

DVC guided tour: a walkthrough

Quick Note

This is a non-interactive demo, there will be no intervals for trying things out.

Feel free to interrupt if you have any question.

- but my knowledge is limited. I haven't integrate DVC in my work yet and my exposure is from reading some documentation and developing this demo.
- expect blind spots and uncovered ground

Baseline run (no DVC yet)

Baseline run (no DVC yet) i

Run the full pipeline (runs `prepare`, `train` and `evaluate` stages)

```
python -m mltoy.cli run-all
```

The default for pipeline params is `params.yaml`

```
split:  
    test_size: 0.2  
    random_state: 42  
  
preprocess:  
    standardize: true  
  
train:  
    C: 1.0  
    max_iter: 200
```

Baseline run (no DVC yet) ii

Expected output/artifacts

- data/processed/train.csv
- data/processed/test.csv
- models/model.joblib
- reports/metrics.json

Content of metrics.json

```
{  
    "accuracy": 0.9333333333333333,  
    "f1_macro": 0.9333333333333332  
}
```

Wire in DVC (data + pipeline)

Initialize DVC

Initialize DVC

```
dvc init # generates .dvc and .dvcignore
```

Commit dvc to git history

```
git add .dvc .dvcignore  
git commit -s -m "Init dvc and add dvc metadata to git"
```

Note: using `.gitignore` we avoid committing DVC cache and temp to Git.

```
.dvc/cache/  
.dvc/tmp/
```

Note: Do not run `git add .` until after `dvc add data/raw/iris.csv` (or use the `.gitignore` allowlist pattern below).

Track the raw dataset with DVC

Add data file to DVC

```
dvc add data/raw/iris.csv
```

This will generate a “DVC pointer” file `data/raw/iris.csv.dvc`

`outs:`

- `md5: 429ae2e52e964aed4081259c27ad9552`

- `size: 2596`

- `hash: md5`

- `path: iris.csv`

Track the raw dataset with DVC ii

Commit the “DVC pointer” file to git

```
git add data/raw/iris.csv.dvc  
git commit -s -m "Track raw iris dataset with dvc"
```

Git tracks the small `.dvc` metadata (“pointer” files), while DVC manages the dataset itself.

Track the raw dataset with DVC iii

Note: `dvc add data/raw/iris.csv` may generate `data/raw/.gitignore`; commit it if it appears. It is intended for hiding data file from Git to avoid ownership conflict between Git and DVC. But we already made data files invisible to Git via `.gitignore` at the root, so `data/raw/.gitignore` might not be generated.

```
# ignore all data files by default
data/**

# do not ignore the directories themselves (so negations below can work)
!data/**/
# allow DVC metadata anywhere under data/
!data/**/*.dvc
!data/**/.gitignore
```

Define the pipeline stages using “dvc stage add” i

Description of args to `dvc stage add`

- `-d` declares dependencies (code + inputs)
- `-o` declares outputs (cached by DVC)
- `-m` declares metrics (so dvc metrics ... can find/show/diff them)
- `-p` declares parameter dependencies from `params.yaml` (granular invalidation)

Define the pipeline stages using “dvc stage add” ii

Add “prepare” stage

```
dvc stage add \
  -n prepare \
  -d mltoy \
  -d data/raw/iris.csv \
  -p split.test_size,split.random_state,preprocess.standardize \
  -o data/processed/train.csv \
  -o data/processed/test.csv \
  python -m mltoy.cli prepare \
  --raw data/raw/iris.csv \
  --train data/processed/train.csv \
  --test data/processed/test.csv \
  --params params.yaml
```

Define the pipeline stages using “dvc stage add” iii

Add “train” stage

```
dvc stage add \
  -n train \
  -d mltoy \
  -d data/processed/train.csv \
  -p train.C,train.max_iter,preprocess.standardize \
  -o models/model.joblib \
  python -m mltoy.cli train \
  --train data/processed/train.csv \
  --model models/model.joblib \
  --params params.yaml
```

Define the pipeline stages using “dvc stage add” iv

Add “evaluate” stage

```
dvc stage add \
  -n evaluate \
  -d mltoy \
  -d data/processed/test.csv \
  -d models/model.joblib \
  -m reports/metrics.json \
  python -m mltoy.cli evaluate \
  --test data/processed/test.csv \
  --model models/model.joblib \
  --metrics reports/metrics.json \
  --params params.yaml
```

Define the pipeline stages using “dvc stage add” v

The sequence of `dvc stage add` commands results in a `dvc.yaml`

```
stages:
```

```
  prepare:  
    cmd: "python -m mltoy.cli prepare --raw ..."  
    deps: [...]  
    params: [...]  
    outs: [...]  
  
  train:  
    ...  
  
  evaluate:  
    cmd: "python -m mltoy.cli evaluate --test ..."  
    deps: [...]  
    metrics: [...]
```

Define the pipeline stages using “dvc stage add” vi

Commit the DVC pipeline definition to Git

```
git add dvc.yaml  
git commit -s -m "Define DVC pipeline stages"
```

Run the pipeline: “dvc repro”

```
dvc repro

'data/raw/iris.csv.dvc' didn't change, skipping

Running stage 'prepare':
> python -m mltoy.cli prepare [...]
Generating lock file 'dvc.lock'
Updating lock file 'dvc.lock'

Running stage 'train':
> python -m mltoy.cli train [...]
Updating lock file 'dvc.lock'

Running stage 'evaluate':
> python -m mltoy.cli evaluate [...]
Updating lock file 'dvc.lock'
```

Commit dvc.lock to git

```
git add dvc.lock
git commit -s -m "Add first version of dvc.lock"
```

Quick note on “dvc.lock” metadata

- `dvc repro` updates `dvc.lock` whenever the resolved deps/params/outs for any stage change (if nothing changes, it may remain identical).
- `dvc.lock` is typically tracked in Git. It is part of the reproducibility story: it pins the exact versions (hashes) of dependencies and outputs for each stage for that Git revision.
- Purpose and how `dvc.lock` is used:
 - `dvc.yaml` is the human-authored pipeline spec (commands, deps, outs, params).
 - `dvc.lock` is the machine-generated “resolved snapshot” with concrete hashes. DVC uses it to know what is up to date, what to rerun, and to reproduce the exact state associated with a commit. It is also what makes `dvc repro` deterministic across machines when combined with a DVC remote/cache.

Visualize the DAG - stage dependencies

```
dvc dag
```

```
+-----+  
| data/raw/iris.csv.dvc |  
+-----+  
    *  
    *  
    *  
    +-----+  
    | prepare |  
    +-----+  
        **      **  
        **          *  
        *          **  
    +-----+          *  
    | train |          **  
    +-----+          *  
        **      **  
        **          **  
        *  *  
    +-----+  
    | evaluate |  
    +-----+
```

The diagram illustrates the dependencies between stages in a Data Pipeline. The stages are represented by boxes with their names: 'data/raw/iris.csv.dvc', 'prepare', 'train', and 'evaluate'. Arrows indicate dependencies from one stage to another. The 'data/raw/iris.csv.dvc' stage has three outgoing arrows pointing to the 'prepare' stage. The 'prepare' stage has two outgoing arrows pointing to the 'train' stage. The 'train' stage has two outgoing arrows pointing to the 'evaluate' stage. The 'evaluate' stage has no outgoing arrows.

Show metrics

```
$ dvc metrics show
Path           accuracy      f1_macro
reports/metrics.json  0.93333  0.93333
```

Cache-aware re-execution

No-op rerun

```
$ dvc status  
Data and pipelines are up to date.
```

```
dvc repro
```

With no changes to dependencies, `repro` would be a “no-op”

```
'data/raw/iris.csv.dvc' didn't change, skipping  
Stage 'prepare' didn't change, skipping  
Stage 'train' didn't change, skipping  
Stage 'evaluate' didn't change, skipping  
Data and pipelines are up to date.
```

Rerun: change a training hyperparameter i

```
In params.yaml set train.C to 0.15
```

```
$ dvc status
train:
    changed deps:
        params.yaml:
            modified:          train.C
```

```
$ dvc params diff
Path      Param   HEAD   workspace
params.yaml  train.C  1.0    0.1
```

Rerun: change a training hyperparameter ii

This should rerun train + evaluate only

```
$ dvc repro
```

```
'data/raw/iris.csv.dvc' didn't change, skipping
```

```
Stage 'prepare' didn't change, skipping
```

```
Running stage 'train':
```

```
> python -m mltoy.cli train [...]
```

```
Updating lock file 'dvc.lock'
```

```
Running stage 'evaluate':
```

```
> python -m mltoy.cli evaluate [...]
```

```
Updating lock file 'dvc.lock'
```

Rerun: change a training hyperparameter iii

Execution of `dvc repro` updates `dvc.lock` and now `dvc status` shows no change.

But `dvc diff` vs HEAD shows changes on both params and metrics

```
$ dvc params diff
Path          Param     HEAD      workspace
params.yaml   train.C  1.0       0.1
```

```
$ dvc metrics diff
Path          Metric    HEAD      workspace      Change
reports/metrics.json accuracy  0.93333  0.86667  -0.06667
reports/metrics.json f1_macro  0.93333  0.86532  -0.06801
```

Rerun: change a training hyperparameter iv

Commit changes to Git

```
git add dvc.lock params.yaml  
git commit -s -m "Update train.c value in param and dvc.lock"
```

Now `dvc status`, `dvc params diff` and `dvc metrics diff` show no changes!

Rerun: change a split hyperparameter in params.yaml

```
In params.yaml set split.test_size to 0.3
```

```
$ dvc status
```

```
prepare:
```

```
    changed deps:
```

```
        params.yaml:
```

```
            modified:           split.test_size
```

```
$ dvc params diff
```

Path	Param	HEAD	workspace
params.yaml	split.test_size	0.2	0.3

Rerun: change a split hyperparameter ii

```
$ dvc repro # Reproduce again - should rerun prepare + downstream
```

```
'data/raw/iris.csv.dvc' didn't change, skipping
```

```
Running stage 'prepare':
```

```
> python -m mltoy.cli prepare [...]
```

```
Updating lock file 'dvc.lock'
```

```
Running stage 'train':
```

```
> python -m mltoy.cli train [...]
```

```
Updating lock file 'dvc.lock'
```

```
Running stage 'evaluate':
```

```
> python -m mltoy.cli evaluate [...]
```

```
Updating lock file 'dvc.lock'
```

Rerun: change a split hyperparameter iii

```
$ dvc params diff
```

Path	Param	HEAD	workspace
params.yaml	split.test_size	0.2	0.3

```
$ dvc metrics diff
```

Path	Metric	HEAD	workspace	Change
reports/metrics.json	accuracy	0.86667	0.84444	-0.02222
reports/metrics.json	f1_macro	0.86532	0.84427	-0.02105

Rerun: change a split hyperparameter iv

Commit changes to Git

```
git add dvc.lock params.yaml  
git commit -s -m "Update split.test_size value in param and dvc.lock"
```

Now `dvc status`, `dvc params diff` and `dvc metrics diff` show no changes!

Data Versioning

Data changes

Make a small data change (example: remove 10 rows)

```
python - <<'PY'  
import pandas as pd  
df = pd.read_csv("data/raw/iris.csv")  
df = df.iloc[:-10]  
df.to_csv("data/raw/iris.csv", index=False)  
PY
```

Check out dvc status

```
$ dvc status  
prepare:  
    changed deps:  
        modified:      data/raw/iris.csv  
data/raw/iris.csv.dvc:  
    changed outs:  
        modified:      data/raw/iris.csv
```

Note that git status won't see any changes since Git is not tracking the data file.

