boar

Release 0.0.1

Alexandre Cameron

May 09, 2020

CONTENTS

Dirty tricks to run python notebooks

CHAPTER

ONE

PURPOSE

Testing an idea is sometimes more easily done by calling a python notebook in a python code. It is as dirty, filthy and ugly as a wild hairy swine with horns... But if you can prove the value of your idea, why not try it out immediatly with `boar`.

Read more at ./POST.md

CHAPTER

TWO

INSTALLATION

Requirements: Python 3 pip

The package is available on pypi: https://pypi.org/project/boar/

To install, execute:

```
pip install boar
```

CHAPTER

THREE

PRACTICE

I used this method in CI tasks to make sure tutorials associated to a project stay up-to-date.

I would not recommend relying this trick on a long term product.

CHAPTER

FOUR

USAGE

More information at ./USAGE.md.

CHAPTER

FIVE

REFERENCE

5.1 Documentation

https://boar.readthedocs.io/en/latest/

5.2 Source

https://github.com/alexandreCameron/boar

CHAPTER

SIX

TABLE OF CONTENT

6.1 Readme

Dirty tricks to run python notebooks

6.1.1 Purpose

Testing an idea is sometimes more easily done by calling a python notebook in a python code. It is as dirty, filthy and ugly as a wild hairy swine with horns... But if you can prove the value of your idea, why not try it out immediatly with `boar`.

Read more at ./POST.md

6.1.2 Installation

Requirements: Python 3 pip

The package is available on pypi: https://pypi.org/project/boar/

To install, execute:

```
pip install boar
```

6.1.3 Practice

I used this method in CI tasks to make sure tutorials associated to a project stay up-to-date.

I would not recommend relying this trick on a long term product.

6.1.4 Usage

More information at ./USAGE.md.

6.1.5 Reference

Documentation

https://boar.readthedocs.io/en/latest/

Source

https://github.com/alexandreCameron/boar

6.2 Usage

6.2.1 Testing

To test your notebook follow:

```
def test_assert_notebook_runs_without_error():
    # Given
    from boar.testing import assert_notebook

    # When / Then
    assert_notebook("my_favorite.ipynb", verbose=True)
```

Other examples are presented at: ./tests/test_testing_e2e.py

6.2.2 Linting

If plots are drawn in notebooks, it is not recommended to commit them to the repo. Therefore, `boar` considers notebook-linting as making sure no data is saved in the notebooks. The function is designed to raise an error when outputs have not been clear. The error will indicate the notebook and the cells at fault.

To lint a notebook (or recursively on all notebooks in a directory), use:

```
from boar.linting import lint_notebook
lint_notebook("my_favorite.ipynb", inline=False, verbose=True)
```

```
from boar.linting import lint_notebook
lint_notebook("my_notebook_directory", inline=False, verbose=True)
```

If the `inline` option is set to `True`, a linted verion of the the notebook will be saved. This version will have `outputs = []` and `execution_count = NOne`.

Other examples are presented at: ./notebook/02-lint-tutorial.ipynb

---

6.2.3 Running

Synthax

To run a notebook use:

```
from boar.running import run_notebook

outputs = run_notebook("my_favorite.ipynb", inputs={"a": 1}, verbose=True)
```

Export

The outputs are defined in the notebook by adding

- `# export_line` for a line
- `# export_start` and `# export_end` for a block.

Examples are presented at: ./notebook/01-test-tutorial.ipynb

Skip

Section of the notebook can be skip using the keywords:

- `# skip_line` for a line
- `# skip_start` and `# skip_end` for a block.

Inputs

Inputs variables can be execute before the notebook using the `inputs` parameter of the `run_notebook` function.

Combined with the skip option, inputs allow run a notebook for differents parameters.

Example

If a variable in a notebook is defined as:

```
data_file = "my_data_file.csv"  # skip_line
```

and an input dictionary defined as:

```
inputs = {"data_file": "data_file_1.csv"}
```

The skip option will prevent the code from executing `data_file = "my_data_file.csv"`.

The input parameter will set `data_file = "data_file_1.csv"`.

This is not the best way to put code in production but it can help out in some occasions.

6.2.4 Caveats

Limits

- Only the graphic package `matplotlib.pyplot` as been tested. Other graphic package may not give back the hand.
- Only python code can be executed. The package will **not** work on julia or R notebooks.
- When executing a notebook via `boar` make sure the environment has all the package to run the notebook.
- The package has not been developped to work recursively. Do not execute boar on notebooks that execute boar on other notebooks!

Forbidden synthax

Some synthax used in notebook can **not** be used with `boar`:

- Magic command starting with `%%`, `!` or any command that cannot be used in a python file.

