**Demo: Integrating Code repository with DagsHub, DVC, Git and MLFlow**

**Step I:** **Exploratory Data Analysis (EDA)**

**Notebook:** [**https://colab.research.google.com/drive/1m-sghZTdPqtMguXIVJOeHqk-mn16Rg5Q?usp=sharing#scrollTo=iR2ALb0TB9vf**](https://colab.research.google.com/drive/1m-sghZTdPqtMguXIVJOeHqk-mn16Rg5Q?usp=sharing#scrollTo=iR2ALb0TB9vf)

**Step2: Setup**

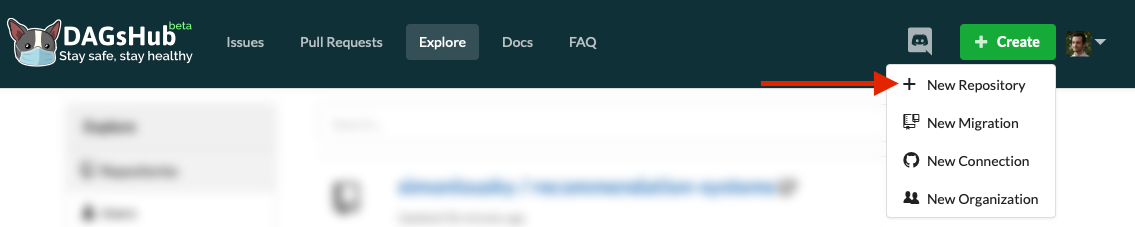
Level Overview

This level of the tutorial covers setting up our project. This includes the following tasks:

* **Creating an account and repo in DagsHub.**
* **Cloning it to your local machine.**
* **Creating a virtual Python environment using venv and installing the needed requirements.**

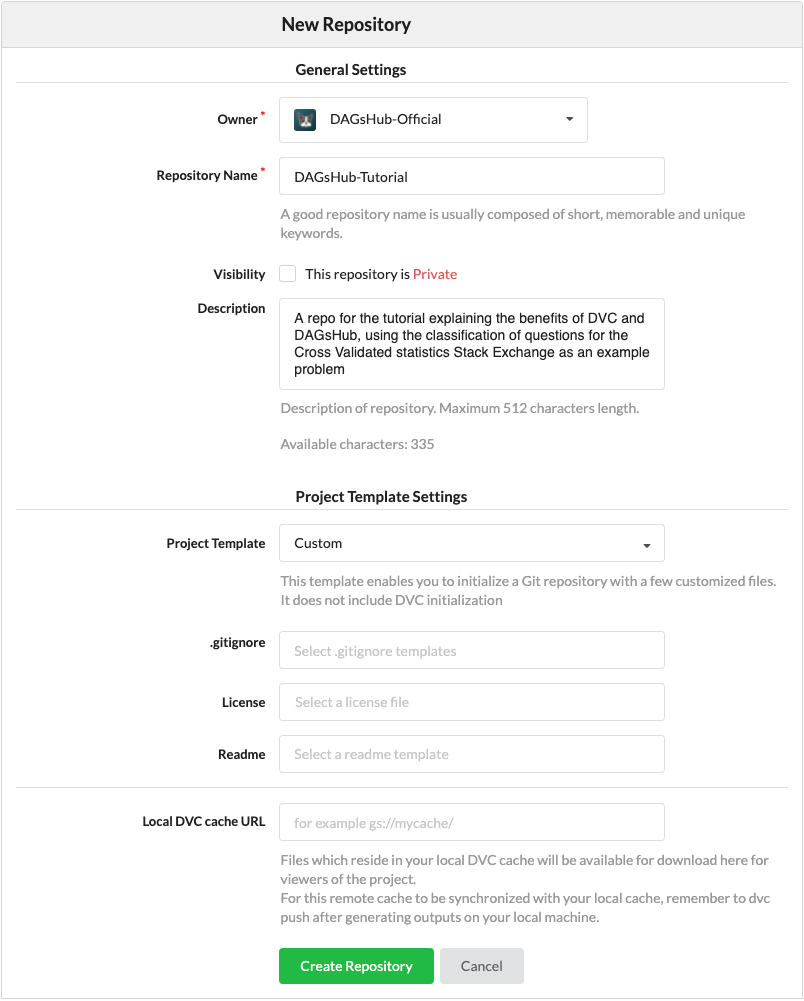
Joining DagsHub

Joining DagsHub is really easy. Just sign up. Then, after logging in, create a new repo by clicking on the plus sign and creating a repository in the navbar.



This opens up a dialog where you can set the repository name, description, and a few other options.

Screenshot: Repo creation dialog



For this tutorial, fill in the name and description, and leave everything else in the default settings. Done with repo creation. On to project initialization.

**Step2: Setting Up Our Project**

* Create a directory named **dagshub\_tutorial** for the project somewhere on your computer. Open a terminal and input the following:

**cd path/to/folder/dagshub\_tutorial**

**git init**

* Now, we will set the remote to our repo on DagsHub. This can be done using the following command:

**git remote add origin https://dagshub.com/<username>/<repo-**

**name>.git**

* Finally, let's create 2 folders for each of our main project components:

**mkdir data,outputs**

* To create and activate our virtual Python environment using venv, type the following commands into your terminal (still in the project folder):

**Linux/Mac:**

python3 -m venv .venv

echo .venv/ >> .gitignore

echo \_\_pycache\_\_/ >> .gitignore

source .venv/bin/activate

**Windows:**

python -m venv .venv

echo .venv/ >> .gitignore

echo \_\_pycache\_\_/ >> .gitignore

.\.venv\Scripts\activate

* The first command creates your virtual environment - a directory named .venv, located inside your project directory, where all the Python packages used by your project will be installed without affecting the rest of your computer.
* The second and third commands make sure the virtual environment packages and pycache are not tracked by Git.
* The fourth command activates our virtual Python environment, ensuring that any Python packages we use don't contaminate our global Python installation.
* The rest of this tutorial should be executed in the same shell session. If you exit the shell session or want to create another, make sure to activate the virtual environment in that shell session first.

