Pre-prerequisite (Install either on your laptops or VMWare image from yesterday):

- pip install dvc
- apt install git
- pip install virtualenv

Building/Deploying ML Apps

➔Data versioning (DVC)

- Step-by-step introduction into DVC[1] features.

Fig 1. CC-BY-2.0: https://dvc.org/doc/tutorials/versioning

Treating data as code

```
$ git clone
https://github.com/iterative/example-versioning.git
$ cd example-versioning
$ ls
$ ls -la
```

```
.dvc/
.gitignore
.git
```

Now, initialize your code and data repos

```
$ rm -fr .dvc
$ rm -fr .git

$ ls -la

$ git init
$ dvc init
```

In practice, DVC uses a /tmp repo local, which syncs with some remote services such as s3/Azure/gdrive, etc.
However, for this lab we are using a local data folder
*** **Don't do this**

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

Preparation of *(**Python3**)* environment; your local environment should run python3.*

```
$ virtualenv -p python3 .env
$ source .env/bin/activate
$ pip install -r requirements.txt
```

Version 1 of our model

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

$ unzip -q data.zip

$ rm -f data.zip
```

The above command obtains a data set from a data repository, i.e., a repository containing references (checksums) to data. In this case, the actual data is also stored in the repo, for demonstration purposes.

Let's capture the current state of this dataset with

DVC:

```
$ dvc add data
```

Next, train our model, which outputs ===> `model.h5`, the trained model, and `metrics.csv`, the metrics history. Then, we capture the current state of the model with DVC:

```
$ python train.py
$ dvc add model.h5
```

Commit the current state into git:

```
$ git add .gitignore model.h5.dvc data.dvc metrics.csv
$ git commit -s -m "First model, trained with 1000 images"
$ git tag -a "v1.0" -m "model v1.0, 1000 images"
```

```
$ git status

---------
...
      bottleneck_features_train.npy
      bottleneck_features_validation.npy`
```

Version 2 of our model

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

$ unzip -q new-labels.zip
$ rm -f new-labels.zip
```

Let's leverage these new labels and retrain our model:

```
$ dvc add data
$ python train.py
$ dvc add model.h5
```

Then, we commit the second version of the data set and,

model and metrics:

```
$ git add model.h5.dvc data.dvc metrics.csv
$ git commit -s -m "Second model, trained with 2000 images"
$ git tag -a "v2.0" -m "model v2.0, 2000 images"
```

That's it! We have tracked a second dataset, model, and metrics with DVC, and the DVC-files that point to them were then committed with Git. Let's now look at how DVC can help us go back to the previous version if we need to.

Switching between workspace versions

To get a particular version of our committed data, we use the `$ dvc checkout` command, which is similar to the `$ git checkout` command.

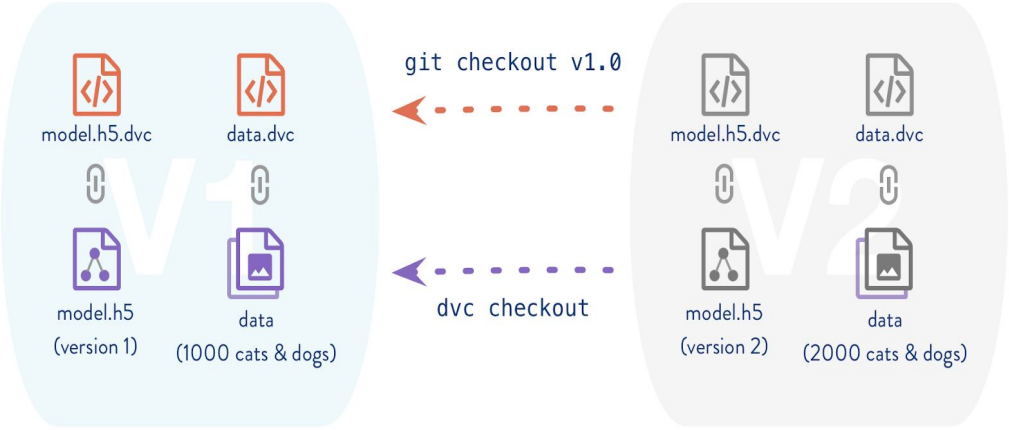

Fig 2. Switching between workspace.

