Day 1 - Guidelines

> Welcome to the Advanced Testing training program, day 1!

Today's main goal is to make sure that the core components of our systems are working properly. This is what end-to-end tests are for.

As these tests are *end-to-end*, we will try to write test that are as close as possible to the end-user's usage, *ie* **we will use the same user interfaces.** The training system and prediction system are Command Line Interfaces (CLI) programs, whereas the serving system is reachable through a REST API.

💡 General tips

- Discuss with your team mates, and share your idea
- Every exercise has a suggested duration. Use them as an indicator as to when you should move on to the next one.
- Your trainer is available to discuss on any problem you may encounter: do not hesitate to ask for help, advice or insights!

⚠️ **THIS IS NOT A RACE, NOR A CONTEST!**

All of the exercises have been designed to make you practice with software testing. Depending on your background, your experience, etc. you will not necessarily have time to finish all of them, **and it's totally OK.**

Do not hesitate to ask your trainer if you have questions along the way 🙂

---

0. Setup

`Suggested time: 15 min`

Since it is our very first day, we need to do a little setup before getting started. This setup should complete in 10/15min.

You may already have done some of these steps ahead of the session. If that's the case, just ensure that everything is done as expected.

0.1 Clone the repo and fetch the data

You first need to clone the project's git repository on your machine.

```
$ # From a terminal
$ git clone https://github.com/nibbleai/demo-machine-learning
```

0.2 Download the dataset

The CSV file used as our base dataset can be downloaded from this link.

Store it in the `data/` directory, at the root of the project (if the directory does not exist, create it). Do not change the file name: `australian_open.csv` is what the app expects.

A data dictionary is also available for download. It gives insights about what columns refer to. It is not strictly needed for the purpose of this training.

0.3 Setup your environment

> ⚠️ **WARNING**
>
> From now on, we highly recommend using a virtual Python environment, weather via the standard `venv` or via Conda.

From a terminal, `cd` into the project's root directory to install the required dependencies, as well as our demo project.

```
# From a terminal, in the project's root dir

# Install the dependencies
$ pip install -r requirements.txt

# Install the project in editable mode. Note the "." at the end
$ pip install -e .
```

We need to set some environment variables, the AWS credentials and your username. Those are not hard-coded into the project because you obviously don't want to record AWS credentials in a git repository... Setting up a username allows us to create a dedicated space in our S3 bucket, so you all have your personal storage space.

Create an `.env` file in the at the project's root, with the following environment variables:

```
# .env
AWS_ACCESS_KEY_ID= (cf. Slack channel)
AWS_SECRET_ACCESS_KEY= (cf. Slack channel)
AWS_S3_BUCKET_NAME=australian-open-outcome-predictions
# replace the following with your own username:
USERNAME=georgeabitbol
```

You can now check it's working correctly, by running the following in a terminal:

```
$ python -m src.main -h
```

If it's working properly you should see the help message in your terminal.

0.4 Run a training and deploy the model

You now need to generate a basic model, and make it accessible to the prediction system (performance does not matter here, we just want a fitted model that can run predictions).

```
$ python -m src.main --train
```

When it is done, you should have a new training log in the `output/` directory, stored in another directory named after the timestamp of the execution, *eg* `20210528-11h05m12s`.

In this directory, open the file `report.json` and copy the unique model identifier, located under the `estimator["id"]` key.

NOTE

In this demo project, *deploying* a model simply means uploading the pickled artifact in an S3 bucket. Of course, on a real project, there would be much more things to do, but we don't really care here.

Now that you have a valid model ID copied, run:

```
$ python -m src.main --deploy
```

and paste the model ID you just copied when prompted.

After uploading the fitted model in an S3 bucket, the command should reply with a `Model successfully deployed` message.

The prediction system has now access to a fitted model, upon which it can run predictions.

0.5 Create the `tests` directory

Create an empty `tests/` directory at the root of the project.

Inside this newly created directory, create an empty `end_to_end/` directory. This `end_to_end/` directory is where we will store all the tests files we will create today.

> You are now ready to start working on tests implementation!

For the following exercises, note that all these systems are controlled via the CLI accessible through the `src/main.py` file. This is our main entrypoint, and flags are used to command any specific task such as generating features, training the model, etc.

Remember that to access the documentation of the CLI, you can always execute this command in a terminal:

```
$ python -m src.main --help
```

You can also open `src/main.py` in your text editor to briefly check the inner workings of the system

> 💡 **Reminder**
>
> Since today's focus is about *end-to-end testing,* there is **no need to check the source code**. The user interface documentation, ie the CLI, should be enough.
>
> Also, for end-to-end tests, tooling doesn't really matter. We suggest you use Python or Bash for all the following assignments.

---

1. Testing the feature generation system

`Suggested time: 15 min`

We will use the CLI to start a python process that will test the feature generation step. Check the documentation of the CLI to see how to generate features.

This one is fairly trivial: the feature generation will be considered working if the process can terminate without any error. If something wrong happens, like input data in the wrong format, the process will exit with an error, and the feature generation process will exit with an error.

Create a file named `test-features.sh` (`tests_features.py` if you're using Python, **but Bash is recommended**), implement the test and run it by executing your script.

```
# tests/end_to_end/test-features.sh
# OR
# tests/end_to_end/test_features.py
```

```
YOUR TEST GOES HERE
```

⚠️ **WARNING for Windows Users** - Python VS Bash

The user interface is a CLI. When using Python, you need to launch the command you want to test in a subprocess. The canonical way of launching a subprocess with Python is by using the standard `subprocess` package.

However, due to the way Windows resolve paths, you will end up with a subprocess using a different Python interpreter (cf. Python bugs tracker) than the one from your virtual env, which is problematic. As a workaround, you could use `os.system` to execute shell commands.

Conclusion : stay as close as possible of the final user behavior, **and use Bash** 🙂

2. Testing the training system

`Suggested time: 30 min`

This test is quite similar to the previous one, but the training process comes in 2 flavors:

- A simple training that uses cross-validation,
- A training with hyper-parameter optimization performed on the same fold as the one used for the cross-validation of the simple training

You can run both commands in a terminal, to get a better idea of the usage.

→ **Create a file** named `test-train.sh` (recommended, but you can also create `test_train.py`) and **implement a test for each version of the training**.

Then run the tests by executing your script, eg:

```
# From a terminal:
$ bash tests/end_to_end/test-train.sh

# Ensure that there was no error, either:
# - by reading the logs (but this is not very robust...)
# - by checking (manually for now) that the exit code of the
#   previous command is 0
```

3. Testing the prediction system

`Suggested time: 30-40 min`

Just like the feature generation and training systems, the predict system uses the CLI as its main interface.

This system is a bit different: running a *prediction* only makes sense if you've got some data to run the inference on! And this data better be similar to the one the system has been trained on...

Take a look at the CLI documentation, you'll see the CLI for the prediction system requires a non-optional argument. **So, prior to running the tests for the prediction system, you'll have to make sure you can feed it some sample data.** As usual with end-to-end testing, some setup is involved.

3.1 A script for sample data

Write a script (Python, bash, whatever you prefer) to prepare sample data "on demand". We suggest creating a Python script, directly within `tests/` **but it's only a suggestion**.

```
# tests/generate_test_data.py

# TODO: write everything you need to build sample data
```

We suggest that this script stores sample data into a file called `data/sample.json`. Then for 3.2, this sample data can be used to feed the prediction system during the tests.

The data format must be something similar to this (based on the data stored in the CSV):

```
[
  {
    "rally": 4,
    "serve": 1,
    "hitpoint": "F",
    "speed": 36.09022579,
    "net.clearance": 1.258680346,
    "distance.from.sideline": 0.244500352,
    ...
  },
  {
    "rally": 8,
    "serve": 1,
    "hitpoint": "B",
    "speed": 18.5794439,
    "net.clearance": 0.998680346,
    "distance.from.sideline": 1.244500352,
    ...
  },
  ...
]
```

The above is the format expected by the `predict` system.

3.2 Test prediction system

Create a file named `test-predict.sh` (**recommended**, although you could also write a `test_predict.py`), and implement an end-to-end test for the prediction system.

Your test should use the sample data created by your script from step 3.1.

Does the output match your expectation? What was the most challenging part?

4. Testing the serving system - BONUS

`Suggested time: 45 min`

> 💡 **Tips**
>
> Before running a test on the serving system, **don't forget to start the web server** (check the CLI documentation).
>
> When a server is listening locally, you can make a web request using the `requests` package (python) or the `curl` command (bash).

This time, the system's main interface is a REST API running on a web server that you can reach through HTTP calls. The API serves 2 endpoints:

1. `/ping`
2. `/predict`

The first one's sole usage is debugging. We will use it as a warmup for the real test, *ie* the prediction endpoint, since **both endpoints require the same setup to be tested.**

> ⚠️ **Warning**
>
> This one is trickier than the previous ones... To properly test the serving system, you need
> to run two different processes in parallel:
>
> - the web server listening to HTTP requests
> - the tests *per se*

4.1 Testing the `/ping` route

This endpoint doesn't do much, it's a simple GET route that just sends back a `'pong!'` response. It's only useful to perform a *health check* of our system.

We want to make sure the web server is able to process the request and responding properly.

This is how the server should respond to a `GET /ping`:

- The HTTP status code should be `200`
- The text response should be `'pong!'`

4.2 Testing the `/predict` route

This is a POST endpoint, so this time you'll make a POST request. This route also takes a non-optional body, as a JSON object string.

The expected body (or payload) is a JSON object string with a root key `data`. This key holds either a single data-point as a JSON object, or an array of data points. A data point is a JSON representation of a single data point in the raw data.

Example of a single data point JSON object:

```
{
  "rally": 3,
  "serve": 1,
  "hitpoint": "F",
  "speed": 43.29604686,
  "net.clearance": 0.163010453,
  "distance.from.sideline": 0.14532310199999998,
  "depth": 5.956976846,
  "outside.sideline": true,
  ...
}
```

When a valid data points, or a valid collection of data points is sent, the server should respond with:

- HTTP status code `200`
- Content being a JSON object containing as many predictions as there are data points. Responses should be `W` for Winners, `UE` for Unforced Errors, or `FE` for Forced Errors.

Create a file named `test_serve.py` (`test-serve.sh`), implement the tests and run them by executing your script.

Running this test properly might involve some subtleties to solve tricky issues that can arise... For example, can you run the tests twice? How much manual work is required to do so?

4.3 Test ill-formatted request

Sometimes, checking that your system returns a specific kind of error under a given circumstance is a guarantee that it behaves as expected.

In the case of our serving system, the web server should respond with the status code `400` when the input data is not properly formatted. `400` is the standard HTTP status code for an error caused by the client (*ie*, the requester).

Add this test in your `test_serve.py` or `test-serve.sh`.

Wrapping up

Congratulations 🎉

> We went from 0 testing to covering every important use cases of the system.

And the good thing is we've done the work once, but can reuse those scripts anytime we want in order to make sure our system didn't break after some changes, this is the basic idea behind automated testing.

On this note, we still have to run each test script one-by-one, **how much better would that be if we could just run them all at once? How would do this?**

Save your thoughts for the interactive session after the break!


That's all Folks!