Lab: Model packaging and serving (2/2)
Professor : Fahd KALLOUBI
Year : 2023/2024
Major : QFM 2

In this lab, we will design a package to move from a research environment to a production environment. Furthermore, since the code is intended for end users unlike the code in a research environment which is intended to carry out local experiments or proof-of-concepts, there are other requirements to take into consideration:

- Testability and maintainability are huge: we need to divide our code in terms of modules so that they are extensible and easy to test
- We also need to separate all configuration from code and we also need to test and document each feature
- We also need to ensure that our code adheres to standards like PEP-8 so that it is easy for others to read.
  - https://www.python.org/dev/peps/pep-0008/
- Scalability and performance are also important: the code must be deployed in infrastructures that can ensure scaling
- Reproduction: the code must be monitored using version control and also other dependency files.

Our package will then be published on PyPI “Python Package Index” (it will be installable via the pip command).

We will base ourselves on the following official tutorial:

- https://packaging.python.org/tutorials/packaging-projects/

1. Research environment

Download the two datasets (test & train): https://www.kaggle.com/c/house-prices-advanced-regression-techniques .

In the lab folder, open the “Research notebooks” folder and then run the notebooks in order to produce our final pipeline.

2. Overview of our package

In this part, we will explore how we were able to move from a research environment (i.e., Notebooks) to a production package that we will consume via an application (later in this lab).

Our package will look like this:

```
parent/
├── regression_model/
│   └── config/
│   └── datasets/
│   └── processing/
│   └── trained_models/
│   ├── config.yml
│   ├── pipeline.py
│   ├── predict.py
│   ├── train_pipeline.py
│   └── VERSION
├── requirements/
│   ├── requirements.txt
│   └── test_requirements.txt
├── setup.py
├── MANIFEST.in
├── pyproject.toml
├── tox.ini
└── tests/
```

We are going to take a walk to understand each part of this tree.

a. the « requirements » folder

You will find two files “requirements.txt” and “test_requirements.txt”. In the “requirements.txt” folder, we specify the acceptable dependencies for our package.

Start by installing these requirements: pip install -r requirements/requirements.txt

We also have the “test_requirements.txt” file, in which we install all the dependencies of the “requirements.txt” file by adding the test and maintenance dependencies of our package:

- Black : for code formatting (https://pypi.org/project/black/)
- Flak8 : to ensure that we adhere to the python conventions (https://flake8.pycqa.org/en/latest/)
- Mypy : type checking tool (http://mypy-lang.org/)
- Isort: to ensure our imports are in the correct order

Go to the “train_pipeline.py” module and add an unnecessary import (for example: import math). Then run the command “tox –e lint”, what do you notice?

b. Management and testing using tox

- Official documentation : https://tox.wiki/en/latest/index.html

Tox (a virtual environment) is a tool for managing and testing a python package which is used to facilitate software development in python.

In the provided folder (our package), open the “trained_models” folder (production-model-package\regression_model\trained_models). You will notice that it only contains one empty python file (__init__.py). Then open your command line on the root and type the command: tox –e train

On a Windows machine, if you get an invocation error visit this link: https://docs.python.org/3/using/windows.html#removing-the-max-path-limitation

Open the “trained_models” folder again, you will notice that the “.pkl” file of our model has been generated.

All tox configurations are in the “tox.ini” file, open it and take a look at its contents.

- Everything in square brackets ([tox], …) are tox environments which will be created as virtual environments in the “.tox” folder
- we can execute commands in a specific environment and we can also inherit commands and dependencies from another environment.

a. The environment [tox] :

This base environment contains a list of environments (i.e., envlist). This says that the content of these environments must run when calling the "tox" command on its own. Run the “tox” command and see the result in your console.

