Get Started: Data Versioning

How cool would it be to make Git handle arbitrarily large files and directories with the same performance that you get with small code files? Imagine doing a `git clone` and seeing data files and machine learning models in the workspace. Or switching to a different version of a 100Gb file in less than a second with a `git checkout`.

The foundation of DVC consists of a few commands you can run along with `git` to track large files, directories, or ML model files. Think "Git for data". Read on or watch our video to learn about versioning data with DVC!

To start tracking a file or directory, use `dvc add`:

⚙ Expand to get an example dataset.

Having initialized a project in the previous section, we can get the data file (which we'll be using later) like this:

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

We use the fancy `dvc get` command to jump ahead a bit and show how a Git repo becomes a source for datasets or models — what we call a "data/model registry". `dvc get` can download any file or directory tracked in a DVC repository. It's like `wget`, but for DVC or Git repos. In this case we download the latest version of the `data.xml` file from the dataset registry repo as the data source.

```
$ dvc add data/data.xml
```

DVC stores information about the added file (or a directory) in a special `.dvc` file named `data/data.xml.dvc` — a small text file with a human-readable format. This metadata file is a placeholder for the original data, and can be easily versioned like source code with Git:

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

The original data, meanwhile, is listed in `.gitignore`.

💡 Expand to see what happens under the hood.

`dvc add` moved the data to the project's cache, and linked it back to the workspace.

```
$ tree .dvc/cache
../.dvc/cache
└── a3
    └── 04afb96060aad90176268345e10355
```

The hash value of the `data.xml` file we just added (`a304afb...`) determines the cache path shown above. And if you check `data/data.xml.dvc`, you will find it there too:

```
outs:
  - md5: a304afb96060aad90176268345e10355
    path: data.xml
```

Storing and sharing

You can upload DVC-tracked data or model files with `dvc push`, so they're safely stored remotely. This also means they can be retrieved on other environments later with `dvc pull`. First, we need to set up a remote storage location:

```
$ dvc remote add -d storage s3://mybucket/dvcstore
$ git add .dvc/config
$ git commit -m "Configure remote storage"
```

> DVC supports many remote storage types, including Amazon S3, SSH, Google Drive, Azure Blob Storage, and HDFS. See `dvc remote add` for more details and examples.

⚙ Expand to set up remote storage.

DVC remotes let you store a copy of the data tracked by DVC outside of the local cache (usually a cloud storage service). For simplicity, let's set up a *local remote*:

```
$ mkdir -p /tmp/dvcstore
$ dvc remote add -d myremote /tmp/dvcstore
$ git commit .dvc/config -m "Configure local remote"
```

> While the term "local remote" may seem contradictory, it doesn't have to be. The "local" part refers to the type of location: another directory in the file system. "Remote" is what we call storage for DVC projects. It's essentially a local data backup.

```
$ dvc push
```

Usually, we also want to `git commit` and `git push` the corresponding `.dvc` files.

💡 Expand to see what happens under the hood.

`dvc push` copied the data cached locally to the remote storage we set up earlier. You can check that the data has been stored in the DVC remote with:

```
$ ls -R /tmp/dvcstore
/tmp/dvcstore/:
a3

/tmp/dvcstore/a3:
04afb96060aad90176268345e10355
```

Retrieving

Having DVC-tracked data and models stored remotely, it can be downloaded when needed in other copies of this project with `dvc pull`. Usually, we run it after `git clone` and `git pull`.

⚙️ Expand to delete locally cached data.

If you've run `dvc push`, you can delete the cache (`.dvc/cache`) and `data/data.xml` to experiment with `dvc pull`:

```
$ rm -rf .dvc/cache
$ rm -f data/data.xml
```

```
$ dvc pull
```

> 📖 See also Sharing Data and Model Files for more on basic collaboration workflows.

Making changes

When you make a change to a file or directory, run `dvc add` again to track the latest version:

⚙️ Expand to make some changes.

