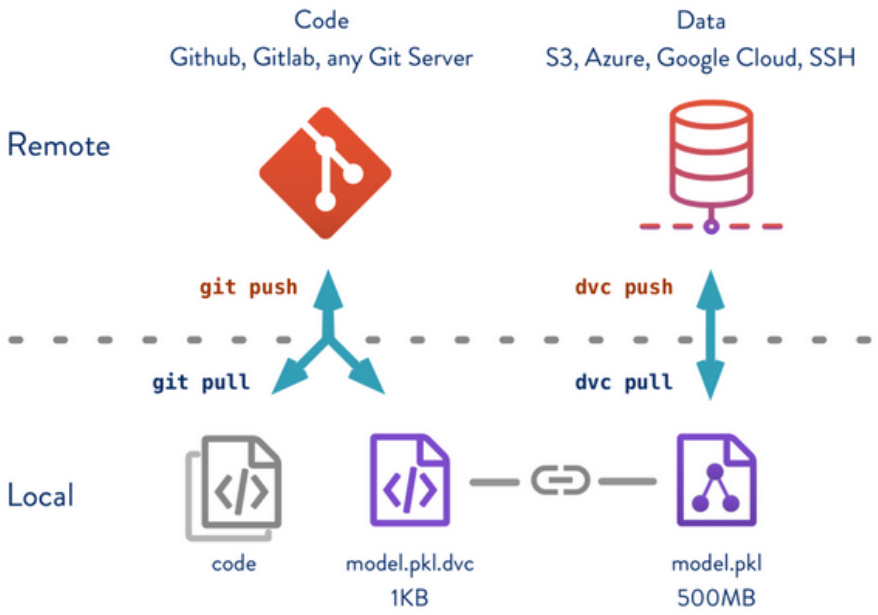


Figure 11: Git quản lý Code/Metadata, DVC quản lý Data thực tế

3.4.3 dvc pull vs. dvc checkout

Khi làm việc nhóm, bạn sẽ dùng hai lệnh này:

- **'dvc pull'**: Tải dữ liệu từ Remote Storage (ví dụ: S3) về kho lưu trữ cục bộ (Local Cache) của bạn (thư mục `.dvc/cache`).
- **'dvc checkout'**: Đồng bộ hóa dữ liệu từ Local Cache ra thư mục làm việc (workspace) của bạn. Lệnh này đọc các file `.dvc` trong workspace và tạo liên kết (symlink) đến các file tương ứng trong cache. Bạn thường chạy lệnh này sau khi `git checkout` một branch mới.

3.4.4 Điều kiện tiên quyết cho DVC Pipeline

Trước khi có thể chạy `dvc repro`, bạn cần đảm bảo:

1. Dự án đã được khởi tạo là một kho Git (`git init`).
2. Đã cài đặt các thư viện cần thiết (ví dụ: `pip install dvc dvc-s3`).
3. Đã định nghĩa các stage trong file `dvc.yaml`.
4. (Tùy chọn) Đã kết nối với kho lưu trữ AWS từ xa bằng lệnh `dvc remote add`, ví dụ: `dvc remote add -d my-s3-storage s3://my-bucket/dvc-storage`. Hướng dẫn setup AWS của TA trên Notion

Phần 4: Case Study: Triển khai DVC cho Dataset MNIST

Chúng ta sẽ thực hiện một case study từng bước để thấy DVC và Git hoạt động song song như thế nào để quản lý các phiên bản thử nghiệm (dữ liệu và mô hình).