The “skipsdist=true” variable means that when executing our command, we do not want to generate the source distribution (.gzip files) because this will be done manually later.

b. The environment [testenv] :

It is a helper environment for testing installation packages which will be inherited by several environments subsequently (each environment must test the installation before its execution).

c. The environment [testenv :test_package] :

It inherits from “testenv” in which we first start by passing the dependencies to the command inherited from “setenv” (i.e., pip install …) and then initialize the environment variables (PYTHONPATH for the virtual environment from which comes the creation of the “.tox” folder and PYTHONHASHSEED for the reproduction of the same results). We then specify the commands to execute:

- The first to execute the pipeline
- And the second to run tests by specifying the test folder and details arguments.

d. The environment [testenv :train] :

- Its “envdir” folder is the same as the “test_package” package
- It has the same dependencies (executed by setenv) and the same environment variables as “test_package”

Run to the “tox –e train” command again to see the sequence of executions.

3. Package configuration

Open the configuration file “config.yml” located in the “regression_model” folder. Among the reasons for using a yaml file instead of a python file for configuration is that a yaml file can be read and modified by a non-python developer (see: https://hitchdev.com/strictyaml/why-not/turing-complete-code/).

Make a description of the contents of the Yaml file. What types of information does this file contain?

Then open the “core.py” file in the “config” folder, this file starts by detecting the paths (using pathlib) to “config.yml”, “datasets” and “trained_models”. Subsequently, we create three classes (which inherit from the BaseModel class from Pydantic) to store the information contained in the “config.yml” file.

For example, change the type of the target attribute (from str to int) in the ModelConfig class and then execute the “tox –e train” command. What do you notice? explain how we could have received this comment?

4. Model training and pipeline execution

Now open the “train_pipeline.py” file that we executed by running the “tox –e train” command.

Note that all input data and all metadata (i.e., features) come from our “config” object returned by the “create_and_validate_config()” function found in the “core.py” module.

Looking at the imports, you will notice that we use several other modules. However, we will start with the “pipeline.py” module which is like the pipeline we used in our last notebook except that in this pipeline we used our “config” object for calling our variables.

Now open the “data_manager” module in the “processing” folder, you will notice that this module is responsible for maintaining the input data (i.e., dataset) and the output data (i.e., the pipeline .pkl file).

5. Features engineering in our package

Looking at the “pipeline.py” module, you will notice that we used several basic transformations apart from two transformations developed in the “features.py” module.

Open the “features.py” module in the “processing” folder. we defined two transformers “TemporalVariableTransformer” and “Mapper” which inherit from “BaseEstimator” and “TransformerMixin”.

To test these transformers, we created a module for this (test_features.py in the test folder). In addition, we created a fixture (i.e., conftest.py) named “sample_input_data” to test on the test dataset. To test, we need to call the “test_package” environment to run the test command. To do this, and while on the root, run the command “tox –e test_package”.

Change the value "49" and rerun the command to see an example of this test failing.

6. Predict using our package

Open the “predict.py” file and look at its contents.

The “make_prediction” function takes “input_data” as a parameter which will subsequently be transformed into a dataframe. This dataframe must be validated using the “validate_inputs” function of the “validation.py” module which will do the basic transformations so that the validation data is acceptable.

Now open the test file “test_prediction.py”. Note that we are testing with the same fixture by adding several tests.

Run the command “tox –e test_package”. Change the expected values (for example the number of predictions or the prediction value for the first individual) and rerun the command.

7. Package construction

We will now build and publish our package so that other applications can install it and use its modules. The files responsible for packaging our package are “pyproject.toml”, “setup.py” and “MANIFEST.in”.

Now open the file “pyproject.toml”, the key lines are from 1 to 6 in which we specify the basic dependencies. The rest of the file is dedicated to configuring other tools like pytest, black, isort, etc.

Now open the “setup.py” file, this file contains all the metadata of our package (name, url, author) as well as all the information necessary for the installation of our package (the version of python, requirements, etc.). Finally, this information is passed to the setup object.

Now open the “MANIFEST.ini” file in which we specify which files to include in our package and which files to exclude from our package.

Open this link https://packaging.python.org/tutorials/packaging-projects/ and then go directly to the “Generating distribution archives” section.

- ➢ Run this command: py -m pip install --upgrade build
- ➢ Then, this command: py -m build

Following this command, you will notice that other folders have been created.

Congratulations, you have built your package.

Now to publish it to Python Package Index, try following the steps described in the “Uploading the distribution archives” section.

<u>To Do :</u>

1. Using the published package, build an API capable of inferring on new data using FastAPI
2. Do the same thing again by operating on another dataset