Data Version Control with DVC

Complete MLOps Reference Guide

A Comprehensive Guide to DVC Commands,
Workflow, Integration with Git, and Best Practices

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to DVC

1.1 What is DVC?

DVC (Data Version Control) is an open-source version control system specifically designed for machine learning projects. It helps manage large datasets, model files, and ML pipelines alongside your code.

Key Definition

DVC extends Git capabilities to handle data versioning, making it possible to version control datasets, models, and ML experiments just like you version control code.

1.2 Why is DVC Needed in MLOps?

In traditional software development, Git is sufficient for version control. However, machine learning projects have unique challenges:

- **Large Datasets**: Data files are too large for Git (GB to TB range)
- **Binary Files**: Models, images, and data are binary, not text
- **Reproducibility**: Need to track which data produced which results
- **Experimentation**: Multiple experiments with different data versions
- **Collaboration**: Teams need consistent data across environments

1.3 The Machine Learning Pipeline

A typical ML pipeline consists of several interconnected components:

1. **Data Ingestion**: Collecting raw data from various sources
2. **Data Preprocessing**: Cleaning, transforming, and preparing data
3. **Feature Engineering**: Creating meaningful features from raw data
4. **Feature Selection**: Selecting the most relevant features
5. **Model Training**: Training ML models on prepared data
6. **Model Evaluation**: Assessing model performance

Important Note

Key Challenge: Each component has its own artifacts (outputs), and changes in any component cascade to all subsequent components. Managing these dependencies and tracking experiments becomes complex without proper versioning.

1.4 Real-World Scenario

Consider a real-world ML project scenario:

Experimentation Challenge

Scenario:

- You run 50 experiments with different data preprocessing techniques
- Each experiment produces different results
- Experiment #23 gives the best accuracy
- Two weeks later, you need to reproduce that exact result
- Problem: Which exact data, preprocessing steps, and parameters were used?

Solution: DVC tracks the exact version of data used with each code version, making experiments fully reproducible.

1.5 Why Not Just Use Git for Data?

Git has significant limitations for data versioning:

Git Limitations	**DVC Solutions**
Storage Issues: Git repos become huge with large files	DVC stores data externally (cloud or local storage)
Performance: Git slows down with binary files	DVC handles large files efficiently
Line-by-line tracking: Inefficient for binary data	DVC uses checksums for tracking
Repository size limits on platforms like GitHub	DVC uses separate storage backends

2 How DVC Works: The Temple Analogy

2.1 Understanding DVC with a Real-World Example

Let's understand DVC's working mechanism through an intuitive analogy:

The Temple Token System

The Scenario:
Imagine you visit a temple where you must deposit personal items before entering:

- Items to deposit: Phone, wallet, bag, shoes, etc.
- Two counters available:
  - **Counter 1**: Handles shoes only
  - **Counter 2**: Handles phone, wallet, bag (everything except shoes)

The Problem:
Without a system, items can get mismatched. Your shoes might be paired with someone else's phone and wallet when collecting items.
The Solution:
The temple implements a token system:

1. Deposit shoes at **Counter 1** → Receive **Token A**
2. Go to **Counter 2** with **Token A**
3. Deposit phone, wallet, bag **+** **Token A** at Counter 2 → Receive **Token B**
4. When returning:
   - Give **Token B** to Counter 2 → Get phone, wallet, bag + **Token A**
   - Give **Token A** to Counter 1 → Get your shoes back

Result: Perfect matching! Your shoes are always linked to your other belongings.

2.2 The DVC-Git Relationship

Now let's map this analogy to DVC and Git:

Temple Component	Technology	What It Stores
Counter 1 (Shoes)	DVC	Large data files
Counter 2 (Phone, Wallet)	Git	Code, configs, small files
Token A	.dvc file	Hash/pointer to data
Token B	Git commit	Code version + .dvc file

2.3 DVC Working Mechanism

1. **Store Data in DVC**:
   - Large datasets stored in DVC remote storage
   - DVC generates a unique hash (token) for this data
   - This hash acts as the "pointer" to your data
2. **Store Pointer in Git**:

- The .dvc file (containing the hash) is tracked by Git
- Git commits this .dvc file along with your code
- Small text file, perfect for Git versioning

3. **Retrieve Data**:
   - Git checkout gives you code + .dvc file (token)
   - DVC pull uses the token to fetch exact data version
   - Complete reproducibility achieved

2.4 What Gets Stored Where

Storage Distribution

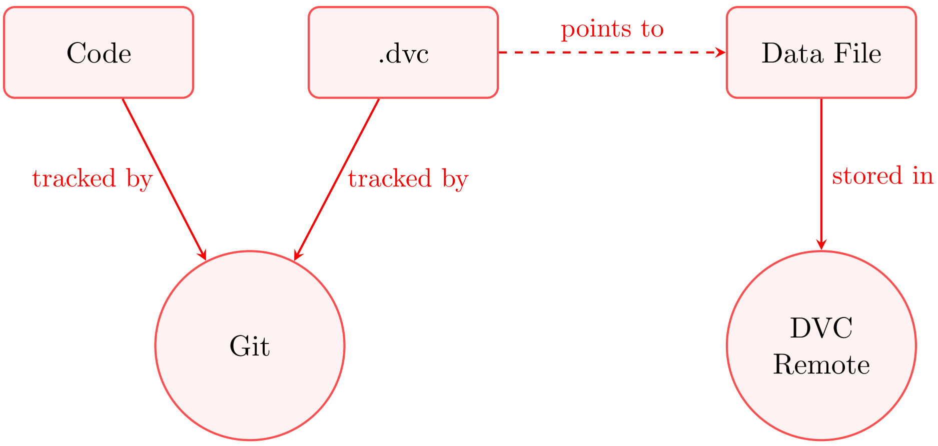
Git Stores:

- Source code (.py, .js, .java files)
- Configuration files (.yaml, .json)
- Documentation (.md, .txt)
- **.dvc files** (tokens/pointers)
- .gitignore and .dvcignore

DVC Stores:

- Large datasets (.csv, .parquet, images)
- Trained models (.pkl, .h5, .pth)
- Preprocessed data
- Feature files
- Evaluation metrics
- Any large binary files

2.5 Visual Representation



3 Setting Up DVC

3.1 Prerequisites

Before installing DVC, ensure you have:

- Python 3.8 or higher installed
- Git installed and configured
- pip (Python package manager)
- Basic understanding of command line/terminal

3.2 Installation

3.2.1 Installing DVC

```
pip install dvc
```

3.2.2 Verifying Installation

```
dvc version
```

Expected output:

```
DVC version: 3.x.x
```

3.3 Essential Terminal Commands

Before working with DVC, familiarize yourself with basic terminal commands:

Command	Description
`cd`	Change directory
`ls`	List files and directories (Linux/Mac)
`dir`	List files and directories (Windows)
`mkdir`	Create new directory
`pwd`	Print working directory (show current location)

3.4 Python Virtual Environment

It's recommended to use a virtual environment for DVC projects:

Setting Up Virtual Environment

```
# Create virtual environment
python -m venv venv

# Activate virtual environment
# On Windows:
venv\Scripts\activate.bat

# On Linux/Mac:
source venv/bin/activate
```

4 Complete DVC Workflow

4.1 Workflow Overview

The DVC workflow integrates seamlessly with Git. Here's the complete process:

1. Initialize Git repository
2. Initialize DVC
3. Configure remote storage
4. Track data with DVC
5. Version code and data pointers with Git
6. Push data to DVC remote
7. Collaborate and reproduce experiments

4.2 Step-by-Step Implementation

4.2.1 Step 1: Create and Clone Git Repository

```
# Create repository on GitHub/GitLab
# Clone to local machine
git clone https://github.com/username/project-name.git
cd project-name
```

4.2.2 Step 2: Create Python Script

Create a file named `DVC_Code.py`:

Version 1 - Initial Code

```python
import pandas as pd
import os

# Create a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Create directory for data
data_dir = 'DVC_data'
os.makedirs(data_dir, exist_ok=True)

# Define file path
file_path = os.path.join(data_dir, 'Sample_data.csv')

# Save DataFrame to CSV
df.to_csv(file_path, index=False)

```

```
print(f"CSV file saved to {file_path}")
```

4.2.3 Step 3: Configure .gitignore

Create/update `.gitignore` file:

```
# Add to .gitignore
DVC_data/
S3/
venv/
__pycache__/
*.pyc
```

Important Note

Adding `DVC_data/` to .gitignore prevents Git from tracking large data files. DVC will handle data versioning instead.

4.2.4 Step 4: Initial Git Commit

```
git add .
git commit -m "Initial Commit before initializing DVC"
git push origin main
```

4.2.5 Step 5: Initialize DVC

```
dvc init
```

This creates:

- `.dvc/` directory (DVC configuration and cache)
- `.dvcignore` file (similar to .gitignore for DVC)

What's Inside .dvc/?

The `.dvc/` directory contains:

- **cache/**: Stores all versions of tracked data locally
- **config**: DVC configuration settings
- **.gitignore**: Prevents cache from being tracked by Git

4.2.6 Step 6: Configure Remote Storage

For this tutorial, we'll use local storage. In production, you'd use cloud storage (S3, GCS, Azure).

```
# Create local storage directory
mkdir S3
```

```
# Add S3 to .gitignore
echo "S3/" >> .gitignore

# Configure DVC remote storage
dvc remote add -d dvc_origin S3
```

Command breakdown:

- `dvc remote add`: Add remote storage
- `-d`: Set as default remote
- `dvc_origin`: Name for this remote
- `S3`: Path to storage location

4.2.7 Step 7: Verify Remote Configuration

```
dvc remote list
```

Expected output:

```
dvc_origin    S3
```

4.2.8 Step 8: Track Data with DVC

```
dvc add DVC_Data/
```

Warning

On first run, if DVC_Data was previously tracked by Git, you'll see:

```
To stop tracking:
git rm -r --cached 'DVC_Data'
git commit -m "stop tracking DVC_Data"
```

Execute these commands, then run `dvc add DVC_Data/` again.

