Day 3 - Guidelines

Mocks or patching should not be necessary for exercise 1.

Starting from exercise 2, we strongly suggest that you keep the unittest.mock documentation open 😉

0. Setup

Create a `tests/units` folder: this is where we are going to store our creations today.

1. Unit testing the features

`Suggested time: 20-30 min`

Feature engineering is a big part of any ML project with structured data. Because of this, this is certainly not something you should ignore when testing.

> Sometimes, like today, testing features may seem so trivial that you could feel you can go without it. But remember: testing is not about today's code: it is about ensuring future code's quality and correctness.

Go ahead and take a look at the `src/train/features/` package. To facilitate your understanding of this package, here are some implementation details:

- `__init__.py` imports all the features created in the `features.py` module.

  Then, we create a list of these features and from it we derive the `FEATURE_STORE`, which is exactly in the form expected by the `FeatureUnion` that it is fed too. This `FeatureUnion` is used by the modeling pipeline.

- `base.py` module consists of a couple of base classes that we will use to build our features.

- `features.py` the most interesting part, where all features are created.

  We can divide those features into 2 groups: those that inherit from the `ColumnExtractorMixin` and those that don't. All features inherit from the `BaseFeature` class which itself inherits from Scikit-Learn's `TransformerMixin` and `BaseEstimators` classes.

- The role of the features inheriting from `ColumnExtractorMixin` is simply to extract a given column from the dataset. Think of it as having selected the most informative raw columns from the dataset.

- Other features have either been created from domain knowledge, data analysis, interaction or custom preprocessing.

> This should leave you with a couple of different tests to implement. You can go with any feature you want, but here are some suggestions:

1. The `FeatureBase` class is very simple, yet very important... Some basic checks can avoid accidentally breaking it.
2. A basic feature such as `Out` can be tested to see if the logic is well implemented.
3. You could check that the `ColumnExtractors` for features that are simply copies of raw data (*eg* `Speed`, `NetClearance`, `PlayerDistanceTravelled`, ...), work as expected.
4. A feature such as `WeirdNetClearance` is designed for a specific value of `net.clearance` that seems like incorrect measurement. Testing it is both simple and useful.
5. Unit test any other feature you want 🙂

> 💡 Tip
>
> Remember that you have access to a data dictionary, available for download. It gives insights about what are the columns, what do they mean, and what do they refer to. It is not strictly needed for the purpose of this training, but can help you to understand the context.

2.Test utils

`Suggested time: 40-45 min`

The `src.utils` module is used extensively in our app. There are several functions to test. We picked two of them that looked interesting for practicing with unit testing techniques.

2.1 Testing the hasher

This function is used in the for experiment tracking in the training system. The `log.py` module contains a single-class which is a custom-made tracking system for training runs. It locally saves a report of the metrics for a given training run, along with serialized artifacts: the model and the data used to train the model.

Within this class, the `hasher` function is used so that **each model can be identified with a unique id** using the SHA1 algorithm, computed using the implementation from the `hashlib` python package.

> ⚠️ **WARNING**
>
> Often when writing tests, the real difficulty resides in framing the right test case.
>
> In our case, what do we want to test? That CPython properly implements the SHA1 algorithm? That's definitely not something you should care about, nor something you have time to invest in. **As a user of Python, you just rightly assume that the SHA1 algorithm *is* properly implemented by Python.**
>
> It all boils down to this question: what do we really want to prevent with our `hasher`?

Define a **pragmatic** test case and implement it in `tests/units/test_utils.py`. Ask yourself:

> What is this function supposed to do? What check(s) can I perform to be sure that this function works as expected?

```
# tests/units/test_utils.py
from src.utils import hasher


def test_hasher(): # you can use patching... or not.
    # TODO: test the core function of the hasher
```

2.2 Testing cache loading

Some background

In our demo project, when we deploy a model "in production", we push the serialized model artifact to an s3 bucket. Then, when the prediction system needs to run inference, it will grab this artifact from s3, de-serialize it and use it to make predictions. However, network calls can be slow, in that regards it's not really efficient to download the artifact from s3 every time we want to use it. The project implements a naïve caching system: it stores the serialized model on disk, and invalidate the cache depending on a expiration time defined as a constant in `src.config`.

The `load_from_cache` function itself relies on a few of other helper functions:

- `cache_has_expired`
- `remove_file`

This function behaves differently depending if a cache file exists or not... *You obviously need to check any expected behavior.*

💡 **Important tips**

We found using a mix of fixtures and some mocking/patching useful for these tests.

Also, note that Pytest has some interesting built-in fixtures and features, in particular:
- `tmp_path` (fixture)
- you can check that an exception was raised during a test.