Let's say we obtained more data from some external source. We can pretend this is the case by doubling the dataset:

```
$ cp data/data.xml /tmp/data.xml
$ cat /tmp/data.xml >> data/data.xml
```

```
$ dvc add data/data.xml
```

Usually you would also run `git commit` and `dvc push` to save the changes:

```
$ git commit data/data.xml.dvc -m "Dataset updates"
$ dvc push
```

Switching between versions

The regular workflow is to use `git checkout` first (to switch a branch or checkout a `.dvc` file version) and then run `dvc checkout` to sync data:

```
$ git checkout <...>
$ dvc checkout
```

⚙️ Expand to get the previous version of the dataset.

Let's go back to the original version of the data:

```
$ git checkout HEAD~1 data/data.xml.dvc
$ dvc checkout
```

Let's commit it (no need to do `dvc push` this time since this original version of the dataset was already saved):

```
$ git commit data/data.xml.dvc -m "Revert dataset updates"
```

Yes, DVC is technically not even a version control system! `.dvc` file contents define data file versions. Git itself provides the version control. DVC in turn creates these `.dvc` files, updates them, and synchronizes DVC-tracked data in the workspace efficiently to match them.

Large datasets versioning

In cases where you process very large datasets, you need an efficient mechanism (in terms of space and performance) to share a lot of data, including different versions. Do you use network attached storage (NAS)? Or a large external volume? You can learn more about advanced workflows using these links:

- A shared cache can be set up to store, version and access a lot of data on a large shared volume efficiently.
- A quite advanced scenario is to track and version data directly on the remote storage (e.g. S3). See Managing External Data to learn more.

Get Started: Data and Model Access

We've learned how to *track* data and models with DVC, and how to commit their versions to Git. The next questions are: How can we *use*these artifacts outside of the project? How do we download a model to deploy it? How to download a specific version of a model? Or reuse datasets across different projects?

> These questions tend to come up when you browse the files that DVC saves to remote storage (e.g. `s3://dvc-public/remote/get-started/fb/89904ef053f04d64eafcc3d70db673` 😱 instead of the original file name such as `model.pkl` or `data.xml`).

Read on or watch our video to see how to find and access models and datasets with DVC.

Remember those `.dvc` files `dvc add` generates? Those files (and `dvc.lock`, which we'll cover later) have their history in Git. DVC's remote storage config is also saved in Git, and contains all the information needed to access and download any version of datasets, files, and models. It means that a Git repository with DVC files becomes an entry point, and can be used instead of accessing files directly.

Find a file or directory

You can use `dvc list` to explore a DVC repository hosted on any Git server. For example, let's see what's in the `get-started/` directory of our dataset-registry repo:

```
$ dvc list https://github.com/iterative/dataset-registry get-started
.gitignore
data.xml
data.xml.dvc
```

The benefit of this command over browsing a Git hosting website is that the list includes files and directories tracked by both Git and DVC (`data.xml` is not visible if you check GitHub).

Download

One way is to simply download the data with `dvc get`. This is useful when working outside of a DVC project environment, for example in an automated ML model deployment task:

```
$ dvc get https://github.com/iterative/dataset-registry \
          use-cases/cats-dogs
```

When working inside another DVC project though, this is not the best strategy because the connection between the projects is lost — others won't know where the data came from or whether new versions are available.

Import file or directory

`dvc import` also downloads any file or directory, while also creating a `.dvc` file (which can be saved in the project):

```
$ dvc import https://github.com/iterative/dataset-registry \
             get-started/data.xml -o data/data.xml
```

This is similar to `dvc get` + `dvc add`, but the resulting `.dvc` files includes metadata to track changes in the source repository. This allows you to bring in changes from the data source later using `dvc update`.

💡 Expand to see what happens under the hood.

> Note that the dataset registry repository doesn't actually contain a `get-started/data.xml` file. Like `dvc get`, `dvc import` downloads from remote storage.

`.dvc` files created by `dvc import` have special fields, such as the data source `repo` and `path` (under `deps`):

```
+deps:
+- path: get-started/data.xml
+  repo:
+    url: https://github.com/iterative/dataset-registry
+    rev_lock: f31f5c4cdae787b4bdeb97a717687d44667d9e62
 outs:
 - md5: a304afb96060aad90176268345e10355
   path: data.xml
```

The `url` and `rev_lock` subfields under `repo` are used to save the origin and version of the dependency, respectively.