4.1 Bước 1: Thiết lập Dự án và Git

Đầu tiên, chúng ta tạo thư mục dự án, cấu trúc thư mục con, môi trường `conda`, và quan trọng nhất là khởi tạo kho Git.

```
$ # 1. Create project directory and basic structure
$ mkdir dvc-mnist-demo
$ cd dvc-mnist-demo
$ mkdir data/raw models scripts

$ # 2. Create conda environment and install libraries
$ conda create -n dvc_mnist python=3.11
$ conda activate dvc_mnist
$ pip install -r requirements.txt

$ # 3. Initialize Git and make the first commit
$ git init
$ git add .
$ git commit -m "Init project"
```

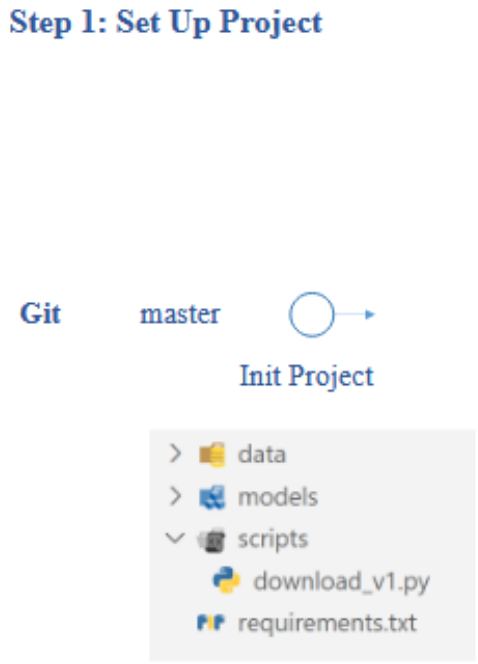


Figure 12: Trạng thái Git và cấu trúc file ban đầu

4.2 Bước 2: Tải Dataset V1 và Theo dõi bằng DVC

Tiếp theo, chúng ta tải phiên bản đầu tiên của dataset (Full MNIST) và dùng DVC để bắt đầu theo dõi nó.

```
$ # 1. Download data (60000 samples)
$ python scripts/download\_v1.py

$ # 2. Initialize DVC in the project
$ dvc init

$ # 3. Ask DVC to track the large data files
```

```
$ dvc add data/raw/x_train\_v1.npy data/raw/y_train\_v1.npy data/raw/x_test.npy data/
    raw/y_test.npy
```

Lệnh `dvc add` sẽ tạo ra các tệp `.dvc` (con trỏ) siêu dữ liệu và tự động thêm các tệp `.npy` lớn vào `.gitignore`.

4.3 Bước 3: Commit Phiên bản Dữ liệu V1

Giờ chúng ta commit các tệp `.dvc` (con trỏ) vào Git để lưu lại "phiên bản" dữ liệu này.

```
$ # Add .dvc files (pointers) and the .gitignore file updated by DVC
$ git add data/raw/.gitignore data/raw/x_train\_v1.npy.dvc data/raw/y_train\_v1.npy.
    dvc data/raw/x_test.npy.dvc data/raw/y_test.npy.dvc

$ # Commit to finalize Version V1
$ git commit -m "Version 1: Full MNIST dataset"
```

