Day 2 - Guidelines

This is day 2 of the Advanced Testing training program.

> In this workshop, instead of focusing on the end-to-end execution of the program, we will zoom in a little bit and start testing specific parts of the systems.

We will talk about Integration, data testing and do a short intro to model performance comparison. Also, we will start using `pytest`, a very popular testing framework for Python.

---

0. Setup

In the `tests/` directory that you created yesterday, you need to create an empty `integration/` directory

We will use these to store the tests we will create today.

1. Testing input data

`Suggested time: 20-30 min`

Not only do we need to test the internal of our system, but we can also use automate tests for testing the inputs of our system. In our case, the data is a structured dataset that has been extracted from a data lake, and that is available on your disk in the `data/` directory.

> **We will consider this dataset to be the ground truth for future extractions,** and will make sure those extractions follow the specifications that we derived from this one.

Create a file called `integration/test_data.py`, and implement all the tests required to check the validity of input data. Make sure they pass before moving on!

Pytest should speed up your development process 🙂

💡 **Tips**

There are quite a few things you can test about your dataset. Here are some ideas that you should implement:

- It includes all the columns we're expecting
- There are no additional columns
- The data types of each columns are what we expect them to be
- ...

Can you think about any other test you could run? Pick a new one, implement it and tell us about it during the live session.

---

2. Testing serve/predict integration

`Suggested time: 30-40 min`

As you already know, the serving system is just a thin wrapper around the predict system: its goal is only to enable getting predictions over HTTP.

> An important integration test would be to check that these two independent systems manage to exchange data with each other.

As with end-to-end testing, there is no need to have a deep knowledge about the inner implementation. The only knowledge needed are information that can be found in the REST API documentation, and most importantly:

- The route on which prediction serving is accessible

- The format of the expected request's body
- The kind of responses we should expect from our serving app, provided that the connection with the predict app is working well

You can access the API documentation here:

API Documentation

> It makes total sense that we need more information about the inner working of our application to perform integration tests, as they are made to check a deeper level of abstraction than end-to-end tests.

💡 **Very Important Tips**

To focus solely on integration between web serving and prediction, it's also good to know that the serving system relies on a Flask app: this info can simplify the testing process.

Indeed, as you saw yesterday, to test a web app we need a web server... However, **most web frameworks, such as Flask, ships with testing tooling**, such as a basic client that can be used in your tests to make fake HTTP requests. Using this testing client allows you to emulate real HTTP requests, without the need to setup/tear-down a server. 🎉🍾

In the official docs, you will see an example of how to use the Flask testing client. Adapt yours to suit your needs.

Read this documentation page carefully! Not only does it show you how to use the web client supplied by Flask, but it also gives you info on how to send JSON data.

Also, note that **our fixture doesn't need to be as complex as this example.** We only need a client, without database setup or file creation/removal.

2.1 A client fixture

Create a Python file dedicated to this test, and start by implementing the fixture you need to make requests on your Flask app.

```
# tests/integration/test_serving_predictions.py
...
import pytest

from src.serve.app import app as flask_app
```

```
...

flask_app.config['TESTING'] = True  # not stricly necessary here


@pytest.fixture
def client():
    # TODO: implement the Flask client fixture
```

2.2 A single test

Yesterday, you developed a function to automate the generation of random data points. Using it, write a Python test that will check that the serving route can communicate with the prediction route and returns a single prediction value if we send the value corresponding to a single data point.

```
# tests/integration/test_serving_predictions.py
...


def test_serving_predictions(client):
    # TODO: implement a test for a prediction on a single
    #    random data point.
    # 1. Get the value
    # 2. Use the client fixture to make a request.
    # 3. Check the response
    # (Cf.API docs to know what assertions need to be implemented)
```

Check that your test is working by running:

```
$ pytest -v tests/integration/test_serving_predictions.py
```

2.3 Parametrization

Now that your test is implemented for a single data point, add a parametrization so Pytest can run test on different counts of data points at the same time. Check the official docs if needed.

For instance, you can test the results with 1, 10 and 100 data points.

Run the tests from a terminal, and check that you have the expected number of tests that run: you must have as many test cases that you have parameters in your parametrization.

```
$ pytest -v tests/integration/test_serving_prediction.py
```

2.4 [bonus] More parametrization

If you're running behind schedule, you can go directly to exercise #3.

Up until now, you were only checking results for successful requests (HTTP status code `200`).

The documentation states that in some cases, the web app may return two different types of error: `400` for errors due to client, and `500` for errors due to the app itself.

Add some tests to check that the system returns the expected responses given an erroneous usage. You may add these new tests as another parametrization, to avoid repeating yourself.

Now, run `pytest` one more time:

```
$ pytest -v tests/integration/test_serving_prediction.py
```

You must end up with **at least** 5 successful checks (probably more): 3 for the successful predictions on 1, 10 and 100 data points, 1 for a failed request due to a client error, and 1 for a failed request due to an app error.

---

3. Testing the ML pipeline

`Suggested time: 30-40 min`

We will focus on the **modeling** part of the pipeline. The current modeling pipeline uses Scikit-Learn pipeline, and has 2 steps:

1. Feature generation using a Scikit-Learn's `FeatureUnion` object
2. ML model/estimator (Scikit-Learn's implementation of Random Forest)

Integration tests will help us pinpoint which part of the system is failing.

You can check the source code in `src/train/features/__init__.py`. It is simply a Scikit-Learn `FeatureUnion` with a feature store as a constructor. For this demo project, the feature store is very simple and is the exact input format expected by the `FeatureUnion`.

Next, take a look at `src/train/pipeline.py`: this is where we instantiate the Random Forest and the overall modeling pipeline using Scikit-Learn `Pipeline` object.

NOTE

Using a `Pipeline` object for such a simple 2-step pipeline can seem overkill, but is a good practice to avoid data leakage during our validation process. It's also a good starting point for implementing more advanced engineering techniques, such as caching.

We can think of the pipeline as a directed graph with 2 nodes. In practice, modeling pipelines can have more nodes and actually, if you dig deeper into the source code, you'll see that the `features_generator` itself consists of multiple sub-nodes (we will explore and test this tomorrow).

Create a file named `integration/test_ml_pipeline.py` and implement the following tests:

- Test that the pipeline can be fitted
- Test that the pipeline can predict after being fitted

  These 2 tests will tell you if the pipeline can do its job properly or not, but if not, it won't tell you which part of it is failing.
- Test that the `features_generator` can be fitted
- Test that the `features_generator` can transform after being fitted
- Test that the `estimator` can be fitted on the output of the fitted `features_generator`
- Test that the `estimator` can predict on the output of the fitted `features_generator`

We suggest you create fixtures, as some inputs need to be reused in several of these tests.

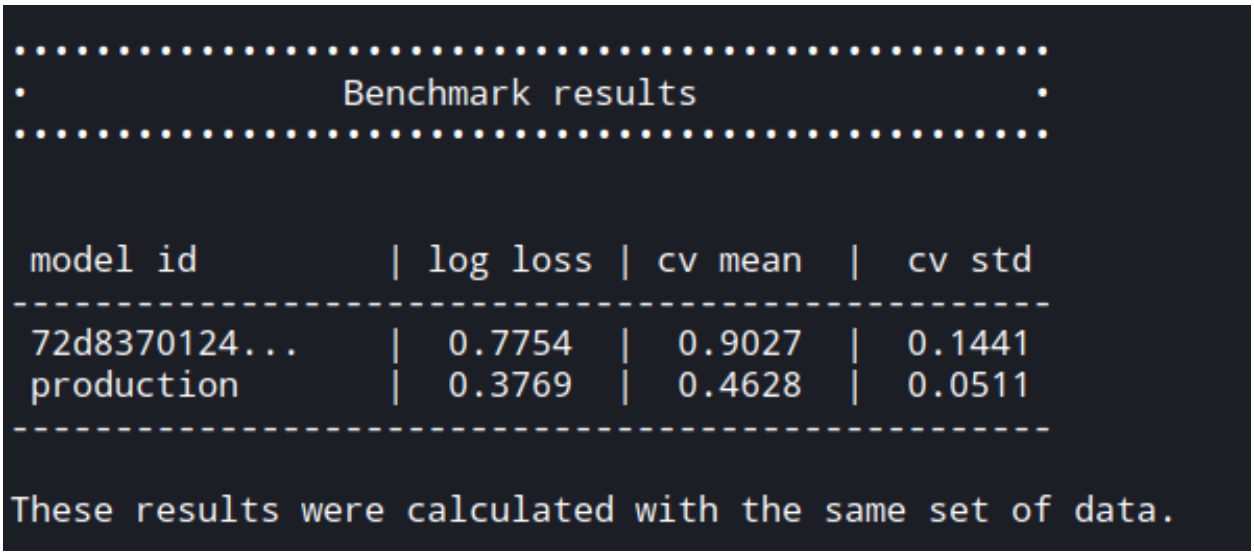
04 - [BONUS] An introduction to model performance testing

`Suggested time: 45 min`

Performance testing is a kind of acceptance test. There are multiple places in the systems where it would make sense to measure the performance of the model, for today we will focus on measuring its performance after retraining.

To make things as simple as possible for the training, we decided that the `log_loss` was the metric on which we would assess the performance of a model. The test consists of comparing the log loss on a train/test split of a newly trained model, with the one currently in production.

Our 'test' will not make any decision, but will simply create a reporting that we can later use to decide if we want to push this new model into production. Here is an example of the output we get:



NOTE

This assignment is a bit different from the rest: you need to have a good understanding of the app's inner mechanics to come up with something useful.

As the goal of this workshop is not to spend hours reading the app's source code — even though it's fairly simple — we'll give you more hints as for how you could achieve the expected result.

Some things to know before trying to figure that out:

- Even though we store this test in our `tests/` folder, it would totally make sense for a next release of the application to make it a full subsystem, at the same level than `serve`, `train` and `predict` (which means that it would need to be tested...) An acceptance test like this one, if carefully crafted, could be an important part of the global application.
- This kind of acceptance test is very difficult to automate: it depends a lot on context. We won't try to make this process fully automated here, as the problems raised by such a situation go beyond the scope of this training.
- Since it will be a manual process, we will simply prompt the testing operator to enter the unique ID of the model he/she wants to compare with the production model.

We suggest you write the implementation in a `tests/performance/test_metrics.py` file.

4.1 Fetch data about the production state

On the first day, you ran a quick training, and then deployed the resulting model into an S3 bucket. This deployment phase was done to give your prediction system access to a fitted model upon which it could run predictions.

To benchmark a new model, you first need to fetch some info about what is currently "in production".

In total, you have uploaded **3 training artifacts** on S3. Indeed, alongside a serialized version of the model, the deployment phase also stored:

1. The dataset used for fitting
2. a JSON report with the following info:
    - Model's unique ID (str)
    - Log loss value of predictions run on test set
    - Mean of cross-validation log losses
    - Standard deviation of cross-validation log losses

If you feel like you need some of these artifacts to do a valid benchmark, use the S3 wrapper functions stored in `src.aws` to download the one(s) you need.

Tip

You can use the caching system implemented in `src.utils` to avoid re-downloading every time you want to run the test.

4.2 Get the model object

The model you want to compare with its production sibling has to be stored somewhere locally. You need to run a `train` process, which, by default, will store training artifacts – fitted model included – in the `output` directory. You can get the unique ID of this new model in the `report.json` file.

Your script should ask the testing operator to enter this unique ID. From there, it can find the corresponding pickled file, and deserialize the data to create a model ready to run predictions.

The `src.utils` module will again help you here. Have a look at the functions you could re-use.

4.3 Putting it all together

On one side, you have access to:

- The "production" model
- The data used to train it
- The training report

On the other side, you have your new model that you would like to benchmark.

The testing operator needs to take quick decisions about the validity of this new model, and how it performs compared to what is running in production today.

Implement a performance comparison test that will print out a comparison report about log losses on test and cross-validation data. Check out the screen-shot from above if you need some inspiration.

That's OK if this test is not run by `pytest`. As previously stated, this is only a very first step to performance testing, and we don't want to automate it yet: we need a human to draw conclusions from the comparison table.