This creates `DVC_Data.dvc` file containing:

```
outs:
- md5: abc123def456...
  size: 1024
  path: DVC_Data
```

4.2.9 Step 9: Track .dvc File with Git

```
git add .gitignore DVC_Data.dvc
git commit -m "First Commit after initializing DVC"
```

4.2.10 Step 10: Check DVC Status

```
dvc status
```

If everything is tracked:

```
Data and pipelines are up to date.
```

4.2.11 Step 11: Push Data to DVC Remote

```
dvc push
```

This uploads:

- The actual data files to S3 (or configured remote)
- Creates hash-named files in remote storage

4.2.12 Step 12: Push Code to Git

```
git push origin main
```

Important Note

At this point, Version 1 of both code and data is saved:

- Git: Has code + .dvc file (pointer)
- DVC Remote: Has actual data
- Local .dvc/cache: Has data copy for fast access

5 Versioning Data Through Iterations

5.1 Creating Version 2

5.1.1 Update the Code

Modify `DVC_Code.py` to add a new row:

Version 2 - Adding One Row

```python
import pandas as pd
import os

# Create initial DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Create directory
data_dir = 'DVC_data'
os.makedirs(data_dir, exist_ok=True)

# Add new row for Version 2
new_row_loc = {'Name': 'GF1', 'Age': 20, 'City': 'City1'}
df.loc[len(df.index)] = new_row_loc

# Save to CSV
file_path = os.path.join(data_dir, 'Sample_data.csv')
df.to_csv(file_path, index=False)

print(f"CSV file saved to {file_path}")
```

5.1.2 Run the Script

```
python DVC_Code.py
```

5.1.3 Check DVC Status

```
dvc status
```

Output:

```
DVC_Data.dvc:
    changed outs:
        modified:  DVC_Data
```

This indicates data has changed since last commit.

5.1.4 Commit and Push Version 2

```
# Add changes to DVC
dvc add DVC_Data/

# Commit DVC changes
dvc commit

# Push data to remote
dvc push

# Track pointer file with Git
git add DVC_Data.dvc
git commit -m "Second Commit after initializing DVC"
git push origin main
```

5.2 Creating Version 3

5.2.1 Update Code Again

Version 3 - Adding Second Row

```python
import pandas as pd
import os

# Create initial DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Create directory
data_dir = 'DVC_data'
os.makedirs(data_dir, exist_ok=True)

# Add first new row (V2)
new_row_loc = {'Name': 'GF1', 'Age': 20, 'City': 'City1'}
df.loc[len(df.index)] = new_row_loc

# Add second new row (V3)
new_row_loc2 = {'Name': 'GF2', 'Age': 30, 'City': 'City2'}
df.loc[len(df.index)] = new_row_loc2

# Save to CSV
file_path = os.path.join(data_dir, 'Sample_data.csv')
df.to_csv(file_path, index=False)

print(f"CSV file saved to {file_path}")
```

5.2.2 Version and Commit

```
# Run script
python DVC_Code.py

# Check status
dvc status

# Version with DVC
dvc add DVC_Data/
dvc commit
dvc push

# Version with Git
git add DVC_Data.dvc DVC_Code.py
git commit -m "Third Commit after initializing DVC"
git push origin main
```

5.3 Understanding the Versioning Process

What Happens During Versioning?

When you run dvc add DVC_Data/ (first time):

- Starts tracking the data directory
- Calculates MD5 hash of data
- Copies data into `.dvc/cache/`
- Creates `DVC_Data.dvc` pointer file (tracked by Git)

When you modify the data later:

- Workspace data changes
- Cache still contains the previous version
- `dvc status` shows "modified"

When you run dvc add DVC_Data/ again (subsequent times):

- Recalculates hash of modified data
- Updates cache with new data version
- Updates `DVC_Data.dvc` with new hash
- No need for `dvc commit` — cache is already updated!

What Happens During Versioning?

When you run dvc push:

- Uploads data from cache to remote storage (S3/GCS/etc.)

- Only uploads changed/new files (efficient!)
- All versions are preserved in remote

Warning

Important: When is dvc commit actually needed?

In the workflow above, **dvc commit is NOT needed** because:

- `dvc add` automatically updates the cache
- Each `dvc add` creates a new cached version

What happens if you run both commands?

```
dvc add DVC_Data/      # Caches data
dvc commit             # Has no effect, redundant
```

- After `dvc add`, workspace = cache (already in sync)
- `dvc commit` checks: workspace matches cache? Yes!
- Result: Has no effect in this workflow and is redundant

dvc commit is only needed when:

- You manually edit tracked files (outside DVC commands)
- Example:

```
vim DVC_Data/Sample_data.csv  # Manual edit
dvc commit DVC_Data.dvc       # Finalize changes
```

- It finalizes changes to already tracked outputs
- It updates cache with manually modified data
- **Note:** This is less common; `dvc add` is preferred for most workflows

Key principle:

- `dvc add` = start tracking + cache updates
- `dvc commit` = finalize manual changes to tracked data
- `dvc add` + `dvc commit` = second command is redundant
- For normal versioning workflow: `dvc add` is sufficient

5.4 Corrected Versioning Workflow

5.4.1 Version 2 - Correct Commands

```
# Run script to modify data
python DVC_Code.py

# Check what changed
dvc status

# Add changes to DVC (caches automatically)
dvc add DVC_Data/

# Push to remote storage
dvc push

# Commit pointer file to Git
git add DVC_Data.dvc DVC_Code.py
git commit -m "Second Commit after initializing DVC"
git push origin main
```

Note: No `dvc commit` needed — `dvc add` already cached the data!