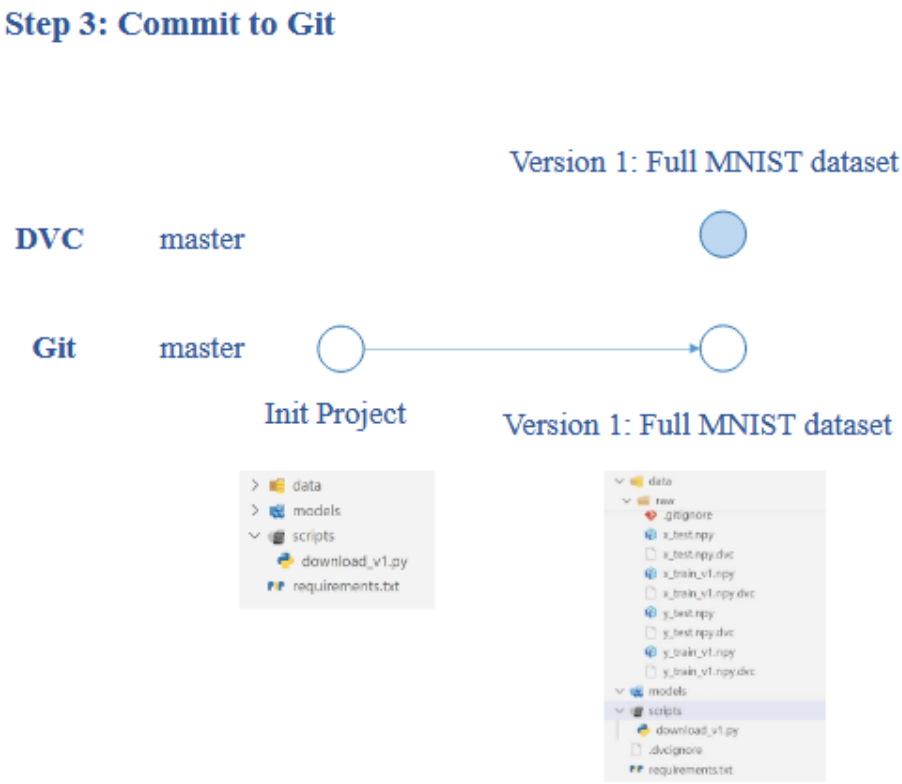


Figure 13: Lịch sử Git và DVC song song sau khi commit dữ liệu V1

4.4 Bước 4: Huấn luyện và Theo dõi Mô hình V1

Với dữ liệu V1, chúng ta huấn luyện mô hình đầu tiên và lại dùng DVC để theo dõi các tệp mô hình và chỉ số (metrics) đầu ra.

```
$ # 1. (Optional) Create symbolic link for train.py script to read data
$ cd data/raw
$ mklink x_train.npy x_train\_v1.npy
$ mklink y_train.npy y_train\_v1.npy
$ cd ../..

$ # 2. Run training (on 60000 samples, achieved 0.9319 accuracy)
```

```
$ python scripts/train.py

$ # 3. Ask DVC to track output files (model and metrics)
$ dvc add models/model.npy
$ dvc add models/metrics.json

$ # 4. Commit V1 model's .dvc pointers to Git
$ git add .
$ git commit -m "Model v1: Full dataset accuracy"
```

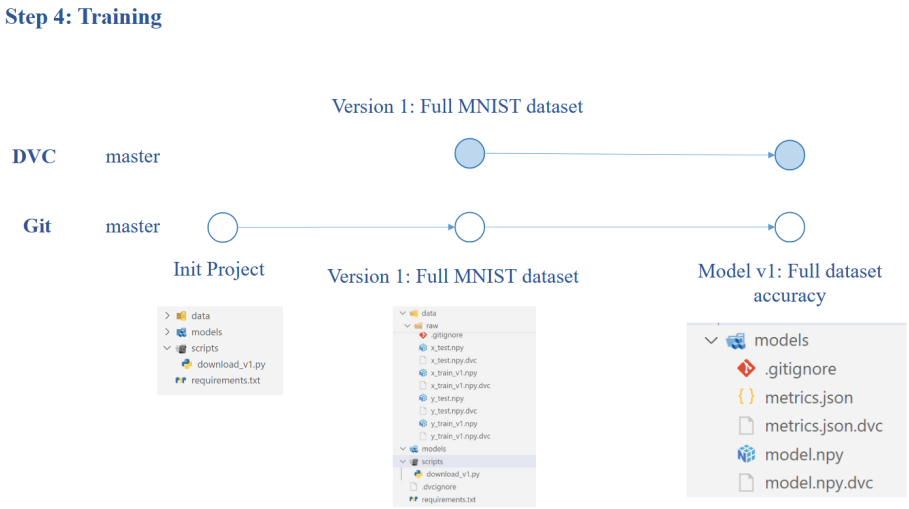


Figure 14: Toàn bộ lịch sử Git và DVC sau khi huấn luyện và commit Mô hình V1

4.5 Bước 5: Tạo Phiên bản Dữ liệu V2

Bây giờ, chúng ta giả lập một thử nghiệm mới bằng cách tạo phiên bản dữ liệu thứ hai (chỉ 1000 mẫu).

```
$ # 1. Run download\_v2.py script to create V2 data
$ python scripts/download\_v2.py

$ # 2. Ask DVC to track the new V2 data files
$ dvc add data/raw/x_train\_v2.npy
$ dvc add data/raw/y_train\_v2.npy

$ # 3. Commit V2's .dvc pointers to Git
$ git add .
$ git commit -m "Dataset V2"
```