Update data version

Update DVC-tracked data pointer

```
dvc add data/raw/iris.csv
```

And DVC status for the file itself is up-to-date.

```
$ dvc status  
prepare:  
    changed deps:  
        modified:          data/raw/iris.csv
```

Note: for `dvc status` to be completely up-to-date, we should `dvc repro` to update `dvc.lock`.

Now the Git-tracked pointer file can be committed to Git.

```
git add data/raw/iris.csv.dvc  
git commit -s -m "Update data version: modify raw iris dataset"
```

Tip: Git log of pointer files is a common way to see data versions through Git history

```
git log --oneline -- data/raw/iris.csv.dvc  
# See what changed between versions (hash/size):  
git diff <old_commit> <new_commit> -- data/raw/iris.csv.dvc
```

Re-run pipeline after date version update

```
dvc repro
```

```
'data/raw/iris.csv.dvc' didn't change, skipping
```

```
Running stage 'prepare':
```

```
> python -m mltoy.cli prepare [...]
```

```
Updating lock file 'dvc.lock'
```

```
Running stage 'train':
```

```
> python -m mltoy.cli train [...]
```

```
Updating lock file 'dvc.lock'
```

```
Running stage 'evaluate':
```

```
> python -m mltoy.cli evaluate [...]
```

```
Updating lock file 'dvc.lock'
```

```
Commit changes of dvc.lock
```

```
git add dvc.lock
```

```
git commit -s -m "Update dvc.lock after data update"
```

DVC Remote (Just a Glance)

Why to add DVC remote

When a DVC remote is not configured and not `dvc push`:

- The actual file contents are stored in the local DVC cache (`.dvc/cache`), and Git commits store the `.dvc` pointer files that reference those cached contents. So switching git commits and `dvc checkout` will reuse the local cache.
- old data versions would be lost if the local cache is cleared, corrupted, or garbage-collected, or if the project is move to another machine. Without a remote, there is no authoritative shared store to fetch missing objects from.

Set a remote and `dvc push` for durable/shareable/reproducible across machines.

Git remote takes care of your code and small files, DVC remote does that for your data.

Storage backend

The following example uses a “local” remote for simplicity.

In addition to local file systems, DVC supports - cloud providers (AWS S3, Google Cloud Storage, Azure Blob Storage), and - network protocols (SSH, HDFS, WebDAV)

In real-life one should use proper data storage backend.

Add DVC remote (push/pull) i

Create a local folder to act as remote:

```
mkdir -p .local_dvc_remote
```

Configure it as default remote:

```
dvc remote add -d local .local_dvc_remote
```

which will create the `.dvc/config` metadata file

```
[core]
    remote = local
['remote "local"']
    url = ../../.local_dvc_remote
```

Add DVC remote (push/pull) ii

Commit to git

```
git add .dvc/config  
git commit -s -m "Add local dvc remote"
```

Push cached data to remote:

```
dvc push
```

To simulate fresh clone:

- delete `data/processed`, `models`, `reports` and even the `data/raw`
- then run: `dvc pull && dvc repro`

Other Useful Features of DVC

We left them out, but you shouldn't

- Multiple DVC subprojects in the same Git repo
- DVC experiments: `dvc exp run/save/show/diff/apply/branch` (rapid iteration without Git commits)
- Remotes beyond local: S3/SSH/Azure/GCS, `dvc push/pull/fetch` (team sharing + reproducibility)
- Cache lifecycle: run-cache, `dvc gc`, “cache is not backup” (storage hygiene and pitfalls)
- Plots: `dvc plots show/diff` (visual diffs across versions/experiments)
- External data and outputs: deps/outs outside repo, `--to-remote` workflows (common in real projects)
- CI integration: `dvc repro / dvc pull` in CI, metrics gates, artifact publishing
- Data Registry: sharing/versioning datasets across repos and teams