11.05 : introduction to Data Versioning with DVC

What is Data Versioning? 00:14 - 00:59

Data Versioning involves tracking data modifications over time, similar to code versioning. By creating and storing multiple data iterations, users can retrieve specific versions, ensuring consistency and accountability. This is valuable in fields like Data Science, Machine Learning, Data Engineering, and Financial Analysis, with a focus on Machine Learning in this course.

Data vs. Code Versioning 00:59 - 01:35

Though similar, data and code versioning differ: code versioning is a mature practice using tools like Git, while data versioning emerged around 2012 and often requires specialized tools alongside Git due to dataset size.

Why Data Versioning in ML? 01:35 - 02:51

In ML, data, code, and hyperparameters all impact model quality. Data serves as the foundation, code defines the model's behavior, and hyperparameters control model tuning. Proper versioning of these components is crucial for ML experiments. In this course, we'll explore how tools like Git and DVC support versioning.

Dataset Influence 02:51 - 03:46

Using the Airbnb dataset, we split it into training sets A and B, and a test set. Testing with a random forest classifier shows minor performance changes between A and B, which could be more substantial with different distributions.

Hyperparameters Influence 03:46 - 04:29

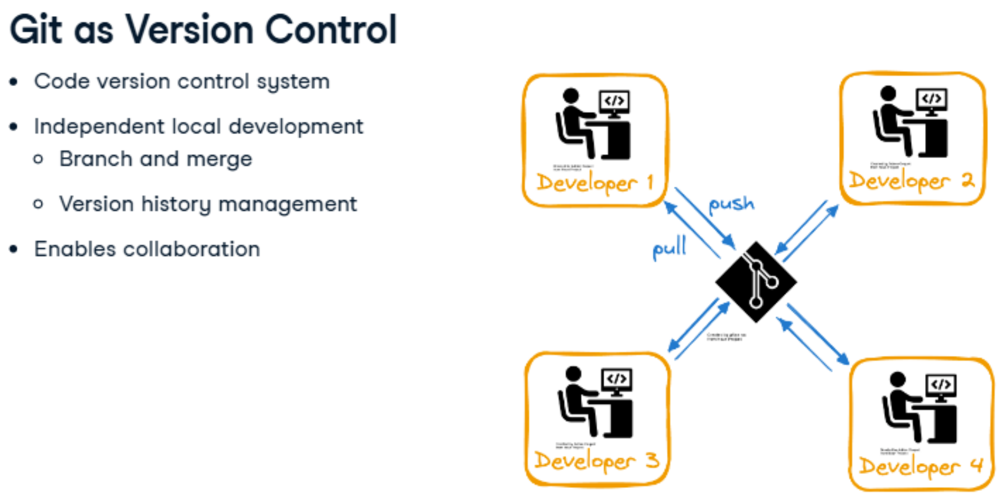
Hyperparameters also affect model performance. Here, increasing model capacity improves results. This emphasizes the need to track code, data, and hyperparameters.

Editor Exercises Layout 04:29 - 04:56

In the exercises, you'll have a layout where the pink box shows folders, the blue box shows file editing, and the green box provides a terminal for bash commands.

Introduction to DVC

Git as Version Control 00:09 - 00:46



Git is a version control system used to manage code changes. Its distributed nature allows users to work locally, supporting offline work, branching, merging, and maintaining robust version histories, which enables decentralized collaboration.

Git CLI 00:46 - 01:17

Git's Command Line Interface (CLI) lets users issue commands in a terminal or shell. A Git repository tracks a folder's contents, storing both files and Git metadata, which is kept in a `.git` folder.

Data Version Control (DVC) 01:17 - 01:48

Data Version Control (DVC) is an open-source tool integrated with Git to manage data, making it easy to version both code and data in a unified workflow.

Git vs DVC CLI 01:48 - 03:21

Git vs DVC CLI

Git

- Initialize repository in working folder

```
$ git init
```

- Adding files to repository (staging changes)

```
$ git add code.py
```

- Commit changes (in version history)

```
$ git commit -m "adding first file"
```

DVC

- Initialize DVC repository in working folder

```
$ dvc init
```

- Adding data files to DVC

```
$ dvc add data/mydata.csv
```

- Updating all tracked data files

```
$ dvc commit
```

Git vs DVC CLI

Git

- Push code changes to remote server

```
$ git push
```

- Pulling changes from remote

```
$ git pull
```

- Cloning an existing repository from remote (Github)

```
$ git clone \
https://github.com/username/repository-name.git
```

DVC

- Push data changes to remote data server

```
$ dvc push
```

- Synchronizing your DVC project

```
$ dvc pull
```

- Download a file or directory tracked by DVC

```
$ dvc get \
https://github.com/username/repo-name model.pkl
```

DVC's CLI mirrors Git's structure. Starting a Git repository uses `git init`, creating a `.git` folder. Likewise, `dvc init` initializes a DVC repository. To track data changes, we use `dvc add [file_path]`, similar to `git add` for code. While `git commit` records code changes, `dvc commit` updates DVC tracking for data without allowing commit messages.

Synchronizing with Remotes 03:21 - 04:37

Git's `git push` and `git pull` send or retrieve code changes from remote repositories. Similarly, `dvc push` and `dvc pull` handle data transfers to and from remote data servers. To clone a Git repository, we use `git clone [repository_url]`, while DVC's `dvc get` downloads specific files or directories from a remote source.