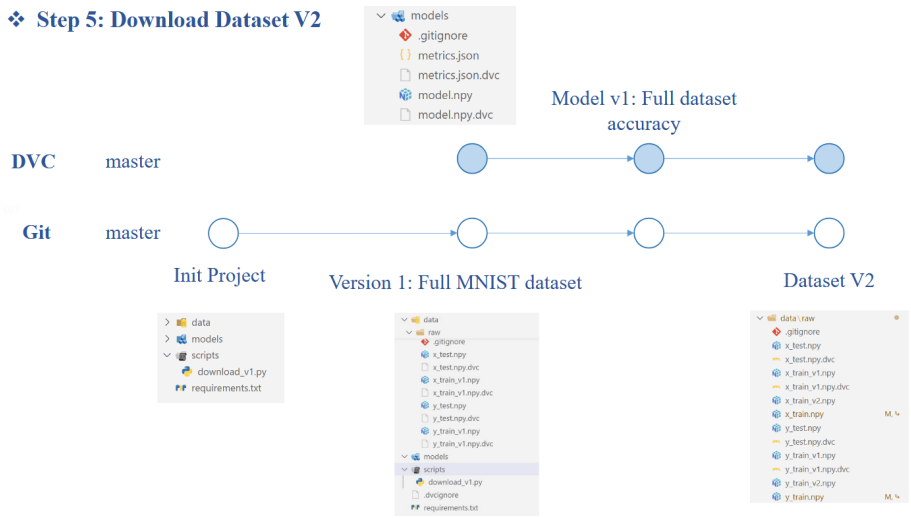


Figure 15: Lịch sử Git và DVC sau khi thêm "Dataset V2"

4.6 Bước 6: Huấn luyện Mô hình V2

Chúng ta lặp lại quy trình huấn luyện, nhưng lần này sử dụng dữ liệu V2 (bằng cách thay đổi symbolic link) để tạo ra mô hình V2.

```
$ # 1. Update symbolic link to point to V2 data
$ cd data/raw
$ del x_train.npy y_train.npy
$ mklink x_train.npy x_train\_v2.npy
$ mklink y_train.npy y_train\_v2.npy
$ cd ../..

$ # 2. Run training (on 1000 samples, achieved 0.8047 accuracy)
$ python scripts/train.py
```

4.7 Bước 7: Theo dõi Mô hình V2

Giờ chúng ta theo dõi các tệp mô hình và chỉ số V2 mới.

```
$ # 1. Ask DVC to track V2 model and metrics files
$ dvc add models/model.npy
$ dvc add models/metrics.json

$ # 2. Commit V2 model's .dvc pointers
$ git add .
$ git commit -m "Model␣v2:␣Small␣dataset␣accuracy"
```

