Complete MLOps DVC Revision

We have always built Machine Learning models in Jupyter notebooks which do not follow a linear or sequential approach in building up ML Pipelines. Notebook Environment is better approach where we have to perform more experimentation and is less tuned towards stitching the whole flow together because ML workflow in notebooks are non-linear.

ML Pipelines

Pipelines are a way to stitch together your ML workflow so that you can work in a sequential approach and work outside the notebook environment in a more script based approach which provides you with :

1. Code Modularity
2. Code Versioning
3. Versioning of Stage **Dependencies** and **Outputs**.

DVC helps you do that where you mention the stages of the pipeline in a `dvc.yaml` file which tracks the dependencies and the outputs of the pipeline where the output of one stage of the pipeline becomes the dependency of other stage.

Tip

Pipelines is a very strong concept in DVC where through pipeline we can do code, model and data versioning and also experimentation alongside of building the pipelines

Dependency - Dependency are those things in the stage which can trigger the re-execution of that stage. Any change in the dependency triggers a change in the stage and the following states connected to that stage. Dependencies can be:

1. Model Files
2. Code Files
3. Data Files
4. Other Artifacts

Outputs - The outputs of the stage are those which are automatically tracked by DVC and are given a unique hash value while tracking. There is no need to manually track the outputs as this is handled automatically by DVC.

Parameters - Parameters are mentioned in the `params.yaml` file. Parameters can be thought of as fine tuning dependencies or smaller dependency whose value change can result in the change of the output. They provide more **granular** approach to the tracking of dependency, where a code file being a major dependency can have 10 parameter values as their sub-dependencies and change in any of the value can trigger the re-execution of the pipeline stage.

Command - Command is the command that is run in the terminal which executes the stage or the python file having the code of the stage. No stage can be built or run without the command so it is the most important entity while building stages in DVC.
Command tells DVC what files to execute when we enter in the stage.

```
dvc stage add -n new_stage \
-d train.py -d data/processed/train.csv \
-p model.n_estimators, model.max_depth \
-o models/model.pkl
python train.py
```

Improving ML Pipelines

Your ML Pipelines can be improved through multiple ways but the most common approach that is used is through three techniques:

1. **Exception Handling** - Properly handle errors in your code
2. **Type Annotation** - Include type hinting in code for proper code linting and help in autocompletion of code.
3. **Logging** - Code logging is another key aspect that can help to improve the code quality in your ML Pipelines.

Exception Handling

Exception Handling is used at every place where we are working with files. Whenever we are opening and closing files and opening some connections it is always advised to do exception handling at that point.
It may be the case that the file is missing at the location or the filename written is not correct, or the path is not correct. In that case Exception Handling comes to the rescue.

Logging

We can log anything in python. Logging can greatly help the developer in Debugging of code instead of printing each and every statement.

Logs in the file can be maintained in those files throughout.

We can get the logger from python using the logging module

```
import logging
# create a logger
logger = logging.getLogger('my_logger')
```

```
# set level for the logger
logger.set_level(logger.INFO)
```

Logging can be done to two areas:

1. **Stream Handler** - Logging directly to console.
2. **File Handler** - Logging to files.

Once we create the file or the stream handler we set the formatter for the logger so that we know in which format the logs will be printed. The formatter is linked to the the handler object.

```
formatter = logging.formatter(format)
handler.setFormatter(formatter)
```

The file or stream handler when created and attached with a formatter should be added to the logger. One logger can have multiple file or stream handlers logging all the messages at the same time. The file/stream handlers can also be set with their specific levels of logging

```
# create stream handler
console_handler = logging.StreamHandler()
console_handler.set_level(logging.DEBUG)

# add handler to the logger
logger.addHandler(console_handler)
```

Version Control

Version control is used in ML to track different versions of the trifecta of **Code**, **Data** and **Models** together. The reason to do is that the end result of the ML pipeline which is our model is dependent on both the code and the data together and any change in either the code or the data can result in the change in the model.

The version control in ML is done by both `git` and `dvc`.

Git handles the versioning of the code whereas DVC helps in versioning of the Code, Data and Models, by using the power of git.
DVC in itself is not a version control tool, but it is tightly integrated with git and manages the version changes in Data, Models and Artifacts.

We can always initialize a repository with DVC using command

```
dvc init
```

This command creates a `./.dvc/` folder and this folder contains 3 files:

1. `.dvcignore` - Files that have to be ignored by DVC.
2. `.config` - The config file contains DVC related config information such as location and name of the remote location for DVC backups.
3. `.gitignore` - Ignore the tracking of `cache` and `temp` folder by git.

The `cache` folder is what maintains the local cache for the DVC versions of data and models.

Data Versioning

Through `dvc add` Command

Data/Model versioning of individual files can be done through DVC. Here when we use the `dvc add <file>` command , DVC creates a `.dvc` file which maintains the metadata around the file and also adds the data/model file into a `.gitignore` file so to tell git not to track the data/model file tracking from now on and this file is now tracked using DVC.

```
dvc add <file>
```

Here a few things are happening:

1. `data.dvc` file tracks the information or metadata about the data and DVC also generates a unique hash for this file which allows it to version track this file.
2. DVC maintains a local copy of the current version of the file in the `cache` folder.
3. The same file can also be pushed to a cloud remote which also maintains the versions of data.
4. The small metadata file created which is `.dvc` file can be git tracked along with the code.
5. Once the user uses `git checkout` to move to some previous version of the code, the data remains the same as the current version.
6. But along with the code now the `.dvc` file is also from the previous commit which contains the hash of the previous version of data.
7. Running command `dvc checkout` after `git checkout` causes DVC to verify the hash value from the metadata file and pull that version of data from the cache folder that matches the hash value from the `.dvc` file.

> **Through this using the power of `git` , DVC is able to track large data files by allowing git to track the small metadata files around the data along with the code commits.**

Staging Area

Staging Area is that area in `git` where the files go after they are added to git tracking by `git add` command. We keep those files in the staging area for which we want to take snapshot or create a commit.
The need for staging area arises because there might be a scenario

where we wish to commit only specific files in the project folder, for this reason there is a staging area.

The cache in DVC is the staging area for DVC.

Files move to DVC cache by using `dvc add` command.

To add remote to DVC we use the command

```
dvc remote add -d <remote name|alias> <remote path>
```

`-d` flag means the default remote repository.

> For DVC the remote can be any folder on the local computer or a remote repository on cloud platforms such as AWS S3, Google Drive etc.

```
outs:
  md5: hash
  size: 66432
  hash: md
  path: data.csv
```

Through Pipelines

Once you define stages in the `dvc.yml` file the command to run the complete pipeline is `dvc repro`.

Imp

Pipelines are the go to technique in DVC for all the **pipelines, versioning and experimentation tracking** needs.

When a pipeline is run in DVC it creates a `dvc.lock` file. This file is also versioned along with the code through git tracking and this file contains the stages of the pipelines, along with the dependency and parameters values.

The `dvc.lock` file maintains the Hashes of all the dependencies and the outs of the pipeline stages.

Through this `dvc.lock` file DVC is able to validate the changes in either the:

1. **Dependencies** - Leads to running of the stage again if any of the code, model or data dependencies of the stage changes.
2. **Parameters** - These are granular dependencies of the stage and there values is also maintained in the lock file.
3. **Outs** - These files/directories are the version tracked outputs of the pipeline stage. Any change in the stage dependencies or parameters leads to change in the outs which can result in re-running of the downstream stages that depends on the output of the current stage.

It is worthy to note that the `dvc.lock` file is also called as the state of the pipeline

```
schema: '2.0'
stages:
  prepare:
    cmd: python src/prepare.py data/data.xml
    deps:
      - path: data/data.xml
        md5: 22a1a2931c8370d3aeedd7183606fd7f
        size: 14445097
      - path: src/prepare.py
        md5: f09ea0c15980b43010257ccb9f0055e2
        size: 1576
    params:
      params.yaml:
        prepare.seed: 20170428
        prepare.split: 0.2
    outs:
      - path: data/prepared
        md5: 153aad06d376b6595932470e459ef42a.dir
        size: 8437363
        nfiles: 2
```

Experimentation Tracking

Through DVC Live

Experimentation Tracking is also handled by DVC through its `DVC Live` functionality. Here we can log individual metrics, parameters and artifacts into the `/dvclive/` folder. This folder contains the current state of the workspace and also holds the `metrics.json` having all the metrics logged and `params.yaml` files which has all the parameters logged.

Once the parameters and metrics have been logged, we can run the file and it will log the experiment.

Experimentation Tracking in DVC

The experimentation tracking of individual metrics and parameters happens in DVC whenever we run the logging code, it generates a new experiment, similar to what happens in MLFlow.

We can compare between the diff results through command

```
dvc exp diff <exp 1> <exp 2>
```

To apply some experiment to the current workspace the command used is:

```
dvc exp apply <exp name>
```

This restores the code, files, data and model artifacts into the current workspace for the experiment.
Specifically, `dvc exp apply` restores any files or directories which exist in the experiment, to their exact states from the experiment. This includes files tracked both with DVC and Git: code, raw data, parameters, metrics, resulting artifacts, etc.

The main use of the apply command is to set the best experiment to the current workspace and then commit the best experiment through `git`.

> **The philosophy of DVC behind experiment tracking is that while conducting experiments the code, data and models should be tracked by DVC for all the experiments but should not clutter the git commit history. Only the best experiment should be committed to git**

Through Pipelines

Everything is tracked automatically through DVC pipelines. The metrics in the evaluation stage mentioned in the `dvc.yaml` file and the parameter values in the `params.yaml` are tracked for each of the experiment done, when experimentation tracking is done through DVC Pipelines.

To run an experiment when done through pipelines is :

```
dvc exp run
```

Executes and tracks experiments in your repository without polluting it with unnecessary Git commits, branches, directories, etc.

Experimentation tracking through pipelines is easy and automatically managed and is a preferred way to do experiment tracking.