Use `import pip ; pip.main(["install", "my-package"])`, to install package within the notebook instead of `! pip install my-package`

- Variable, list, dictionary comprehension.

```
# This synthax will **fail**
b = [a for a in range(10) if a > 3]
```

```
# This synthax will **pass**
b = []
for a in range(10):
    if a > 3:
        b.append(a)
```

- Function calls for functions defined within the notebook scope.

```
# This synthax will **fail**
def f2(a):
    return a**2

def f2plus1(a):
    return f2(a) +1
```

```
# This synthax will **pass**
def f2plus1(a):
    def f2(a):
        return a**2
    return f2(a) +1
```

- Package imports in the notebook scope.

```
# This synthax will **fail**
import numpy as np

def f2plus1(a):
    return np.square(a) +1
```

```
# This synthax will **pass**
def f2plus1(a):
    import numpy as np
    return np.square(a) +1
```

- Use of `export` or `skip` tags in a indented section.

6.3 Start testing your python notebooks

6.3.1 TL;DR : boar helps test notebook in CI pipelines

https://github.com/alexandreCameron/boar

https://pypi.org/project/boar/

It provides functions to:

- test: check that a notebook can be executed until its last cell
- lint: check that a notebook does not have output secretly stored in a cell
- run: execute a notebook and collect the output

6.3.2 Uncertainty of ML projects

I had been working on the PoC for the end of the past sprint. I was still not sure I would get any results for this model. I had tried different feature constructions, data splits and classifiers. Two weeks had gone by since the sprint planning. After half a dozen stand-up meetings spent on the same task, I would have to tell Scott my boss that I was not sure the problem could have a simple solution. Even though Scott understands agile methodology, he would not be amused. Wait! What about this unconventional idea I had a week ago? I still had one day to throw a Hail Mary pass. A bit discouraged by my previous failed attempts, but sill eager to try, I opened my notebook and started frenetically pouring lines of code in the cells.

[A Hail Mary pass is a desperate last attempt long forward pass to win a US football game wikipedia]

6.3.3 Deciphering notebooks

What happens next to the model is of little importance. What is more interesting is how to handle the notebook where "lines of codes were written frenetically". What are the options?

- Was the idea of any interest? Should it just be deleted?
- Could it be useful? Should it be "archived"?
- Is it the start of something big? Should it be converted in a tested production-grade code?

[Disclaimer: I'm not advocating to keep dead code. Dead code should be eradicated without any remorse otherwise zombie bugs might come to viciously haunt the project.]

There are probably other options. But, in practice, Scott will have to deliver his promise to the customer and he will push the task until the job is done. If a task is listed on the sprint board, the cold hard truth is that the team will have to solve the issue. In short, I will have to decipher the notebook and integrate the useful part to a production-grade software. When this inevitably happens, my future-self will curse a lot against my past-self. My future-self will regret not having a reproducible code on a different machine, basics tests and tutorials to document what is happening.

6.3.4 TDD on uncharted territory

Some colleagues with a solid software development background may point out that best practice would suggest to use TDD. At the heart of TDD, the feature to develop should be well defined. The specs can still have some questions marks here and there, but the general map is clear. With the map in mind, developers can build the roads to bring users to their objective with a value adding tool. First, the main highways are paved and then smaller roads branch out of the highway grid.

In the case of an ML PoC, the general map does not always exist, it's uncharted territory. Before bringing out the asphalt paving machine, data scientists have to scout the region, identify the best areas to settle and most importantly locate quicksands. They should also try their best to sketch existing trails that could be used in the future.

6.3.5 Crossing the gap

With the library boar now available on github and pypi, I tried to provide tools to help data scientists go from notebook to tested code. I'm sure other developers might also find it useful. The library enables the easy integration of notebook in CI by:

- testing : check that a notebook can be executed until its last cell
- linting : check that a notebook does not have output secretly stored in a cell
- running : execute a notebook and collect the output

At first glance, testing notebooks in CI may seem dirty, filthy and ugly as a wild hairy swine with horns. But dirty, filthy and ugly tests are better than no test.

We may not be able to bridge the gap, but boar can help us leap over it.

6.4 Manual

6.4.1 boar.testing

boar.testing.**assert_error_notebook**(*notebook_path: Union[str, pathlib.Path]*, *expected_error_type: Optional[type]*, *expected_error_msg: Optional[str]*, *error_label: str = 'Assertion'*, *verbose: bool = True*) → None

Assert that notebook raise specific error.