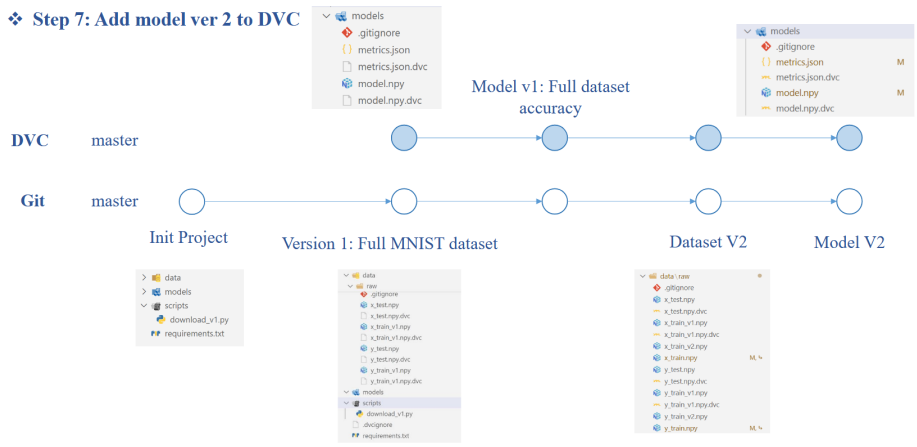


Figure 16: Lịch sử Git/DVC hoàn chỉnh với 2 phiên bản dữ liệu và 2 phiên bản mô hình

4.8 Bước 8: Cấu hình Kho lưu trữ (Storage)

Các tệp dữ liệu lớn thực sự nằm trong cache (`.dvc/cache`). Để chia sẻ chúng, ta cần thiết lập một kho lưu trữ từ xa (remote storage).

4.8.1 Sử dụng Local Storage (Lưu trữ Cục bộ)

Đây là cách đơn giản để chia sẻ cache trong cùng một máy hoặc qua mạng nội bộ.

```
$ # 1. Create a directory outside the project to act as "storage"
$ mkdir dvc_storage
$ # 2. Add it as the "localremote" (save configuration to .dvc/config)
$ dvc remote add -d localremote ./dvc_storage
$ # 3. Push data from cache (.dvc/cache) to localremote
$ dvc push
```

4.8.2 Sử dụng Cloud Storage (ví dụ: AWS S3)

Đây là cách làm phổ biến nhất khi làm việc nhóm.

```
$ # 1. Install the S3 support library
$ pip install dvc-s3
$ # 2. Add the S3 bucket as the "mys3" remote
$ dvc remote add -d mys3 s3://dvc-mnist-demo-bucket/data
$ # 3. Push data from cache to S3 (the default remote)
$ dvc push
```

4.9 Bước 9: Kiểm tra Chuyển đổi Phiên bản

Đây là sức mạnh lớn nhất của DVC. Chúng ta có thể quay lại bất kỳ thử nghiệm nào trong quá khứ. Ví dụ, để quay lại commit của Mô hình V1:

```
$ # 1. Find the commit_id of V1 (e.g., "Model v1: Full dataset accuracy")
$ git log

$ # 2. Revert to the code state of that commit
$ git checkout <commit_id_cua_V1>

$ # 3. Ask DVC to sync the corresponding data/model for that commit
$ dvc checkout
```

```
$ # 4. Check (will see V1 shape is 60000)
$ python -c "import numpy as np; print('V1 Data shape:', np.load('data/raw/x_train.npy
    ').shape)"
```

Nếu bạn `git checkout master` và `dvc checkout` một lần nữa, dữ liệu sẽ quay về V2 (1000 mẫu).

4.10 Bước 10: Push và Clone (Làm việc nhóm)

Quy trình làm việc nhóm điển hình:

```
$ # --- Person A (Pushing the project) ---
$ # 1. Push code and .dvc pointers to GitHub
$ git remote add origin <your-repository>
$ git push origin master
$ # 2. Push the actual data to Cloud Storage
$ dvc push

$ # --- Person B (Cloning the project) ---
$ # 1. Clone code and .dvc pointers from GitHub
$ git clone <your-repository>
$ # 2. Connect to Cloud Storage
$ dvc remote add -d mys3 s3://dvc-mnist-demo-bucket/data
$ # 3. Pull the actual data from Cloud Storage to cache
$ dvc pull
```

4.11 Bước 11: Tổng kết Thao tác DVC cơ bản

Workflow trên minh họa các lệnh DVC cơ bản bạn sẽ sử dụng hàng ngày:

Figure 17: Các lệnh DVC workflow cơ bản

```
$ # Ask DVC to track a file. Note: Track data file
$ dvc add data/raw/file.npy

$ # Sync data from cache to working directory (when checking out Git). Note: Switch
    data versions
$ dvc checkout

$ # Push data (large files) from cache to remote storage (S3, GCS...). Note: Send to
    remote
$ dvc push
```

```
$ # Pull data (large files) from remote storage to cache. Note: Get from remote
$ dvc pull

$ # Check status of data files compared to Git commit. Note: Check changes
$ dvc status

$ # List files currently tracked by DVC in the project
$ dvc list .
```

Phần 5: Tự động hóa Pipelines và Các khái niệm Versioning

Một trong những tính năng mạnh mẽ nhất của DVC là khả năng tự động hóa toàn bộ quy trình Machine Learning (ML pipeline) thông qua các tệp cấu hình (Configure File).

Tệp cấu hình chính là `dvc.yaml`. Bạn có thể coi nó như một "bản công thức" hay một "bản kế hoạch chi tiết" cho dự án của bạn. Nó là một tệp văn bản đơn giản mà con người có thể đọc được, thường nằm ở thư mục gốc (root) của dự án. Tệp này định nghĩa tất cả các **stages** (giai đoạn) trong quy trình của bạn, ví dụ: "bước 1: tải dữ liệu", "bước 2: xử lý dữ liệu", "bước 3: huấn luyện mô hình".

Mục đích của nó là giải quyết vấn đề "phải chạy thủ công" (manual work) và đảm bảo **khả năng tái lập (reproducibility)**. Thay vì phải nhớ chạy 5 tệp Python theo đúng thứ tự, bạn chỉ cần định nghĩa chúng một lần trong `dvc.yaml`. Sau đó, DVC sẽ tự động biết phải chạy gì, theo thứ tự nào, và quan trọng nhất: chỉ chạy lại những bước bị ảnh hưởng khi code hoặc dữ liệu của bạn thay đổi, giúp tiết kiệm rất nhiều thời gian.

5.1 DVC Automation Pipeline (Quy trình Tự động hóa DVC)

Hãy xem cách tệp `dvc.yaml` định nghĩa một pipeline.

Configure File: dvc.yaml

```yaml
stages:
  download_v1:
    cmd: python scripts/download_v1.py
    deps:
      - scripts/download_v1.py
    outs:
      - data/raw/x_train_v1.npy
      - data/raw/y_train_v1.npy
      - data/raw/x_test.npy
      - data/raw/y_test.npy

  download_v2:
    cmd: python scripts/download_v2.py
    deps:
      - scripts/download_v2.py
      - data/raw/x_train_v1.npy
      - data/raw/y_train_v1.npy
    outs:
      - data/raw/x_train_v2.npy
      - data/raw/y_train_v2.npy
```

```yaml
train_v1:
  cmd: python scripts/train.py --version v1
  deps:
    - scripts/train.py
    - data/raw/x_train_v1.npy
    - data/raw/y_train_v1.npy
    - data/raw/x_test.npy
    - data/raw/y_test.npy
  outs:
    - models/model_v1.npy
  metrics:
    - models/metrics_v1.json:
        cache: false

train_v2:
  cmd: python scripts/train.py --version v2
  deps:
    - scripts/train.py
    - data/raw/x_train_v2.npy
    - data/raw/y_train_v2.npy
    - data/raw/x_test.npy
    - data/raw/y_test.npy
  outs:
    - models/model_v2.npy
  metrics:
    - models/metrics_v2.json:
        cache: false
```

Figure 18: Cấu hình các giai đoạn (stages) trong `dvc.yaml`

Trong tệp `dvc.yaml`, mọi thứ được tổ chức trong `stages`:

- **'stages:'**: Khai báo bắt đầu danh sách các giai đoạn.
- **'download_v1:'**: Đây là **tên** của một stage.
- **'cmd:'**: Lệnh (command) sẽ được thực thi khi chạy stage này (ví dụ: `python scripts/download_v1.py`).
- **'deps:'**: Các tệp phụ thuộc (dependencies). DVC sẽ theo dõi các tệp này. Nếu tệp `scripts/download_v1.py` thay đổi, DVC sẽ biết stage này "lỗi thời" và cần chạy lại.
- **'outs:'**: Các tệp đầu ra (outputs). Đây là kết quả của stage (ví dụ: `data/raw/x_train_v1.npy`). DVC sẽ tự động theo dõi (giống như `dvc add`) các tệp này.