**Installing Requirements**

To install the requirements for the first part of this project, simply copy paste the below packages into the requirements.txt into your project folder.

These are the direct dependencies:

makefile

dagshub==0.3.8.post2

dvc==3.49.0

dvc-s3==3.1.0

joblib==1.3.2

mlflow==2.8.0

pandas==2.1.2

scikit-learn==1.3.2

Now, to install them, type:

pip install -r requirements.txt

**Downloading the Data**

We'll keep our data in a folder named, oddly enough, data.

It's also important to remember to add this folder to .gitignore! We definitely don't want to accidentally commit large data files to Git.

The following commands should take care of everything:

mkdir -p data

echo /data/ >> .gitignore

wget https://dagshub-public.s3.us-east-2.amazonaws.com/tutorials/stackexchange/CrossValidated-Questions-Nov-2020.csv -O data/CrossValidated-Questions.csv

**Committing Progress to Git**

Let's check the Git status of our project:

git status -s

**Now let's commit this to Git and push to DagsHub using the command line:**

git add .

git commit -m "Initialized project"

git push -u origin master

You can now see the setup files on your DagsHub repo. So far so good!

**Step3: Data Versioning**

Writing the basic training code

Let's use our existing insights and code from the data exploration level to get started with a single Python script which:

* Loads the data
* Processes the data
* Trains a classification model
* Evaluates the trained model and reports relevant metrics.

We'll put all this in a single script called main.py for now. You can download the complete file here: [main.py](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial/raw/3d56f5ad4202e6d96a01a33dec2b380f387342dd/main.py) and save it to your project folder.

Or copy/paste it the below code in main.py

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import roc\_auc\_score, average\_precision\_score, accuracy\_score, precision\_score, recall\_score, \

f1\_score

from sklearn.model\_selection import train\_test\_split

def feature\_engineering(raw\_df):

df = raw\_df.copy()

df['CreationDate'] = pd.to\_datetime(df['CreationDate'])

df['CreationDate\_Epoch'] = df['CreationDate'].astype('int64') // 10 \*\* 9

df['MachineLearning'] = df['Tags'].str.contains('machine-learning').fillna(False)

df = df.drop(columns=['Id', 'Tags'])

df['Title\_Len'] = df.Title.str.len()

df['Body\_Len'] = df.Body.str.len()

# Drop the correlated features

df = df.drop(columns=['FavoriteCount'])

df['Text'] = df['Title'].fillna('') + ' ' + df['Body'].fillna('')

return df

def fit\_tfidf(train\_df, test\_df):

tfidf = TfidfVectorizer(max\_features=25000)

tfidf.fit(train\_df['Text'])

train\_tfidf = tfidf.transform(train\_df['Text'])

test\_tfidf = tfidf.transform(test\_df['Text'])

return train\_tfidf, test\_tfidf, tfidf

def fit\_model(train\_X, train\_y):

clf\_tfidf = LogisticRegression(solver='sag')

clf\_tfidf.fit(train\_X, train\_y)

return clf\_tfidf

def eval\_model(clf, X, y):

y\_proba = clf.predict\_proba(X)[:, 1]

y\_pred = clf.predict(X)

return {

'roc\_auc': roc\_auc\_score(y, y\_proba),

'average\_precision': average\_precision\_score(y, y\_proba),

'accuracy': accuracy\_score(y, y\_pred),

'precision': precision\_score(y, y\_pred),

'recall': recall\_score(y, y\_pred),

'f1': f1\_score(y, y\_pred),

}

if \_\_name\_\_ == '\_\_main\_\_':

print('Loading data...')

df = pd.read\_csv('data/CrossValidated-Questions.csv')

train\_df, test\_df = train\_test\_split(df)

del df

train\_df = feature\_engineering(train\_df)

test\_df = feature\_engineering(test\_df)

print('Fitting TFIDF...')

train\_tfidf, test\_tfidf, tfidf = fit\_tfidf(train\_df, test\_df)

print('Fitting classifier...')

train\_y = train\_df['MachineLearning']

model = fit\_model(train\_tfidf, train\_y)

train\_metrics = eval\_model(model, train\_tfidf, train\_y)

print('Train metrics:')

print(train\_metrics)

test\_metrics = eval\_model(model, test\_tfidf, test\_df['MachineLearning'])

print('Test metrics:')

print(test\_metrics)

We can see that the script works by running:

python main.py

The output should look more or less like this:

Loading data...

Fitting TFIDF...

Fitting classifier...

Train metrics:

{'roc\_auc': 0.9264913034812006, 'average\_precision': 0.5975913323426428, 'accuracy': 0.90584, 'precision': 0.7115251897860594, 'recall': 0.24879343629343628, 'f1': 0.3686751296263186}

Test metrics:

{'roc\_auc': 0.877444815878766, 'average\_precision': 0.4621588861511042, 'accuracy': 0.89408, 'precision': 0.5871369294605809, 'recall': 0.20099431818181818, 'f1': 0.29947089947089944}

It's a good idea to commit this to Git so we can always get back to a working version:

git add main.py

git commit -m "Basic training script"

**Things to improve in the script**

That script was nice just to see that everything works, but before we start experimenting, there are some issues we should fix:

* Right now, the test set will be different every time we run the script.  
  If we want to compare different runs, we need to make sure the test set stays the same across different runs or risk introducing noise and uncertainty into our decision making.  
  To fix this, we should do the train-test split as a separate step which we run only once, and train the model in a different step which we will run several times, with different configurations, using the same test set.
* It's also a good idea to [stratify our train-test split](https://scikit-learn.org/0.22/modules/cross_validation.html#cross-validation-iterators-with-stratification-based-on-class-labels) by the MachineLearning class, since our classes are imbalanced.
* We didn't set random seeds - to get reproducible research and leave as little to chance as possible, this is also an important practice.
* We should save our trained model as a file - otherwise, how will we use it in real life?

