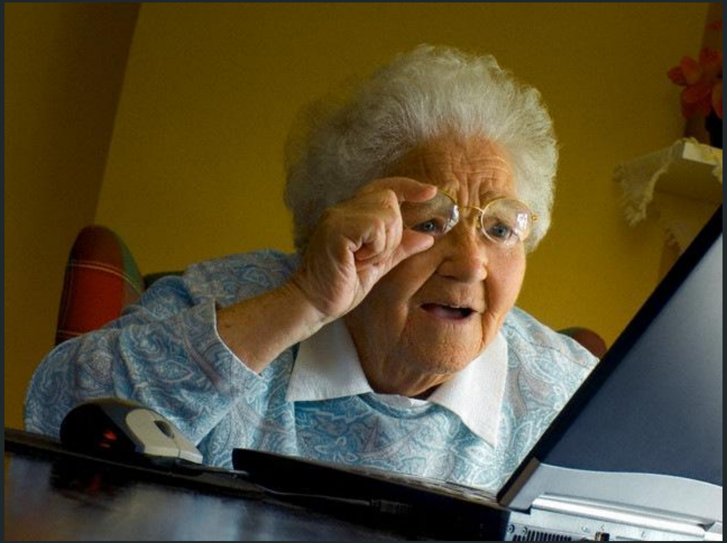


What is DVC?

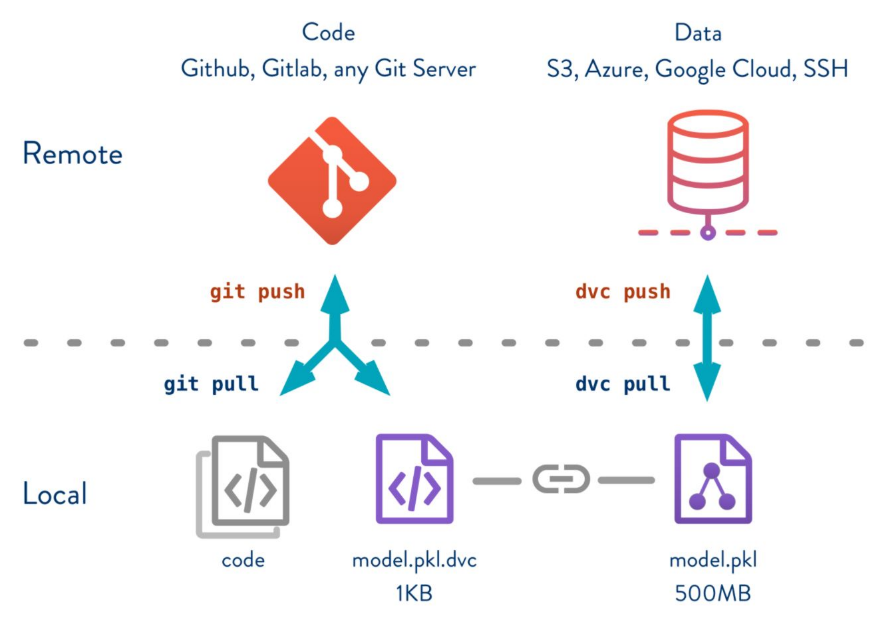
What is DVC?[ref]

- **Simple command line Git-like experience.**
  - Does not require installing and maintaining any databases.
  - Does not depend on any proprietary online services.
- Management and **versioning** of **datasets and ML models.**
  - Data is saved in S3, Google cloud, Azure, SSH server, HDFS, or even local HDD RAID.
- Makes **projects reproducible** and shareable; answers questions on **how a model was built**.
- Helps **manage experiments with Git tags**/branches and **metrics tracking**.

*"**DVC aims to replace spreadsheet and document sharing tools** (such as Excel or Google Docs) which are being used frequently as both **knowledge repositories and team ledgers**.*

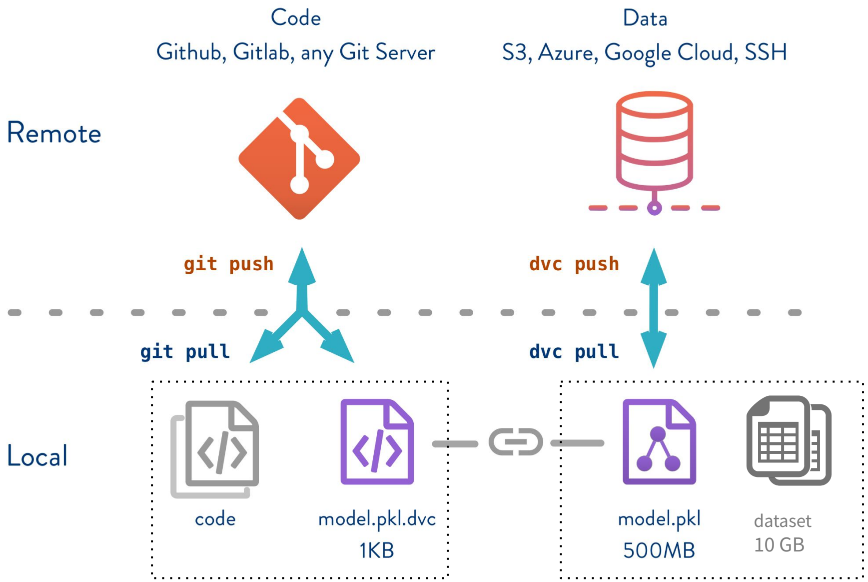
***DVC also replaces both ad-hoc scripts to track, move, and deploy different model versions;** as well as ad-hoc data file suffixes and prefixes."*

What is DVC?