Python API

It's also possible to integrate your data or models directly in source code with DVC's Python API. This lets you access the data contents directly from within an application at runtime. For example:

```python
import dvc.api

with dvc.api.open(
    'get-started/data.xml',
    repo='https://github.com/iterative/dataset-registry'
) as fd:
    # fd is a file descriptor which can be processed normally
```

Get Started: Data Pipelines

Versioning large data files and directories for data science is great, but not enough. How is data filtered, transformed, or used to train ML models? DVC introduces a mechanism to capture *data pipelines* — series of data processes that produce a final result.

DVC pipelines and their data can also be easily versioned (using Git). This allows you to better organize projects, and reproduce your workflow and results later — exactly as they were built originally! For example, you could capture a simple ETL workflow, organize a data science project, or build a detailed machine learning pipeline.

Watch and learn, or follow along with the code example below!

Pipeline stages

Use `dvc run` to create *stages*. These represent processes (source code tracked with Git) which form the steps of a *pipeline*. Stages also connect code to its corresponding data *input* and *output*. Let's transform a Python script into a stage:

Expand to download example code.

Get the sample code like this:

```
$ wget https://code.dvc.org/get-started/code.zip
$ unzip code.zip
$ rm -f code.zip
$ tree
.
├── params.yaml
└── src
    ├── evaluate.py
    ├── featurization.py
    ├── prepare.py
    ├── requirements.txt
    └── train.py
```

Now let's install the requirements:

> We **strongly** recommend creating a virtual environment first.

```
$ pip install -r src/requirements.txt
```

Please also add or commit the source code directory with Git at this point.

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

A `dvc.yaml` file is generated. It includes information about the command we ran (`python src/prepare.py data/data.xml`), its dependencies, and outputs.

💡 Expand to see what happens under the hood.

The command options used above mean the following:

- `-n prepare` specifies a name for the stage. If you open the `dvc.yaml` file you will see a section named `prepare`.
- `-p prepare.seed,prepare.split` defines special types of dependencies — parameters. We'll get to them later in theMetrics, Parameters, and Plots page, but the idea is that the stage can depend on field values from a parameters file (`params.yaml` by default):

```
prepare:
  split: 0.20
  seed: 20170428
```

- `-d src/prepare.py` and `-d data/data.xml` mean that the stage depends on these files to work. Notice that the source code itself is marked as a dependency. If any of these files change later, DVC will know that this stage needs to be reproduced.
- `-o data/prepared` specifies an output directory for this script, which writes two files in it. This is how the workspaceshould look like now:

```
 .
 ├── data
 │   ├── data.xml
 │   ├── data.xml.dvc
+│   └── prepared
+│       ├── test.tsv
+│       └── train.tsv
+├── dvc.yaml
+├── dvc.lock
 ├── params.yaml
 └── src
   ├── ...
```

- The last line, `python src/prepare.py data/data.xml` is the command to run in this stage, and it's saved to `dvc.yaml`, as shown below.

The resulting `prepare` stage contains all of the information above:

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

There's no need to use `dvc add` for DVC to track stage outputs (`data/prepared` in this case); `dvc run` already took care of this. You only need to run `dvc push` if you want to save them to remote storage, (usually along with `git commit` to version `dvc.yaml` itself).

Dependency graphs (DAGs)

By using `dvc run` multiple times, and specifying outputs of a stage as dependencies of another one, we can describe a sequence of commands which gets to a desired result. This is what we call a *data pipeline* or *dependency graph*.

Let's create a second stage chained to the outputs of `prepare`, to perform feature extraction:

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

The `dvc.yaml` file is updated automatically and should include two stages now.

💡 Expand to see what happens under the hood.

The changes to the `dvc.yaml` should look like this:

```
  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
+   featurize:
+     cmd: python src/featurization.py data/prepared data/features
+     deps:
+     - data/prepared
+     - src/featurization.py
+     params:
+     - featurize.max_features
+     - featurize.ngrams
+     outs:
+     - data/features
```

⚙ Expand to add more stages.

Let's add the training itself. Nothing new this time; just the same `dvc run` command with the same set of options:

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

Please check the `dvc.yaml` again, it should have one more stage now.

This should be a good time to commit the changes with Git. These include `.gitignore`, `dvc.lock`, and `dvc.yaml` — which describe our pipeline.

Reproduce

The whole point of creating this `dvc.yaml` file is the ability to easily reproduce a pipeline:

```
$ dvc repro
```

⚙️ Expand to have some fun with it.

Let's try to play a little bit with it. First, let's try to change one of the parameters for the training stage:

1. Open `params.yaml` and change `n_est` to `100`, and
2. (re)run `dvc repro`.

You should see:

```
$ dvc repro
Stage 'prepare' didn't change, skipping
Stage 'featurize' didn't change, skipping
Running stage 'train' with command: ...
```

DVC detected that only `train` should be run, and skipped everything else! All the intermediate results are being reused.

Now, let's change it back to `50` and run `dvc repro` again:

```
$ dvc repro
Stage 'prepare' didn't change, skipping
Stage 'featurize' didn't change, skipping
```

As before, there was no need to rerun `prepare`, `featurize`, etc. But this time it also doesn't rerun `train`! The previous run with the same set of inputs (parameters & data) was saved in DVC's run-cache, and reused here.

💡 Expand to see what happens under the hood.

`dvc repro` relies on the DAG definition from `dvc.yaml`, and uses `dvc.lock` to determine what exactly needs to be run.

The `dvc.lock` file is similar to a `.dvc` file — it captures hashes (in most cases `md5` s) of the dependencies and values of the parameters that were used. It can be considered a *state* of the pipeline:

```
schema: '2.0'
stages:
  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:
    - path: data/prepared
      md5: 20b786b6e6f80e2b3fcf17827ad18597.dir
```

> `dvc status` command can be used to compare this state with an actual state of the workspace.

DVC pipelines (`dvc.yaml` file, `dvc run`, and `dvc repro` commands) solve a few important problems:

- *Automation*: run a sequence of steps in a "smart" way which makes iterating on your project faster. DVC automatically determines which parts of a project need to be run, and it caches "runs" and their results to avoid unnecessary reruns.
- *Reproducibility*: `dvc.yaml` and `dvc.lock` files describe what data to use and which commands will generate the pipeline results (such as an ML model). Storing these files in Git makes it easy to version and share.
- *Continuous Delivery and Continuous Integration (CI/CD) for ML*: describing projects in way that can be reproduced (built) is the first necessary step before introducing CI/CD systems. See our sister project CML for some examples.

Visualize

Having built our pipeline, we need a good way to understand its structure. Seeing a graph of connected stages would help. DVC lets you do so without leaving the terminal!

```
$ dvc dag
         +---------+
         | prepare |
         +---------+
              *
              *
              *
        +-----------+
        | featurize |
        +-----------+
         **        **
```

```
       **             *
      *                **
+-------+                *
| train |              **
+-------+             *
         **         **
           **     **
             *  *
        +----------+
        | evaluate |
        +----------+
```

> Refer to `dvc dag` to explore other ways this command can visualize a pipeline.

Get Started: Metrics, Parameters, and Plots

DVC makes it easy to track metrics, update parameters, and visualize performance with plots. These concepts are introduced below.

> All of the above can be combined into experiments to run and compare many iterations of your ML project.

Collecting metrics

First, let's see what is the mechanism to capture values for these ML attributes. Let's add a final evaluation stage to our pipeline from before:

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

💡 Expand to see what happens under the hood.

The `-M` option here specifies a metrics file, while `--plots-no-cache` specifies a plots file (produced by this stage) which will not be cached by DVC. `dvc run` generates a new stage in the `dvc.yaml` file:

```
evaluate:
  cmd: python src/evaluate.py model.pkl data/features ...
  deps:
    - data/features
    - model.pkl
    - src/evaluate.py
  metrics:
    - scores.json:
        cache: false
  plots:
    - prc.json:
        cache: false
    - roc.json:
        cache: false
```

The biggest difference to previous stages in our pipeline is in two new sections: `metrics` and `plots`. These are used to mark certain files containing ML "telemetry". Metrics files contain scalar values (e.g. `AUC`) and plots files contain matrices and data series (e.g. `ROC curves` or model loss plots) meant to be visualized and compared.

