SciUnit Documentation

Rick Gerkin and Cyrus Omar

Jan 23, 2021

Contents:

SciUnit

CHAPTER 1

Concept

The conference paper

CHAPTER 2

Tutorials With Jupyter NoteBook

CHAPTER 3

Basic Usage

```
my_model = MyModel(**my_args) # Instantiate a class that wraps your model of interest.
my_test = MyTest(**my_params) # Instantiate a test that you write.
score = my_test.judge() # Runs the test and return a rich score containing test␣
↪results and more.
```

CHAPTER 4

Domain-specific libraries and information

NeuronUnit for neuron and ion channel physiology See others here

CHAPTER 5

Mailing List

There is a mailing list for announcements and discussion. Please join it if you are at all interested!

CHAPTER 6

Contributors

- Rick Gerkin, Arizona State University (School of Life Science)
- Cyrus Omar, Carnegie Mellon University (Dept. of Computer Science)

CHAPTER 7

Reproducible Research ID

RRID:SCR_014528

CHAPTER 8

License

SciUnit is released under the permissive MIT license, requiring only attribution in derivative works. See the LICENSE file for terms.

CHAPTER 9

Table of Contents

9.1 What's SciUnit and how to install it?

Everyone hopes that their model has some correspondence with reality. Usually, checking whether this is true is done informally. But SciUnit makes this formal and transparent.

SciUnit is a framework for validating scientific models by creating experimental-data-driven unit tests.

9.1.1 Installation

Setting up Miniconda before the installation of SciUnit.

Then, in the Conda environment, using pip to install SciUnit as a Python package.

Note: SciUnit is no longer support Python 2.

9.2 Quick tutorial

You can read the SciUnit Basic before starting this tutorial for understanding the components of SciUnit.

9.2.1 Creating a Model and a Test instance from scratch

Let's create a model that can output a constant number.

Importing sciunit at the beginning.

```
import sciunit
```

Creating a subclass of SciUnit Capability class. The Capability subclass contains one or more unimplemented methods. It can be included in a Test class as `required_capabilities`, and only the models which implements the methods in the Capability subclass can be tested by the Test instance.

Here we define a simple capability through which a model can return a single number.

```
class ProducesNumCapability(sciunit.Capability):
    """An example capability for producing some generic number."""

    def produce_number(self):
        """The implementation of this method should return a number."""
        raise NotImplementedError("Must implement produce_number.")
```

And creating a subclass of SciUnit Model. A model we want to test is always an instance (with specific model arguments) of a more generic model class.

```
class ConstModel(sciunit.Model, ProducesNumCapability):
    """A model that always produces a constant number as output."""

    def __init__(self, constant, name=None):
        self.constant = constant
        super(ConstModel, self).__init__(name=name, constant=constant)

    def produce_number(self):
        return self.constant
```

Now we have a model and a capability. Let's create a Test class and include the capability in a subclass of Test. Note that a SciUnit test class must contain:

1. the capabilities a model requires to take the test.
2. the type of score that it will return.
3. an implementation of generate_prediction, which will use the model's capabilities to get some values out of the model.
4. an implementaiton of compute_score, to use the provided observation and the generated prediction to compute a sciunit Score.

```
class EqualsTest(sciunit.Test):
    """Tests if the model predicts
    the same number as the observation."""

    # The one capability required for a model to take this test.
    required_capabilities = (ProducesNumCapability,)

    # Set the type of score returned by judge method in a Test instance
    score_type = sciunit.scores.BooleanScore

    def generate_prediction(self, model):
        return model.produce_number()

    def compute_score(self, observation, prediction):
        score = self.score_type(observation['value'] == prediction) # Returns a␣
↪BooleanScore.
        score.description = 'Passing score if the prediction equals the observation'
        return score
```

After defining the subclass of SciUnit Model, we can create an instance of the model that always produce number 37.

```
const_model_37 = ConstModel(37, name="Constant Model 37")
```

Suppose we have a observation value, and we want to test if the value match the number predicted (produced) by the model instance defined above.

```
observation = {'value':37}
equals_37_test = EqualsTest(observation=observation, name='Equal 37 Test')
```

Simply call the `judge` method of the Test instance with the model instance as an argument.

```
score = equals_37_test.judge(model=const_model_37)
```

Now we got the score instance.

```
>>> print(score)
Pass
```

Printing out the score and we can see that the test was passed. We can also summarize the score in its entirety, printing information about the associated model and test.

```
>>> score.summarize()
=== Model Constant Model 37 achieved score Pass on test 'Equal 37 Test'. ===
```

How was that score computed again?

```
>>> score.describe()
Passing score if the prediction equals the observation
```

Next, let's create some other test instances that suppose to fail.

```
observation = {'value':36}
equals_36_test = EqualsTest(observation, name='Equal 36 Test')
observation = {'value':35}
equals_35_test = EqualsTest(observation, name='Equal 35 Test')
score1 = equals_36_test.judge(model=const_model_37)
score2 = equals_36_test.judge(model=const_model_37)
```

```
>>> print(score1)
Fail
```

```
>>> print(score2)
Fail
```

We can also put these test instances together in a TestSuite instance. The TestSuite also contains a `judge` method that can run every Test instance's `judge` methods.

```
tests = [equals_35_test, equals_36_test, equals_37_test]
equals_suite = sciunit.TestSuite(tests=tests, name="Equals test suite")
score_matrix = equals_suite.judge(const_model_37)
```

```
>>> print(score_matrix)
                  Equal 35 Test Equal 36 Test Equal 37 Test
Constant Model 37          Fail          Fail          Pass
```

In the result, we can see a 1*3 score matrix that shows the results of each test.

We can create more models and subject those to the test suite to get a more extensive score matrix.

```
const_model_36 = ConstModel(36, name='Constant Model 36')
const_model_35 = ConstModel(35, name='Constant Model 35')
score_matrix = equals_suite.judge([const_model_36, const_model_35, const_model_37])
```

```
>>> print(score_matrix)
                   Equal 35 Test Equal 36 Test Equal 37 Test
Constant Model 37          Fail          Fail          Pass
Constant Model 36          Fail          Pass          Fail
Constant Model 35          Pass          Fail          Fail
```

Now, we can see the result is a 3*3 matrix, and each model pass the corresponding test.

We can also examine the results only for one of the tests in the suite.

```
>>> print(score_matrix[equals_35_test])
Constant Model 37    Fail
Constant Model 36    Fail
Constant Model 35    Pass
Name: Equal 35 Test, dtype: object
```

Or examine the results only for one of the models.

```
>>> print(score_matrix[const_model_35])
Equal 35 Test    Pass
Equal 36 Test    Fail
Equal 37 Test    Fail
Name: Constant Model 35, dtype: object
```

In the next section we'll see how to build slightly more sophisticated tests using objects built-in to SciUnit.

9.2.2 Testing with help from the SciUnit standard library

The `ConstModel` class we defined in the last section was included in SciUnit package as an example, and we can just import it.

```
import sciunit

from sciunit.models.examples import ConstModel
from sciunit.capabilities import ProducesNumber

from sciunit.scores import ZScore # One of many SciUnit score types.
from sciunit.errors import ObservationError # An exception class raised when a test
```

Let's create the instance of ConstModel.

```
const_model_37 = ConstModel(37, name="Constant Model 37")
```

And a new subclass of SciUnit Test class.

```
class MeanTest(sciunit.Test):
    """Tests if the model predicts the same number as the observation."""

    # The one capability required for a model to take this test.
    required_capabilities = (ProducesNumber,)

    # This test's 'judge' method will return a BooleanScore.
    score_type = ZScore

    def validate_observation(self, observation):
        if type(observation) is not dict:
```

(continues on next page)

(continued from previous page)

```
            raise ObservationError("Observation must be a python dictionary")
        if 'mean' not in observation:
            raise ObservationError("Observation must contain a 'mean' entry")

    def generate_prediction(self, model):
        return model.produce_number()

    def compute_score(self, observation, prediction):

        # Compute and return a ZScore object.
        score = ZScore.compute(observation,prediction)

        score.description = ("A z-score corresponding to the normalized location of␣
↪the"
                             "observation relative to the predicted distribution.")
        return score
```

Compared with the sruff in last section, we've done two new things here:

- The optional `validate_observation` method checks the observation to make sure that it is the right type, that it has the right attributes, etc. This can be used to ensures that the observation is exactly as the other core test methods expect. If we don't provide the right kind of observation:
- Instead of returning a BooleanScore, encoding a True/False value, we return a ZScore encoding a more quantitative summary of the relationship between the observation and the prediction.

Let's create a observation and attach it to the MeanTest instance.

```
observation = {'mean':37.8, 'std':2.1}
mean_37_test = MeanTest(observation, name='Equal 37 Test')
score = mean_37_test.judge(const_model_37)
```

And let's see what's the result:

```
>>> score.summarize()
=== Model Constant Model 37 achieved score Z = -0.38 on test 'Equal 37 Test'. ===
```

```
>>> score.describe()
A z-score corresponding to the normalized location of theobservation relative to the␣
↪predicted distribution.
```

9.2.3 Example of RunnableModel and Backend

Beside the usual model in previous sections, let's create a model that run a Backend instance to simulate and obtain results.

Firstly, import necessary components from SciUnit package.

```
import sciunit, random
from sciunit import Test
from sciunit.capabilities import Runnable
from sciunit.scores import BooleanScore
from sciunit.models import RunnableModel
from sciunit.models.backends import register_backends, Backend
```

Let's define subclasses of SciUnit Backend, Test, and Model.

Note that:

1. A SciUnit Backend subclass should implement `_backend_run` method.
2. A SciUnit Backend subclass should implement `run` method.

```
class RandomNumBackend(Backend):
    '''generate a random integer between min and max'''

    def set_run_params(self, **run_params):

        # get min from run_params, if not exist, then 0.
        self.min = run_params.get('min', 0)

        # get max from run_params, if not exist, then self.min + 100.
        self.max = run_params.get('max', self.min + 100)

    def _backend_run(self):
        # generate and return random integer between min and max.
        return random.randint(self.min, self.max)

class RandomNumModel(RunnableModel):
    """A model that always produces a constant number as output."""

    def run(self):
        self.results = self._backend.backend_run()


class RangeTest(Test):
    """Tests if the model predicts the same number as the observation."""

    # Default Runnable Capability for RunnableModel
    required_capabilities = (Runnable,)

    # This test's 'judge' method will return a BooleanScore.
    score_type = BooleanScore

    def generate_prediction(self, model):
        model.run()
        return model.results

    def compute_score(self, observation, prediction):
        score = BooleanScore(
            observation['min'] <= prediction and observation['max'] >= prediction
        )
        return score
```

Let's define the model instance named `model 1`.

```
model = RandomNumModel("model 1")
```

We must register any backend isntance in order to use it in model instances.

`set_backend` and `set_run_params` methods can help us to set the run-parameters in the model and its backend.

```
register_backends({"Random Number": RandomNumBackend})
model.set_backend("Random Number")
model.set_run_params(min=1, max=10)
```

Next, create an observation that requires the generated random integer between 1 and 10 and a test instance that use the observation and against the model

```
observation = {'min': 1, 'max': 10}
oneToTenTest = RangeTest(observation, "test 1")
score = oneToTenTest.judge(model)
```

print the score, and we can see the result.

```
>>> print(score)
Pass
```

9.2.4 Real Example

For real example of using SciUnit, you can read Chapter 5 and 6 of the Jupyter notebook tutorial.

Tutorial Chapter 5

Tutorial Chapter 6

9.3 SciUnit basics

This page will give you a basic view of the SciUnit project, and you can read the quick tutorials for some simple examples.

The major parts of SciUnit are Score, Test, and Model.

9.3.1 Model

`Model` is the abstract base class for sciunit models. Generally, a model instance can generate predicted or simulated results of some scientific fact.

Runnable Model

`Runnable model` is a kind of model that implements Runnable capability, and it can be executed to simulate and output results.

Backend

After being registered by `register_backends` function, a `Backend` instance can be executed by a Ruunable Model at the back end. It usually does some background computing for the runnable model.

9.3.2 Score

`Score` is the abstract base class for scores. The instance of it (or its subclass) can give some types of results for test and/or test suite against the models.

The complete scores type in SciUnit are `BooleanScore`, `ZScore`, `CohenDScore`, `RatioScore`, `PercentScore`, and `FloatScore`.

Each type of score has their own features and advantage.

There are also incomplete score types. These type does not contain any information regarding how good the model is, but the existing of them means there are some issues during testing or computing process. They are `NoneScore`, `TBDScore`, `NAScore`, and `InsufficientDataScore`

ScoreArray, ScoreArrayM2M

Can be used like this, assuming n tests and m models:

```
>>> sm[test]
    (score_1, ..., score_m)
```

```
>>> sm[model]
    (score_1, ..., score_n)
```

`ScoreArray` represents an array of scores derived from a test suite. Extends the pandas Series such that items are either models subject to a test or tests taken by a model. Also displays and computes score summaries in sciunit-specific ways.

`ScoreArrayM2M` represents an array of scores derived from `TestM2M`. Extends the pandas Series such that items are either models subject to a test or the test itself.

ScoreMatrix, ScoreMatrixM2M

Can be used like this, assuming n tests and m models:

```
>>> sm[test]
(score_1, ..., score_m)
```

```
>>> sm[model]
(score_1, ..., score_n)
```

`ScoreMatrix` represents a matrix of scores derived from a test suite. Extends the pandas DataFrame such that tests are columns and models are the index. Also displays and compute score summaries in sciunit-specific ways.

`ScoreMatrixM2M` represents a matrix of scores derived from `TestM2M`. Extends the pandas DataFrame such that models/observation are both columns and the index.

9.3.3 Test, TestM2M

`Test` is a abstract base class for tests.

`TestM2M` is an abstract class for handling tests involving multiple models.

A test instance contains some observations which are considered as the fact. The test instance can test the model by comparing the predictions with the observations and generate a specific type of score.

Enables comparison of model to model predictions, and also against experimental reference data (optional).

Note: `TestM2M` would typically be used when handling mutliple (>2) models, with/without experimental reference data. For single model tests, you can use the 'Test' class.

9.3.4 TestSuite

A collection of tests. The instance of `TestSuite` can perform similar things that a test instance can do.

9.3.5 Converter

A `Converter` instance can be used to convert a score between two types. It can be included in a test instance.

9.3.6 Capability

`Capability` is the abstract base class for sciunit capabilities. A capability instance can be included in a test instance to ensure the model, which is tested by the test instance, implements some methods.

9.4 Config File And Using SciUnit In A Shell

9.4.1 Create Config File And Execute Tests In A Shell

We can build a scientific computing project with a SciUnit config file. Then, we will be able to run sciunit In A Shell

Here is an example of well written SciUnit config file. This file was generated by executing `sciunit create` in the shell. A SciUnit config file is always named `sciunit.ini`.

```
[misc]
config-version = 1.0
nb-name = scidash

[root]
path = .

[models]
module = models

[tests]
module = tests

[suites]
module = suites
```

`config-version` is the version of the config file.

`nb-name` is the name of the IPython Notebook file that can be create with `sciunit make-nb`.

`root` is the root of the project. The `path` is the path to the project from the directory that contains this config file.

`module` in the `models` section is the path from the root of the project to the file that contains `models`, which is a list of `Model` instances.

`module` in the `tests` section is the path the root of the project to the file that contains `tests`, which is a list of `Test` instances.

`module` in the `suites` section is the path the root of the project to the file that contains `suites`, which is a list of `TestSuite` instances.

Let's use the config file above and create corresponding files that contain definitions models, tests, and suites.

In the root directory of the project, let's create three files.

tests.py

```
import sciunit
from sciunit.scores import BooleanScore
from sciunit.capabilities import ProducesNumber



class EqualsTest(sciunit.Test):
```

(continues on next page)

(continued from previous page)

```
    """Tests if the model predicts
    the same number as the observation."""

    required_capabilities = (ProducesNumber,)
    score_type = BooleanScore

    def generate_prediction(self, model):
        return model.produce_number() # The model has this method if it inherits from␣
↪the 'ProducesNumber' capability.

    def compute_score(self, observation, prediction):
        score = self.score_type(observation['value'] == prediction)
        score.description = 'Passing score if the prediction equals the observation'
        return score

tests = []
```

suites.py

```
import sciunit
from tests import EqualsTest


equals_1_test = EqualsTest({'value':1}, name='=1')
equals_2_test = EqualsTest({'value':2}, name='=2')
equals_37_test = EqualsTest({'value':37}, name='=37')

equals_suite = sciunit.TestSuite([equals_1_test, equals_2_test, equals_37_test], name=
↪"Equals test suite")

suites = [equals_suite]
```

models.py

```
import sciunit
from sciunit.capabilities import ProducesNumber


class ConstModel(sciunit.Model,
                 ProducesNumber):
    """A model that always produces a constant number as output."""

    def __init__(self, constant, name=None):
        self.constant = constant
        super(ConstModel, self).__init__(name=name, constant=constant)

    def produce_number(self):
        return self.constant

const_model_1 = ConstModel(1, name='Constant Model 1')
const_model_2 = ConstModel(2, name='Constant Model 2')
const_model_37 = ConstModel(37, name="Constant Model 37")

models = [const_model_1, const_model_2, const_model_37]
```

We have `tests` at the end of `tests.py`, `models` at the end of `models.py`, and `suites` at the end of `suites.py`. Since we are using test suites instead of tests, `tests` is an empty list in this example. They will be taken by

sciunit when command `sciunit run` is being executing

Execute `sciunit run` in the root directory, and then sciunit will run each test in the suites against each model and give us the result.

```
$ sciunit run


Executing test =1 on model Constant Model 1... Score is Pass
Executing test =2 on model Constant Model 1... Score is Fail
Executing test =37 on model Constant Model 1... Score is Fail
Executing test =1 on model Constant Model 2... Score is Fail
Executing test =2 on model Constant Model 2... Score is Pass
Executing test =37 on model Constant Model 2... Score is Fail
Executing test =1 on model Constant Model 37... Score is Fail
Executing test =2 on model Constant Model 37... Score is Fail
Executing test =37 on model Constant Model 37... Score is Pass

Suite Equals test suite:
                     =1    =2   =37
Constant Model 1   Pass  Fail  Fail
Constant Model 2   Fail  Pass  Fail
Constant Model 37  Fail  Fail  Pass
```

9.4.2 Create and Run IPython Notebook File

Next, let's move to creating and executing IPython Notebook file with `sciunit make-nb` and `sciunit run-nb` commands.

Let's add a file, `__init__.py`, to our project directory and import everything including suites, tests, and models in the file. This is necessary because the made notebook file will try to import everything in `__init__.py` and run each suite (a collection of tests instances) against each model.

__init__.py

```
from . import models
from . import tests
from . import suites
```

Now, let's execute `sciunit make-nb` SciUnit will automatically generate a notebook file.

```
$ sciunit make-nb
Created Jupyter notebook at:
/the_path_to_the_project.../test_sciunit.ipynb
```

The notebook file will contains two blocks of code:

Note:

1. the name of generated notebook file will be the value of `nb-name` attribute in the config file, `sciunit.ini`
2. The path to the project's root can be different on different machine. So, The notebook file generated usually only be valid on the machine where it is generated. If you want to execute it on different machine, try to re-generate it or change the path.

Let's execute `sciunit run-nb` command.

```
$ sciunit run-nb
Entering run function
/the_path_to_the_project_config_file..././test_sciunit.ipynb
/the_path_to_the_project_config_file.../.
```

The result of running the notebook will be in the notebook file. You can open it by many tools like VS Code and Jupyter Lab

CHAPTER 10

Indices and tables

- genindex
- modindex
- search