Software Testing for Data Science

January 12, 2017

From the title *"Data Scientist"* you would guess that we work with data, and we do some sort of science, but what you might miss is that we write a lot of code. This means that good development practices **are** good data science practices! A good

practice to start with, one that is not only easy to do but extremely useful, is **testing**.

Working tests provide several benefits for your data science projects:

- They give you confidence that your code works as intended.
- They allow rapid development because breaks introduced by new code are caught sooner.
- They encode the assumptions you have about how your code works in a standard form.

Below I'll go through a few ways to test data science code using examples from my SWITRS-to-SQLite project (which I've discussed previously).

Data Extraction Tests

The first step in a data driven analysis is reading your data. Often data is read from a database, but sometimes (as was the case in SWITRS) it is from a file on disk. Testing your data loading involves generating a fake data source and attempting to read it. For example, here is the test which verifies that my code can read `.gzip` data files:

```
from etl import open_record_file

def test_read_gzipped_file(tmpdir):
    # Write a file to read back
    f = tmpdir.join("test.csv.gz")
    contents = "Test contents\nsecond line"
    file_path = f.dirname + "/" + f.basename
    with gzip.open(file_path, 'wt') as f:
        f.write(contents)

    # Read back the file
    with open_record_file(file_path) as f:
        assert f.read() == contents
```

In this test I create a very small gzip file, read it with the function being tested, and verify that it reports the same content that I wrote to the file. It was helpful to have this function when writing the library because I wanted to be able to read both zipped files and plain text (since compressing the files can save several gigabytes of disk space). This test ensured I didn't break the ability to read one of these file types while working on the other.

Data Cleaning and Transformation Tests

After the raw data is loaded, it is often necessary to parse (read, clean, and transform) it. I've found that the parser code benefits enormously from tests for three reasons:

- It has to deal with high number of edge cases because data is never as clean as we would like it to be.
- It often has the most bespoke code because everyone's data is different.
- It must change when the raw data format changes, which is more likely than not to be outside of your control.

Testing the parser is simple: create pairs of examples of raw data and correctly parsed results, then test that the code produces the right result when provided the raw data. You should have (at least) one of these pairs for each variation of data you expect to encounter. For example, here is a simplified version of the tests for my `convert` function:

```
from etl import convert

def test_convert():
    convert_vals = (
        # Pass through
        ("9", None, None, "9"),
        # Standard dtypes
        ("9", int, None, 9),
        # With spaces
        ("9 ", int, None, 9),
        (" 9", int, None, 9),
        # Nulls that do nothing
        ("9", int, [""], 9),
        # Nulls that return None
        ("9", int, ["9"], None),
        ("9", None, ["9"], None),
        # Conversion failure
        ("a", int, None, None),
    )
    for val, dtype, nulls, answer in convert_vals:
        assert convert(val=val, dtype=dtype, nulls=nulls) == answer
```

The `convert` function takes a string, a type, and a list of values that should map to the `NULL` value. If the string is in the `NULL` list then `None` should be returned, otherwise the string should be converted to the type and returned. The tests provide example inputs and check that they match the example output.

The SWITRS data files contained many edge cases—as is common in raw data! These edge cases include: multiple input values that all mean `NULL`, multiple input values that all map to `True` or `False`, a mix of strings and numbers in the same

column, and occasionally spaces prepended or appended to values. A solid set of unit tests gave me confidence that the parsing functions worked correctly and allowed me to track all of the edge cases as I discovered them.

Integration Tests

Once all components have been tested in isolation, you should test the system as a whole. This ensures that the components, which we are already confident in individually, are put together correctly into a working program. For example, I test the class that parses a row from one of the CSV files using the `convert` function (and a few others) tested above as follows:

```
from etl import VictimRow

# You should do this with a fixture, but this is a simple example
ROWS = (
  (
    #Case_ID    Party_Number  Victim_Role  Victim_Sex  Victim_Age
    # Input
    ['097293',  '1',          '2',         '-',        '20',],
    # Output
    ['097293',  1,            '2',         None,        20,],
  ),
)

def test_victimrows():
    for row, answer in ROWS:
        c = VictimRow(row)
        assert c.values == answer
```

I provide example rows from the CSV as input along with the expected correct output. You'll notice it has to read strings and convert them into strings, integers, and `NULL`. Verifying this manually by hand would be quite tedious!

Tests: Remembering So You Don't Have To

A final remark: tests are great because they keep track of the complexity of your input data! When you find a new value in your data that your code doesn't handle correctly, add it to the tests! This does two things for you:

- You'll know your code is fixed when the test starts passing.
- You will never mishandle that value again, regardless of what happens in the future.

This takes a lot of mental load off you, allowing you to focus more on the science, and less on the data parsing!

Recent Posts