2.2.1 Testing the `remove_file` function

The `remove_file` function should:

- delete a file from disk if the file exists
- raise a `FileNotFoundError` if the file does not exist
- do nothing if we try to remove a folder

```
# tests/units/test_utils.py
from src.utils import remove_file

...

class TestRemoveFile:
    # using a Test class is not strictly necessary here,
    # but it can help with code readability: we know that
    # all the test methods are bound to the same business logic.

    def test_remove_existing_file(???):
        # TODO: check that `remove_file` does its job when a file exists

    def test_remove_non_existing_file(???):
         # TODO: check that an error is raised when trying to remove
         # a file that does not exist.

    def test_remove_directory(???):
```

```
        # TODO: check that `remove_file` does NOTHING if we try to delete
        # a directory.
```

2.2.2 Testing the `cache_has_expired` function

The `cache_has_expired` function should:

- **not raise** if the file path given as argument does not exists, and simply return `False`
- return `False` if the file was created recently (less than `CACHE_MAX_AGE` ago)
- return `True` if the file creation date is older than what `CACHE_MAX_AGE` authorizes

```
# tests/units/test_utils.py
from src.utils import CACHE_MAX_AGE, cache_has_expired

...

class TestCacheHasExpired:

    def test_non_existing(???):
        # TODO: check that it does not raise, and simply returns False

    def test_existing_not_expired(???):
         # TODO: check it returns True for recently created file

    @patch(...) # TODO: what do we need to patch?
    def test_existing_expired(???):
        # TODO: check it returns True if file is older than CACHE_MAX_AGE
```

2.2.3 Testing the `load_from_cache` function

Now that we're sure that the 2 sub-functions work, we can test `load_from_cache` as a whole.

This function should:

- raise a `FileNotFoundError` if the path given as argument points to a non-existing location, **or** if cache has expired
- return the file's content otherwise.

```
# tests/units/test_utils.py
from src.utils import load_from_cache

...

class TestLoadFromCache:

    def test_non_existing(???):
        # TODO: check that it raises a FileNotFoundError

    @patch(...) # TODO: what do we need to patch?
    def test_existing_not_expired(???):
         # TODO: check it returns the content of the file

    @patch(...) # TODO: what do we need to patch?
    def test_existing_expired(???):
        # TODO: check it raises a FileNotFoundError
```

Don't forget to run `pytest` from a terminal to check the correctness of your tests.

3. [BONUS] Unit testing the predict system

`Suggested time: 30-40 min`

This exercise will make you practice with unit testing, with advanced usage of mocking and patching.

Have a look at the source code of the `src.predict` package. It's only 2 files, pretty easy to understand. You can experiment with the functions by calling them in a Python shell if it helps you to better understand their behavior.

The 3 main functions are:

- `get_model` to fetch a serialized model from S3, or from a local cache file
- `parse` used to convert the feed from the web app to a pandas DataFrame
- `predict` the core function of this `predict` package

Those are the 3 functions we want to test.

> Take some time to criticize the implementation of the `predict` system. There is room for improvement, and this improvements that may make the tests writing process easier.

```
# tests/units/test_prediction_system.py

def test_parser():
    # TODO: check that `parse` works as expected
    #    Parametrization can be useful...

# -------------

# Use patching to ensure that we test ONLY the logic
# implemented in `get_model`.
# You probably need some parametrization as well...
@patch('src.predict.main.load_model_from_s3')
@patch('src.predict.main.load_model_from_cache')
def test_get_model(...):
    # TODO: implement a test for `get_model`


# -------------

# we don't want to test `get_model` or `parse` here: we just did!
@patch('src.predict.main.get_model')
@patch('src.predict.main.parse')
def test_predict(...):
    # TODO: implement the test.
```

```
    # NOTE: this test is kind of hard to implement, which suggests
    # that the `predict` implementation could be improved and simplified.
```

---

4. [Bonus²] Test AWS-related logic

`Suggested time: 30-40 min`

> This test can be seen as in-between integration and unit testing... Another example that sometimes, labeling a test is not easy. But we don't care: what we want is to ensure that our `src.aws` module is reliable!

Let's do some stubbing! Now that you are used to `patch` and mocks, just check how you can `stub` a function/method call to make your test simpler to understand, and easier to write.

All components of our app relies on the `src.aws` module to read/write from/to S3.

However, we did not build our S3 interface ourselves: we simply used the `boto3` package, officially maintained by Amazon, giving us guarantees that, as long as we call the methods of S3 client documented in `boto3`, we *will* reach our S3 infrastructure.

Depending on the approach you take when implementing unit tests, it can make total sense to rely on *stubbing.* But what's a stub?

> A Stub is a temporary replacement of a module (or function), that doesn't implement the whole logic of the module (function) it replaces, but simply simulate data communication between your test and the module (function).

In other words, if we trust `boto3` (which we do), simply stubbing it correctly should be enough to give us the guarantee that the communication between our wrapper functions and the actual AWS infrastructure is handled correctly.

This is such a basic need for a library like `boto3` that a stubbing system is included by default in the `botocore` package (which is a core dependency of `boto3`).

Tips

There are several ways to upload/download data from/to S3 using `boto3`.

The app uses the most basic ones, from the `boto3.client('s3')`:
- `put_object()` to upload (docs)
- `get_object()` to download (docs)

You need to look at the source code of `src.aws` to check the parameters we are using for both methods.

1. Create the file `tests/units/test_aws.py`
2. Read the docs of the stubber object provided by `boto`. With that small docs and the information above, you have everything you need to implement a stubbing.
3. Write the tests for both functions

4. Run them using `pytest`
5. Sit back, and enjoy the green flags popping up everywhere on your screen

---