Thus, we can either do a complete workspace checkout or checkout of our model.

Complete checkout

```
$ git checkout v1.0
$ dvc checkout
```

We restore our workspace to the first snapshot that includes: code, data files, model, all of it. Moreover, DVC optimizes this operation to avoid

copying data or model files each time.

Thus, `$ dvc checkout` is fast, irrespective of the size of your datasets or models.

Specific checkout

```
$ git checkout v1.0 data.dvc
$ dvc checkout data.dvc
```

In this case, we want to keep the current revisions of our code files, but want to go back to the earliest dataset version.

Run `$ git status` , and notice that `data.dvc` is modified and points to **`v1.0` version** of the dataset, meanwhile **code** and **model** still point to the `v2.0` tag.

Go deeper

Up To now, we have been using a simpler model. Now, we want to dive deeper into more advanced DVC features and experiments.

Text classification problem with DVC[5]

The ML community needs best practices to effectively organize projects and to collaborate. One possible reason for this lag is that ML algorithms are difficult to implement, manage and reuse; especially in ***reproducible research***.

This is one **area*** where DVC comes in handy (reproducibility).

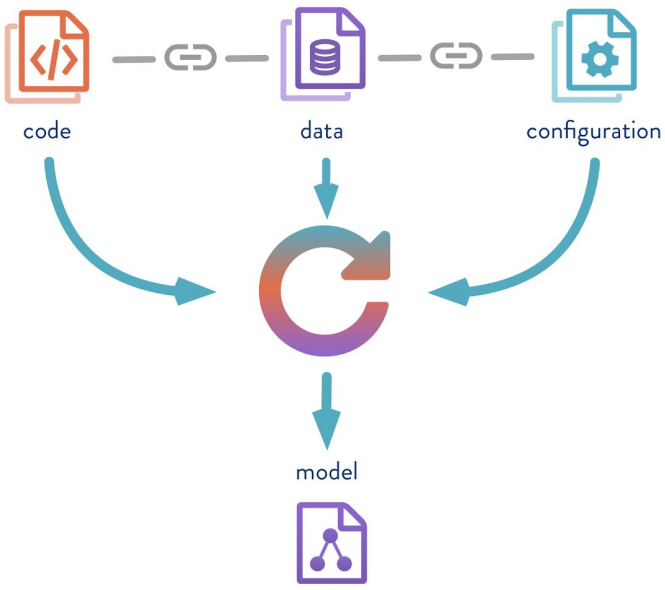

Fig 3.

Git was designed with a linear structure of branches in mind, which is not the case for an ML process, where each hypothesis represents a Git branch. However, Git can’t keep track of the data, and the dependencies between code and data that is used to build a model

DVC Workflow

Fig 4 shows the DVC commands and relationships between a local cache and remote storage.

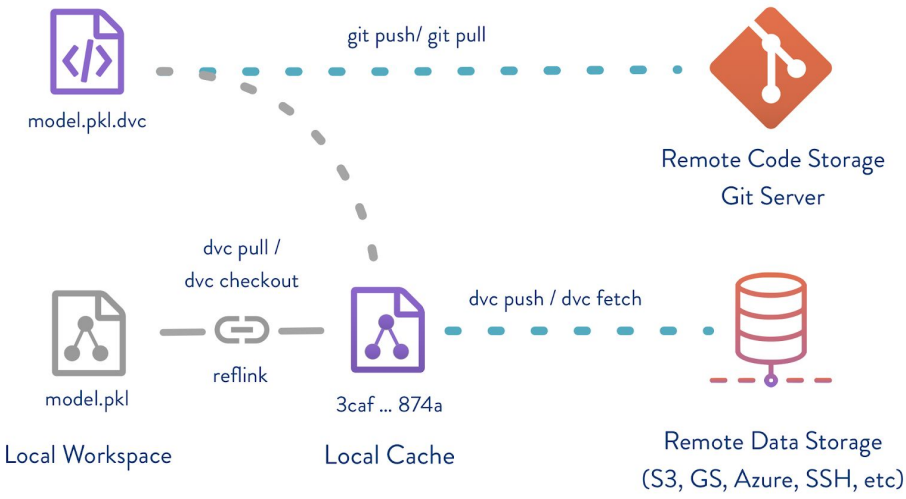

Fig 4.