5.4.2 Version 3 - Correct Commands

```
# Run script to add more data
python DVC_Code.py

# Check status
dvc status

# Version with DVC (caches automatically)
dvc add DVC_Data/

# Push to remote
dvc push

# Version with Git
git add DVC_Data.dvc DVC_Code.py
git commit -m "Third Commit after initializing DVC"
git push origin main
```

5.5 Understanding Cache Behavior

How DVC Cache Works

Cache Structure:

- Located in `.dvc/cache/`
- Each file/directory version gets a unique hash
- Multiple versions coexist in cache

- Cache is content-addressable (hash-based)

Version History:

- Each `git commit` captures one `.dvc` file state
- Each `.dvc` file points to one cache version
- Git history = timeline of data versions
- `git checkout` + `dvc checkout` = restore any version

5.6 Viewing Version History

5.6.1 Check Git Log

```
git log --oneline
```

Example output:

```
32fe46b (HEAD -> main) Third Commit after initializing DVC
2a8dc29 Second Commit after initializing DVC
361d7ac First Commit after initializing DVC
ec2e1d7 Initial Commit before initializing DVC
776595f (origin/main) Initial commit
```

Each commit represents a specific version of code + data pointer.

6 Rolling Back to Previous Versions

6.1 The Need for Rollback

After running multiple experiments, you might need to:

- Reproduce results from an earlier experiment
- Compare current version with previous version
- Debug issues by reverting to working version
- Retrieve specific data configuration

6.2 Rollback Process

Rolling back involves two steps:

1. **Git checkout**: Retrieves code + .dvc file (data pointer)
2. **DVC pull**: Uses pointer to fetch actual data

6.3 Step-by-Step Rollback

6.3.1 Step 1: View Available Versions

```
git log --oneline
```

Output shows all commits with their hash IDs.

6.3.2 Step 2: Checkout Specific Version

To rollback to Version 1:

```
git checkout 361d7ac
```

Important Note

Detached HEAD State:
You'll see a message about "detached HEAD" state. This is normal when checking out a specific commit. You're viewing a specific point in history.

Result: Your code now matches Version 1:

```python
# DVC_Code.py - Version 1
import pandas as pd
import os

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)
data_dir = 'DVC_data'
os.makedirs(data_dir, exist_ok=True)
file_path = os.path.join(data_dir, 'Sample_data.csv')
```

```
df.to_csv(file_path, index=False)
print(f"CSV file saved to {file_path}")
```

6.3.3 Step 3: Check DVC Status

```
dvc status
```

Output:

```
DVC_Data.dvc:
    changed outs:
        modified:  DVC_Data
```

This indicates your local data doesn't match the version specified in the .dvc file.

6.3.4 Step 4: Pull Data from DVC

```
dvc pull
```

Output:

```
Collecting                                  |2.00 [00:00, 69.9entry/s]
Fetching
Building workspace index                    |3.00 [00:00,  255entry/s]
Comparing indexes                           |3.00 [00:00,  735entry/s]
Applying changes                            |1.00 [00:00,  176file/s]
M       DVC_Data\
1 file modified
```

6.3.5 Step 5: Verify Rollback

```
dvc status
```

Output:

```
Data and pipelines are up to date.
```

Check the data file:

Sample_data.csv - Version 1

```
Name,Age,City
Alice,25,New York
Bob,30,Los Angeles
Charlie,35,Chicago
```

6.4 Returning to Latest Version

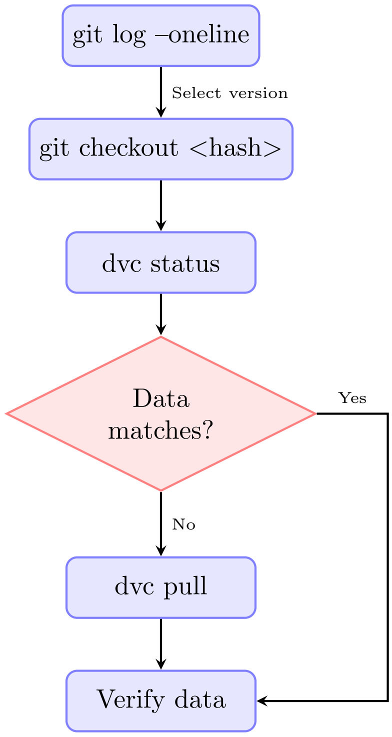
To return to the most recent version:

```
# Return to main branch
git checkout main

# Pull latest data
```

```
dvc pull
```

6.5 Rollback Workflow Diagram



7 Complete DVC Commands Reference

7.1 Initialization and Configuration

Command	Description
`dvc init`	Initialize DVC in current Git repository
`dvc version`	Display DVC version
`dvc remote add <name> <url>`	Add remote storage location
`dvc remote add -d <name> <url>`	Add and set as default remote
`dvc remote list`	List configured remote storage locations
`dvc remote remove <name>`	Remove a remote storage
`dvc config`	View/modify DVC configuration

7.2 Data Tracking

Command	Description
`dvc add <file/dir>`	Start tracking file or directory
`dvc remove <file.dvc>`	Stop tracking (remove .dvc file)
`dvc move <src> <dst>`	Move tracked file/directory
`dvc unprotect <file>`	Make tracked file writable