Bạn có thể thấy các stage `train_v1` và `train_v2` phức tạp hơn, chúng phụ thuộc vào cả script (`train.py`) và dữ liệu đầu vào (ví dụ: `data/raw/x_train_v1.npy`). Điều này tạo ra một chuỗi liên kết, hay một **DAG (Directed Acyclic Graph)**, nơi DVC hiểu rằng phải chạy `download_v1` trước rồi mới được chạy `train_v1`.

5.1.1 Sử dụng params.yaml để Tối ưu Pipeline

Việc sao chép và dán các stage `train_v1` và `train_v2` (như trong hình trên) rất dễ gây lỗi. Một cách tốt hơn là sử dụng tệp `params.yaml` để lưu trữ các tham số (hyperparameters).

Configure File: params.yaml

```yaml
vars:
  - data_version: ["v1", "v2"]

stages:
  download_v1:
    cmd: python scripts/download_v1.py
    deps:
      - scripts/download_v1.py
    outs:
      - data/raw/x_train_v1.npy
      - data/raw/y_train_v1.npy
      - data/raw/x_test.npy
      - data/raw/y_test.npy

  download_v2:
    cmd: python scripts/download_v2.py
    deps:
      - scripts/download_v2.py
      - data/raw/x_train_v1.npy
      - data/raw/y_train_v1.npy
    outs:
      - data/raw/x_train_v2.npy
      - data/raw/y_train_v2.npy
```

```yaml
train_${data_version}:
  foreach: ${data_version}
  do:
    cmd: python scripts/train.py --version ${item}
    deps:
      - scripts/train.py
      - data/raw/x_train_${item}.npy
      - data/raw/y_train_${item}.npy
      - data/raw/x_test.npy
      - data/raw/y_test.npy
    params:
      - model
    outs:
      - models/model_${item}.npy
    metrics:
      - models/metrics_${item}.json:
          cache: false
```

Figure 19: Kết hợp `dvc.yaml` (phải) và `params.yaml` (trái)

Thay vì định nghĩa hai stage riêng biệt, chúng ta sử dụng "templating" (tạo mẫu):

1. **'params.yaml'**: Chúng ta định nghĩa một biến `data_version: ["v1", "v2"]`.
2. **'dvc.yaml'**: Chúng ta tạo một stage `train_$data_version` (tên stage động).
3. **'foreach:'**: DVC sẽ lặp qua từng mục (item) trong `data_version`.
4. **'$item'**: Biến giữ chỗ này sẽ được thay thế bằng "v1" và "v2" khi chạy.

Với cách này, bạn chỉ cần định nghĩa stage `train` một lần. Nếu sau này bạn muốn thêm "v3", bạn chỉ cần thêm "v3" vào tệp `params.yaml` mà không cần sửa `dvc.yaml`.

5.2 Ý tưởng của Versioning (Phiên bản hóa)

Khái niệm cốt lõi của DVC là mở rộng Git workflow (quy trình làm việc của Git) cho dữ liệu và các thử nghiệm (experiments). Cách hoạt động tương tự git, mỗi nhánh tương ứng với 1 hướng phát triển.