Getting the example code

DVC works atop Git, so you have to initialise Git first before you can initialize DVC.

```
$ mkdir classify
$ cd classify
$ git init
$ wget https://code.dvc.org/tutorial/nlp/code.zip
$ unzip code.zip -d code && rm -f code.zip
$ git add code
$ git commit -m "download code"
```

Create a virtual environment

```
$ virtualenv -p python3 .env
$ source .env/bin/activate
$ echo ".env/" >> .gitignore
$ pip install -r code/requirements.txt
```

Initialize DVC

```
$ dvc init
$ ls -a .dvc
```

```
$ git status -s
$ cat .dvc/.gitignore
$ git add .
$ git commit -s -m "init DVC"
```

The `cache` directory is contained in `.dvc/.gitignore`, which means that it won't be tracked by Git — It's a local-only directory, and you cannot push it to any Git remote.

Define ML Pipeline

Get data file

Next, we create a repo for our data.

```
$ mkdir data
$ wget -P data
https://data.dvc.org/tutorial/nlp/100K/Posts.xml.zip
$ du -sh data/*
 41M data/Posts.xml.zip
```

Recall that **`data/Posts.xml.zip`** is a regular (untracked) file. However, we will use DVC (`$ dvc add`) to track it. After executing the command you will see a new file **`data/Posts.xml.zip.dvc`** and a change in `data/.gitignore`. Both of these files have to be committed to the repository.

```
$ dvc add data/Posts.xml.zip
$ du -sh data/*
-----
    41M data/Posts.xml.zip
    4.0K data/Posts.xml.zip.dvc
```

```
$ git status -s data/
-----
   ?? data/.gitignore
   ?? data/Posts.xml.zip.dvc

$ git add .
$ git commit -s -m "add raw dataset"
```

Data file internals

https://dvc.org/doc/user-guide/dvc-file-format

```
$ cat data/Posts.xml.zip.dvc
-----
   md5: 7559eb45beb7e90f192e836be8032a64
   outs:
     - cache: true
     md5: ec1d2935f811b77cc49b031b999cbf17
     path: Posts.xml.zip

$ du -sh .dvc/cache/ec/*
-----
   41M .dvc/cache/ec/1d2935f811b77cc49b031b999cbf17
```

By looking at the contents of the textual DVC-files, one can easily identify the relationships between the data file-path in a repository and the path in cache directory.

DVC was designed with large data files in mind. DVC can create ref-links or other file link types.

Creating file links is a quick file system operation. So, with DVC you can easily `checkout` a few dozen files of any size. A file link prevents you from using twice as much space in the hard drive. Even if each of the files contains 41MB of data, the overall size of the repository is still 41MB. Both of the files correspond to the same inode (a file metadata record) in the file system.

```
$ ls -i data/Posts.xml.zip
-----
     78483929 data/Posts.xml.zip

$ ls -i .dvc/cache/ec/
------
     78483929 88519f8465218abb23ce0e0e8b1384

$ du -sh .  ## Also try this with the df command.
----
    41M .
```

Note that `$ ls -i` prints the index number(78483929) of each file and inode for data/Posts.xml.zip and .dvc/cache/ec/88519f8465218abb23ce0e0e8b1384 remained same.

Data Processing and Model Training

DVC comes handy with `$ dvc run` or `$ dvc add` to define stages of your ML process and link them to an ML pipeline.

ML pipeline Description:

ML pipelines typically start with large raw datasets, include intermediate featurization and training stages, and produce a final model, as well as accuracy metrics.

Let's see how an extraction command `unzip` works under DVC, for example:

```
$ dvc run -d data/Posts.xml.zip -o data/Posts.xml \
        unzip data/Posts.xml.zip -d data/
------
    Running command:
```

```
      unzip data/Posts.xml.zip -d data/
    Archive:  data/Posts.xml.zip
        inflating: data/Posts.xml
    Saving information to 'Posts.xml.dvc'.

    To track the changes with git run:

      git add data/.gitignore Posts.xml.dvc

$ du -sh data/*
------
     145M data/Posts.xml
     41M data/Posts.xml.zip
     4.0K data/Posts.xml.zip.dvc
```