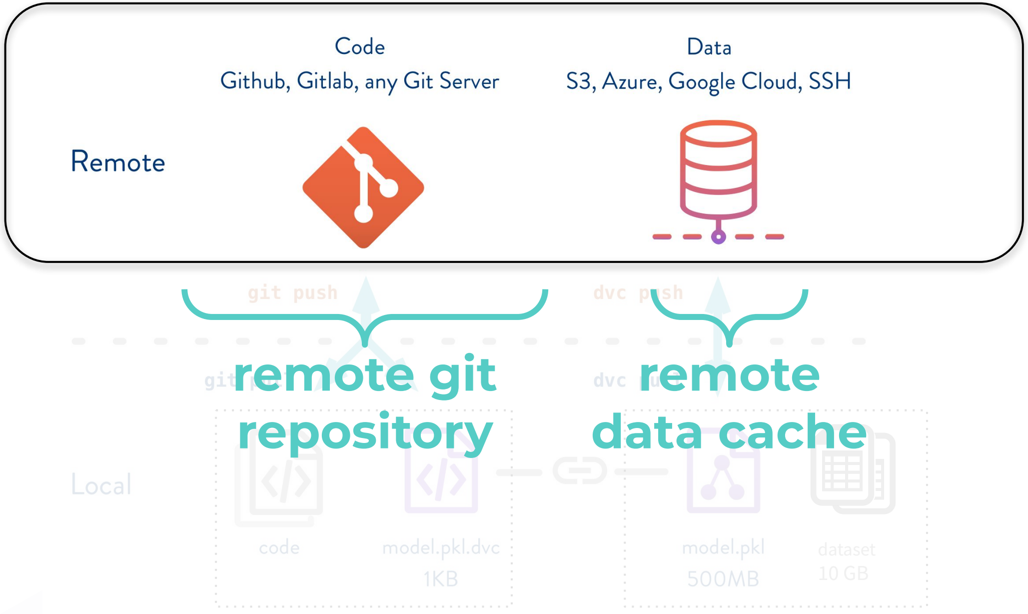
- **dvc and git**
  - `git`: version code, small files
  - `dvc`: version data, intermed. results, models
  - `dvc` uses `git`, w/o storing file content in repo
- **versioning and storing large files**
  - `dvc` save info on data in special `.dvc` files
  - `.dvc` files can then be versioned using `git`
  - actual storage happens w remote storage
  - `dvc` supports many remote storage types
- **dvc main features**
  - data versioning
  - data access
  - data pipelines

How DVC works with data?

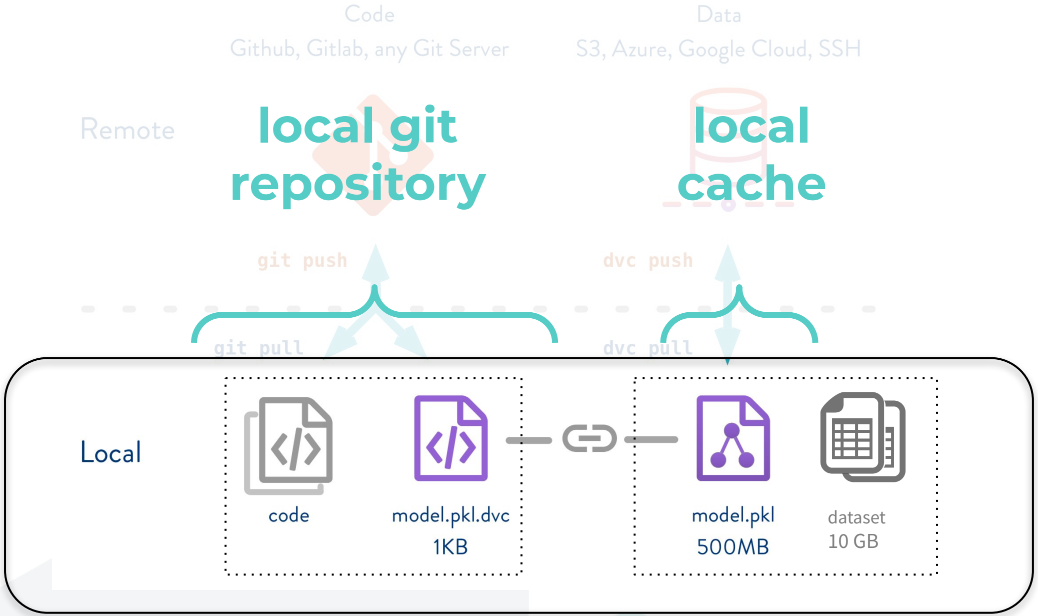


Original image source: https://dvc.org/doc/use-cases/data-and-model-files-versioning

Store data in remote storage



Bring data to local workspace



Simplify a team collaboration

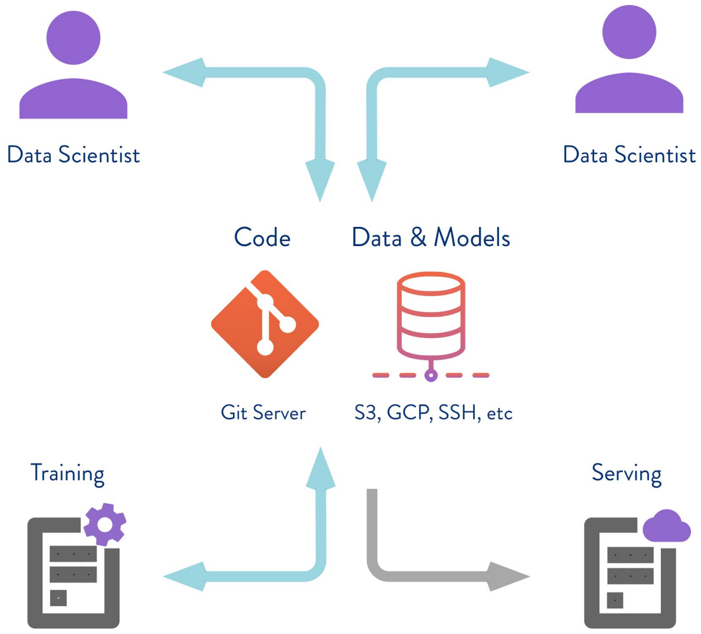


Image source: https://dvc.org/doc/use-cases/data-and-model-files-versioning

Getting Started

Install

- Install as a python package.

```
$ pip install dvc
```

- Depending on remote storage you will use, you may want to install specific dependencies.

```
$ pip install dvc[s3]  # support Amazon S3
$ pip install dvc[ssh]  # support ssh
$ pip install dvc[all]  # all supports
```

Initialization

- We must work inside a **Git repository**. If it does not exist yet, we create and initialize one.

```
$ mkdir ml_project & cd ml_project
$ git init
```

- **Initializing a DVC project** creates and automatically `git add` a few important files.

```
$ dvc init
$ git status -s
A  .dvc/.gitignore
A  .dvc/config
A  .dvc/plots/confusion.json
A  .dvc/plots/default.json
A  .dvc/plots/scatter.json
A  .dvc/plots/smooth.json
A  .dvcignore

$ git commit -m "Initialize dvc project"
```

Tell Git not to track `.dvc/cache` and `.dvc/tmp`

TOML file with configurations for
- dvc remote storage — name, url
- dvc cache – reflink/copy/hardlink, location, ...
- ...