> With `cache: false`, DVC skips caching the output, as we want `scores.json`, `prc.json`, and `roc.json` to be versioned by Git.

`evaluate.py` writes the model's ROC-AUC andaverage precision to `scores.json`, which in turn is marked as a `metrics` file with `-M`. Its contents are:

```
{ "avg_prec": 0.5204838673030754, "roc_auc": 0.9032012604172255 }
```

`evaluate.py` also writes `precision`, `recall`, and `thresholds` arrays (obtained using `precision_recall_curve`) into the plots file `prc.json`:

```
{
  "prc": [
    { "precision": 0.021473008227975116, "recall": 1.0, "threshold": 0.0 },
    ...,
    { "precision": 1.0, "recall": 0.009345794392523364, "threshold": 0.6 }
  ]
}
```

Similarly, it writes arrays for the roc_curve into `roc.json` for an additional plot.

> DVC doesn't force you to use any specific file names, nor does it enforce a format or structure of a metrics or plots file. It's completely user/case-defined. Refer to `dvc metrics` and `dvc plots` for more details.

You can view tracked metrics and plots with DVC. Let's start with the metrics:

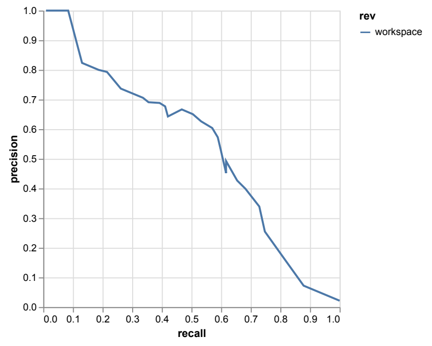
```
$ dvc metrics show
Path         avg_prec    roc_auc
scores.json  0.52048     0.9032
```

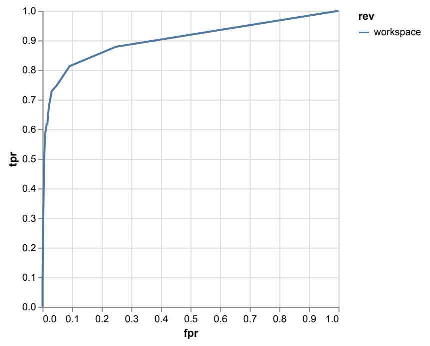
To view plots, first specify which arrays to use as the plot axes. We only need to do this once, and DVC will save our plot configurations.

```
$ dvc plots modify prc.json -x recall -y precision
Modifying stage 'evaluate' in 'dvc.yaml'
$ dvc plots modify roc.json -x fpr -y tpr
Modifying stage 'evaluate' in 'dvc.yaml'
```

Now let's view the plots:

```
$ dvc plots show
file:///Users/dvc/example-get-started/plots.html
```





Let's save this iteration, so we can compare it later:

```
$ git add scores.json prc.json roc.json
$ git commit -a -m "Create evaluation stage"
```

Later we will see how to compare and visualize different pipeline iterations. For now, let's see how can we capture another important piece of information which will be useful for comparison: parameters.

Defining stage parameters

It's pretty common for data science pipelines to include configuration files that define adjustable parameters to train a model, do pre-processing, etc. DVC provides a mechanism for stages to depend on the values of specific sections of such a config file (YAML, JSON, TOML, and Python formats are supported).

Luckily, we should already have a stage with parameters in `dvc.yaml`:

```
featurize:
  cmd: python src/featurization.py data/prepared data/features
  deps:
    - data/prepared
    - src/featurization.py
  params:
    - featurize.max_features
    - featurize.ngrams
  outs:
    - data/features
```

Expand to recall how it was generated.

The `featurize` stage was created with this `dvc run` command. Notice the argument sent to the `-p` option (short for `--params`):

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

The `params` section defines the parameter dependencies of the `featurize` stage. By default, DVC reads those values (`featurize.max_features` and `featurize.ngrams`) from a `params.yaml` file. But as with metrics and plots, parameter file names and structure can also be user- and case-defined.

Here's the contents of our `params.yaml` file:

```
prepare:
  split: 0.20
  seed: 20170428

featurize:
  max_features: 500
  ngrams: 1

train:
  seed: 20170428
  n_est: 50
  min_split: 2
```

Updating params and iterating

We are definitely not happy with the AUC value we got so far! Let's edit the `params.yaml` file to use bigrams and increase the number of features:

```
 featurize:
-  max_features: 500
-  ngrams: 1
+  max_features: 1500
+  ngrams: 2
```

The beauty of `dvc.yaml` is that all you need to do now is run:

```
$ dvc repro
```

It'll analyze the changes, use existing results from the run-cache, and execute only the commands needed to produce new results (model, metrics, plots).

The same logic applies to other possible adjustments — edit source code, update datasets — you do the changes, use `dvc repro`, and DVC runs what needs to be.

Comparing iterations

Finally, let's see how the updates improved performance. DVC has a few commands to see changes in and visualize metrics, parameters, and plots. These commands can work for one or across multiple pipeline iteration(s). Let's compare the current "bigrams" run with the last committed "baseline" iteration:

```
$ dvc params diff
Path         Param                   Old    New
params.yaml  featurize.max_features  500    1500
params.yaml  featurize.ngrams        1      2
```

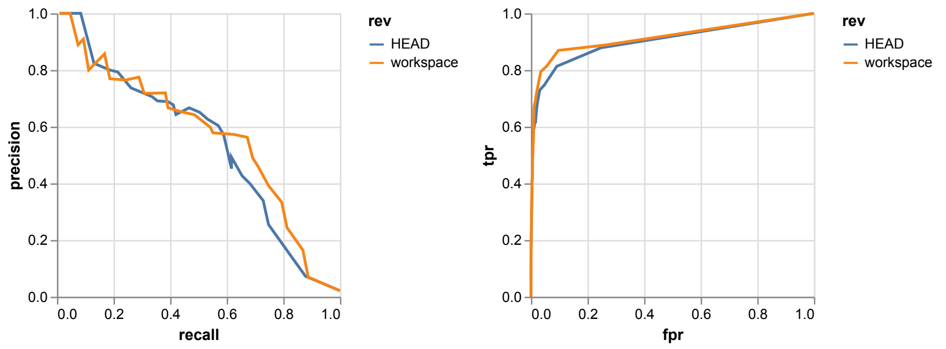
`dvc params diff` can show how params in the workspace differ vs. the last commit.

`dvc metrics diff` does the same for metrics:

```
$ dvc metrics diff
Path         Metric    Old      New      Change
scores.json  avg_prec  0.52048  0.55259  0.03211
scores.json  roc_auc   0.9032   0.91536  0.01216
```

And finally, we can compare both `precision recall` and `roc` curves with a single command!

```
$ dvc plots diff
file:///Users/dvc/example-get-started/plots.html
```



> See `dvc plots diff` for more info on its options.

> All these commands also accept Git revisions (commits, tags, branch names) to compare.

On the next page, you can learn advanced ways to track, organize, and compare more experiment iterations.

Get Started: Experiments

⚠️ This feature is only available in DVC 2.0 ⚠️

Experiments proliferate quickly in ML projects where there are many parameters to tune or other permutations of the code. We can organize such projects and keep only what we ultimately need with `dvc experiments`. DVC can track experiments for you so there's no need to commit each one to Git. This way your repo doesn't become polluted with all of them. You can discard experiments once they're no longer needed.

> 📖 See Experiment Management for more information on DVC's approach.

Running experiments

Previously, we learned how to tune ML pipelines and compare the changes. Let's further increase the number of features in the `featurize` stage to see how it compares.

`dvc exp run` makes it easy to change hyperparameters and run a new experiment:

```
$ dvc exp run --set-param featurize.max_features=3000
```

💡 Expand to see what happens under the hood.