For reproducibility purposes, `$ dvc run` creates the `**Posts.xml.dvc**` stage file in the project with information about this pipeline stage. Note that the name of this file could be specified by using the `-f` option, for example `-f extract.dvc`
Now, let's look inside the resulting stage file

```
$ cat Posts.xml.dvc
------
cmd: ' unzip data/Posts.xml.zip -d data/'
deps:
- md5: ec1d2935f811b77cc49b031b999cbf17
  path: data/Posts.xml.zip
md5: 16129387a89cb5a329eb6a2aa985415e
outs:
- cache: true
  md5: c1fa36d90caa8489a317eee917d8bf03
  path: data/Posts.xml
```

Notice that the `data/.gitignore` file was modified.

```
$ git status -s
------
   M data/.gitignore
  ?? Posts.xml.dvc

$ cat data/.gitignore
------
```

```
   Posts.xml.zip
   Posts.xml
```

The output file `Posts.xml` was transformed by DVC into a data file in accordance with the `-o` option.

You can find the corresponding cache file with the hash value, as a path starting in `c1/fa36d`:

```
$ ls .dvc/cache/
2f/ a8/

$ du -sh .dvc/cache/c1/* .dvc/cache/ec/*
 41M .dvc/cache/ec/1d2935f811b77cc49b031b999cbf17
145M .dvc/cache/c1/fa36d90caa8489a317eee917d8bf03

$ du -sh .
186M .
```

Let's commit the result of the `unzip` command. This will be the first stage of our ML pipeline.

```
$ git add .
$ git commit -m "extract data"
```

Running in bulk

You can create stages and commit them with Git later.

Thus, let's create the following stages:

converting an XML file to TSV, and then separating training and testing datasets:

```
dvc run -d data/Posts.xml -d code/xml_to_tsv.py -d code/conf.py \
        -o data/Posts.tsv \
        python code/xml_to_tsv.py
```

```
-----
    Using 'Posts.tsv.dvc' as a stage file
    Reproducing 'Posts.tsv.dvc':
           python code/xml_to_tsv.py
```

```
$ dvc run -d data/Posts.tsv -d code/split_train_test.py \
        -d code/conf.py \
        -o data/Posts-test.tsv -o data/Posts-train.tsv \
        python code/split_train_test.py 0.33 20180319

-----
     Using 'Posts-test.tsv.dvc' as a stage file
     Reproducing 'Posts-test.tsv.dvc':
             python code/split_train_test.py 0.33 20180319
     Positive size 2049, negative size 97951
```

Notice that code/conf.pyc was created and we don't want Git to track this type of files

```
git status -s
-------
      M data/.gitignore
      ?? Posts-test.tsv.dvc
      ?? Posts.tsv.dvc
      ?? code/conf.pyc

$ echo "*.pyc" >> .gitignore
```

Also, both stage files can be committed to the repository together

```
$ git add .
$ git commit -m "Process to TSV and separate test and train"
```

We can define the feature extraction stage, which takes **train** and **test** `**TSVs**` and generates corresponding matrix files:

```
$ dvc run -d code/featurization.py -d code/conf.py \
        -d data/Posts-train.tsv -d data/Posts-test.tsv \
        -o data/matrix-train.p -o data/matrix-test.p \
```

```
        python code/featurization.py

-----
    Using 'matrix-train.p.dvc' as a stage file
    Reproducing 'matrix-train.p.dvc':
        python code/featurization.py
    The input data frame data/Posts-train.tsv size is (66999, 3)
    The output matrix data/matrix-train.p size is (66999, 5002)
and data         type is float64
    The input data frame data/Posts-test.tsv size is (33001, 3)
    The output matrix data/matrix-test.p size is (33001, 5002) and
data type is float64
```

Train a model using the train matrix file:

```
$ dvc run -d data/matrix-train.p -d code/train_model.py \
        -d code/conf.py -o data/model.p \
        python code/train_model.py 20180319

-----
    Using 'model.p.dvc' as a stage file
    Reproducing 'model.p.dvc':
        python code/train_model.py 20180319
    Input matrix size (66999, 5002)
    X matrix size (66999, 5000)
    Y matrix size (66999,)
```

Now, evaluate the result of the trained model using the test feature matrix:

```
$ dvc run -d data/model.p -d data/matrix-test.p \
        -d code/evaluate.py -d code/conf.py -M data/eval.txt \
        -f Dvcfile \
        python code/evaluate.py

-----
     Reproducing 'Dvcfile':
         python code/evaluate.py
```

Note that the output file `data/eval.txt` was transformed by DVC into a metric file in accordance with the `-M` option.

The result of the last three `dvc run` commands

execution is three stage files and a modified .gitignore file. All the changes should be committed with Git:

```
$ git status -s
 M data/.gitignore
?? Dvcfile
?? data/eval.txt
?? matrix-train.p.dvc
?? model.p.dvc

$ git add .
$ git commit -m Evaluate
```

The output of the evaluation stage contains the target value in a simple text form:

```
$ cat data/eval.txt
-----
    AUC: 0.624652
```

You can also show the metrics using the `DVC metrics` command:

```
$ dvc metrics show
-----
    data/eval.txt:AUC: 0.624652
```

This is probably not the best AUC that you have seen. In this document, our focus is DVC, not ML modeling and we use a relatively small dataset without any advanced ML techniques.

Model Optimization

Reproducibility

DVC facilitates reproducibility.

```
Reproduce complete or partial pipelines by executing
commands defined in their stages, in the correct
order. The commands to be executed are determined by
recursively analyzing dependencies and outputs of
the target stages.
```

DVC finds and reads all the DVC-files in a repository and builds a dependency graph (pipeline) based on these files.

This is one of the differences between DVC reproducibility and software build automation tools (Make, Maven, Ant, Rakefile etc). It was designed in such a way to localize specification of the graph nodes (pipeline stages).

If you run `$ repro` on any DVC-file from our repository, nothing happens because nothing was changed in the pipeline defined in the project.

```
$ dvc repro model.p.dvc
```

By default, `dvc repro` read the `**Dvcfile**`

```
$ dvc repro
```

This command wants to reproduce the same pipeline, but there is still nothing to reproduce.

Adding bigrams

Thus far, we have used unigram to build our model. However, we can improve our model with a bigrams.

Bigrams model extracts signals from separate and two-word combinations. This can increase the number of features for the model, which might also improve the target metric.

We will edit the `**code/featurization.py**` file.

First, create and checkout to a new branch`***bigrams***`.

```
$ git checkout -b bigrams

# Use any editor to modify this file
$ vim code/featurization.py
```

Specify **ngram** parameter in **CountVectorizer** **(lines 50–53)** and increase the number of features to **6000**:

```
bag_of_words = CountVectorizer(stop_words='english',
                               max_features=6000,
                               ngram_range=(1, 2))
```

Reproduce our changed pipeline:

```
$ dvc repro

Reproducing 'matrix-train.p.dvc':
    python code/featurization.py

-----
The input data frame data/Posts-train.tsv size is (66999, 3)
The output matrix data/matrix-train.p size is (66999, 6002) and
data type is float64
The input data frame data/Posts-test.tsv size is (33001, 3)
The output matrix data/matrix-test.p size is (33001, 6002) and
data type is float64

Reproducing 'model.p.dvc':
    python code/train_model.py 20180319
Input matrix size (66999, 6002)
X matrix size (66999, 6000)
Y matrix size (66999,)

Reproducing 'Dvcfile':
```

```
    python code/evaluate.py
```

Reproducibility (repo) process started with the feature creation stage because one of its parameters was changed — the edited source code file code/featurization.py.

All dependent stages were executed as well.

Let's take a look at the metric's change. The improvement is close to zero (+0.0075% to be precise):

```
$ cat data/eval.txt
------
    AUC: 0.624727
```

This is not a great result but it gives us some information about the model.

Let’s compare this result to the previous AUC.

```
$ dvc metrics show -a
------
bigrams:
      data/eval.txt: AUC: 0.624727

master:
      data/eval.txt: AUC: 0.624652
```

It's convenient to keep track of information even for failed experiments. Sometimes a failed hypothesis gives more information than a successful one.

Let's keep the result in the repository. Later we can find out why bigrams don't add value to the current model and change that.

Many DVC-files were changed. This happened due to

file hash changes.

```
$ git status -s
-----
    M Dvcfile
    M code/featurization.py
    M matrix-train.p.dvc
    M model.p.dvc
```

Let’s commit these changes to Git:

```
$ git add .  # Adds every changes in the current folder
$ git commit -s -m “… Bigrams …”
```

Checkout code and data files

The previous experiment was done in the 'featurization' stage and provided no improvements. This might be caused by not having perfect model hyperparameters. Let's try to improve the model by changing the hyperparameters.

