Collaborative Testing for Scientific Model Validation

Cyrus Omar, Jonathan Aldrich
Carnegie Mellon University
{comar,aldrich}@cs.cmu.edu

Richard C. Gerkin
Arizona State University
rgerkin@asu.edu

ABSTRACT

One of the pillars of the modern scientific method is *model validation*: comparing a scientific model's predictions against empirical observations. Today, a scientist demonstrates the validity of a model by making an argument in a paper and submitting it for peer review, a process comparable to *code review* in software engineering. While human review helps to ensure that contributions meet high-level goals, software engineers typically supplement this with *unit testing* to get a more complete picture of the status of a software project, particularly for complex projects involving many developers.

We argue that a similar test-driven methodology would be valuable to scientific communities as they seek to validate increasingly complex models against growing collections of empirical data. The dynamics of scientific communities and software communities differ in several key ways, however. In this paper, we introduce *SciUnit*, a framework for test-driven scientific model validation. We describe how SciUnit, supported by new and existing collaborative infrastructure, can integrate into the modern scientific process.

1. INTRODUCTION

Scientific theories often take the form of a *quantitative model*: a formal structure that can generate predictions about observable quantities. Such a model is characterized by its *scope*: the set of observable quantities that the model can predict, and by its *validity*: the extent to which its predictions agree with experimental observations of these quantities.

Quantitative models are validated by *peer review*. For a model to be accepted by the scientific community, scientists must submit a paper describing it and providing evidence that it predicts some quantity of interest more accurately than previous models, or that it makes a desirable tradeoff between accuracy and complexity [**?**]. Other members of the relevant community are then tasked with ensuring that validity was measured properly and that relevant data and competing models were adequately considered, drawing on knowledge of statistical methods and the prior literature. Publishing is a primary motivator for most scientists [**?**].

Quantitative scientific modeling and software development have much in common. Indeed, quantitative models are increasingly implemented as computer programs and in some cases, the program *is* the model (e.g. complex simulations). The peer review process for papers is similar in many ways to the *code review* process used in many development teams, where team members look for errors, enforce style and architectural guidelines and check that the code is *valid* (i.e. that achieves its intended goal [**?**]) before permitting it to be committed to the primary source code repository [**?**].

Code review can be quite effective [**?**, **?**], but this requires that developers expend considerable effort [**?**]. Code review is also most effective for resolving issues related to software architecture [**?**]. Most development teams thus supplement code reviews with automated approaches to verification and validation, the most widely-used of which is *unit testing* [**?**]. In brief, unit tests are functions that check a single portion of a program against a single well-defined correctness criterion. A suite of such tests can be seen as a partial specification of a program or component. Test suites complement code review by allowing developers to answer questions like these more easily:

1. Which functionality has been adequately implemented? What remains to be done?
2. What modes of usage does the team consider most important? How does the team measure correctness?
3. Does a code contribution cause *regressions* in other parts of a program?

Scientists ask analagous questions:

1. Which observations are already well-explained by existing models? What is the state-of-the-art? What are the open modeling problems of interest?
2. What are the contemporary community standards for measuring goodness-of-fit?
3. How do newly-made experimental observations impact the validity of previous models?

But while software engineers can rely on a program's test suite, scientists today must extract this information from a body of scientific publications. This is increasingly difficult. Each publication focuses on just one model and is frozen in time, so it does not consider the latest experimental data or statistical methods. Discovering, precisely

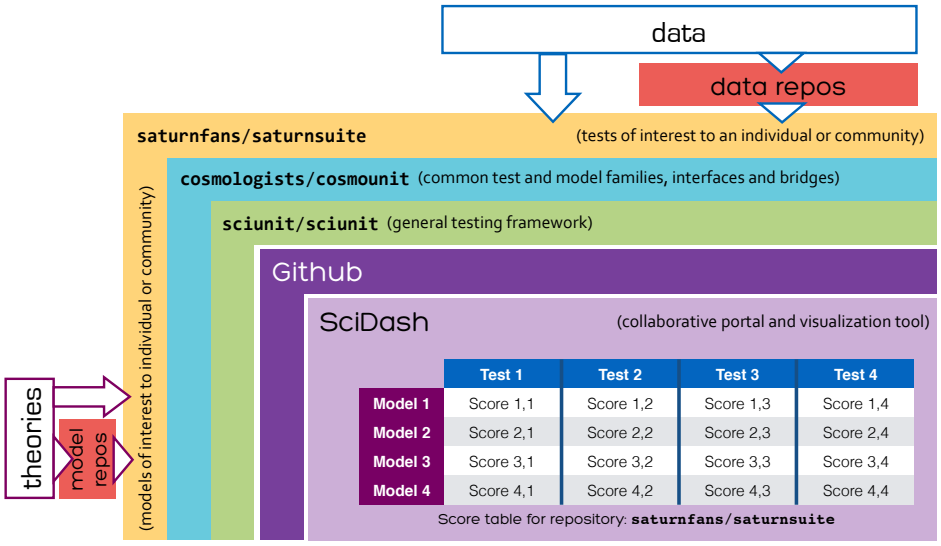

Figure 1: The central table summarizes the performance of a collection of models against a suite of tests. These are collaboratively curated in *suite repositories* using social coding infrastructure – Github – combined with a portal that organizes and summarizes them. Tests are derived from data and models are derived from scientific theories, parameterized by *training data*. Tests communicate with models using interface specifications that are also collaboratively developed in repositories that build upon the core SciUnit testing framework. These repositories can also contain common test and model families and bridges to existing infrastructure and tools.

