Hands-on activity: Data Versioning with DVC

Data Version Control (DVC) is an open-source tool designed to manage data and model versioning with a workflow that is similar to using Git for code. This tutorial will guide you through setting up DVC for versioning data.

Objectives:

To gain practical experience in using DVC for versioning data and models in machine learning projects, emphasizing reproducibility and version control fundamentals.

Step 1: Initialize Your Project

Scenario:
Imagine a data science team beginning a new project to develop a machine learning model for image recognition. The project will involve experimenting with various models and datasets. The team needs a way to track and manage different versions of their datasets and models effectively. They decide to use DVC to handle large datasets and model files, and Git to manage code changes. The first step is to initialize DVC in their project repository so that all subsequent data-related operations are managed through DVC.

- Create a New Branch for the Tutorial:
  ```
  git checkout -b tutorial
  ```

  This command helps manage different development streams within the same project. Creating a branch in Git allows developers to work on features or experiments in isolated environments. Branching is crucial for maintaining the main project (often called 'master' or 'main') stable while development continues on the other parts of the project.

- Initialize DVC:
  ```
  dvc init
  ```

  dvc init sets up the current directory for DVC usage by creating a .dvc directory and several configuration files. This is similar to git init, which initializes a Git repository. The .dvc directory includes a configuration file for the project settings and a .gitignore file to prevent Git from tracking files that should only be handled by DVC, like the data files themselves.

- Check initialization results:
  ```
  ls .dvc/
  git status
  ```

Listing the .dvc directory contents and checking the Git status helps visualize what DVC added to the project. It is essential to understand that while DVC tracks the data, it uses Git to track the history of the DVC files, which describes how to reproduce the data files.

- Commit DVC setup to Git:

```
git add .dvc
git commit -m "DVC init"
```

These commands add the DVC configuration files to Git's version control. This step is critical because it ensures that changes to the DVC setup are tracked alongside the project's source code, allowing for coordinated version control of code and data.

Step 2: Version a Single File

Scenario:
The team has decided to start with a publicly available dataset for initial model training. They download a baseline dataset, data.xml, containing sample images annotated with labels. To ensure that every team member can reproduce the project setup without manual downloads and to track changes in the dataset, they use DVC to add and version this file. This allows any modifications to the dataset to be detected and versioned, ensuring that the entire team can work with consistent and correct versions of the data.

- Download and Add a File Under DVC Control:

```
dvc get https://github.com/iterative/dataset-registry get-started/data.xml -o
data/data.xml
dvc add data/data.xml -v
```

dvc get is a command to download data from a remote DVC storage to the local environment, which is useful for fetching datasets without manually downloading them. dvc add starts tracking the file with DVC, creating a DVC file (data.xml.dvc), which acts like a pointer to the data file and stores metadata such as the file hash (checksum). This checksum is used to detect changes in the data at later stages.

- Commit the file version to Git:

```
git add data/.gitignore data/data.xml.dvc
git commit -m "Add raw data"
```

After adding a file to DVC, the command git add is used to stage changes, including the .dvc file and changes to .gitignore (which now ignores the actual data file). Committing these changes to Git ensures that the project's data tracking is maintained alongside the source code.

Step 3: Version a Directory

Scenario:
As the project progresses, the team expands their dataset by including a new collection of images featuring cats and dogs. This data is stored in a directory structure separating images by label. To manage this dataset effectively, track changes, and ensure reproducibility across all team members' environments, they decided to version the entire directory using DVC. This step helps in handling directories containing multiple files or subdirectories, making it simpler to manage complex datasets.

- Checkout a New Branch and Manage a Dataset Directory:
  ```
  git checkout -b cats-dogs-v1
  ```

  This command branches off from the main project to allow separate management of a specific version of the dataset. It's useful for experimenting with data without affecting the main or other branches.

- Fetch and add the directory:
  ```
  dvc get --rev cats-dogs-v1 https://github.com/iterative/dataset-registry
  use-cases/cats-dogs -o datadir
  dvc add datadir
  ```

  dvc get --rev fetches a specific version of a dataset from a DVC repository. dvc add tracks the entire directory (datadir), creating a .dvc file for the directory which includes references to all files within. This process is crucial for managing datasets with multiple files, ensuring all components of the dataset are versioned together.

- Commit the directory to Git:
  ```
  git add .gitignore datadir.dvc
  git commit -m "Add datadir"
  git tag -a cats-dogs-v1 -m "Create data version v1"
  ```

  Similar to tracking individual files, tracking a directory involves committing the .dvc file that describes it. Tagging the commit with cats-dogs-v1 marks this point in the project history as the version 1 release of the dataset, which is critical for later reference and rollback if necessary.

Step 4: Track and Update Data Versions

Scenario:
After some initial experiments, the team improves the dataset by adding more images and refining the labels for better model accuracy. They need to update the versioned dataset without losing the ability to revert to the original version if needed. They create a new branch, update the dataset, and use DVC to track these changes. This allows them to maintain

multiple versions of the dataset, each corresponding to different stages of project development, enabling easy comparison and rollback if necessary.

- Creating a New Branch:
  `git checkout -b cats-dogs-v2`

  This command creates a new branch named cats-dogs-v2 and switches to it. In version control systems like Git, branches are used to develop features isolated from each other. The main idea here is to make changes to the dataset without affecting the current stable version of the project (likely on another branch such as master or main). Branching is crucial when you need to work on different versions of datasets or models simultaneously.

- Fetching a specific revision of the dataset:
  `dvc get --rev cats-dogs-v2 https://github.com/iterative/dataset-registry use-cases/cats-dogs -o datadir`

  dvc get --rev retrieves a specific version (cats-dogs-v2 revision in this case) of a dataset from a DVC repository and places it in the specified directory (datadir). The --rev option specifies the exact version of the dataset you want to work with. This step ensures that you are using the precise version of the dataset intended for this branch, which is essential for reproducibility and consistency across different stages of the project.

- Adding the updated dataset to DVC:
  `dvc add datadir`

  After downloading and placing the new dataset version in your project, the next step is to use dvc add to track the dataset with DVC. This command creates a .dvc file for the directory datadir. This .dvc file acts as a pointer to the data files and contains metadata about the dataset, including its checksum, which DVC uses to track changes in the data. If the dataset changes, DVC will detect these changes through the altered checksums.

- Staging and Committing Changes to Git:
  `git add datadir.dvc`
  `git commit -m "Change data"`

  These commands add the new .dvc file to Git's staging area and then commit it to the Git repository. The commit message "Change data" indicates that the dataset has been updated. It’s important to commit these changes to maintain consistency between the data being used and the version control history. This makes sure that every change to the data can be traced through Git's commit history, which is crucial for going back to previous states if necessary.

- Tagging the Commit in Git:

```
git tag -a cats-dogs-v2 -m "Create data version v2"
```

Tagging in Git is used to mark specific points in the repository's history as important. Typically, this is used for releases. In the context of data versioning, tagging helps you mark the release of a new dataset version. The tag cats-dogs-v2 allows you to easily switch between different versions of the dataset for comparisons or regression tests. Tags are immutable references that do not change as additional commits are made, which makes them ideal for marking version releases.

Step 5: Switch Between Data Versions

Scenario:
During model training, the team wants to test the model's performance across different versions of the dataset to understand how changes in data affect the model's accuracy. They need to switch between different dataset versions easily without manually replacing files. By using branches and DVC's checkout feature, they can quickly switch the entire working environment (both code and data) to a specific dataset version, facilitating efficient testing and evaluation of different data versions.

- Checkout Different Branches to Switch Data Versions:

```
git checkout tutorial
dvc checkout
```

Using git checkout to switch branches changes the project's code and DVC files to those stored in the branch. Running dvc checkout after switching branches updates the data files in the workspace to match the versions specified in the .dvc files currently checked out. This command is crucial for ensuring that the workspace reflects the correct versions of both code and data for the branch.

Step 6: Remote Storage and Sharing

Scenario:
The team is distributed across different locations, and they need a reliable way to share data and ensure that everyone is working with the same version of datasets and models. They set up a DVC remote storage on a shared server where everyone can push to and pull from. This centralizes the data storage, making it accessible to all team members regardless of their location. This setup is crucial for collaborative projects where consistency and synchronization of data state are critical for successful project outcomes.

- Create a directory for DVC to use as remote storage:

```
mkdir -p /tmp/dvc
dvc remote add -d local /tmp/dvc
git add .dvc/config
git commit -m "Add remote storage"
```

```
dvc push -v
```

Setting up a remote storage in DVC is analogous to setting up a remote in Git. It allows you to push and pull data to a centralized server, facilitating collaboration among team members. The command dvc push uploads data from the local cache to the remote storage, ensuring that all team members can access the same version of the data.

- Remove local data and simulate a fresh pull from remote:

```
rm -rf .dvc/cache
rm -rf datadir
dvc pull -v
```

This step simulates a scenario where you need to fetch data from the remote storage into a new or cleaned workspace. dvc pull retrieves data based on the current .dvc files. It is essential for restoring data after loss or when setting up new environments.