7.3 Status and Information

Command	Description
`dvc status`	Check status of tracked data
`dvc status -c`	Show status with cloud comparison
`dvc diff`	Show changes between commits
`dvc list <repo> <path>`	List repository contents
`dvc dag`	Visualize pipeline dependencies

7.4 Data Synchronization

Command	Description
`dvc push`	Upload tracked data to remote storage
`dvc push -r <remote>`	Push to specific remote
`dvc pull`	Download data from remote storage
`dvc pull -r <remote>`	Pull from specific remote
`dvc fetch`	Download data without modifying workspace
`dvc checkout`	Update workspace with tracked data
`dvc commit`	Save current state to cache

7.5 Pipeline Management

Command	Description
`dvc run`	Create a pipeline stage
`dvc repro`	Reproduce pipeline or specific stage
`dvc pipeline show`	Display pipeline structure
`dvc params`	Manage and show parameters
`dvc metrics`	Track and compare metrics

7.6 Other Useful Commands

Command	Description
`dvc gc`	Garbage collect unused cache files
`dvc cache dir`	Show cache directory location
`dvc import <url> <path>`	Download and track data from repository
`dvc get <url> <path>`	Download data without tracking
`dvc doctor`	Check DVC installation and setup

8 Remote Storage Configuration

8.1 Supported Storage Types

DVC supports various remote storage backends:

1. **Local Storage**: Local directory or network drive
2. **Amazon S3**: AWS cloud storage
3. **Google Cloud Storage (GCS)**: Google cloud storage
4. **Azure Blob Storage**: Microsoft cloud storage
5. **Google Drive**: Personal cloud storage
6. **SSH/SFTP**: Remote servers via SSH
7. **HDFS**: Hadoop Distributed File System
8. **HTTP/HTTPS**: Web servers

8.2 Local Storage Configuration

Local Storage Setup

```
# Create local storage directory
mkdir /path/to/storage

# Add as DVC remote
dvc remote add -d myremote /path/to/storage

# Verify
dvc remote list
```

8.3 Amazon S3 Configuration

AWS S3 Setup

```
# Add S3 bucket as remote
dvc remote add -d s3remote s3://mybucket/path

# Configure AWS credentials (Option 1: AWS CLI)
aws configure

# Or configure directly in DVC (Option 2)
dvc remote modify s3remote access_key_id 'mykey'
dvc remote modify s3remote secret_access_key 'mysecret'

# Optional: Set region
dvc remote modify s3remote region us-west-2
```

8.4 Google Cloud Storage Configuration

GCS Setup

```
# Add GCS bucket as remote
dvc remote add -d gcsremote gs://mybucket/path

# Authenticate with Google Cloud
gcloud auth application-default login

# Or set service account key
dvc remote modify gcsremote \
  credentialpath /path/to/key.json
```

8.5 Azure Blob Storage Configuration

Azure Setup

```
# Add Azure container as remote
dvc remote add -d azureremote \
  azure://mycontainer/path

# Configure connection string
dvc remote modify azureremote \
  connection_string 'myconnectionstring'

# Or use account name and key
dvc remote modify azureremote account_name 'myaccount'
dvc remote modify azureremote account_key 'mykey'
```

8.6 Google Drive Configuration

Google Drive Setup

```
# Add Google Drive as remote
dvc remote add -d gdriveremote \
  gdrive://folder_id

# First push will prompt for authentication
dvc push
```

8.7 SSH/SFTP Configuration

SSH Setup

```
# Add SSH remote
dvc remote add -d sshremote \
  ssh://user@example.com/path/to/storage

# Configure SSH key
dvc remote modify sshremote keyfile /path/to/key
```

```
# Or use password (not recommended)
dvc remote modify sshremote password 'mypassword'
```

8.8 Managing Multiple Remotes

You can configure multiple remotes and switch between them:

```
# Add multiple remotes
dvc remote add s3prod s3://prod-bucket/data
dvc remote add s3dev s3://dev-bucket/data
dvc remote add local /mnt/storage

# Set default
dvc remote default s3prod

# Push to specific remote
dvc push -r s3dev

# Pull from specific remote
dvc pull -r local
```

9 Collaboration with DVC

9.1 Team Workflow

DVC enables seamless team collaboration on ML projects:

1. **Shared Remote Storage**: All team members access same data versions
2. **Git for Coordination**: Code and .dvc files synced through Git
3. **Reproducible Experiments**: Everyone can reproduce exact results
4. **No Data Duplication**: Data shared efficiently through remotes

9.2 Collaborator Setup

9.2.1 Initial Setup for New Team Member

Onboarding Workflow

```
# 1. Clone the Git repository
git clone https://github.com/team/ml-project.git
cd ml-project

# 2. Install dependencies
pip install -r requirements.txt

# 3. DVC is already initialized (from repo)
# Check DVC remotes
dvc remote list

# 4. Configure credentials if needed
# (For S3, GCS, etc.)

# 5. Pull all data
dvc pull

# 6. Verify everything is ready
dvc status
# Should show: "Data and pipelines are up to date."
```