Arbitrary files that you may create yourselves.
~~Plot templates (visualize & compare metrics)~~

Tell dvc what not to track (empty for now)

v1.0 v2.0 v3.0

Data Versioning

Getting some data

Use "tutorials/versioning/data.zip" instead

- Let's download some data to train and validate a "cat VS dog" CNN classifier.
  We use `dvc get`, which is like `wget` to download data/models from a remote dvc repo.

```
$ dvc get  https://github.com/iterative/dataset-registry   tutorial/ver/data.zip
$ unzip data.zip   &   rm -f data.zip
  inflating: data/train/cats/cat.001.jpg
  ...
```

- This folder contains 43 MB of JPG figures organized in a hierarchical fashion.

```
data
├── train
│    ├── dogs         # 500 pictures
│    └── cats         # 500 pictures
└── validation
     ├── dogs         # 400 pictures
     └── cats         # 400 pictures
```

Start versioning data

- Tracking data with DVC is very similar to tracking code with git.

```
$ dvc add  data/
100% Add|██████████|1/1    [00:30, 30.51s/file]
To track the changes with git, run:
    git add .gitignore data.dvc

$ git add .gitignore  data.dvc
$ git commit -m "Add first version of data/"
$ git tag  -a "v1.0"  -m "data v1.0, 1000 images"
```

Tell git not to track the `data/` directory

DVC-generated, contains hash to track `data/` :

```
outs:
- md5: b8f4d5a78e55e88906d5f4aeaf43802e.dir
  path: data
```

→ human readable, can be versioned with git!

- Quite a few things happened when calling `dvc add` :
  - The **hash** of the content of `data/` was computed and added to a new `data.dvc` file
  - DVC updates `.gitignore` to tell Git **not to track the content of `data/`**
  - The **physical content** of `data/` —i.e. the jpg images— **has been moved to a cache** (by default the cache is located in `.dvc/cache/` but using a remote cache is possible!)
  - The files were **linked back to the workspace** so that it looks like nothing happened (the user can configure the link type to use: hard link, soft link, reflink, copy)

Make changes to tracked data (add)

- Let's download some more data for our "cat VS dog" dataset.
  Running **`dvc diff`** will confirm that dvc is aware the data has changed!

"tutorials/versioning/new-labels.zip"

```
$ dvc get  https://github.com/iterative/dataset-registry   tutorial/ver/new-labels.zip
$ unzip new-labels.zip   &   rm -f new-labels.zip
  inflating: data/train/cats/cat.501.jpg
  ...
$ dvc diff
Modified:
    data/
```

- To track the changes in our data with dvc, we follow the same procedure as before.

```
$ dvc add data/
$ git diff data.dvc
 outs:
-- md5: b8f4d5a78e55e88906d5f4aeaf43802e.dir
+- md5: 21060888834f7220846d1c6f6c04e649.dir
   path: data
$ git commit -am "New version of data/ with more training images"
$ git tag  -a "v2.0"  -m "data v2.0, 2000 images"
```

Switch between versions

- To switch version, first run **`git checkout`.**
  This **affects `data.dvc` but not workspace files in `data/` !**

```
$ git checkout v1.0
$ dvc diff
Modified:
    data/
```

- To fix this mismatch we simply call **`dvc checkout`.**
  This reads the cache and **updates the data in the workspace based on the current `*.dvc` files**.

```
$ dvc checkout
M       data/
$ dvc status
Data and pipelines are up to date.
```

Working with Storages

Configure remote storage

- A **remote storage** is for dvc, what a GitHub is for git:
  - push and pull files from your workspace to the remote
  - easy sharing between developers
  - safe backup should you ever do a terrible mistake à la `rm -rf *`

- Many remote storages are supported (Google Drive, Amazon S3, Google Cloud, SSH, HDFS, HTTP, …)
  But we (as for Git) nothing prevents us to use a “local remote”!

```
$ mkdir -p  ~/tmp/dvc_storage
$ dvc remote add  --default loc_remote  ~/tmp/dvc_storage
Setting 'loc_remote' as a default remote.


$ git add .dvc/config
$ git commit -m "Configure remote storage loc_remote"
```

DVC-generated, contains remote storage config

```
[core]
    remote = loc_remote
['remote "loc_remote"']
    url = /root/tmp/dvc_storage
```

Storing, sharing, retrieving from storage

- Running basically `dvc push` uploads the content of the cache to the remote storage.
  This is pretty much like `git push`.

```
$ dvc push
1800 files pushed
```

- Now, even if all the data is deleted from our workspace and cache, we can download it with `dvc pull`.
  This is pretty much like `git pull`.

```
$ rm -rf   .dvc/cache   data
$ dvc pull        # update .dvc/cache with contents from remote
1800 files fetched

$ dvc checkout   # update workspace, linking data from .dvc/cache
A       data/
```

Access data from storage

- First, we can explore the content of a DVC repo hosted on a Git server.

```
$ dvc list https://github.com/iterative/dataset-registry
README.md
get-started/
tutorial/
...
```

- When working outside of a DVC project —e.g. in automated ML model deployment— use **`dvg get`**

```
$ dvc get https://github.com/iterative/dataset-registry  tutorial/ver/new-labels.zip
```

- When working inside of another DVC project, we want to keep the connection between the projects. In this way, others can know where the data comes from and whether new versions are available.

```
$ dvc import https://github.com/iterative/dataset-registry tutorial/ver/new-labels.zip
$ git add new-labels.zip.dvc .gitignore
$ git commit -m "Import data from source"
```

`dvc import` is like `dvc get` + `dvc add`, but the resulting `data.dvc` also includes a ref to the source repo!

- **Note**. For all these commands we can specify a git revision (sha, branch, or tag) with `--rev <commit>`.

Data Registries

- We can build a DVC project dedicated only to tracking and versioning datasets and models.
  The repository would have all the metadata and history of changes in the different datasets.

- This a **data registry**, a middleware between ML projects and cloud storage.
  This introduces quite a few advantages.
  - Reusability — reproduce and organize feature stores with a simple `dvc get / import`
  - Optimization — track data shared by multiple projects centralized in a single location
  - Data as code — leverage Git workflow such as commits, branching, pull requests, CI/CD ...
  - Persistence — a DVC registry-controlled remote storage improves data security

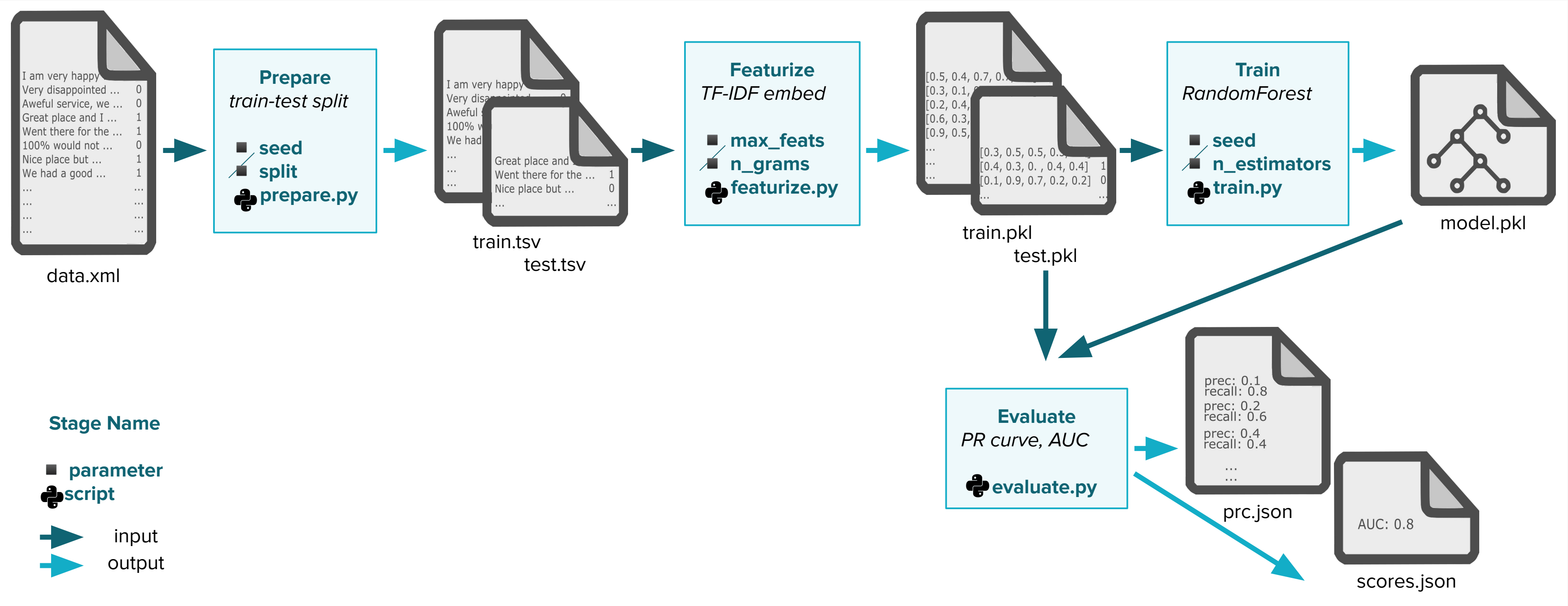
- But versioning large data files for data science is great, is not all DVC can do:
  DVC **data pipelines** capture how is data filtered, transformed, and used to train models!

DVC features:
Data and ML pipelines automation

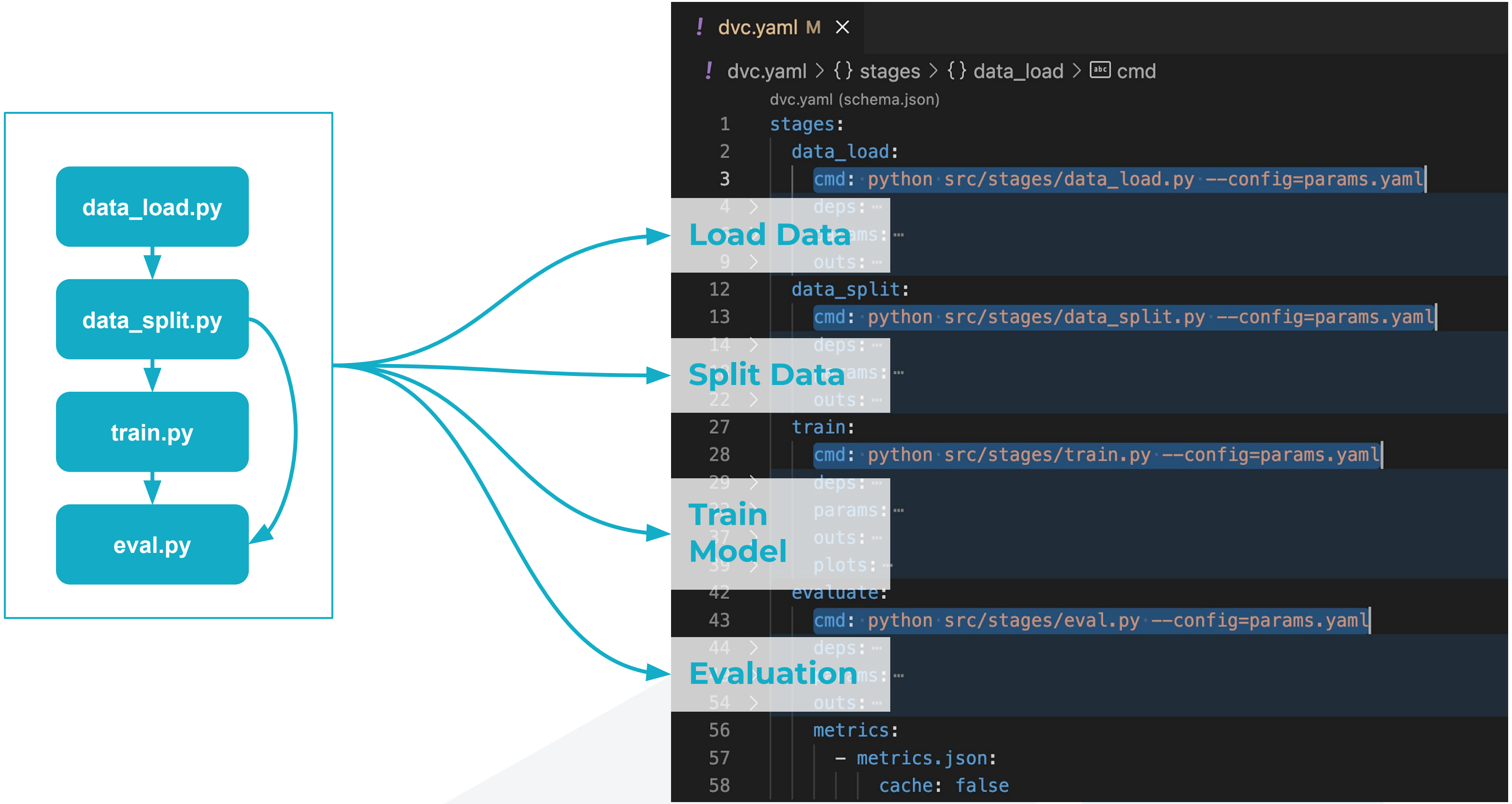
Motivation

- With `dvc add` we can track large files—this includes files such as: trained models, embeddings, etc. However, we also want to **track how such files were generated** for reproducibility and better tracking!

The following is an example of a **typical ML pipeline**. Its structure is a **DAG** (**direct acyclic graph**).

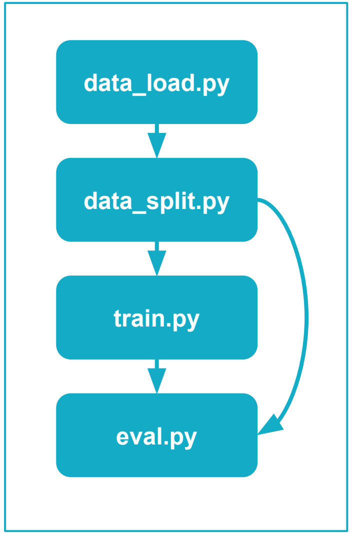


Configure pipelines in a simple dvc.yaml



Source: Alex Kim, Optimizing Image Segmentation Projects with DVC, Iterative.ai

Use any executable script as a stage job



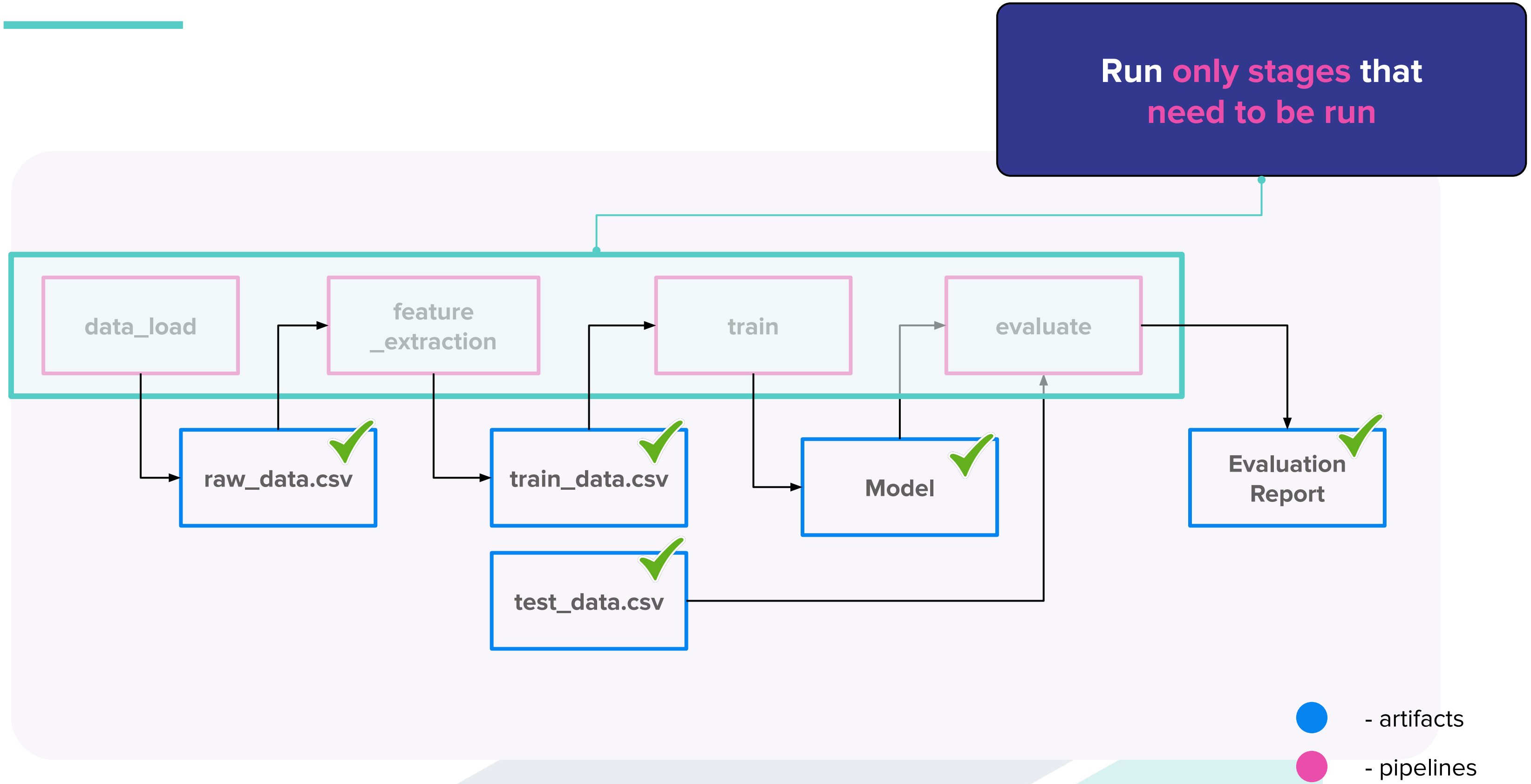
Python module

Docker container

Any script (bash)

Jupyter Notebook

Run as simple as: dvc exp run



Tracking ML Pipelines

- **Option A**: run pipeline stages, then track output artifacts with **`dvc add`**

```
$ python   src/prepare.py   data/data.xml
$ dvc add  data/prepared/train.tsv  data/prepared/train.tsv
```

~~instead of 'dvc run', read it as 'dvc stage add --run -v -f'~~

- **Option B**: run pipeline stage and track them together with all dependencies with **`dvc run`**

```
$ dvc run -n prepare \
          -p prepare.seed \
          -p prepare.split \
          -d src/prepare.py \
          -d data/data.xml \
          -o data/prepared \
          python src/prepare.py data/data.xml
```

stage name

parameters — read from `params.yaml`

dependencies (including script!)

outputs to track

```
prepare:
 split: 0.20
 seed: 20170428

featurize:
 max_feats: 500
 ngrams: 1

   ...
```

→ **Advantages of Option B**

1. outputs are automatically tracked (i.e. saved in `.dvc/cache`)
2. pipeline stages with parameters names are saved in **`dvc.yaml`**
3. deps, params, outs are all hashed and tracked in **`dvc.lock`**
4. **like a `Makefile`**, can reproduce by **`dvc run prepare`**—re-run only if deps changed!

Example

- Let's create a DVC repo for an NLP project.

```
$ mkdir nlp_project & cd nlp_project
$ git init & dvc init & git commit -m "Init dvc repo"
```

- Then we download some data + some code to prepare the data and train/evaluate a model

```
$ dvc get https://github.com/iterative/dataset-registry get-started/data.xml \
          -o data/data.xml
$ dvc add data/data.xml
$ git add data/.gitignore data/data.xml.dvc & git commit -m "Add data, first version"
$ wget https://code.dvc.org/get-started/code.zip
$ unzip code.zip & rm -f code.zip
```

```
$ tree
.
├── data
│   ├── data.xml
│   └── data.xml.dvc
├── params.yaml
└── src
    ├── evaluate.py
    ├── featurization.py
    ├── prepare.py
    ├── requirements.txt
    └── train.py
```

YAML file with params for all the stages

```
prepare:
 split: 0.20
 seed: 20170428

featurize:
 max_features: 500
 ngrams: 1

train:
 seed: 20170428
 n_estimators: 50
```

pipeline steps

Example

Use "dvc stage add --run -v -f" instead!!

- Let's run the `prepare` stage.

```
$ dvc run -n prepare \
          -p prepare.seed \
          -p prepare.split \
          -d src/prepare.py \
          -d data/data.xml \
          -o data/prepared \
          python src/prepare.py data/data.xml

$ git add  data/.gitignore  dvc.yaml  dvc.lock
```

```
stages:
 prepare:
   cmd: python src/prepare.py data/data.xml
   deps:
   - data/data.xml
   - src/prepare.py
   params:
   - prepare.seed
   - prepare.split
   outs:
   - data/prepared
```

Describe data pipelines, similar to how `Makefiles` work for building software.

```
prepare:
  cmd: python src/prepare.py data/data.xml
  deps:
  - path: data/data.xml
    md5: a304afb96060aad90176268345e10355
  - path: src/prepare.py
    md5: 285af85d794bb57e5d09ace7209f3519
  params:
    params.yaml:
      prepare.seed: 20170428
      prepare.split: 0.2
  outs:
  - params: data/prepared
    md5: 20b786b6e6f80e2b3fcf17827ad18597.dir
```

Matches the `dvc.yaml` file.

Created and updated by DVC commands like `dvc run`.

Describes latest pipeline state for:

1. track intermediate and final artifacts (like a `.dvc` file)
2. allow DVC to detect when stage defs or dependencies changed, triggering re-run.

- **Note**: dependencies and artifacts are automatically tracked, **no need to dvc add them!**

Example

~~"dvc stage add --run -v -f " instead~~

- Then we run the `featurize` and `train` stages in the same way.

```
$ dvc run -n featurize \
          -p featurize.max_features \
          -p featurize.ngrams \
          -d src/featurize.py \
          -d data/prepared \
          -o data/features \
          python src/featurization.py data/prepared data/features

$ git add  data/.gitignore  dvc.yaml  dvc.lock
```

~~featurization.py~~

```
$ dvc run -n train \
          -p train.seed \
          -p train.n_estimators \
          -d src/train.py \
          -d data/features
          -o model.pkl
          python src/train.py data/features model.pkl

$ git add  data/.gitignore  dvc.yaml  dvc.lock
```

Example

Use "dvc stage add --run -v -f" instead of "dvc run"

- And finally we run the `evaluation` stage.

```
$ dvc run -n evaluate \
          -d src/evaluate.py \
          -d model.pkl \
          -d data/features \
          --metrics-no-cache scores.json \
          --plots-no-cache prc.json \
          python src/evaluate.py model.pkl data/features scores.json prc.json

$ git add  dvc.yaml  dvc.lock
```

eval/metrics.json

eval/plots

Declare output **plot file.**

A special kind of output file (**-o**), must be JSON and can be used to make comparisons across experiments in a **plot form**.

E.g. here it contains data for **ROC curve plot.**

The `-no-cache` prevents DVC to store the file in cache.

Declare output **metrics file.**

A special kind of output file (**-o**), must be JSON and can be used to make comparisons across experiments in a **tabular form**.

E.g. here it contains data for **AUC score.**

The `-no-cache` prevents DVC to store the file in cache.

Plot dependency graphs

```
$ dvc dag

    +-------------------+
    | data/data.xml.dvc |
    +-------------------+
              *
              *
              *
         +---------+
         | prepare |
         +---------+
              *
              *
              *
        +-----------+
        | featurize |
        +-----------+
         **        **
       **            *
      *               **
+-------+               *
| train |             **
+-------+            *
         **        **
           **    **
             *  *
        +----------+
        | evaluate |
        +----------+
```

Reproducing Pipelines

- **`dvc repro`** regenerate data pipeline results, by restoring the DAG defined by stages listed in **`dvc.yaml`**. This compares file hashes with **`dvc.lock`** to re-run only if needed. This is **like `make` in software builds.**

- **Case 1: nothing changed**, re-running pipeline stages is skipped.

```
$ dvc repro train
      'data/data.xml.dvc' didn't change, skipping
      Stage 'prepare' didn't change, skipping
      Stage 'featurize' didn't change, skipping
      Stage 'train' didn't change, skipping
      Data and pipelines are up to date.
```

- **Case 2: a dependency changed**, pipeline stages are re-run if needed.

```
$ sed -i -e "s@max_features: 500@max_features: 1500@g" params.yaml
$ dvc repro train
      'data/data.xml.dvc' didn't change, skipping
      Stage 'prepare' didn't change, skipping
      Running stage 'featurize' with command:
            python src/featurization.py data/prepared data/features
      Updating lock file 'dvc.lock'
      Running stage 'train' with command:
            python src/train.py data/features model.pkl
      Updating lock file 'dvc.lock'
      To track the changes with git, run:
            git add dvc.lock
```

Track Experiments in CLI

```
$ dvc exp show
 Experiment                  Created          loss      acc   train.epochs   model.conv_units
 workspace                   -             0.25183   0.9137   10             64
 mybranch                    Oct 23, 2021        -        -   10             16
 ├── 9a4ff1c [exp-333c9]     10:40 AM      0.25183   0.9137   10             64
 ├── 138e6ea [exp-55e90]     10:28 AM      0.25784   0.9084   10             32
 └── 51b0324 [exp-2b728]     10:17 AM      0.25829   0.9058   10             16
```

dvc exp show
to visualize metrics

dvc exp push
to save (commit) experiment

```
$ dvc exp push origin exp-333c9
Pushed experiment 'exp-333c9'to Git remote 'origin'.
```

https://iterative.ai/blog/DVC-VS-Code-extension

Comparing experiments

- **`dvc params diff rev_1 rev_2`** shows how parameters differ in two different git revisions/tags. Without arguments, it shows how they differ in workspace vs. last commit.

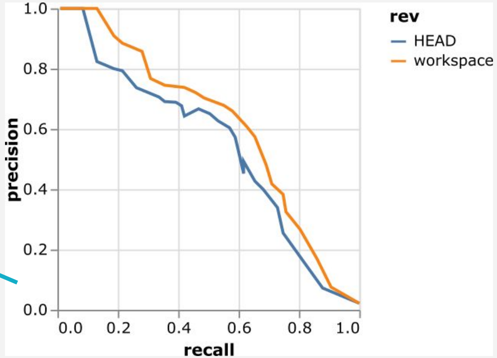
```
$ dvc params diff
Path         Param                   Old    New
params.yaml  featurize.max_features  500    1500
```

- **`dvc metrics diff rev_1 rev_2`** does the same for metrics.

```
$ dvc params diff
Path         Metric    Value    Change
scores.json  auc       0.61314  0.07139
```

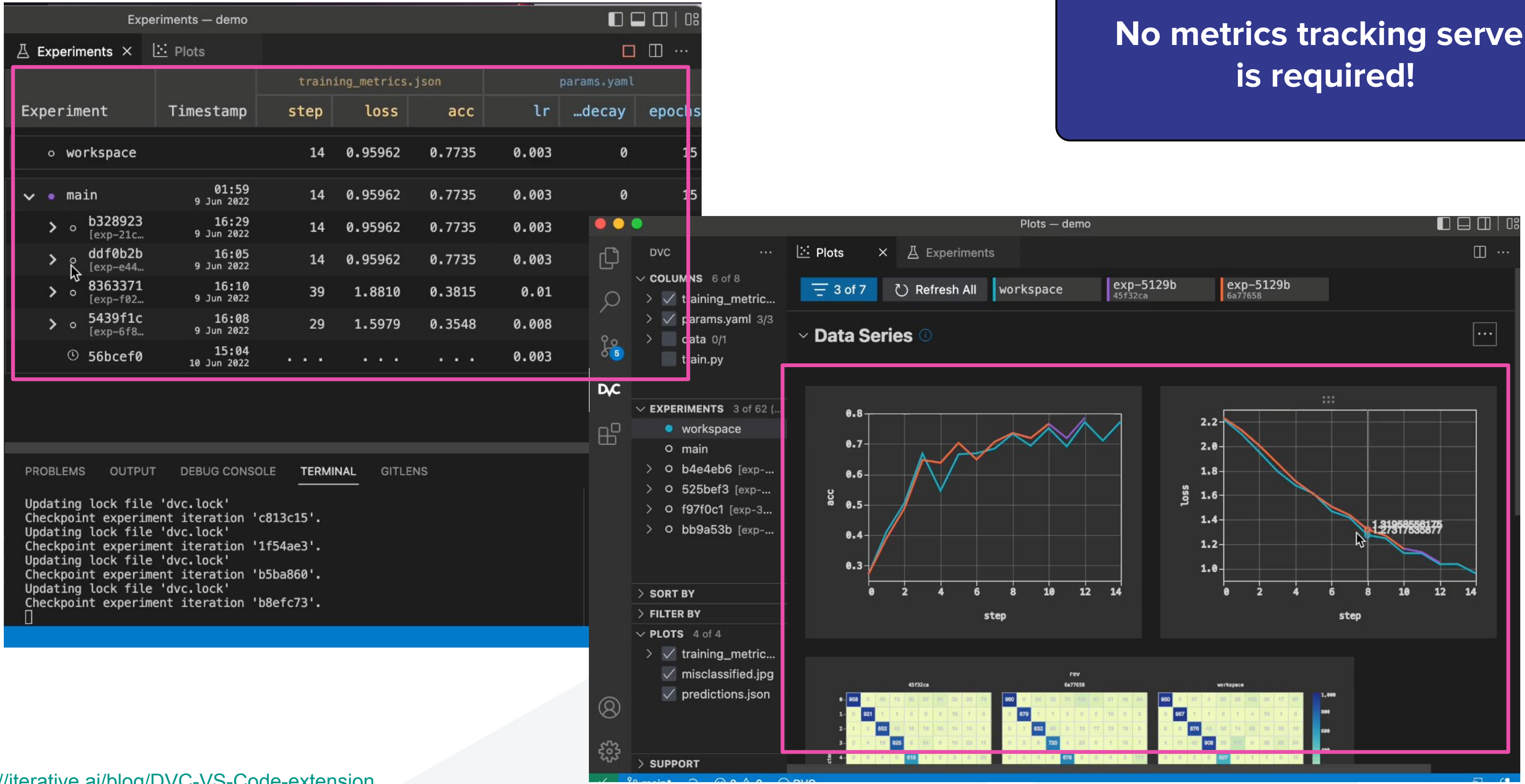
- **`dvc plots diff rev_1 rev_2`** does the same for plots.

```
$ dvc plots diff -x recall -y precision
file:///Users/dvc/example-get-started/plots.html
```



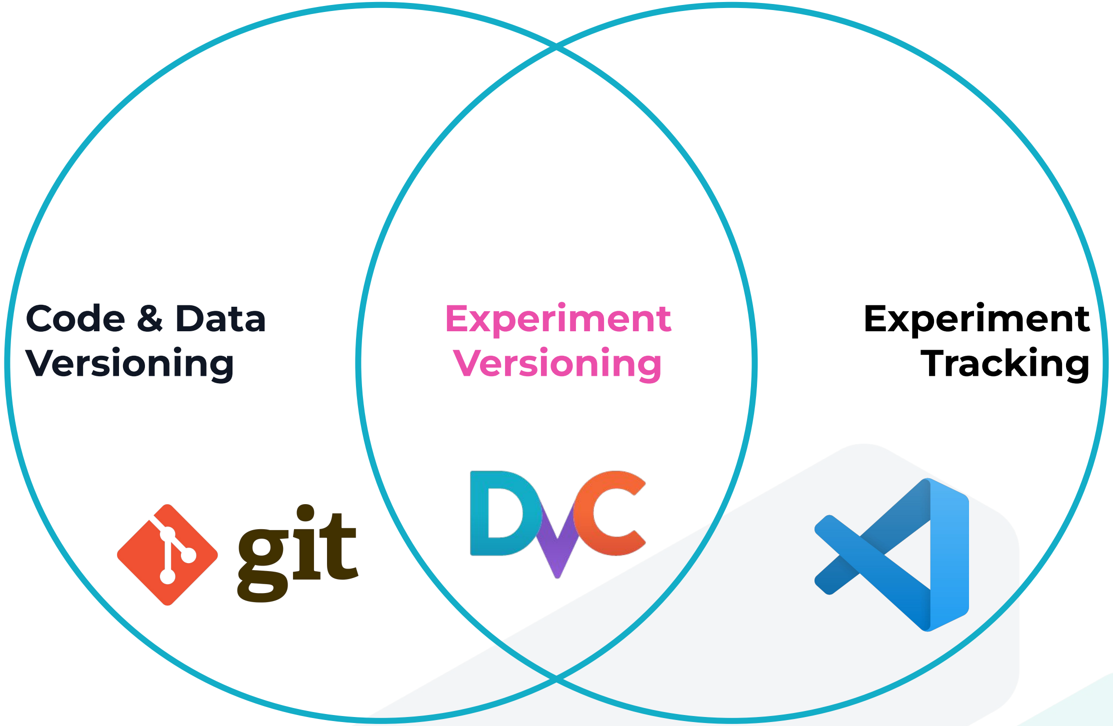
...or, use DVC extension UI in VSCode

No metrics tracking server is required!



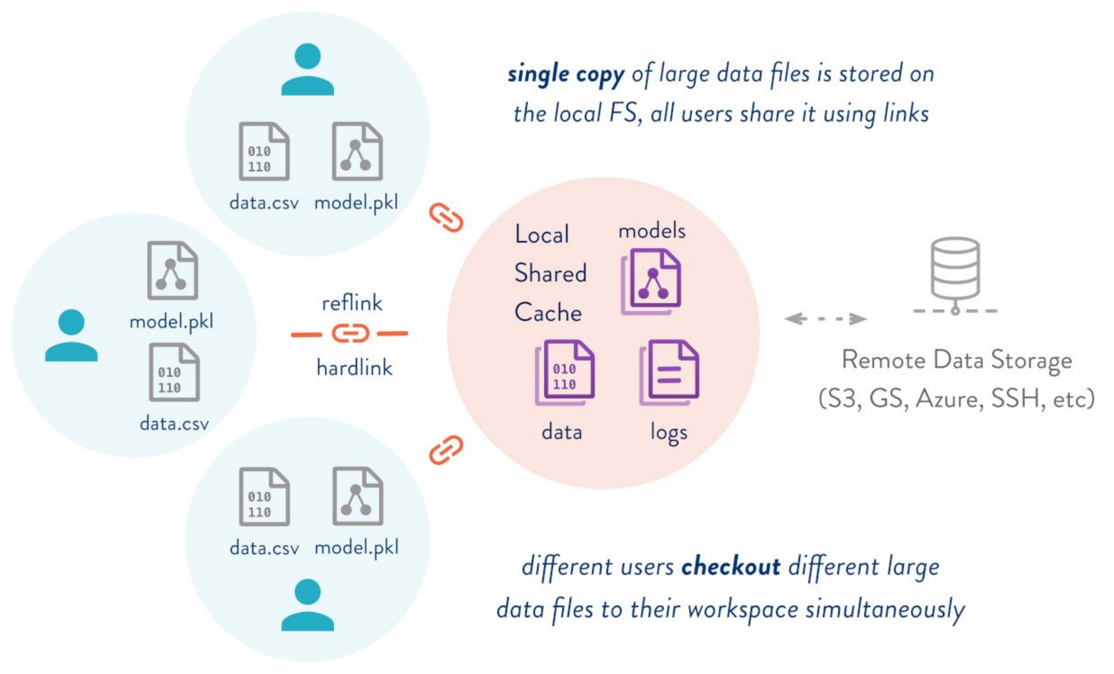
https://iterative.ai/blog/DVC-VS-Code-extension

All experiments are versioned



Shared Development Server

Shared Development Server



- **Disk space optimization**
  Avoid having 1 cache per user!

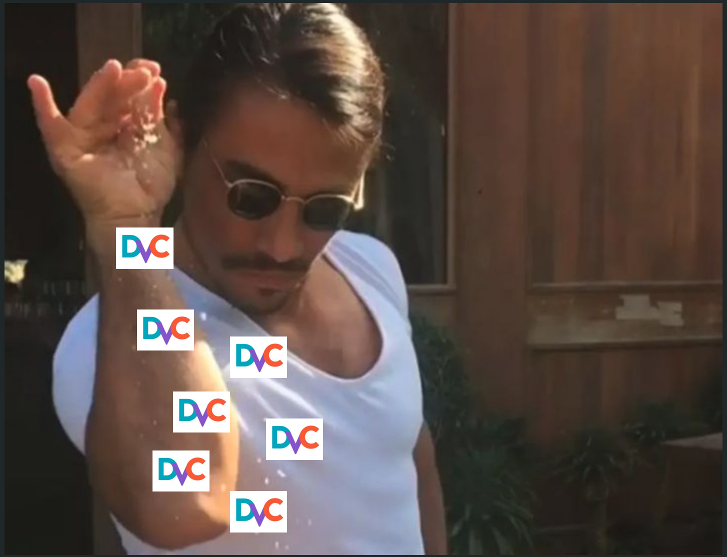
- **Use DVC as usual**
  - Each `dvc add` or `dvc run` moves data to the shared external cache!
  - Each `dvc checkout` links required data to the workspace!

- See here for implementation details, but basically it's not too difficult:

```
$ mkdir -p path_shared_cache/
$ mv .dvc/cache/* path_shared_cache/

$ dvc cache dir path_shared_cache/
$ dvc config cache.shared group

$ git commit -m "config shared cache"
```



Conclusions

Conclusions

- DVC is a **version control** system for **large ML data and artifacts**.
- DVC **integrates with Git** through **`*.dvc`** and **`dvc.lock`** files, to version files and pipelines, respectively.
- DVC repos can work as **data registries**, i.e. a middleware between cloud storage and ML projects
- To **track raw ML data files**, use **`dvc add`**—e.g. for input dataset.
- To **track intermediate or final results** of a ML pipeline, use **`dvc run`**—e.g. for model weights, dataset.
- Consider using a **shared development server** with a **unified, shared external cache**