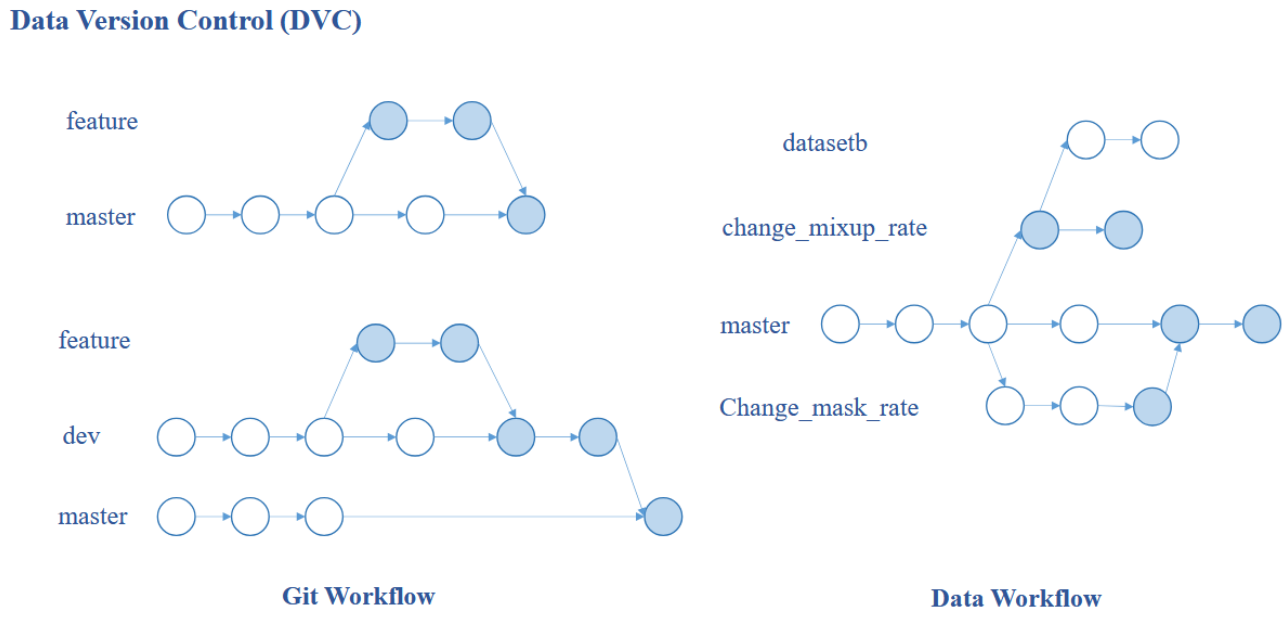


Figure 20: So sánh Git Workflow (trái) và Data Workflow (phải)

- **Git Workflow (Bên trái):** Trong phát triển phần mềm, khi muốn phát triển một tính năng mới (feature), bạn tạo một **nhánh (branch)** mới từ `master`, thêm code (các commit màu xanh), và khi hoàn thành, bạn gộp (merge) nó trở lại.
- **Data Workflow (Bên phải):** Chúng ta áp dụng ý tưởng tương tự cho các thử nghiệm ML.
  - Nhánh `master` là mô hình chính (production model) của bạn.
  - Khi bạn muốn thử một ý tưởng mới, ví dụ thay đổi tham số `mask_rate`, bạn tạo một nhánh Git mới tên là `Change_mask_rate`.
  - Trên nhánh này, bạn thay đổi tệp `params.yaml`, sau đó chạy `dvc repro`. DVC sẽ tạo ra mô hình và chỉ số mới, sau đó bạn `git commit` các thay đổi đó (ví dụ: `dvc.lock`).
  - Tương tự, bạn có thể tạo nhánh `change_mixup_rate` hoặc nhánh `datasetb` để thử nghiệm dữ liệu mới.

Bằng cách này, mỗi nhánh Git đại diện cho một thử nghiệm ML hoàn chỉnh và có thể tái lập. Bạn có thể dễ dàng chuyển đổi giữa các thử nghiệm (`git checkout`) và so sánh kết quả (`dvc metrics diff`) mà không làm lộn xộn nhánh `master` chính.

References

[Ite25] Iterative. *Data Version Control · DVC*. https://dvc.org/. Accessed: 2025-10-18. 2025.