9.3 Making Changes and Sharing

9.3.1 Developer Workflow

Typical Development Cycle

```
# 1. Ensure you're up to date
git pull
dvc pull

# 2. Create feature branch
git checkout -b experiment-new-feature

# 3. Make changes to code and data
python train.py  # Generates new data
```

```

# 4. Track new/modified data
dvc add data/processed/
dvc add models/

# 5. Commit DVC changes
dvc commit
dvc push

# 6. Commit to Git
git add .
git commit -m "Experiment: new feature with improved accuracy"
git push origin experiment-new-feature

# 7. Create pull request for review
```

9.4 Reviewing Changes

Team members can review experiments:

```
# Checkout colleague's branch
git checkout experiment-new-feature

# Pull corresponding data
dvc pull

# Run and verify results
python evaluate.py

# Compare with main branch
git checkout main
dvc pull
python evaluate.py
```

9.5 Best Practices for Teams

1. **Naming Conventions**:
   - Use descriptive branch names: `exp-optimizer-adam`, `data-augmentation-v2`
   - Clear commit messages linking code and data changes
2. **Data Management**:
   - Don't track generated/intermediate files unnecessarily
   - Use `.dvcignore` for temporary files
   - Regularly clean cache: `dvc gc`
3. **Communication**:
   - Document experiments in commit messages
   - Maintain README with DVC setup instructions
   - Share remote storage credentials securely

4. **Access Control**:
    - Set appropriate permissions on remote storage
    - Use separate remotes for dev/staging/prod
    - Monitor storage usage and costs

10 Advanced DVC Features

10.1 DVC Pipelines

DVC pipelines automate ML workflows and track dependencies.

10.1.1 Creating a Pipeline

Simple Pipeline Example

```
# Stage 1: Data preprocessing
dvc run -n preprocess \
  -d data/raw \
  -o data/processed \
  python preprocess.py

# Stage 2: Feature extraction
dvc run -n features \
  -d data/processed \
  -o data/features \
  python extract_features.py

# Stage 3: Model training
dvc run -n train \
  -d data/features \
  -o models/model.pkl \
  -M metrics/train.json \
  python train.py

# Stage 4: Evaluation
dvc run -n evaluate \
  -d models/model.pkl \
  -d data/features \
  -M metrics/eval.json \
  python evaluate.py
```

10.1.2 Reproducing Pipelines

```
# Reproduce entire pipeline
dvc repro

# Reproduce specific stage
dvc repro train

# Force reproduce (ignore cache)
dvc repro -f
```

10.2 Parameters and Metrics

10.2.1 Tracking Parameters

Create params.yaml:

```
train:
```

```
  learning_rate: 0.001
  epochs: 100
  batch_size: 32

model:
  layers: [64, 32, 16]
  dropout: 0.2
```

Access in code:

```python
import yaml

with open('params.yaml', 'r') as f:
    params = yaml.safe_load(f)

lr = params['train']['learning_rate']
```

View parameters:

```
dvc params diff
```

10.2.2 Tracking Metrics

Save metrics in JSON format:

```python
import json

metrics = {
    'accuracy': 0.95,
    'loss': 0.12,
    'f1_score': 0.93
}

with open('metrics.json', 'w') as f:
    json.dump(metrics, f)
```

Compare metrics:

```
dvc metrics show
dvc metrics diff
```

10.3 Experiments

DVC experiments feature helps manage ML experiments.

```
# Run experiment
dvc exp run

# Run with modified parameters
dvc exp run --set-param train.lr=0.01

# List experiments
dvc exp show

# Compare experiments
dvc exp diff
```

```
# Apply best experiment
dvc exp apply <exp-name>
```

10.4 Data Registry

Share data across projects using DVC:

Data Registry Workflow

```
# In data repository
dvc add dataset.csv
git add dataset.csv.dvc .gitignore
git commit -m "Add dataset"
git push

# In ML project
dvc import https://github.com/team/data-repo \
  dataset.csv
```

10.5 Cache Management

10.5.1 Cache Location

```
# Show cache directory
dvc cache dir

# Change cache directory
dvc cache dir /new/path/to/cache
```

10.5.2 Cleaning Cache

```
# Remove unused cached files
dvc gc

# Remove all cache files not in workspace
dvc gc --workspace

# Remove cache for specific remote
dvc gc --cloud
```

11 Troubleshooting and Common Issues

11.1 Common Errors and Solutions

11.1.1 Error: "Failed to push data to remote"

Cause: Connection issues or incorrect remote configuration

Solution:

```
# Check remote configuration
dvc remote list
dvc config remote.myremote.url

# Test connection
dvc push -v  # Verbose output

# Verify credentials (for cloud storage)
# Check AWS/GCS/Azure credentials
```

11.1.2 Error: "Output is already tracked"

Cause: File is tracked by both Git and DVC

Solution:

```
# Remove from Git tracking
git rm --cached file.csv

# Add to .gitignore
echo "file.csv" >> .gitignore

# Track with DVC
dvc add file.csv

# Commit changes
git add file.csv.dvc .gitignore
git commit -m "Move tracking from Git to DVC"
```

11.1.3 Error: "Failed to pull data"

Cause: Data not available in remote or local cache

Solution:

```
# Check status
dvc status -c

# Fetch data from remote
dvc fetch

# Checkout fetched data
dvc checkout

# Or pull directly
```

```
dvc pull -v
```