`dvc exp run` is similar to `dvc repro` but with some added conveniences for running experiments. The `--set-param` (or `-S`) flag sets the values for parameters

Check that the `featurize.max_features` value has been updated in `params.yaml`:

```
 featurize:
-  max_features: 1500
+  max_features: 3000
```

Any edits to dependencies (parameters or source code) will be reflected in the experiment run.

`dvc exp diff` compares experiments:

```
$ dvc exp diff
Path         Metric    Value    Change
scores.json  avg_prec  0.56191  0.009322
scores.json  roc_auc   0.93345  0.018087

Path         Param                    Value    Change
params.yaml  featurize.max_features   3000     1500
```

Queueing experiments

So far, we have been tuning the `featurize` stage, but there are also parameters for the `train` stage (which trains arandom forest classifier).

These are the `train` parameters from `params.yaml`:

```
train:
  seed: 20170428
  n_est: 50
  min_split: 2
```

Let's set up experiments with different hyperparameters. We can use the `--queue` flag to define all the combinations we want to try without executing anything (yet):

```
$ dvc exp run --queue -S train.min_split=8
Queued experiment 'd3f6d1e' for future execution.
$ dvc exp run --queue -S train.min_split=64
Queued experiment 'f1810e0' for future execution.
$ dvc exp run --queue -S train.min_split=2 -S train.n_est=100
Queued experiment '7323ea2' for future execution.
$ dvc exp run --queue -S train.min_split=8 -S train.n_est=100
Queued experiment 'c605382' for future execution.
$ dvc exp run --queue -S train.min_split=64 -S train.n_est=100
Queued experiment '0cdee86' for future execution.
```

Next, run all (`--run-all`) queued experiments in parallel (using `--jobs`):

```
$ dvc exp run --run-all --jobs 2
```

Comparing many experiments

To compare all of these experiments, we need more than `diff`. `dvc exp show` compares any number of experiments in one table:

```
$ dvc exp show --no-timestamp \
               --include-params train.n_est,train.min_split

┏━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Experiment    ┃ avg_prec ┃ roc_auc ┃ train.n_est┃ train.min_split ┃
```

```
 ┃━━━━━━━━━━━━━━━┃━━━━━━━━━━━┃━━━━━━━━━━┃━━━━━━━━━━━━┃━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┃
 │ workspace     │  0.56191 │ 0.93345 │ 50         │ 2                              │
 │ master        │  0.55259 │ 0.91536 │ 50         │ 2                              │
 │ ├── exp-bfe64 │  0.57833 │ 0.95555 │ 50         │ 8                              │
 │ ├── exp-b8082 │  0.59806 │ 0.95287 │ 50         │ 64                             │
 │ ├── exp-c7250 │  0.58876 │ 0.94524 │ 100        │ 2                              │
 │ ├── exp-b9cd4 │  0.57953 │ 0.95732 │ 100        │ 8                              │
 │ ├── exp-98a96 │  0.60405 │  0.9608 │ 100        │ 64                             │
 │ └── exp-ad5b1 │  0.56191 │ 0.93345 │ 50         │ 2                              │
 └───────────────┴──────────┴─────────┴────────────┴────────────────────────────────┘
```

Each experiment is given an arbitrary name by default (although we can specify one with `dvc exp run -n`.) We can see that `exp-98a96` performed best among both of our metrics, with 100 estimators and a minimum of 64 samples to split a node.

> See `dvc exp show --help` for more info on its options.

Persisting experiments

Now that we know the best parameters, let's keep that experiment and ignore the rest.

`dvc exp apply` rolls back the workspace

```
$ dvc exp apply exp-98a96
Changes for experiment 'exp-98a96' have been applied to your workspace.
```

💡 Expand to see what happens under the hood.

`dvc exp apply` is similar to `dvc checkout`, but works with experiments instead. DVC tracks everything in the pipeline for each experiment (parameters, metrics, dependencies, and outputs), retrieving things later as needed.

Check that `scores.json` reflects the metrics in the table above:

```
{ "avg_prec": 0.6040544652105823, "roc_auc": 0.9608017142900953 }
```

Once an experiment has been applied to the workspace, it is no different from reproducing the result without `dvc exp run`. Let's make it persistent in our regular pipeline by committing it in our Git branch:

```
$ git add dvc.lock params.yaml prc.json roc.json scores.json
$ git commit -a -m "Preserve best random forest experiment"
```

Sharing experiments

After committing the best experiments to our Git branch, we can store and share them remotely like any other iteration of the pipeline.

```
dvc push
git push
```

💡 Important information on storing experiments remotely.