Parameters

- **notebook_path** (*Union[str, Path]*) – Path of notebook
- **expected_error_type** (*Union[type, None]*) – Expected error of the notebook

- **expected_error_msg** (*Union[str, None]*) – Expected error message of the notebook
- **error_label** (*str, optional*) – Name of the error
- **verbose** (*bool, optional*) – Option to print more information, by default False

boar.testing.**assert_file**(*notebook_path: Union[str, pathlib.Path]*, *error_label: str = 'Assertion'*, *verbose: bool = True*) → None

Check that notebook runs without error.

Parameters

- **notebook_path** (*Union[str, Path]*) – Path of notebook
- **error_label** (*str, optional*) – Name of the error
- **verbose** (*bool, optional*) – Option to print more information, by default False

boar.testing.**assert_notebook**(*notebook_path: Union[str, pathlib.Path]*, *error_label: str = 'Assertion'*, *verbose: bool = True*, *recursion_level: int = 0*, *max_recursion: Optional[int] = None*) → None

Check that notebook runs without error.

Applied on a directory, all the notebook will be lint down to the level defined by *max_recursion*.

Parameters

- **notebook_path** (*Union[str, Path]*) – Path of notebook
- **error_label** (*str, optional*) – Name of the error
- **verbose** (*bool, optional*) – Option to print more information, by default False
- **recursion_level** (*int, optional*) – Level of recurssion, by default 0 Set to -1000 if you wish to avoid raising Error
- **max_recursion** (*Union[int, None], optional*) – Depth of directory to explore, by default None

Returns Posix of notebook that failed

Return type List[str]

Raises **BoarError** – At list one notebook as failed, the message will list all failed notebooks

boar.testing.**get_error_notebook**(*notebook_path: Union[str, pathlib.Path]*, *verbose: bool*) → Tuple[Optional[type], Optional[str]]

Get notebook error.

Parameters

- **notebook_path** (*Union[str, Path]*) – Path of notebook
- **verbose** (*bool, optional*) – Option to print more information, by default False

Returns error_type: class of error raised error_msg: error message

Return type Tuple[Union[type, None], Union[str, None]]

6.4.2 boar.linting

boar.linting.**lint_file**(*file_path: Union[str, pathlib.Path]*, *error_label: str = 'Linting'*, *verbose: Any = True*, *inline: bool = False*) → Union[None, str]

Lints one file.

Parameters

- **file_path** (*Union[str, Path]*) – Päth of the notebook, must be file
- **error_label** (*str, optional*) – Name of the error
- **verbose** (*Any, optional*) – Verbosity optional
- **inline** (*bool,optional*) – Replace existing notebook with linted version

Returns Path in posix format if notebook fail else None

Return type Union[None, str]

Raises **BoarError** – Notebook is not a file or not linted.

boar.linting.**lint_notebook**(*notebook_path: Union[str, pathlib.Path]*, *error_label: str = 'Linting'*, *verbose: Any = True*, *inline: bool = False*, *recursion_level: int = 0*, *max_recursion: Optional[int] = None*) → List[str]

Lint notebook.

Applied on a directory, all the notebook will be lint down to the level defined by *max_recursion*.

Parameters

- **notebook_path** (*Union[str, Path]*) – Notebook path or notebook directory
- **error_label** (*str, optional*) – Name of the error
- **verbose** (*Any, optional*) – Verbosity option, by default True
- **inline** (*bool, optional*) – Replace existing notebook with linted version, by default False
- **recursion_level** (*int, optional*) – Level of recurssion, by default 0 Set to -1000 if you wish to avoid raising Error
- **max_recursion** (*Union[int, None], optional*) – Depth of directory to explore, by default None

Returns Posix of notebook that failed

Return type List[str]

Raises **BoarError** – At list one notebook as failed, the message will list all failed notebooks

6.4.3 boar.running

boar.running.**run_notebook**(*notebook_path: Union[str, pathlib.Path]*, *inputs: dict = {}*, *verbose: Union[bool, object] = True*, *Tag: enum.EnumMeta = <enum 'Tag'>*) → dict

Run notebook one cell and one line at a time.

Parameters

- **notebook_path** (*Union[str, Path]*) – Path of notebook
- **inputs** (*dict, optional*) – Parameter to set before launching the script, by default {}

- **verbose** (*Union[bool, object], optional*) – Option to print more information, by default False
- **Tag** (*EnumMeta, optional*) – Name of the tags, by default Tag

Returns Outputs to return if *export*-tags set in notebook

Return type dict

Raises

- **BoarError** – If *export** and *skip** tags in the same source
- **BoarError** – If **start* and **line* tags in the same source

CHAPTER

SEVEN

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX