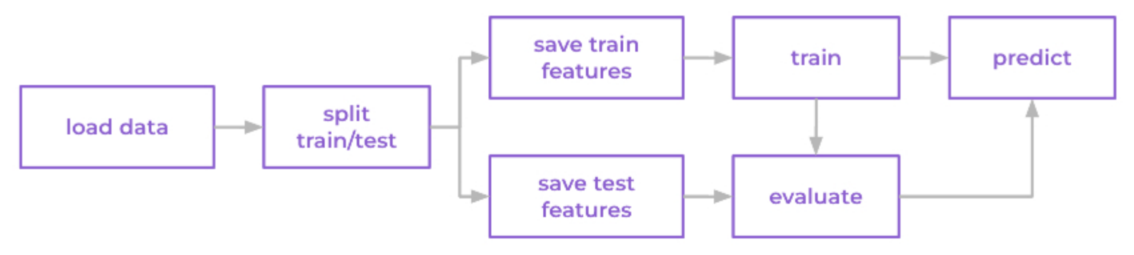
Step 5: Automate pipelines with DVC

Now that we have defined the stages for our pipeline, we can automate it from front to end. This requires a few extra steps.

There are multiple ways to achieve this automation, ranging from bash scripts to triggering the entire pipeline from a Jupyter Notebook. Here we will discuss how we can automate the pipeline using DVC.

The benefit of using DVC for our automation is that we don't need to make further changes to our code. Moreover, DVC provides additional benefits such as tracking dependencies and versions of our models.

In DVC, a pipeline is a sequence of stages. Each stage is a computation task that may produce data for next stages. You define a pipeline as a series of stages and dependencies, and DVC derives a directed acyclical graph (DAG) from that: in order to start with a stage, all preceding stages need to have been completed. In the DAG below, for example, the `evaluate` stage will only run once `save test features` and `train` have been completed.



Once we have this DAG set up, we can run the entire pipeline with just one command: `dvc repro`. *Repro* in this case stands for *reproduce*. Using this command, DVC will check whether anything has changed since the latest pipeline run. If it has, it will start the pipeline from there and run through all downstream stages. If `save test features` in the pipeline above was the only component that had changed, for example, DVC would run that stage as well as `evaluate` and `predict`.

[TODO: make this an admon] There is another command we can use to trigger our pipeline: `dvc exp run`. We will explore this in a future module.

Here's what we need to do to automate our pipeline with DVC:

Installing DVC

DVC is freely available as a package. The installation command will generally look something like this:

```
# Install with pip
pip install dvc

# Install with support for Amazon S3 storage
pip install "dvc[s3]" # Or use [all] to support all remote storages

# Install with conda
conda install -c conda-forge dvc
```

The precise method for installing may vary depending on your machine. Check out our installation guide over at dvc.org/doc/install for step-by-step instructions.

Once we have installed DVC, we need to initialize it. This works on a project-by-project basis, much like how you would initialize a Git repository.

You initialize DVC with this command:

```
dvc init
```

DVC will then create a number of files to track your DVC repository. Most of these are hidden files, and for now, we don't need to think about them too much. In further modules we will dive deeper into the files and directories created, and how DVC uses them.

For now it suffices to know that DVC keeps track of changes to the artifacts in our Git repository (such as datasets or pipeline stages). It does so by creating a number of metadata files. These files should be stored in our Git history, so therefore we also need to commit the initialization of DVC.

```
git add .
git commit -m "Initialize DVC"
```

Creating DVC pipeline stages

Now that DVC is initialized we can start creating the pipeline consisting of the stages we created in the previous lesson. To add a stage to our pipeline we use the `dvc run` command. We specify a name for our stage and the command that DVC should run when this stage is triggered. This is why we needed to define a CLI in the previous lesson.

```
dvc run --name data_load \
    pyton src/data_load.py --config=params.yaml
```

Here the `--name` (or `-n`) option specifies the name of our stage, in this case `data_load`, and the subsequent command is what we want to actually run. The `\` continues the command on the next line.

We can also specify dependencies and outputs when creating a stage. Dependencies are artifacts that are required before the stage is run, and outputs are artifacts that result from the stage running. Continuing from our example above, the `data_load` stage could look something like this:

```
dvc run --name data_load \
    --deps src/data_load.py \
    --outs data/iris.csv \
    --params data_load \
    python src/data_load.py --config=params.yaml
```

After we run this command, DVC will create a `dvc.yaml` file that contains the specification for our pipeline. The configuration for our `data_load` stage will be inside of the `params.yaml` file:

```
stages:
    data_load:
        cmd: python src/stages/data_load.py --config=params.yaml
        deps:
        - src/stages/data_load.py
        params:
        - base
        - data_load
        outs:
        - data/raw/iris.csv
```

The contents of the `yaml` file can be manually changed when you need to change stages. This is also an easy way to add more stages to our pipeline, for example by adding the `featurize` stage:

```
stages:
    data_load:
        cmd: python src/stages/data_load.py --config=params.yaml
        deps:
        - src/stages/data_load.py
        params:
        - base
        - data_load
        outs:
        - data/raw/iris.csv
    featurize:
        cmd: python src/stages/featurize.py --config=params.yaml
        deps:
        - data/raw/iris.csv
        - src/stages/featurize.py
        params:
        - base
        - data_load
        - featurize
        outs:
        - data/processed/featured_iris.csv
```

Notice how one of the dependencies for the second stage, namely `data/raw/iris.csv`, is the output of the first stage. This means that the `featurize` stage will not run until the first stage has completed and produced its output. This is how we define the DAG and make sure different stages are triggered in order. We can visualize this DAG in our CLI by running dvc dag.

Now when we run `dvc repro` DVC would first run `data_load`, thus executing the contents of the `data_load.py` file. After that it would run `featurize` by executing the contents of `featurize.py`. As such, we now have a first version of our automated pipeline that we can run with one simple command! From here on out we can expand the pipeline by adding the remaining stages.