Simple things first - let's create a directory to save our outputs:

mkdir -p outputs

echo /outputs/ >> .gitignore

Note that our outputs are also in .gitignore - you usually won't want to save these using Git, especially if dealing with large models like neural networks.  
In our case, the [TFIDF](https://scikit-learn.org/0.22/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.text.TfidfVectorizer) object is fairly large.

Now, we'll mostly change our main function so that it supports running one of the two steps (train-test split and training), as well as a few other code changes to address all the points above. You can download the complete file here: [main.py](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial/raw/b5fd2a63674aba1d156511509c1572c8cddbfdb5/main.py)

import argparse

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import roc\_auc\_score, average\_precision\_score, accuracy\_score, precision\_score, recall\_score, \

f1\_score

from sklearn.model\_selection import train\_test\_split

import joblib

# Consts

CLASS\_LABEL = 'MachineLearning'

train\_df\_path = 'data/train.csv.zip'

test\_df\_path = 'data/test.csv.zip'

def feature\_engineering(raw\_df):

df = raw\_df.copy()

df['CreationDate'] = pd.to\_datetime(df['CreationDate'])

df['CreationDate\_Epoch'] = df['CreationDate'].astype('int64') // 10 \*\* 9

df = df.drop(columns=['Id', 'Tags'])

df['Title\_Len'] = df.Title.str.len()

df['Body\_Len'] = df.Body.str.len()

# Drop the correlated features

df = df.drop(columns=['FavoriteCount'])

df['Text'] = df['Title'].fillna('') + ' ' + df['Body'].fillna('')

return df

def fit\_tfidf(train\_df, test\_df):

tfidf = TfidfVectorizer(max\_features=25000)

tfidf.fit(train\_df['Text'])

train\_tfidf = tfidf.transform(train\_df['Text'])

test\_tfidf = tfidf.transform(test\_df['Text'])

return train\_tfidf, test\_tfidf, tfidf

def fit\_model(train\_X, train\_y, random\_state=42):

clf\_tfidf = LogisticRegression(solver='sag', random\_state=random\_state)

clf\_tfidf.fit(train\_X, train\_y)

return clf\_tfidf

def eval\_model(clf, X, y):

y\_proba = clf.predict\_proba(X)[:, 1]

y\_pred = clf.predict(X)

return {

'roc\_auc': roc\_auc\_score(y, y\_proba),

'average\_precision': average\_precision\_score(y, y\_proba),

'accuracy': accuracy\_score(y, y\_pred),

'precision': precision\_score(y, y\_pred),

'recall': recall\_score(y, y\_pred),

'f1': f1\_score(y, y\_pred),

}

def split(random\_state=42):

print('Loading data...')

df = pd.read\_csv('data/CrossValidated-Questions.csv')

df[CLASS\_LABEL] = df['Tags'].str.contains('machine-learning').fillna(False)

train\_df, test\_df = train\_test\_split(df, random\_state=random\_state, stratify=df[CLASS\_LABEL])

print('Saving split data...')

train\_df.to\_csv(train\_df\_path)

test\_df.to\_csv(test\_df\_path)

def train():

print('Loading data...')

train\_df = pd.read\_csv(train\_df\_path)

test\_df = pd.read\_csv(test\_df\_path)

print('Engineering features...')

train\_df = feature\_engineering(train\_df)

test\_df = feature\_engineering(test\_df)

print('Fitting TFIDF...')

train\_tfidf, test\_tfidf, tfidf = fit\_tfidf(train\_df, test\_df)

print('Saving TFIDF object...')

joblib.dump(tfidf, 'outputs/tfidf.joblib')

print('Training model...')

train\_y = train\_df[CLASS\_LABEL]

model = fit\_model(train\_tfidf, train\_y)

print('Saving trained model...')

joblib.dump(model, 'outputs/model.joblib')

print('Evaluating model...')

train\_metrics = eval\_model(model, train\_tfidf, train\_y)

print('Train metrics:')

print(train\_metrics)

test\_metrics = eval\_model(model, test\_tfidf, test\_df[CLASS\_LABEL])

print('Test metrics:')

print(test\_metrics)

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

subparsers = parser.add\_subparsers(title='Split or Train step:', dest='step')

subparsers.required = True

split\_parser = subparsers.add\_parser('split')

split\_parser.set\_defaults(func=split)

train\_parser = subparsers.add\_parser('train')

train\_parser.set\_defaults(func=train)

parser.parse\_args().func()

Now we've updated the script, lets run both it's stages:

Train-test split step:

python3 main.py split

Here the output is:

Loading data...

Saving split data...

Training step:

python3 main.py train

Here the output is:

Loading data...

Engineering features...

Fitting TFIDF...

Saving TFIDF object...

Training model...

Saving trained model...

Evaluating model...

Train metrics:

{'roc\_auc': 0.9254119108511665, 'average\_precision': 0.5958732255869963, 'accuracy': 0.90528, 'precision': 0.7059219380888291, 'recall': 0.25192122958693564, 'f1': 0.37132743362831855}

Test metrics:

{'roc\_auc': 0.8808051840002658, 'average\_precision': 0.46366929635715587, 'accuracy': 0.89472, 'precision': 0.5756302521008403, 'recall': 0.19740634005763688, 'f1': 0.2939914163090129}

And let's commit these changes to Git:

git status -s

M .gitignore

M main.py

git add .gitignore main.py

git commit -m "Training script with outputs"

**Installing DVC**

Installing DVC is as simple as To start, we need to initialize our git repo to also use DVC for data versioning:

dvc init

This is somewhat similar to the .git folder contained in every git repo, except some of its contents will be tracked using git.

* .dvc/config is similar to .git/config. By default, it's empty. More on this later on.
* .dvc/.gitignore makes sure git ignores DVC internal files that shouldn't be tracked by Git.
* .dvc/plots contains predefined templates for plots you can generate using dvc - more info [here](https://dvc.org/doc/command-reference/plots).
* .dvc/tmp is used by DVC to store temporary files, this shouldn't interest the average user.
* .dvc/cache doesn't exist yet, but it is where DVC will keep the different versions of our data files. It's very similar in principle to .git/objects.

Some of the files generated by dvc init should be tracked by Git, so let's start by committing that:

git add .dvc

git commit -m "dvc init"

Instructing DVC to track data and outputs

dvc add data

dvc add outputs

You should see two new metadata files, created by DVC:

**git status -s**

?? data.dvc

?? outputs.dvc

**type data.dvc**

outs:

- md5: 714b1181c5d7cb9dda66272be8be33ac.dir

path: data

**cat outputs.dvc**

outs:

- md5: bc939fd1899e52dd1a5c65be0443986a.dir

path: outputs

Now, we can commit these .dvc files to Git:

git add data.dvc outputs.dvc

git commit -m "Added data and outputs to DVC"

**Another training run**

Now, let's try re-running the training with a different configuration.

We'll try to use a SGDClassifier with loss='modified\_huber' (since this type of loss supports all of the metric types we calculate).

You can download the complete file here: [main.py](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial/raw/fa46f37d6b120fe271b23f5e2f24965b40d12b7c/main.py)

Or copy paste from here

import argparse

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import SGDClassifier

from sklearn.metrics import roc\_auc\_score, average\_precision\_score, accuracy\_score, precision\_score, recall\_score, \

f1\_score

from sklearn.model\_selection import train\_test\_split

import joblib

# Consts

CLASS\_LABEL = 'MachineLearning'

train\_df\_path = 'data/train.csv.zip'

test\_df\_path = 'data/test.csv.zip'

def feature\_engineering(raw\_df):

df = raw\_df.copy()

df['CreationDate'] = pd.to\_datetime(df['CreationDate'])

df['CreationDate\_Epoch'] = df['CreationDate'].astype('int64') // 10 \*\* 9

df = df.drop(columns=['Id', 'Tags'])

df['Title\_Len'] = df.Title.str.len()

df['Body\_Len'] = df.Body.str.len()

# Drop the correlated features

df = df.drop(columns=['FavoriteCount'])

df['Text'] = df['Title'].fillna('') + ' ' + df['Body'].fillna('')

return df

def fit\_tfidf(train\_df, test\_df):

tfidf = TfidfVectorizer(max\_features=25000)

tfidf.fit(train\_df['Text'])

train\_tfidf = tfidf.transform(train\_df['Text'])

test\_tfidf = tfidf.transform(test\_df['Text'])

return train\_tfidf, test\_tfidf, tfidf

def fit\_model(train\_X, train\_y, random\_state=42):

clf\_tfidf = SGDClassifier(loss='modified\_huber', random\_state=random\_state)

clf\_tfidf.fit(train\_X, train\_y)

return clf\_tfidf

def eval\_model(clf, X, y):

y\_proba = clf.predict\_proba(X)[:, 1]

y\_pred = clf.predict(X)

return {

'roc\_auc': roc\_auc\_score(y, y\_proba),

'average\_precision': average\_precision\_score(y, y\_proba),

'accuracy': accuracy\_score(y, y\_pred),

'precision': precision\_score(y, y\_pred),

'recall': recall\_score(y, y\_pred),

'f1': f1\_score(y, y\_pred),

}

def split(random\_state=42):

print('Loading data...')

df = pd.read\_csv('data/CrossValidated-Questions.csv')

df[CLASS\_LABEL] = df['Tags'].str.contains('machine-learning').fillna(False)

train\_df, test\_df = train\_test\_split(df, random\_state=random\_state, stratify=df[CLASS\_LABEL])

print('Saving split data...')

train\_df.to\_csv(train\_df\_path)

test\_df.to\_csv(test\_df\_path)

def train():

print('Loading data...')

train\_df = pd.read\_csv(train\_df\_path)

test\_df = pd.read\_csv(test\_df\_path)

print('Engineering features...')

train\_df = feature\_engineering(train\_df)

test\_df = feature\_engineering(test\_df)

print('Fitting TFIDF...')

train\_tfidf, test\_tfidf, tfidf = fit\_tfidf(train\_df, test\_df)

print('Saving TFIDF object...')

joblib.dump(tfidf, 'outputs/tfidf.joblib')

print('Training model...')

train\_y = train\_df[CLASS\_LABEL]

model = fit\_model(train\_tfidf, train\_y)

print('Saving trained model...')

joblib.dump(model, 'outputs/model.joblib')

print('Evaluating model...')

train\_metrics = eval\_model(model, train\_tfidf, train\_y)

print('Train metrics:')

print(train\_metrics)

test\_metrics = eval\_model(model, test\_tfidf, test\_df[CLASS\_LABEL])

print('Test metrics:')

print(test\_metrics)

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

subparsers = parser.add\_subparsers(title='Split or Train step:', dest='step')

subparsers.required = True

split\_parser = subparsers.add\_parser('split')

split\_parser.set\_defaults(func=split)

train\_parser = subparsers.add\_parser('train')

train\_parser.set\_defaults(func=train)

parser.parse\_args().func()

Now run

python main.py train

Now, we have a new version of a trained model in outputs/model.joblib.

We can see that by running

dvc status

Output will look like this

changed outs:

modified: outputs

To record the md5 of the new model, and save it to .dvc/cache, we can run:

dvc commit -f

This updates the outputs.dvc file with the hash of the new output files, as well as store the new model version in .dvc/cache:

git diff outputs.dvc

Finally, we commit everything to git:

git add main.py outputs.dvc

git commit -m "Tried SGDClassifier with modified\_huber loss"

**Reproducing an old experiment**

Now, after we run a few experiments and want to reproduce one which looked promising, we can just do that with a git & dvc checkout

For example, to return to our previous commit, we can do:

git checkout HEAD~

dvc checkout

And we will then have the older version of the model at outputs/model.joblib.

**Pushing code, data, and models to DagsHub**

It's great to have saved versions of our data and models in our local workspace, but what if we have team members? Or if we want to continue work on some other machine?

DagsHub has you covered - not only can you push your Git code history to DagsHub, but you can also push (and later pull) all DVC managed files!

First of all, make sure you return to the latest version of the master branch:

git checkout master

dvc checkout

Now, we need to define DagsHub as our DVC remote.

**If you don't know what your DagsHub password is (for instance, if you signed up via Github), then first**[**create an access token**](https://dagshub.com/user/settings/tokens)**and use that token instead of a password.**

Add credentials and then run these commands

Now, copy the following commands into your terminal:

set DAGSHUB\_USER="Username: "

set DAGSHUB\_REPO="Repo name: "

set DAGSHUB\_TOKEN="Token: "

dvc remote add origin s3://dvc

dvc remote modify origin endpointurl https://dagshub.com/%DAGSHUB\_USER%/%DAGSHUB\_REPO%.s3

dvc remote modify origin --local access\_key\_id %DAGSHUB\_TOKEN%

dvc remote modify origin --local secret\_access\_key %DAGSHUB\_TOKEN%

set DAGSHUB\_TOKEN=

You can see that some DVC stores some configurations in .dvc/config, which should be committed to Git:

git diff

So, let's commit these configuration changes to git:

git add .dvc/config

git commit -m "Configured the DVC remote"

And push to our repo:

git push -u origin master

dvc remote default origin

dvc push --all-commits

Now, any future collaborator can git clone and then dvc pull the data and models from any version.

What's more, you and your collaborators can now **explore and download the code and model files directly from the DagsHub UI!**

**Level 3 - Experimentation**

Now that we [have a project and the raw data](https://dagshub.com/docs/experiment_tutorial/1_setup/), the next step is to try different types of data processing and models to learn what works better.

In real life, this part is often where things get complicated, difficult to remember, track, and reproduce.

The [data versioning](https://dagshub.com/docs/experiment_tutorial/2_data_versioning/) we set up will help us keep track of data and model versions, and easily reproduce and share them. But how will we compare the different experiments we're going to run?

It's a sadly common tale, of a data scientist getting really good results with some combination of data, model, and hyperparameters, only to later forget exactly what they did and having to rediscover it. This situation gets much worse when multiple team members are involved.

**This level of the tutorial shows how using [DagsHub's integration with MLflow](https://dagshub.com/docs/feature_guide/experiment_tracking/) allows us to easily keep a reproducible record of our experiments, both for ourselves and for our teammates.**

The full resulting project can be found here:

[**See the project on DagsHub**](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial)

Using MLflow to track experiments

We're now at a point where we can start experimenting with different models, hyperparameters, and data preprocessing. However, we don't have a way to record and compare results yet.

To solve this, we can use [MLflow](https://mlflow.org/), which will record information about each of our experiments to the MLflow server provided with each DagsHub repository. Then, we can [search, visualize, and compare our experiments](https://dagshub.com/docs/feature_guide/experiment_tracking/) on DagsHub.

MLflow is already installed since it was already [included in our requirements.txt](https://dagshub.com/DAGsHub-Official/DAGsHub-Tutorial/raw/2bf23a5d7aaf077d4e21b9229aab67a56d632a6c/requirements.txt), so we can start right away with adjusting our code.

Alternatively, you can download the complete file here: [***main.py***](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial/raw/master/main.py)

add the dagshub credentials in main.py file

Notice the calls made to MLflow to log the hyperparameters of the experiment as well as metrics.

Commit the changed file:

git add main.py

git commit -m "Added experiment logging"

Now, we can run the first experiment which will be recorded:

python main.py train

Now, let's record this baseline experiment's parameters and results. Remember that since we trained a new model, our outputs have changed:

dvc status

So we should commit them to DVC before committing to Git:

dvc commit -f outputs.dvc

# DVC will change the contents of outputs.dvc, to record the new hashes of the models saved in the outputs directory

git add outputs.dvc

git commit -m "Baseline experiment"

Running a few more experiments

Now, we can let our imaginations run free with different configurations for experiments.

Here are a few examples (with a link to the code for them):

* We can change the type of model:
  + [AdaBoost](https://scikit-learn.org/0.22/modules/generated/sklearn.ensemble.AdaBoostClassifier.html) model – [main.py with AdaBoost](https://dagshub.com/DAGsHub-Official/DAGsHub-Tutorial/raw/29f9a9cd4bc659ffd082ea0382cedd2b0aba13e3/main.py)
  + [Random Forest](https://scikit-learn.org/0.22/modules/generated/sklearn.ensemble.RandomForestClassifier.html) model – [main.py with Random Forest](https://dagshub.com/DAGsHub-Official/DAGsHub-Tutorial/raw/a2171fa313e120aeac3220d17126ab40385aac20/main.py)
* We can play around with parameters:
  + We can try out different values for random forest's max\_depth parameter – [main.py with different max depth](https://dagshub.com/DAGsHub-Official/DAGsHub-Tutorial/raw/285c153742ccc4671374be487c71b9aea76b1538/main.py)
* Etc.

After each such modification, we'll want to save our code and models. We make sure to commit our code first, because MLflow will point any runs done to a particular commit, if run from a Git repository. This lets you match up code changes with experiment results.

We can do that by running a set of commands like this:

git add main.py

git commit -m "Description of the experiment"

python3 main.py train

dvc commit -f outputs.dvc

git add outputs.dvc

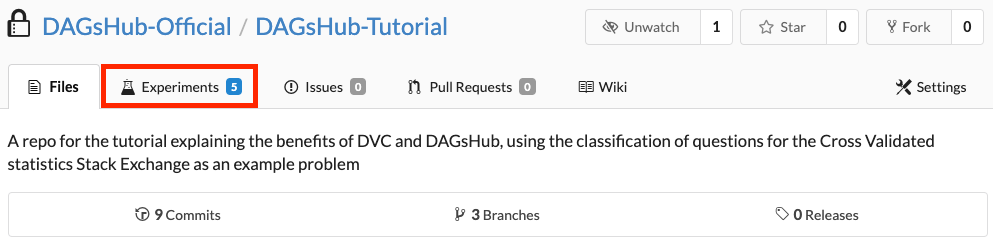
git commit -m "Results of the experiment"

Of course, it's a good (but optional) idea to change the commit message to something meaningful.

***Our recommendation*** is to separate distinct experiments (for example, different types of models) into separate branches, while smaller changes between runs (for example, changing model parameters) are consecutive commits on the same branch.

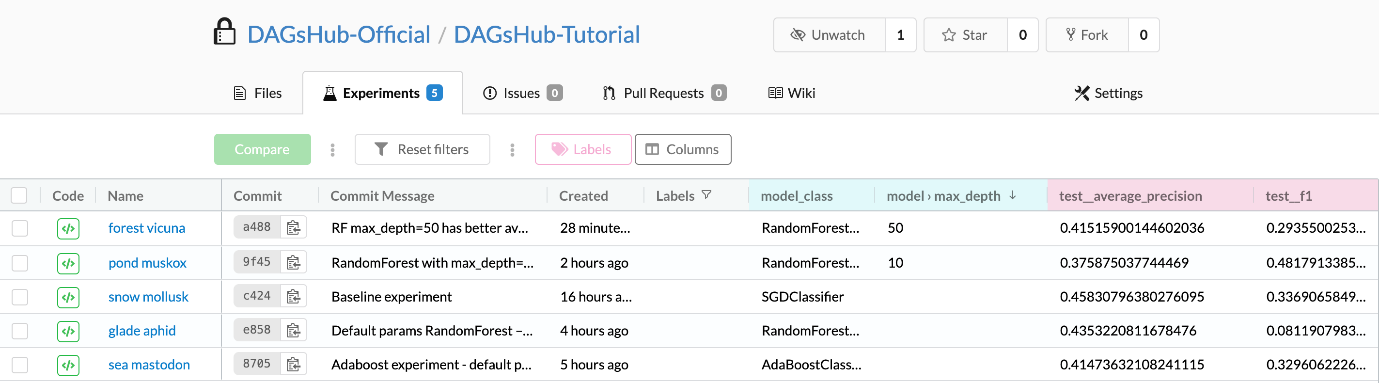
Visualizing experiments on DagsHub[¶](https://dagshub.com/docs/experiment_tutorial/3_experiments/" \l "visualizing-experiments-on-dagshub" \o "Permanent link)

To see our experiments visualized, we can navigate to the "Experiments" tab in our DagsHub repo:



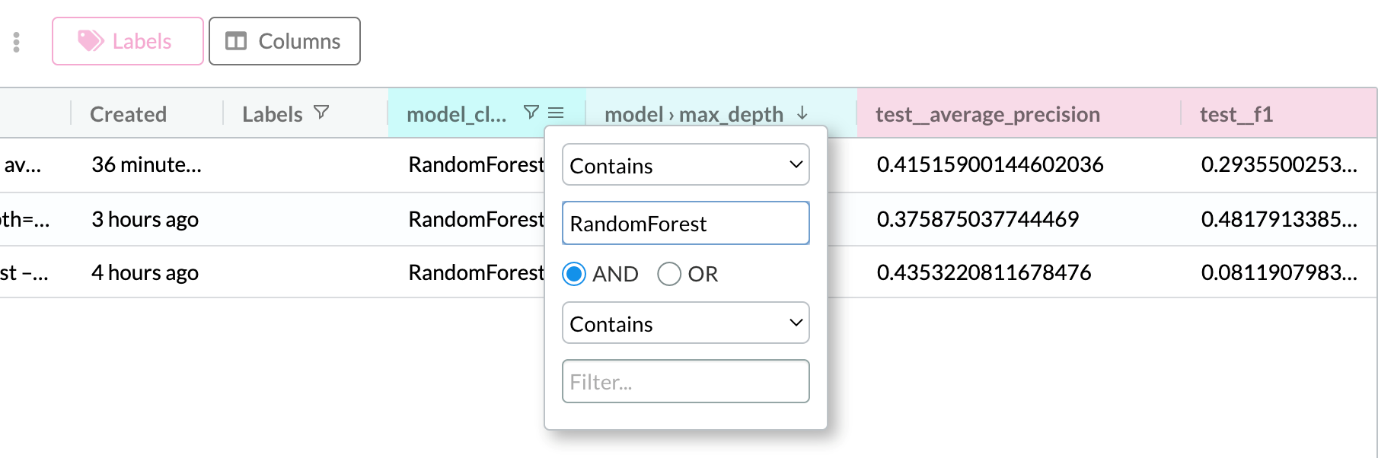
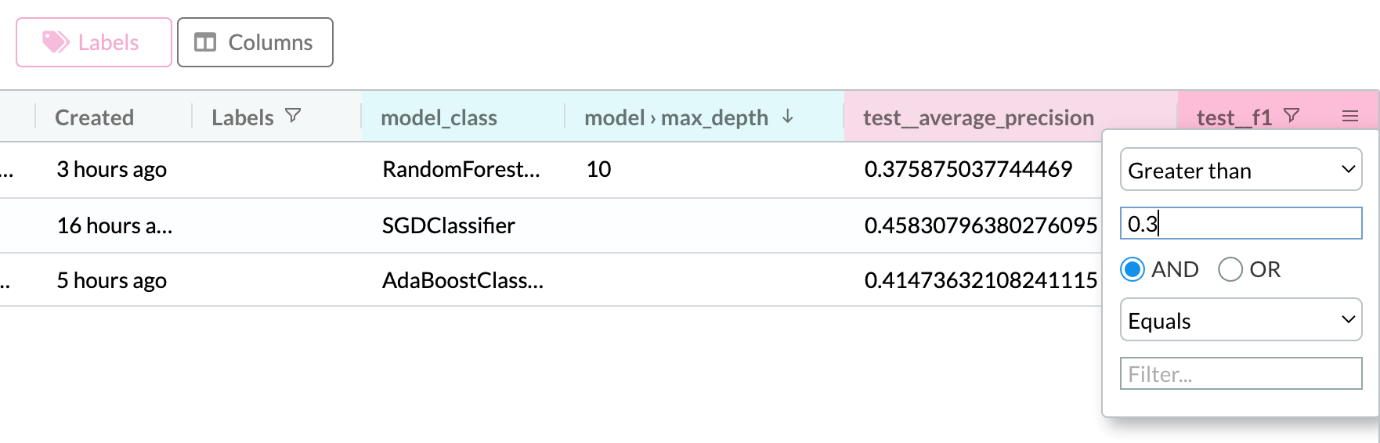
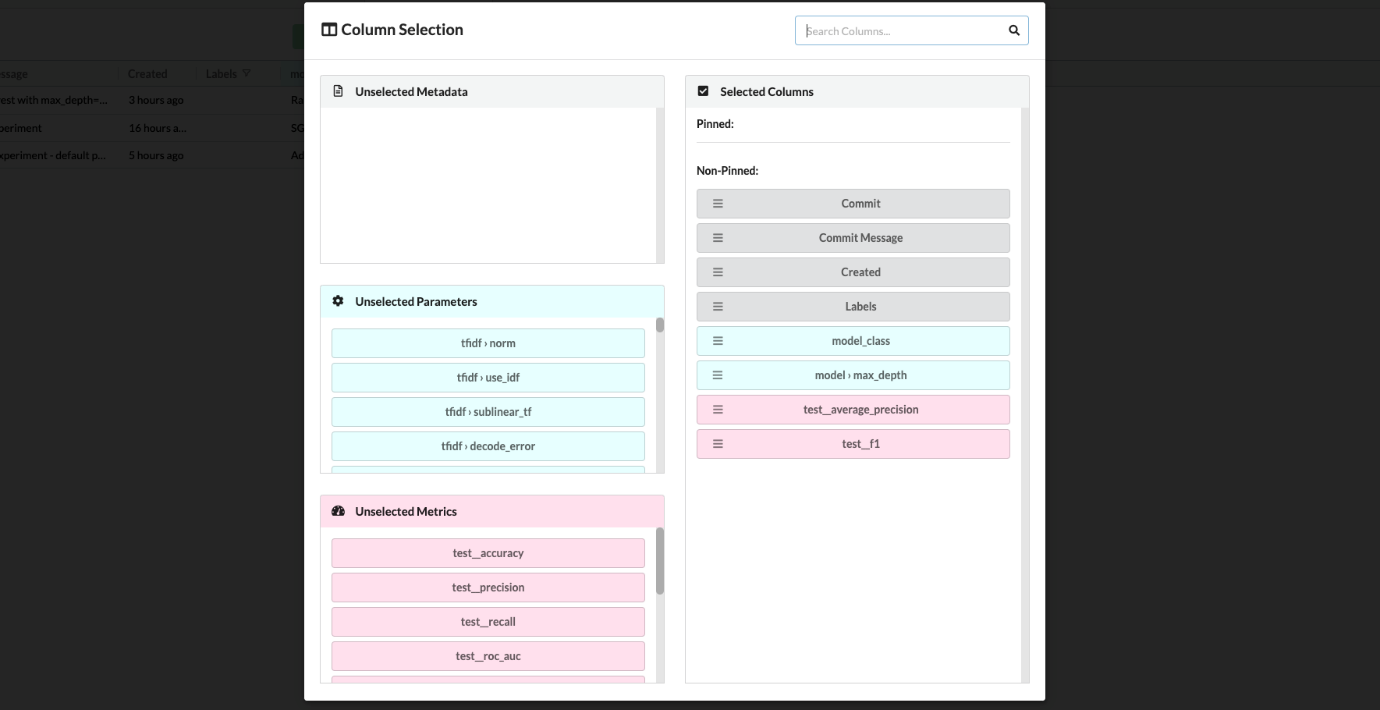
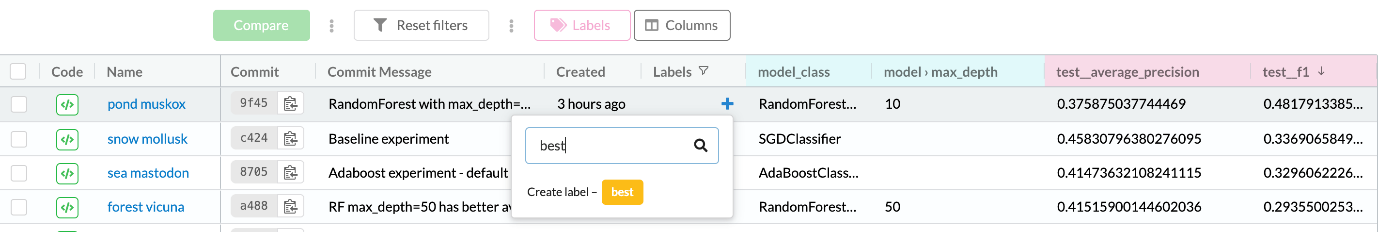
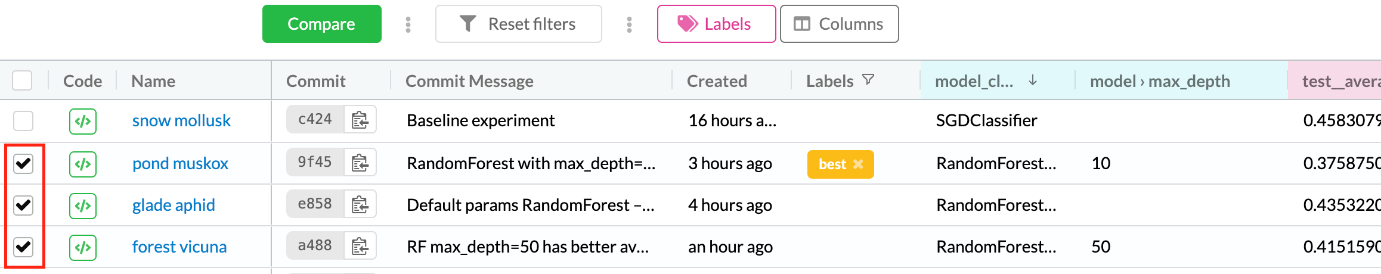
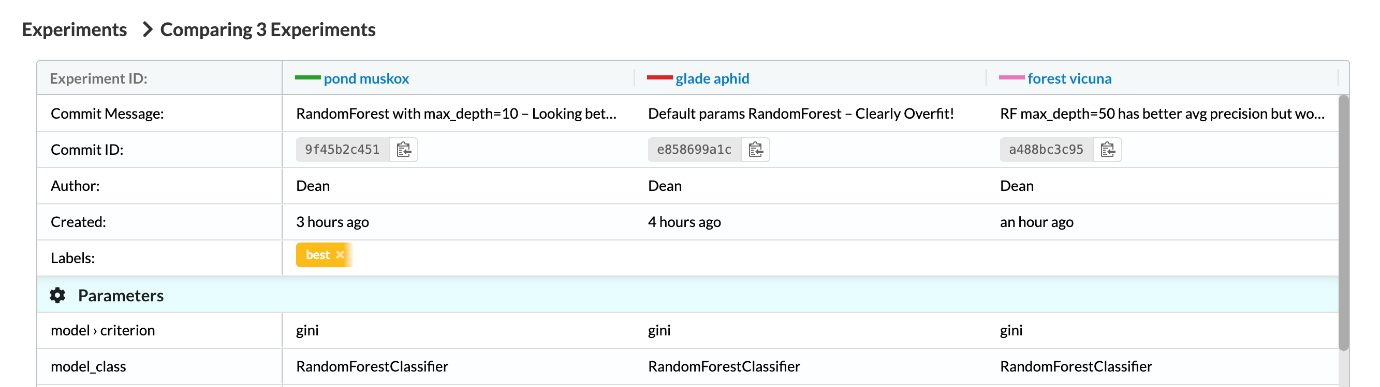
If you want to interact with the experiments table of our pre-made repo, [you can find it here](https://dagshub.com/DagsHub-Official/DagsHub-Tutorial/experiments/#/).

Here is what our experiments table looked like at this stage, after running a few different configurations:

[](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table.png)

This table has a row for each detected experiment in your Git history, showing its information and columns for hyperparameters and metrics. Each of these rows corresponds to a single Git commit.

You can interact with this table to:

* **Filter experiments by hyperparameters:** [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table_filter_model_class.png)
* **Filter & sort experiments by numeric metric values** - i.e. easily find your best experiments: [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table_filter_f1.png)
* **Choose the columns to display in the table** - by default, we limit the number of columns to a reasonable number: [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table_columns.png)
* **Label experiments for easy filtering.**  
  Experiments labeled hidden are automatically hidden by default, but you can show them anyway by removing the default filter. [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table_labels.png)
* **See the commit IDs and code of each experiment, for easy reproducibility.**
* **Select experiments for comparison.**  
  For example, we can check the top 3 best experiments: [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_table_select.png)  
  Then click on the Compare button to see all 3 of them side by side: [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_compare_1.png)  
  [](https://dagshub.com/docs/experiment_tutorial/assets/experiments_compare_2.png)

Next Steps

The next logical steps for this project would be to:

* Experiment more with data preprocessing and cleaning, and do so in a separate step to save processing time.
* Add more training data, and see if it improves results.
* Store the trained models & pipelines in a centrally accessible location, so it's easy to deploy them to production or synchronize with team members.
* [Track different versions of raw data, processed data, and models using DVC](https://dagshub.com/docs/use_cases/reproduce_experiment_results/), to make it easy for collaborators (and yourself) to reproduce experiments.

Stay tuned for updates to this tutorial, where we will show you how to implement these steps.

In the meantime, **if you want to learn more about how to use**[**DVC**](https://dvc.org/)**with DagsHub, you can follow our**[**other tutorial**](https://dagshub.com/docs/tutorial/)**, which focuses on data pipeline versioning & reproducibility.**