Let's checkout the original model from the master branch, both Git and DVC.

```
$ git checkout master
$ dvc checkout

$ dvc repro
-----
    Data and pipelines are up to date.
```

After proper checkout, there is nothing to reproduce because all source files were checked out by Git, and all data files by DVC.

Precisely, `$ git checkout master` checked out the code

and DVC-files.

The DVC-files from the master branch point to old (unigram based) dependencies and outputs. `$ dvc checkout` command found all the DVC-files and restored the data files based on them.

Tune the model

Now, create a new branch called ***tuning*** for this new experiment.

```
$ git checkout -b tuning
# Please use your favorite text editor:
$ vi code/train_model.py
```

Also, increase the number of trees in the forest to 700 by changing the ***n_estimators*** parameter and the number of jobs in the `RandomForestClassifier class (line 27)`:

```
clf = RandomForestClassifier(n_estimators=700,
                             n_jobs=6, random_state=seed)
```

We reproduce only the modeling and the evaluation stage.

```
$ dvc repro
-----
    Reproducing 'model.p.dvc':
        python code/train_model.py 20180319
    Input matrix size (66999, 5002)
    X matrix size (66999, 5000)
    Y matrix size (66999,)
    Reproducing 'Dvcfile':
        python code/evaluate.py
```

Validate the metric and commit all the changes.

```
$ cat data/eval.txt
AUC: 0.637561
```

This seems like a good model improvement (+1.28%).

Now, commit all the changes:

```
$ git add .
$ git commit -s -m '700 trees in the forest'
```

Merge the model to master

Revisiting the failing hypothesis with bigrams, which didn't provide any model improvement even with one thousand more features.

Git merge logic works for data files and respectively for DVC models.

```
$ git checkout -b train_bigrams
$ git merge bigrams
-----
    Auto-merging model.p.dvc
    CONFLICT (content): Merge conflict in model.p.dvc
    Auto-merging Dvcfile
    CONFLICT (content): Merge conflict in Dvcfile
    Automatic merge failed; fix conflicts and then commit the result.
```

The merge has a few conflicts. All of the conflicts are related to file hash mismatches in the branches.

Another way to solve git merge conflicts is to simply replace all file hashes with empty strings ''. The only disadvantage of this trick is that DVC will need to recompute the output hashes.

Resolve the conflicts[1] and checkout the data files:

```
# Replace conflicting hashes with empty string ''
$ vim model.p.dvc
$ vim Dvcfile
$ dvc checkout
```

Then reproduce the result:

```
$ dvc repro
-----
Reproducing 'model.p.dvc':
    python code/train_model.py 20180319
Input matrix size (66999, 6002)
X matrix size (66999, 6000)
Y matrix size (66999,)
Reproducing 'Dvcfile':
    python code/evaluate.py
```

Check the target metric:

```
$ cat data/eval.txt
AUC: 0.640389
```

The bigrams increased the target metric by 0.28% and the last change looks like a reasonable improvement to the ML model. So, the result should be committed:

```
$ git add .
$ git commit -m 'Merge bigrams into the tuned model'
```

Now our current branch contains the best model and it can be merged into master.

```
$ git checkout master
$ dvc checkout
$ git merge train_bigrams
-------
Updating f5ff48c..4bd09da
Fast-forward
 Dvcfile               | 6 +++---
```

[1] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line

```
 code/featurization.py | 3 ++-
 code/train_model.py   | 2 +-
 matrix-train.p.dvc    | 6 +++---
 model.p.dvc           | 6 +++---
 5 files changed, 12 insertions(+), 11 deletions(-)
```

Fast-forward strategy was applied to this merge. It means that we have all the changes in the right place and reproduction is not needed.

```
$ dvc checkout
$ dvc repro
------
Data and pipelines are up to date.
```

References:

1. https://dvc.org/doc/get-started/agenda
2. https://git-scm.com
3. https://dvc.org/doc/install
4. https://dvc.org/doc/tutorials/versioning
5. https://dvc.org/doc/tutorials/deep
6. Very Deep Convolutional Networks for Large-Scale Image Recognition
   K. Simonyan, A. Zisserman
   arXiv:1409.1556