11.1.4 Error: "Corrupted cache"

Cause: Cache files corrupted or modified
Solution:

```
# Clear corrupted cache
rm -rf .dvc/cache

# Pull fresh data
dvc pull
```

11.2 Performance Issues

11.2.1 Slow Push/Pull Operations

Solutions:

1. Use jobs parameter for parallel transfers:

   ```
   dvc push -j 4  # Use 4 parallel jobs
   ```

2. Configure cloud provider CLI for better performance

3. Use local cache effectively:

   ```
   dvc config cache.type hardlink,symlink
   ```

11.2.2 Large Cache Size

Solutions:

1. Remove unused files:

   ```
   dvc gc --workspace
   ```

2. Move cache to external drive:

   ```
   dvc cache dir /mnt/external/cache
   ```

3. Use shared cache for team:

   ```
   dvc config cache.dir /shared/cache
   dvc config cache.shared group
   ```

11.3 Debugging Tips

1. **Use Verbose Mode**:

   ```
   dvc push -v
   dvc pull -v
   ```

2. **Check System Health**:

```
dvc doctor
```

3. **Verify Configuration**:

```
dvc config --list
cat .dvc/config
```

4. **Check File Integrity**:

```
dvc status
dvc diff
```

12 DVC Best Practices

12.1 Project Organization

12.1.1 Recommended Directory Structure

```
ml-project/
|-- data/
|   |-- raw/              # Original, immutable data
|   |-- processed/        # Cleaned data
|   +-- features/         # Feature files
|-- models/               # Trained models
|-- notebooks/            # Jupyter notebooks
|-- src/                  # Source code
|   |-- data/             # Data processing scripts
|   |-- features/         # Feature engineering
|   |-- models/           # Model training
|   +-- visualization/    # Plotting scripts
|-- metrics/              # Evaluation metrics
|-- params.yaml           # Parameters
|-- dvc.yaml              # Pipeline definition
|-- .dvc/                 # DVC configuration
|-- .dvcignore            # DVC ignore file
|-- .gitignore            # Git ignore file
+-- README.md             # Project documentation
```

12.2 What to Track with DVC

12.2.1 DO Track

- Raw datasets
- Processed/cleaned data
- Feature files
- Trained model files (.pkl, .h5, .pth)
- Large binary files (images, videos, audio)
- Evaluation results
- Pre-trained embeddings

12.2.2 DON'T Track

- Temporary files
- Cache files
- Virtual environment folders
- IDE configuration files
- System files (.DS_Store, Thumbs.db)
- Logs (unless necessary for experiments)

12.3 .dvcignore Configuration

Sample .dvcignore

```
# Temporary files
*.tmp
*.temp
*~

# Logs
*.log
logs/

# OS files
.DS_Store
Thumbs.db

# Python
__pycache__/
*.pyc
*.pyo

# Jupyter
.ipynb_checkpoints/

# Virtual environments
venv/
env/
```

12.4 Commit Message Conventions

Use clear, descriptive commit messages that link code and data changes:

Good Commit Messages

```
[OK] "feat: Add data augmentation pipeline"
[OK] "exp: Test BERT model with lr=0.001"
[OK] "data: Update training set with 10k new samples"
[OK] "fix: Correct normalization in preprocessing"
[OK] "perf: Optimize feature extraction (20% faster)"
```

Warning

Avoid Vague Messages:

```
[x] "Update"
[x] "Fix bug"
[x] "Changes"
[x] "WIP"
```

12.5 Data Versioning Strategy

1. **Version by Dataset Splits**:

  - Track train/val/test sets separately
  - Ensures consistent evaluation

2. **Version Preprocessing Steps**:
  - Track both raw and processed data
  - Enables reproducing preprocessing

3. **Use Meaningful Versions**:
  - Tag important versions: `git tag v1.0-baseline`
  - Document what changed between versions

4. **Regular Cleanup**:
  - Periodically run `dvc gc`
  - Archive old experiments

12.6 Security Considerations

Warning

Never Commit Credentials:

- Don't put API keys in code or configs
- Use environment variables for secrets
- Add credential files to .gitignore
- Use separate configs for dev/prod

Secure Credential Management

```python
import os

# Good: Use environment variables
AWS_KEY = os.getenv('AWS_ACCESS_KEY_ID')
AWS_SECRET = os.getenv('AWS_SECRET_ACCESS_KEY')

# Bad: Hardcoded credentials
# AWS_KEY = "AKIAIOSFODNN7EXAMPLE"  # DON'T DO THIS!
```

12.7 Documentation

Maintain comprehensive documentation:

Essential Documentation

README.md should include:

- Project overview and goals
- Setup instructions (Git + DVC)
- Remote storage configuration

- How to reproduce experiments
- Data sources and descriptions
- Pipeline overview
- Contact information

13 Real-World Case Studies

13.1 Case Study 1: Image Classification Project

Scenario

Project: Medical image classification for disease detection
Challenges:

- 500GB of medical images
- Multiple preprocessing pipelines tested
- 50+ experiments with different architectures
- Team of 5 data scientists

DVC Solution:

- Images tracked with DVC, stored in AWS S3
- Each preprocessing version tracked separately
- Pipeline defined in dvc.yaml for reproducibility
- Parameters tracked in params.yaml
- Metrics logged for each experiment

Results:

- Reduced onboarding time from 2 days to 2 hours
- Full experiment reproducibility
- 70% reduction in storage costs (deduplicated data)
- Easy rollback to best-performing model

13.2 Case Study 2: NLP Pipeline

Scenario

Project: Sentiment analysis for customer reviews
Challenges:

- Text data updated monthly
- Multiple feature engineering approaches
- Different models (BERT, GPT, custom)
- Need to track embeddings (5GB each)

DVC Solution:

- Raw text data tracked with DVC
- Preprocessing pipeline automated with dvc repro

- Embeddings versioned and cached
- Model checkpoints tracked
- Automatic retraining when data updated

Results:

- Automated monthly retraining
- Easy A/B testing of models
- Clear audit trail of changes
- Improved collaboration across teams

14 Quick Reference Guide

14.1 Essential Commands Cheat Sheet

Setup & Configuration

```
dvc init                        # Initialize DVC
dvc remote add -d name url      # Add remote storage
dvc remote list                 # List remotes
```

Data Tracking

```
dvc add data/                   # Track data
dvc push                        # Upload to remote
dvc pull                        # Download from remote
dvc status                      # Check status
```

Version Control

```
git checkout <commit>           # Switch to version
dvc pull                        # Get corresponding data
git checkout main               # Return to latest
```

Pipelines

```
dvc run -n stage ...            # Create stage
dvc repro                       # Reproduce pipeline
dvc dag                         # View pipeline
```

14.2 Common Workflows

14.2.1 Initial Project Setup

```
# 1. Initialize
git init
dvc init

# 2. Configure storage
dvc remote add -d storage s3://bucket/path

# 3. Track data
dvc add data/

# 4. Commit
git add .
git commit -m "Initial setup"
dvc push
git push
```

14.2.2 Daily Development

```
# 1. Start work
git pull
dvc pull
```

```

# 2. Make changes
python train.py

# 3. Track changes
dvc add models/
dvc push

# 4. Commit
git add .
git commit -m "Updated model"
git push
```

14.2.3 Experiment Workflow

```
# 1. Create branch
git checkout -b experiment-1

# 2. Modify parameters
vim params.yaml

# 3. Run experiment
dvc repro

# 4. Version results
dvc add models/ metrics/
dvc push

# 5. Commit
git add .
git commit -m "Experiment: increased learning rate"
git push
```

14.3 Troubleshooting Checklist

Issue	**Solution**
Can't push data	Check `dvc remote list` and credentials
Data not pulling	Run `dvc status -c` to check remote
Cache too large	Run `dvc gc --workspace`
Slow operations	Use `-j` flag for parallel jobs
Git tracking data	Add to .gitignore, track with DVC
Corrupted cache	Delete cache, run `dvc pull`

15 Conclusion

15.1 Key Takeaways

DVC is an essential tool for modern machine learning workflows, providing:

1. **Data Versioning**: Track large datasets efficiently
2. **Reproducibility**: Reproduce any experiment with exact data
3. **Collaboration**: Team can work on same project seamlessly
4. **Pipeline Automation**: Automate ML workflows
5. **Storage Flexibility**: Support for various cloud providers
6. **Git Integration**: Works alongside existing Git workflow

15.2 When to Use DVC

Use DVC when you need:

- Version control for large datasets
- Track ML experiments systematically
- Reproduce results reliably
- Collaborate on ML projects
- Automate ML pipelines
- Manage model files efficiently

DVC might be overkill if:

- Working with small datasets (< 100MB)
- Single-person, one-time analysis
- No need for reproducibility
- Simple data that fits in Git

15.3 The DVC Advantage

Why DVC Matters

DVC bridges the gap between traditional software engineering (Git) and modern machine learning workflows. It extends version control to data and models, making ML projects:

- **Reproducible**: Anyone can reproduce any experiment
- **Collaborative**: Teams work together effectively
- **Auditable**: Complete history of all changes
- **Efficient**: Smart caching and storage optimization
- **Scalable**: Handles projects from MB to TB

15.4 Next Steps

To continue your DVC journey:

1. **Practice**: Set up DVC in a personal project
2. **Experiment**: Try different storage backends
3. **Pipelines**: Create automated ML pipelines
4. **Integrate**: Combine with MLflow, Jupyter, CI/CD
5. **Contribute**: Join the DVC community

15.5 Additional Resources

- Official Documentation: `https://dvc.org/doc`
- DVC GitHub Repository: `https://github.com/iterative/dvc`
- DVC Community: `https://discord.com/invite/dvwXA2N`
- DVC Blog: `https://dvc.org/blog`
- Tutorials: `https://dvc.org/doc/start`
- Use Cases: `https://dvc.org/doc/use-cases`

15.6 Final Thoughts

DVC represents a paradigm shift in how we approach machine learning project management. By treating data as a first-class citizen alongside code, DVC enables the same level of rigor and best practices in ML that software engineering has enjoyed for decades.

Important Note

Remember the Temple Analogy:
Just as the temple token system ensures your belongings stay together, DVC ensures your data and code stay in sync across all versions and team members. This simple but powerful concept makes ML projects more reliable, reproducible, and collaborative.

End of DVC Complete Reference Guide

"In God we trust. All others must bring data."
— W. Edwards Deming

With DVC, bring versioned data!