The commands in this section require both a `dvc remote default` and a Git remote. A DVC remote stores the experiment data, and a Git remote stores the code, parameters, and other metadata associated with the experiment. DVC supports various types of remote storage (local file system, SSH, Amazon S3, Google Cloud Storage, HTTP, HDFS, etc.). The Git remote is often a central Git server (GitHub, GitLab, BitBucket, etc.).

Experiments that have not been made persistent will not be stored or shared remotely through `dvc push` or `git push`.

`dvc exp push` enables storing and sharing any experiment remotely.

```
$ dvc exp push gitremote exp-bfe64
Pushed experiment 'exp-bfe64' to Git remote 'gitremote'.
```

`dvc exp list` shows all experiments that have been saved.

```
$ dvc exp list gitremote --all
72ed9cd:
        exp-bfe64
```

`dvc exp pull` retrieves the experiment from a Git remote.

```
$ dvc exp pull gitremote exp-bfe64
Pulled experiment 'exp-bfe64' from Git remote 'gitremote'.
```

> All these commands take a Git remote as an argument. A `dvc remote default` is also required to share the experiment data.

Cleaning up

Let's take another look at the experiments table:

```
$ dvc exp show --no-timestamp \
               --include-params train.n_est,train.min_split
┏━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Experiment ┃ avg_prec ┃ roc_auc ┃ train.n_est┃ train.min_split ┃
┡━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ workspace  │  0.60405 │  0.9608 │ 100        │ 64              │
│ master     │  0.60405 │  0.9608 │ 100        │ 64              │
└────────────┴──────────┴─────────┴────────────┴─────────────────┘
```

Where did all the experiments go? By default, `dvc exp show` only shows experiments since the last commit, but don't worry. The experiments remain cached and can be shown or applied. For example, use `-n` to show experiments from the previous *n* commits:

```
$ dvc exp show -n 2 --no-timestamp \
                    --include-params train.n_est,train.min_split
┏━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Experiment    ┃ avg_prec ┃ roc_auc ┃ train.n_est┃ train.min_split ┃
┡━━━━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ workspace     │  0.60405 │  0.9608 │ 100        │ 64              │
│ master        │  0.60405 │  0.9608 │ 100        │ 64              │
│ 64d74b2       │  0.55259 │ 0.91536 │ 50         │ 2               │
│ ├── exp-bfe64 │  0.57833 │ 0.95555 │ 50         │ 8               │
│ ├── exp-b8082 │  0.59806 │ 0.95287 │ 50         │ 64              │
│ ├── exp-c7250 │  0.58876 │ 0.94524 │ 100        │ 2               │
│ ├── exp-98a96 │  0.60405 │  0.9608 │ 100        │ 64              │
│ ├── exp-b9cd4 │  0.57953 │ 0.95732 │ 100        │ 8               │
│ └── exp-ad5b1 │  0.56191 │ 0.93345 │ 50         │ 2               │
└───────────────┴──────────┴─────────┴────────────┴─────────────────┘
```

Eventually, old experiments may clutter the experiments table.

`dvc exp gc` removes all references to old experiments:

```
$ dvc exp gc --workspace
$ dvc exp show -n 2 --no-timestamp \
                    --include-params train.n_est,train.min_split
┏━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Experiment ┃ avg_prec ┃ roc_auc ┃ train.n_est┃ train.min_split ┃
┡━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ workspace  │  0.60405 │  0.9608 │ 100        │ 64              │
│ master     │  0.60405 │  0.9608 │ 100        │ 64              │
│ 64d74b2    │  0.55259 │ 0.91536 │ 50         │ 2               │
└────────────┴──────────┴─────────┴────────────┴─────────────────┘
```

> `dvc exp gc` only removes references to the experiments; not the cached objects associated with them. To clean up the cache, use `dvc gc`.