Data Version Control (DVC) :-

◇ What is DVC?

Data Version Control (DVC) is an open-source version control system specifically built for machine learning projects. While Git is great for tracking code, it struggles with handling large files such as datasets, models, or intermediate data outputs. DVC bridges this gap by enabling version control of large files, ML pipelines, and experiment tracking, while being compatible with Git.

◇ Why Use DVC?

In a typical machine learning workflow, you deal with:

- Large datasets
- Trained models
- Preprocessing steps
- Feature engineering
- Model evaluation metrics

Git alone cannot efficiently handle large binary files (like .csv, .h5, .pkl), nor can it recreate the flow of how a model was trained. DVC solves these issues by:

- Keeping your Git repository lightweight (no large files)
- Allowing you to recreate the exact environment and data that led to a model
- Sharing ML projects easily with collaborators (code + data + models)

- Tracking metrics and experiments over time

◇ Key Features of DVC

Feature	Description
Data versioning	Tracks versions of datasets or models without storing them in Git
Experiment tracking	Easily compare results of different hyperparameters or code versions
Pipelines	Build and reproduce workflows (data preprocessing, training, evaluation)
Remote storage	Push and pull data from shared storage (Google Drive, S3, etc.)
Metrics/plots	Track model accuracy, loss, and other performance metrics visually

◇ DVC vs Git

Concept	Git	DVC
Purpose	Track source code	Track data, models, ML pipeline
File types	Text files, code	Large binary files (CSV, model files)
Storage	Git repo	Local cache + Remote (e.g., S3, GDrive)
Collaboration	Via Git repo	Git repo + DVC remote storage

◇ Installing DVC

You can install DVC using pip. If you are using a particular storage backend, install its extension:

```
pip install dvc

pip install "dvc[gdrive]"

pip install "dvc[s3]"
```

◇ Getting Started with DVC

Step 1: Initialize Git & DVC

```
git init
dvc init
git commit -m "Initialize DVC"
```

- dvc init creates a .dvc/ directory and adds a .dvcignore file.
- It sets up DVC configuration for tracking files.

Step 2: Add a Data File

```
dvc add data.csv
```

This will:

- Move the actual file into .dvc/cache/ (by content hash)
- Create a data.csv.dvc file (metadata file that tracks the dataset)
- Add an entry to .gitignore to prevent Git from tracking data.csv

You should now :

```
git add data.csv.dvc .gitignore
git commit -m "Add dataset"
```

Step 3: Setup Remote Storage

To store data in the cloud (e.g. Google Drive):

```
    dvc remote add -d myremote
gdrive://<your-folder-id>
```

To push your dataset to remote

```
    dvc push
```

This uploads the dataset to the remote storage.

To pull it later (on another system):

```
    dvc pull
```

◇ DVC Cache System

DVC uses a content-addressable cache stored in .dvc/cache/.
When you dvc add a file:

- DVC computes the MD5 hash of the file content.
- The file is stored in the cache using its hash as a path.
- The original file is replaced by a reference to the cache.

This avoids data duplication and allows efficient versioning.

◇ Understanding .dvc Files

A .dvc file (e.g.,data.csv.dvc) contains metadata:

```
    Outs:
    - md5: a1b2c3...
```

```
size: 1048576
path: data.csv
```

This file is tracked by Git and tells DVC how to retrieve the actual file.

◇ Pipelines and dvc.yaml

DVC pipelines let you define end-to-end workflows for ML.

```
dvc run -n train_model \
  -d train.py -d data.csv \
  -o model.pkl \
  python train.py
```

This will:

- Create dvc.yaml (pipeline definition)
- Create dvc.lock (input/output hash tracking)

Run pipelines with:

```
dvc repro
```

Only changed stages will be rerun.

◇ Track Metrics

DVC can track metrics like loss, accuracy, etc. in JSON/YAML files:

```
dvc metrics add metrics.json
```

Then you can view them across experiments:

```
dvc metrics show --all-branches
```

◇ Track Experiments

You can use dvc exp to try out multiple models:

```
dvc exp run
```

Each run is stored with a unique hash. Compare experiments:

```
dvc exp show
```

Apply a specific experiment:

```
dvc exp apply <exp_id>
```

◇ Checkpoints (for Deep Learning)

You can enable intermediate checkpointing (e.g. every epoch):

```
dvc exp run -checkpoint
```

This helps you recover or compare progress during long-running training.

◇ Cleaning Cache and Data

To remove unused cache files:

```
dvc gc -w  # Garbage collect workspace
```

◇ Sharing Your Project

1. **Push code to GitHub:**

   ```
   git push origin main
   ```

2. **Push data and models to remote:**

   ```
   dvc push
   ```

3. **Others clone and pull data:**

```
git clone <repo>
dvc pull
```

◇ Summary: Git + DVC Workflow

```
git init
dvc init

#Track dataset
dvc add data.csv
git add data.csv.dvc
dvc remote add -d gdrive gdrive://...
dvc push

#Run training pipeline
dvc run -n train -d train.py -d data.csv -o
model.pkl python train.py
dvc repro
dvc metrics show
dvc exp run
```

◇ Real Use Case Example

Imagine you're building a cancer prediction model:

1. Dataset: cancer.csv (100MB)
2. Code: train.py, evaluate.py
3. Model: model.pkl
4. Pipeline: preprocess → train → evaluate

Use DVC to:

- Track cancer.csv with dvc add

- Create pipeline stages with dvc run
- Version each trained model
- Share with team (push code + data)
- Reproduce results with dvc repro

Final Tips

- Always commit .dvc files and dvc.yaml to Git
- Never commit large data files directly to Git
- Use dvc status to check pipeline changes
- Use dvc diff to compare changes in data
- Use dvc plots to visualize model performance