Unit tests in python

June 18, 2021

Introduction

Unit tests

Unit tests test that functions in the source code behave as expected.

For an example, see here .

Conventional unit testing

Conventional unit testing

In conventional unit testing, you write tests that assert that a *particular* input is transformed into a *particular* answer known to be correct.

We will practice writing tests like this using the `catinabox` module.

Property-based testing

Property-based testing

In property-based testing, you write tests that assert that something should be true for every case, not just the ones you happen to think of.

- **Conventional unit tests**: Assert that a *particular* input is transformed to a *particular* answer known to be correct.
- **Property-based tests**: Assert that some property holds for *all* data matching some specification.

Property-based testing

In property-based testing, you write tests that assert that something should be true for every case, not just the ones you happen to think of.

- **Conventional unit tests**: Assert that a *particular* input is transformed to a *particular* answer known to be correct.
- **Property-based tests**: Assert that some property holds for *all* data matching some specification.

The `hypothesis` package

`hypothesis` is a Python library supporting property-based testing.

It works by generating arbitrary data matching your specification and checking that your guarantee still holds in that case.

It can find edge cases in your code you wouldn't have thought to look for.

```python
import hypothesis.strategies as st
import numpy as np
import pytest
from hypothesis import given

from pypolyagamma import PyPolyaGamma

from fall_2020.logistic_models.polya_gamma_vi.bayes_logreg.inference import (
    compute_polya_gamma_expectation
)

@given(
    b = st.floats(min_value=1, max_value=1),
    c = st.floats(min_value=-100, max_value=100),
)

def test_compute_polya_gamma_expectation(b, c):
    # Test that our computed polya gamma expectation for a PG(b,c) distribution
    # is close to the empirical mean of a bunch of samples (obtained from the pypolyagamma library)
    pg  = PyPolyaGamma()
    empirical_mean = np.mean([pg.pgdraw(b, c) for i in range(10000)])
    computed_mean = compute_polya_gamma_expectation(b,c)
    print(f"For b={b}, c={c}, the Monte Carlo mean was {empirical_mean}, and my function's value was {computed_mean}")
    assert np.isclose(empirical_mean, computed_mean, atol=.01, rtol=.05)
```

Code fails to raise error when user specifies a non-positive real number for the first parameter `b` #49

Edit | New issue

Open **mikewojnowicz** opened this issue on Feb 19 · 1 comment

mikewojnowicz commented on Feb 19 • edited

For example:

```
from pypolyagamma import PyPolyaGamma
pg          = PyPolyaGamma()
b,c = -2, -1
values=[pg.pgdraw(b, c) for i in range(10000)]
assert all([x==0 for x in values])
```

The sampler returns all 0's, although one would expect it to raise an error.

I would have opened a PR myself, but I do not know how to code in C.

slinderman commented on Feb 20 — Owner

Good catch! I think you can add an assert in `pgdrawv` before the C code gets called.

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

None yet

Good practices

Some good practices

- Should be fast (5-10 seconds to run *all* tests).
- Run tests before committing changes to source code.
- Each test function name should have a postfix describing what we're testing:

  ```
  test__function_name__what_property_we_are_testing
  ```

- One assertion per test.