DVC features and use cases

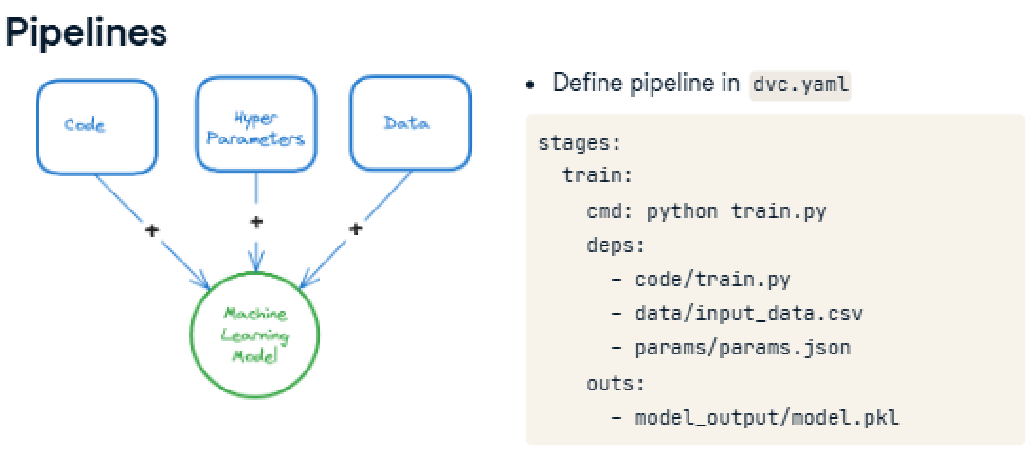
DVC Features and Use Cases 00:00 - 00:42

DVC provides capabilities for managing data and model versions, DVC pipelines for reproducible ML tasks, and monitoring metrics and plots. Advanced use cases include experiment tracking, CI/CD for ML, and data registry, though these will not be covered in this course.

Versioning Data and Models 00:42 - 01:20

DVC tracks data and model versions by creating metadata files, which record changes without duplicating data. This makes it easy to switch between different versions of data, models, and parameters, working seamlessly with Git.

Pipelines 01:20 - 02:25

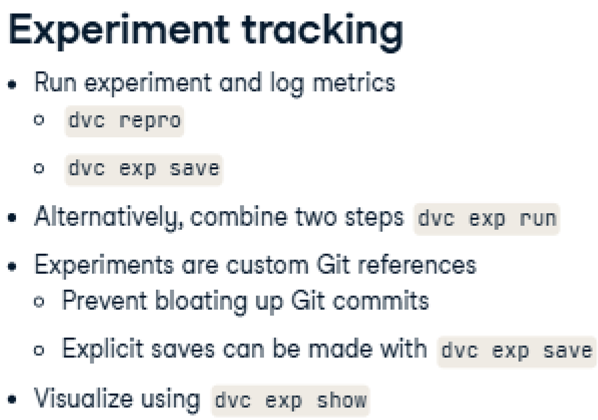


DVC pipelines allow users to define ML workflows with dependencies listed in YAML files. Each step in a pipeline specifies what command to run (`cmd`), dependencies (`deps`), and expected outputs (`outs`). Pipelines ensure reproducible workflows that can be run end-to-end.

Tracking Metrics and Plots 02:25 - 03:09

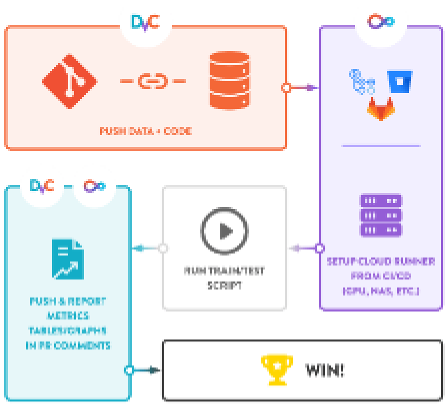
DVC can track metrics and plots by specifying them in a YAML file. The `dvc metrics diff` command compares metrics across different runs, and DVC automatically tracks changes in related data files.

Experiment Tracking 03:09 - 03:50



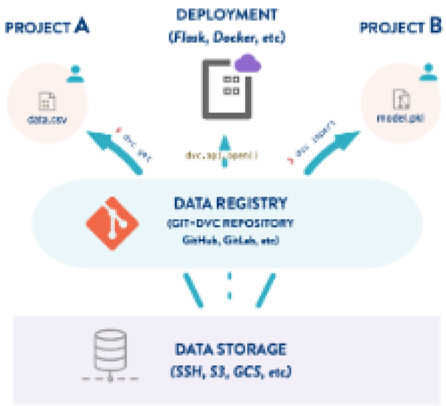
DVC supports efficient logging and retrieval of metrics with `dvc exp save` and `dvc exp run`. Experiments can be visualized in a table using `dvc exp show` without requiring permanent Git commits.

CI/CD for Machine Learning 03:50 - 04:57



DVC and CML support CI/CD for ML, automating model updates, testing, and deployments. Together, they run pipelines based on Git events, such as pull requests, allowing comparison of metrics and plots and posting them on pull requests for review.

Data Registry 04:57 - 05:32



DVC acts as a data registry, facilitating a centralized data store for multiple projects. It leverages Git to track version metadata, with actual data stored in cloud solutions like S3.

DVC Setup and Initialization

Installation 00:13 - 00:43

- DVC is a Python package
  - Universally install with `pip`

```
$ pip install dvc
```

- Remember to install in virtual environments
- Ensure Git is installed

To install DVC, use the command `pip install dvc` in a virtual environment to avoid conflicts. Ensure Git is installed for optimal functionality.

Verify Installation 00:43 - 01:09

```
$ dvc version
```

```
DVC version: 3.40.1 (pip)
Platform: Python 3.9.16 on macOS-14.2.1-arm64-arm-64bit
Config:
    Global: /Users/<username>/Library/Application Support/dvc
    System: /Library/Application Support/dvc
```

Run `dvc version` to confirm installation details, including DVC version, installation method, platform, and config locations.

Initializing DVC 01:09 - 01:27

- Ensure Git is initialized

```
$ git init
```

```
Initialized empty Git repository in /path/to/repo/.git/
```

- Initialize DVC in the repository

```
$ dvc init
```

```
Initialized DVC repository.

You can now commit the changes to git.
```

DVC works best within a Git repository. Run `git init` before `dvc init` to maximize functionality.

DVC Hidden Files 01:27 - 01:57

- Initialization creates internal files that should be tracked with Git

```
$ git status
```

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .dvc/.gitignore
    new file:   .dvc/config
    new file:   .dvcignore
```

- Commit the changes

```
$ git commit -m "initialized dvc"
```

DVC initialization creates a `.dvc` directory with configuration files and cache locations. This directory is automatically staged with `git add`, allowing easy commits.

.dvcignore File 01:57 - 02:46

- Similar to `.gitignore` file
  - Follows the same pattern
  - Outline files/directories that DVC will ignore
- Useful when tracking many data files not needed
  - Improves execution time of DVC operations

Similar to `.gitignore`, `.dvcignore` specifies files or directories for DVC to ignore. This improves efficiency, especially with large data sets.

Example 02:46 - 03:37

```
# .dvcignore
# Ignore all files in the 'data' directory
data/*

# But don't ignore 'data/data.csv'
!data/data.csv

# Ignore all .tmp files
*.tmp
```

Example rules for `.dvcignore`:

- `data/*` ignores all files in the data directory.
- `!data/data.csv` allows data.csv in the data directory to be tracked, overriding the previous rule.
- `.tmp` ignores all `.tmp` files in the project.

Checking Ignored Files 03:37 - 04:42

- Use `dvc check-ignore` command

```
$ dvc check-ignore data/file.txt
```

```
data/file.txt
```

- Use with `-d` flag to get details

```
$ dvc check-ignore -d data/file.txt
```

```
.dvcignore:3:data/*    data/file.txt
```

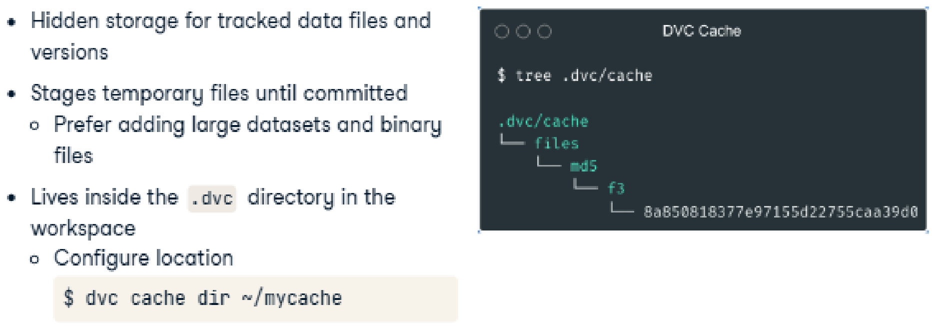
The `dvc check-ignore` command verifies if a file is ignored by DVC. Use the `-d` option to see the ignore rule causing a file to be excluded.

Summary 04:42 - 04:53

A quick reference of DVC commands discussed in this lesson is provided for use in exercises.

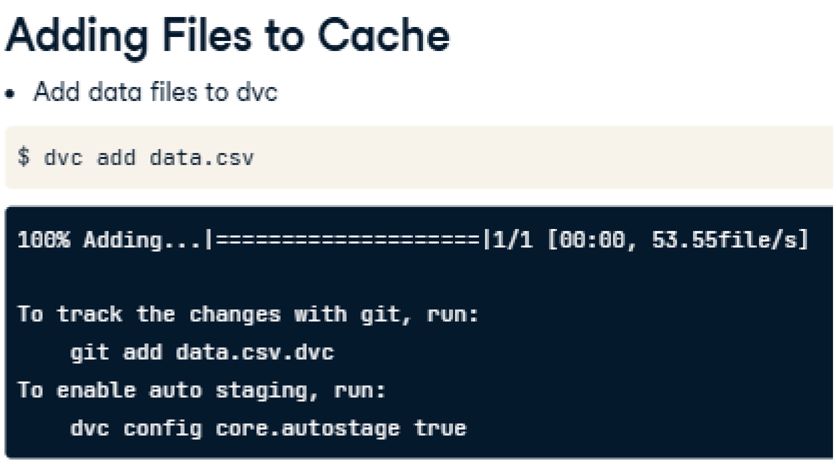
DVC Cache and Staging Files

DVC Cache 00:12 - 01:35



DVC cache provides hidden storage for files and directories tracked by DVC, aiding in efficient versioning of large datasets and models. This cache typically resides in the `.dvc` directory but can be reconfigured using `dvc cache dir`. DVC cache stages files temporarily before they're committed to DVC, making it ideal for large datasets and non-text files.

Adding Files to Cache 01:35 - 02:18



Files are added to the DVC cache with `dvc add`. This command creates a metadata file (e.g., `data.csv.dvc`) that stores information about the tracked file. The `.dvc` files keep track of data changes, and by versioning these files with Git, we maintain a lightweight repository.

.dvc Files 02:18 - 03:51

- Each DVC tracked file has its corresponding `.dvc` file
  - `data.csv -> data.csv.dvc`
- To version the data file, use `git commit -m "data.csv.dvc"`
- Content of `.dvc` files

```
outs:
- md5: f38a850818377e97155d22755caa39d0
  size: 16
  hash: md5
  path: data.csv
```

Each `.dvc` file corresponds to a tracked data file. Inside a `.dvc` file:

- `md5`: Hash of the file, used to detect changes.
- `size`: File size in bytes.
- `hash`: Specifies the hash function (MD5).
- `path`: Path to the tracked data file.

Interaction with DVC Cache 03:51 - 04:57

- The path of cache file uses the MD5 value

```
$ find .dvc/cache -type f
```

```
.dvc/cache/f3/8a850818377e97155d22755caa39d0
```

- Compute MD5 of dataset

```
$ md5 data.csv
```

```
MD5 (data.csv) = f38a850818377e97155d22755caa39d0
```

- Use `dvc add -v` for verbose output

```
DVC Add Verbose

$ dvc add -v data.csv

2024-01-24 08:15:52,749 DEBUG: v3.40.1 (pip), CPython 3.9.16 on macOS-14.2.1-arm64-arm-64bit
2024-01-24 08:15:52,749 DEBUG: command: /Users/<username>/miniconda3/envs/ml-cicd/bin/dvc add -v data.csv
2024-01-24 08:15:52,932 DEBUG: Preparing to transfer data from 'memory://dvc-staging-md5/d71db5f54bc5b422d456b87e0d1c2de1576ee960ac6011da388188d5edfd826a' to '/Users/<username>/development/dvc-test/.dvc/cache/files/md5'

2024-01-24 08:15:52,932 DEBUG: Preparing to collect status from '/Users/<username>/development/dvc-test/.dvc/cache/files/md5'
2024-01-24 08:15:52,932 DEBUG: Collecting status from '/Users/<username>/development/dvc-test/.dvc/cache/files/md5'
2024-01-24 08:15:52,932 DEBUG: Preparing to collect status from 'memory://dvc-staging-md5/d71db5f54bc5b422d456b87e0d1c2de1576ee960ac6011da388188d5edfd826a'

2024-01-24 08:15:52,933 DEBUG: Removing '/Users/<username>/development/dvc-test/.AlspXky4WBJrt2KnThTN2g.tmp'
2024-01-24 08:15:52,934 DEBUG: Removing '/Users/<username>/development/dvc-test/.AlspXky4WBJrt2KnThTN2g.tmp'
2024-01-24 08:15:52,934 DEBUG: Removing '/Users/<username>/development/dvc-test/.dvc/cache/files/md5/.k3hsk3ziCvWMZN2ST_CrnA.tmp'
2024-01-24 08:15:52,934 DEBUG: Removing '/Users/<username>/development/dvc-test/data.csv'
2024-01-24 08:15:52,936 DEBUG: Saving information to 'data.csv.dvc'.
100% Adding...|████████████████████████████████████████|1/1 [00:00, 53.55file/s]

To track the changes with git, run:

    git add data.csv.dvc
```

Using `dvc add`, data is moved to the cache and linked back to the workspace. You can confirm this with `find .dvc/cache`. Adding the `-v` flag to `dvc add` provides detailed output showing the file being copied to the cache and information saved to the `.dvc` file.

Removing from and Cleaning Cache 04:57 - 05:32

- Remove added files using `dvc remove`

```
$ dvc remove data.csv.dvc
```

- To clear the cache, use the `dvc gc`
  - Use with `-w` flag to remove workspace cache

```
$ dvc gc -w
```

```
WARNING: This will remove all cache except items used in the workspace of the current repo.
Are you sure you want to proceed? [y/n]: y
Removed 1 objects from repo cache.
```

To remove files, use `dvc remove` on the `.dvc` files, which removes DVC's reference. To delete these files from the cache, use `dvc gc -w`, which cleans up unlinked files after confirmation.

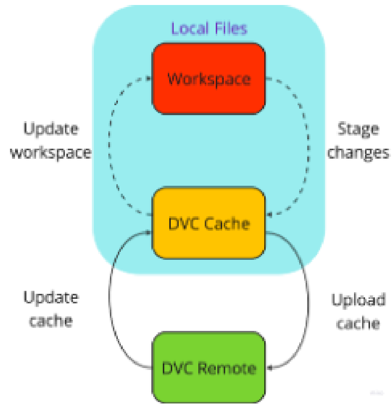
Summary

- DVC cache stages data files before commit
- Configure cache location
  - `dvc cache dir ~/mycache`
- Add files to cache
  - `dvc add data.csv`
  - Creates a `.dvc` file with metadata
- Remove added files `dvc remove data.csv.dvc`
  - Clean workspace cache with `dvc gc -w`

Configuring DVC Remotes

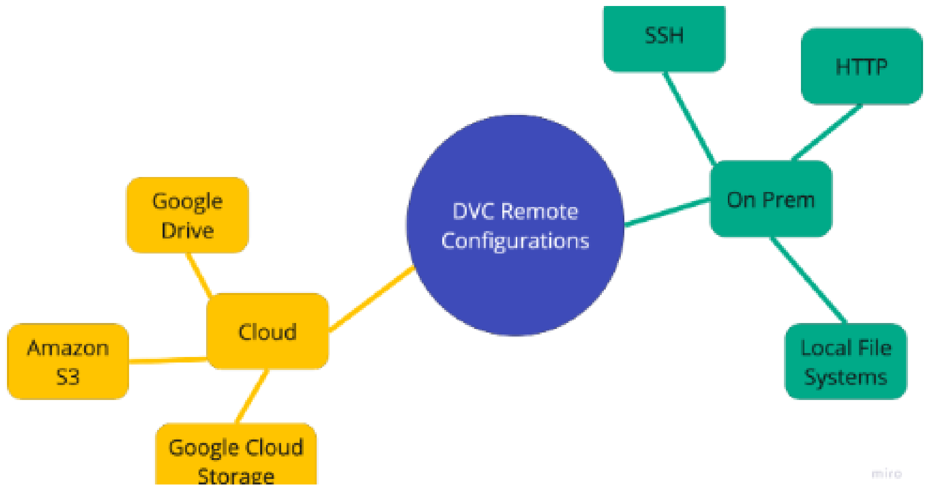
The Need for DVC Remotes 01:00 - 02:13

- DVC Remotes: Location for Data Storage
- Similar to Git remotes, but for *cached* data
- Benefits of using remotes
  - Synchronize large files and directories
  - Centralize or distribute data storage
  - Save local space



DVC remotes act as centralized storage for data and ML models, much like Git remotes but for large data. Services like GitHub have storage limits, which DVC remotes help address by allowing the syncing of large files, establishing a shared repository, and saving local storage.

Supported Storage Types 02:13 - 02:28



DVC remotes support a variety of storage types, including cloud providers like AWS, GCP, Azure, and on-prem storage through SSH and HTTP.

Setting up Remotes 02:28 - 03:25

- Setting remotes
  - `dvc remote add <name> <location>`
- S3 bucket

```
$ dvc remote add s3_remote \
    s3://mys3bucket
```

- DVC config changes

```
['remote "s3_remote"']
    url = s3://mys3bucket
```

- GCP bucket

```
$ dvc remote add gcp_remote \
    gs://myGCPbucket
```

- Azure

```
$ dvc remote add azure_remote \
    azure://mycontainer/path
```

To configure storage locations, use `dvc remote add` with a reference name and location. For instance, to set up an S3 bucket, run `dvc remote add s3_remote mys3bucket`. This command updates the `.dvc/config` file, where DVC saves these configurations. Common cloud providers' configurations are often automatically detected.

Local Remotes 03:25 - 04:31

- Local remotes are used for rapid prototyping
- Use system directories or Network Attached Storage

```
$ dvc remote add mylocalremote /tmp/dvc
```

- Set default remotes with `-d` flag

```
$ dvc remote add -d mylocalremote /tmp/dvc
```

- Default remote assigned in the `core` section of `.dvc/config`

```
[core]
remote = mylocalremote
```

Local remotes, useful for testing or when external storage isn't needed, include directories, network-attached storage (NAS), and external devices. Use the `-d` flag with `dvc remote add` to set a default remote, which will save it in the DVC config.

Listing Remotes 04:31 - 05:11

- Listing remotes

```
$ dvc remote list
```

```
s3_remote    s3://mys3bucket
local_remote /tmp/dvcremote
```

- Reads from `.dvc/config`

```
['remote "s3_remote"']
    url = s3://mys3bucket
['remote "local_remote"']
    url = /tmp/dvcremote
```

Run `dvc remote list` to see all configured remotes. This command reads the `.dvc/config` file and displays the remotes, including names, paths, and configuration settings.

Modifying Remote Configuration 05:11 - 05:30

- Customizations can be done with `dvc remote modify`

```
$ dvc remote modify s3_remote connect_timeout 300
```

- DVC config file change

```
['remote "s3_remote"']
    url = s3://mys3bucket
    connect_timeout = 300
```

If you need to adjust remote settings, use `dvc remote modify`. This command edits the `.dvc/config` file, where changes are saved.

Summary 05:30 - 06:00

DVC remotes store data and ML models externally, configured using `dvc remote add` with optional `-d` for default. List and modify remotes with `dvc remote list` and `dvc remote modify`, respectively.

Interacting with DVC Remotes

Uploading and Retrieving Data 00:07 - 01:07

- Push entire cache

```
$ dvc push
```

- Update the cache without changing workspace contents

```
$ dvc fetch
```

- Override default remote with `-r` flag

```
$ dvc push -r aws_remote data.csv
```

With configured remotes, we can transfer data between the local cache and remote storage. Use `dvc push` to upload data to the remote and `dvc pull` to retrieve data from the remote. These commands are essential for sharing data across environments and maintaining versions of datasets, models, and DVC metrics. You can specify specific files or directories to target.

Selective Data Transfer 01:07 - 02:14

Without a specified target, `dvc push` transfers all cached contents by default. To update the cache without changing the workspace, use `dvc fetch`, useful for loading data across multiple project branches or tags. The `-r` flag specifies a particular remote if multiple are configured. If unspecified, interactions occur with the default remote.

Similarities with Git 02:14 - 03:17

`dvc pull`

- **Function:** Downloads remote data to DVC workspace
- **Use Case:** Large datasets or model artifacts

`dvc push`

- **Function:** Uploads data to remote storage
- **Use Case:** Sharing or storing data artifacts

`git pull`

- **Function:** Fetch/Merge data remote Git repo
- **Use Case:** Local branch in sync with remote

`git push`

- **Function:** Uploads local changes to remote
- **Use Case:** Share changes to Git remote

While similar to Git, `dvc push` and `dvc pull` manage data files rather than code. `dvc pull` retrieves large datasets or models from remote storage, whereas `git pull` fetches code commits from a Git repository. Similarly, `dvc push` uploads data, while `git push` sends code commits.

Versioning Data 03:17 - 04:15

- `.dvc` is tracked by Git, not DVC
- Leverage this to checkout specific version of data file
- Checkout `.dvc` file

```
$ git checkout <commit_hash|tag|branch>
```

- Retrieve data with MD5 specified in `.dvc` file

```
$ dvc checkout <target>
```

To manage dataset versions, each data file is linked to a `.dvc` metadata file tracked by Git. Use `git checkout` to revert to a specific commit or branch, updating the `.dvc` file to the chosen version. Follow up with `dvc checkout` to retrieve the exact data version as specified by the MD5 in the `.dvc` file.

Tracking Data Changes 04:15 - 05:00

- Change data file contents, then add dataset changes

```
$ dvc add <target>
```

- Commit changed `.dvc` file to Git

```
$ git add <target>.dvc
$ git commit <target>.dvc \
    -m "Dataset updates"
```

- Push metadata to Git

```
$ git push origin main
```

- Upload changed data file

```
$ dvc push
```

For any data changes, use the following steps to sync with DVC and Git:

1. Stage the data change with `dvc add`, updating the `.dvc` file.
2. Track the `.dvc` file using `git add` and commit with `git commit`.
3. Push metadata changes to Git with `git push`.
4. Finally, push the modified data to the DVC remote with `dvc push`.

Code organization and refactoring

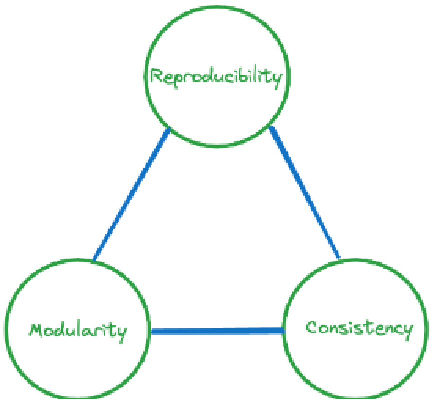
Prototyping vs Production Code 00:10 - 00:46

- Prototyping code allows rapid iteration
- But not suitable for production
  - Untested and prone to errors
  - Not modular with many repeated code blocks
  - Likely not reproducible

Prototyping tools like Jupyter Notebook allow quick iterations during model development, but they aren't suitable for production. Prototype code often lacks testing, contains repetitive blocks, and isn't reproducible across environments, making it unreliable for scalable use.

Features of Good Production Code 00:46 - 01:42

- **Reproducible:** recreate same outputs in different environments and time
- **Modular:** written as distinct, independent, and testable modules
- **Consistent:** Single source of truth for all parameters
  - A configuration/parameter file



Production-grade code should consistently reproduce results by replicating the original environment and data. It should be modular, with independent, testable components, and all configuration parameters (e.g., hyperparameters) should be stored in a central configuration file for easier tracking.

Configuration Files and YAML 01:42 - 02:48

- Files should be in supported format
  - YAML, JSON, TOML, Python
  - Default is `params.yaml`
- We'll work with YAML
  - **YAML Ain't Markup Language**
  - Allows a standard format to transfer data between languages or applications
  - Simple and clean format
  - Valid file extensions: `.yaml` or `.yml`

Parameters should be stored in configuration files, preferably in YAML, JSON, TOML, or Python formats. DVC uses `params.yaml` as its default configuration file. YAML is user-friendly and readable, with keys and values separated by colons and support for nested dictionaries, making it ideal for structuring parameters.

YAML Syntax 02:48 - 03:49

- Specify parameters as dictionaries
  - Keys and values separated by `:`
- Comments start with `#`
- Data types:
  - Integer, Floats, Strings
- Data structures:
  - Arrays
  - Nested Dictionaries

```yaml
# Key-value pairs
a: 1
b: 1.2
c: "String value"
```

```yaml
# Arrays
a: [1, 2.2, 3, 4.8]
b:
  - 5
  - "String value"
```

```yaml
# Nested dictionaries
a:
  b: "Some value"
  c: "Some other value"
```

YAML organizes data with key-value pairs, supports comments with the hash symbol, and recognizes multiple data types (e.g., integers, strings, arrays). Indentation is essential in YAML to define hierarchy, allowing for nested dictionaries that group related parameters effectively.

Example Configuration File 03:49 - 04:21

```yaml
# Data preprocessing paramters
preprocess:
  ...
  target_column: RainTomorrow
  categorical_features:
    - Location
    - WindGustDir
    - ...
# Model training/evaluation paramters
train_and_evaluate:
  rfc_params:
    n_estimators: 2
    ...
```

A YAML file can group parameters for data preprocessing and model training. For instance, the preprocessing section could define target and categorical columns, while the training section could specify model hyperparameters.

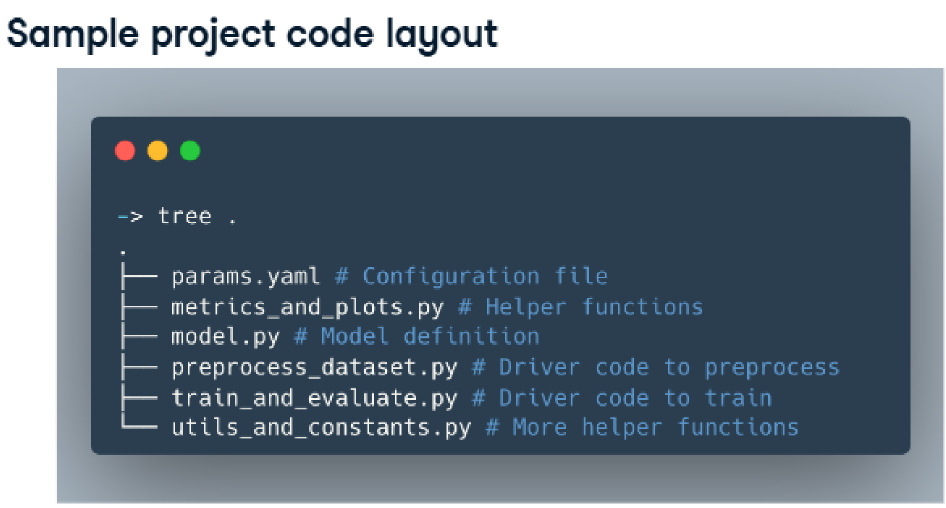
Example Modular Function 04:21 - 05:00

```python
# In model.py
def evaluate_model(model, X_test, y_test):
    """Evaluate a model on a test set and return metrics."""
    y_pred = model.predict(X_test)
    precision = precision_score(y_test, y_pred)
    ...
    return { "accuracy": accuracy, "precision": precision,
        "recall": recall, "f1_score": f1 }
```

```python
# In entry-point code (train_and_evaluate.py)
from model import evaluate_model
metrics = evaluate_model(model, X_test, y_test)
```

Modularizing code (e.g., in separate `.py` files) enhances reusability by allowing common functions to be imported as modules, reducing redundancy and errors.

Sample Project Code Layout 05:00 - 05:44



A well-organized project layout includes:

1. Configuration or parameter file.
2. Helper functions grouped in separate files.
3. Entry-point code files, each dedicated to a single workflow step (e.g., preprocessing or training).
4. Model definitions in a separate module.

Writing and visualizing DVC pipelines

DVC Pipelines

00:10 - 01:11

- Sequence of stages defining Machine Learning workflow and dependencies
  - Versioned and tracked with Git
- Defined in `dvc.yaml` file
  - Input data and scripts (`deps`)
  - Parameters (`params`)
  - Stage execution commands (`cmd`)
  - Output artifacts (`outs`)
    - Special data e.g. `metrics` and `plots`

A DVC pipeline is a sequence of stages that define workflows and dependencies for machine learning tasks, versioned and tracked with Git. The pipeline's stages are set up in the `dvc.yaml` file, with keys specifying dependencies (`deps`), parameters (`params`), commands (`cmd`), and outputs (`outs`). Special outputs like metrics and plots have dedicated keys.

Adding Preprocessing Stage

01:11 - 02:33

- Create stages using `dvc stage add`

```
dvc stage add \
-n preprocess \
-p params.yaml:preprocess \
-d raw_data.csv \
-d preprocess.py \
-o processed_data.csv \
python3 preprocess.py
```

```
stages:
  preprocess:
    cmd: python3 preprocess.py
    params:
      # Keys from params.yaml
      - params.yaml
        - preprocess
    deps:
    - preprocess.py
    - raw_data.csv
    outs:
    - processed_data.csv
```

We create pipeline stages using the `dvc stage add` command. For example, a preprocessing stage specifies the name (`-n`), parameters (`-p`), dependencies (`-d`), and outputs (`-o`). Parameters are specified by key from `params.yaml` , ensuring reproducibility. This stage setup organizes code, data, and parameters for consistent execution.

Adding Training and Evaluation Stage

02:33 - 03:35

- Add a training step using output from previous step

```
dvc stage add \
-n train_and_evaluate \
-p train_and_evaluate \
-d train_and_evaluate.py \
-d processed_data.csv \
-o plots.png \
-o metrics.json \
python3 train_and_evaluate.py
```

```
stages:
  train_and_evaluate:
    cmd: python3 train_and_evaluate.py
    params:
      # Skip specifying parameter file
      # Defaulted to params.yaml
      - train_and_evaluate
    deps:
    - processed_data.csv
    - train_and_evaluate.py
    outs:
    - plots.png
    - metrics.json
```

Similar to preprocessing, training and evaluation stages can be added, connecting outputs of one stage to inputs of the next. DVC automatically defaults parameters to `params.yaml`, creating a Directed Acyclic Graph (DAG) that outlines dependencies between stages.

Updating Stages

03:35 - 03:52

- Running `dvc stage add` multiple times

```
ERROR: Stage 'train_and_evaluate'
already exists in 'dvc.yaml'.
Use '--force' to overwrite.
```

- Use `dvc stage add --force`

```
dvc stage add --force \
-n train_and_evaluate \
-p train_and_evaluate \
-d train_and_evaluate.py \
-d processed_data.csv \
-o plots.png \
-o metrics.json \
python3 train_and_evaluate.py
```

Using `dvc stage add` to modify a stage causes errors; instead, use `--force` to overwrite existing stages in the `dvc.yaml`.

Visualizing DVC Pipelines

03:52 - 04:29

```
# Print DAG on terminal
dvc dag
```

```
# Display DAG up to a certain step
dvc dag <target>
```

```
+------------+
| preprocess |
+------------+
       *
       *
       *
+--------------------+
| train_and_evaluate |
+--------------------+
```

```
# Display step outputs as nodes
dvc dag --outs
```

```
          +-------------------------------+
          | processed_dataset/weather.csv |
          +-------------------------------+
                ***              ***
             ***                    ***
           **                          **
+---------------+                  +-----------+
| metrics.json  |                  | plots.png |
+---------------+                  +-----------+
```

The pipeline's dependency graph can be visualized with `dvc dag`, presenting stages in sequence from top to bottom. To focus on specific sections, add a target argument.

Visualizing Pipeline Outputs

04:29 - 04:59

The `dvc dag --outs` command displays a DAG focused on outputs, highlighting data flow through stages, which can clarify workflow structure.

Generating DOT Files

04:59 - 05:29

Using `--dot` generates DOT scripts, which create visualizations for documentation. DOT files are useful for showing dependencies, but are beyond the scope of this course.

Executing DVC pipelines

Reproducing a Pipeline

00:22 - 01:16

Reproducing a pipeline

- Reproduce the pipeline using `dvc repro`

```
$ dvc repro
```

```
Running stage 'preprocess':
> python preprocess.py
Running stage 'train_and_evaluate':
> python train_and_evaluate.py
Updating lock file 'dvc.lock'
```

- A state file `dvc.lock` is generated
  - Similar to `.dvc` file, captures MD5 hashes

```
$ git add dvc.lock && git commit -m "first pipeline run"
```

With `dvc.yaml`, we can rerun the pipeline using the `dvc repro` command to process new data. Each stage runs sequentially, creating a `dvc.lock` file that captures the pipeline's state. It's best practice to commit the `dvc.lock` file to Git to document the current state.

Using Cached Results

01:16 - 01:38

- Using cached results to speed up iteration

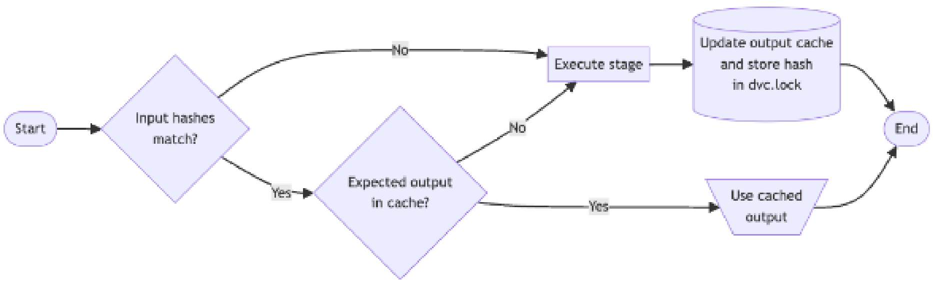
```
$ dvc repro
```

```
Stage 'preprocess' didn't change, skipping
Running stage 'train_and_evaluate' with command: ...
```

If no changes are detected in certain stages, DVC will use cached results and skip those steps, saving time in complex pipelines.

Stage Caching in DVC

01:38 - 02:43



DVC uses md5 checksums to track dependencies, code, and outputs. If changes are detected in any of these, DVC will rerun the stage. Otherwise, it will use cached outputs, optimizing the workflow by avoiding redundant computations.

Dry Running a Pipeline

02:43 - 03:16

- Use `--dry` flag to only print commands without running the pipeline

```
$ dvc repro --dry
```

```
Running stage 'preprocess':
> python3 preprocess_dataset.py

Running stage 'train_and_evaluate':
> python3 train_and_evaluate.py
```

The `--dry` flag with `dvc repro` shows the commands to be executed without actually running them. This preview is helpful for verifying stage execution.

Additional Arguments

03:16 - 04:46

- Running specific files `dvc repro linear/dvc.yaml`
  - Multiple `dvc.yaml` in one folder are not allowed
- Running specific stages `dvc repro <target_stage>`
  - This will *also* run upstream dependencies
- Force run a pipeline/stage `dvc repro -f`

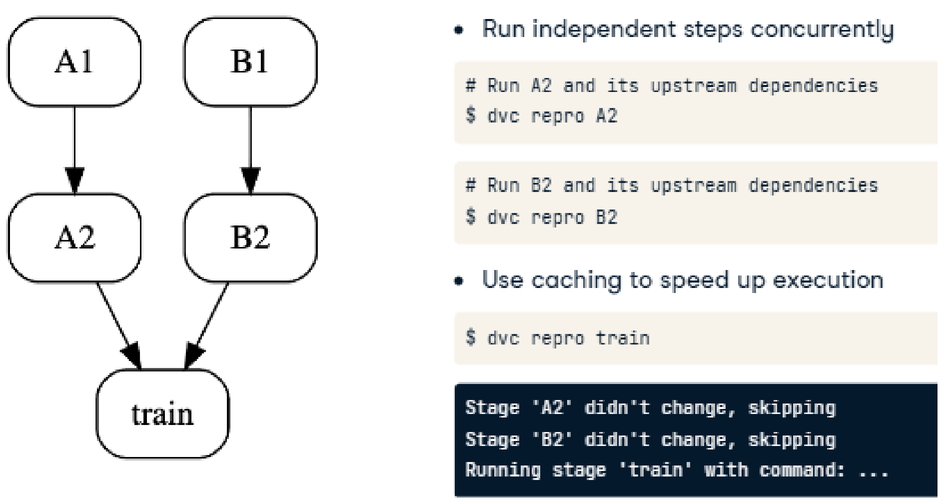
DVC provides flexibility with flags for `dvc repro`:

- Specify a particular `dvc.yaml` file by path (one per folder).

- Provide a stage name to run it along with its dependencies.
- Use `f` to force rerun all stages, even if cached.
- Avoid caching results for exploration by not storing outputs temporarily; finalize with `dvc commit`.

Parallel Stage Execution

04:46 - 05:34

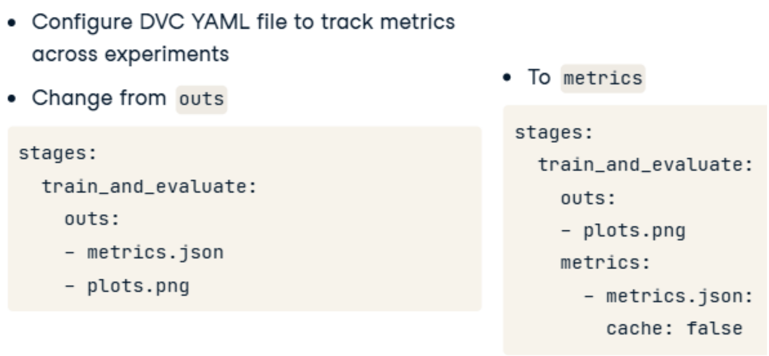


To run stages concurrently, execute `dvc repro` in multiple terminals. For example, in a pipeline with parallel branches, run `dvc repro A2` and `dvc repro B2` at the same time, then execute `dvc repro train` to complete the final stage, utilizing stage caching.

Evaluation: Metrics and plots in DVC

Metrics: Changes in `dvc.yaml`

00:11 - 00:57



DVC supports monitoring model performance metrics, visualizing them through graphs, and comparing results across experiments, aiding in selecting the best model. To enable metric tracking, add the `metrics.json` file under the `metrics` key in the `dvc.yaml` file, instead of `outs`. Set `cache: false` to store metrics with Git, as metrics files are typically small and text-based.

Printing DVC Metrics

00:57 - 01:06

Use `dvc metrics show` to display current metrics on the terminal.

Compare Metrics Across Runs

01:06 - 01:41

- Change a hyperparameter and rerun `dvc repro`

```
$ dvc metrics diff
```

```
Path          Metric     HEAD    workspace  Change
metrics.json  accuracy   0.947   0.9995     0.0525
metrics.json  f1_score   0.8656  0.9989     0.1333
metrics.json  precision  0.988   0.9993     0.0113
metrics.json  recall     0.7702  0.9986     0.2284
```

DVC allows metric comparison across experiments. After committing an initial experiment, make hyperparameter changes, run `dvc repro`, and execute `dvc metrics diff` to see improvements in model performance.

Plots: Changes in `dvc.yaml`

01:41 - 02:25

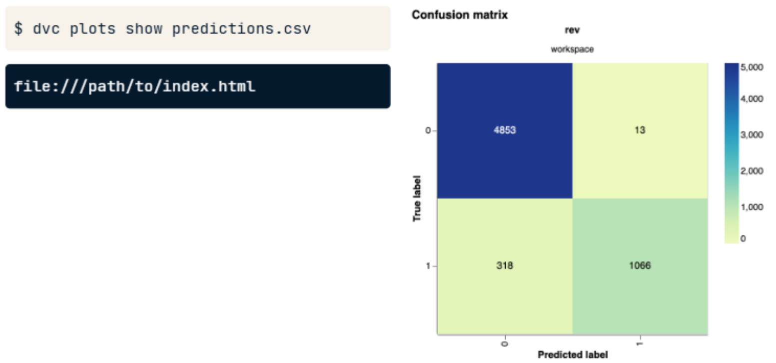
```
stages:
  train_and_evaluate:
    ...
    plots:
    - predictions.csv: # Name of file containing predictions
        template: confusion # Style of plot
        x: predicted_label # X-axis column name in csv file
        y: true_label # Y-axis column name in csv file
        x_label: 'Predicted label'
        y_label: 'True label'
        title: Confusion matrix
        cache: false # Save in Git
```

https://dvc.org/doc/user-guide/experiment-management/visualizing-plots#plot-templates-data-series-only

Instead of tracking plot images, DVC tracks data files used for generating plots. Define these under the `plots` key in `dvc.yaml`, specifying a template for plot styling. Organize data by column names, axis labels, and title. Set `cache: false` to track the plot file via Git.

Printing DVC Plots to File

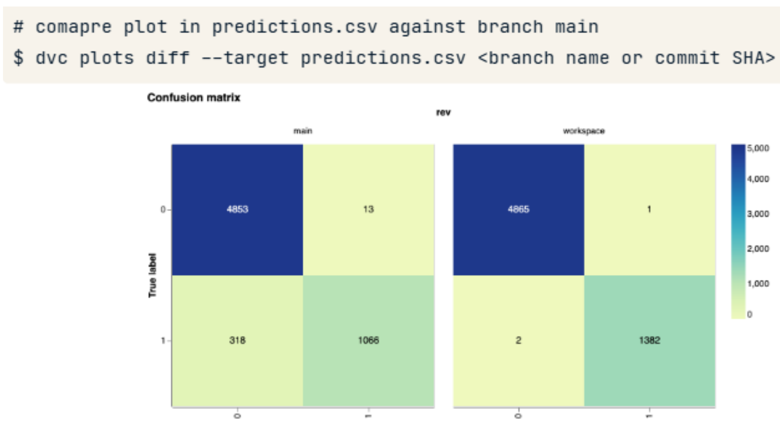
02:25 - 02:54



To show plots, use `dvc plots show`, which will output an HTML file with an interactive plot based on the target in `dvc.yaml`.

Comparing DVC Plots

02:54 - 03:29



To compare plots across branches or commits, use `dvc plots diff`, specifying the target from `dvc.yaml` and the branch or commit SHA. This enables side-by-side plot comparisons between branches or commits.