characterizing and comparing several models to discover the state of the art and find open modeling problems can often require an encyclopedic knowledge of the literature. Senior scientists can attempt to fill this need by publishing review papers, but in many areas, the number of publications generated every year can be overwhelming [**?**], and comprehensive reviews are published relatively infrequently for any given area. Statisticians commonly complain that scientists are not following best practices and that community standards evolve too slowly because canonical papers or popular reviews advocate outdated methods [**?**]. One might compare this process to a software development team relying primarily on carefully-prepared summaries of code review sessions, peppered with cross-references to previous summaries, to understand and validate a complex software project.

This suggests that the scientific process could be improved by the adoption of test-driven methodologies alongside peer review. However, the scientific community presents several unique challenges that must first be addressed:

1. Unit tests are typically pass/fail, while goodness-of-fit is typically a continuous value (e.g. a p-value).

2. Different communities, groups and individual scientists prefer different goodness-of-fit measures and focus on different sets of observable quantities. This can evolve as, for example, statisticians develop better measures.

3. Model developers compete with one another to predict data, rather than collaborating with one another on a common codebase. They are sometimes reluctant to share certain details of their implementations [**?**].

4. Experimental data can itself be erroneous, noisy or inconsistent with other available data.

5. Data formats can differ, sometimes drastically, between experimentalists. Similarly, different modelers may prefer different programming languages or libraries.

6. Professional software developers are typically trained in testing practices and tools, while scientists rarely have training or experience with testing practices [**?**].

Test-driven methodologies have started to see success in neuroscience as well. Modeling competitions in neuroscience, for example, are typically organized around a collection of simple validation criteria, implemented as executable tests. These competitions continue to drive important advances and improve scientists' understanding of the relative merits of different models. For example, the quantitative single neuron modeling competition (QSNMC) [**?**] investigates the complexity-accuracy tradeoff among reduced models of excitable membranes; the "Hopfield" challenge [**?**] tested techniques for generating neuronal network form given its function; the Neural Prediction Challenge sought the best stimulus reconstructions, given neuronal activity (*http://neuralprediction.berkeley.edu*); the Diadem challenge is advancing the art of neurite reconstruction (*http://www.diademchallenge.org*); and examples from other subfields of biology abound (*http://www.the-dream-project.org*).

Each of these examples has leveraged *ad hoc* infrastructure to support test generation. While the specific criteria used to evaluate models varies widely between disciplines in neuroscience, the underlying test-driven methodology has many common features that could be implemented once. Recognizing this, we developed a discipline-agnostic framework for developing scientific validation test suites called *SciUnit* (*http://www.sciunit.org*). Here we describe *NeuronUnit*, which builds upon *SciUnit*, allowing neuroscientists to build *SciUnit* tests that validate neurophysiology models against

```
1  class SpikeCountTest(sciunit.Test):
2    """Tests spike counts produced in response to
         several current stimuli against observed means
         and standard deviations.
3
4    goodness of fit metric: Computes p-values based on a
          chi-squared test statistic, and pools them
         using Fisher's method.
5    parameters:
6      inputs: list of numpy arrays containing input
           currents (nA)
7      means, stds: lists of observed means and standard
           deviations, one per input
8    """
9    def __init__(self, inputs, means, stds):
10     self.inputs, self.means, self.stds = inputs, means
           , stds
11
12   required_capabilities = [SpikeCountFromCurrent]
13
14   def _judge(self, model):
15     inputs, means, stds = self.inputs, self.means,
           self.stds
16     n = len(inputs)
17     counts = numpy.empty((n,))
18     for i in xrange(n):
19       counts[i] = model.spike_count_from_current(
             inputs[i])
20     chisquared = sum((counts-means)**2 / means) # An
           array of chi-squared values.
21     p = scipy.stats.chi2.cdf(chisquared,n-1) # An
           array of p-values.
22     pooled_p = sciunit.utils.fisherp(p_array) # A
           pooled p-value.
23     return sciunit.PValue(pooled_p, related_data={
24       "inputs": inputs, "counts": counts, "obs_means":
             means, "obs_stds": stds
25     })
```

Figure 2: An example single neuron spike count test class implemented using *SciUnit*. Because this implementation contains logic common to many different systems, *NeuronUnit* was developed to provide a simpler means to deliver it (see Sec. ??).

electrophysiological data. We provide a concrete example pipeline, showing how models described using NeuroML and provided freely by the *Open Source Brain Project* (OSB, [?], *http://www.opensourcebrain.org*) can be tested in fully automated fashion using published, curated data available through the *NeuroElectro Project* (Neuroelectro, [?], *http://neuroelectro.org*), leveraging facilities from the *NeuroTools* library (*http://neuralensemble.org/NeuroTools*) to extract relevant features of model output. This is summarized in Figure 1, which shows the relationships between the layers described here.

2. VALIDATION TESTING WITH *SCIUNIT*

2.1 Example: The Quantitative Single Neuron Modeling Competition

We first illustrate the form of a generic example *SciUnit* test suite that could be used in neurophysiology. Suppose we have collected data from an experiment where current stimuli (measured in pA) are delivered to neurons in some brain region, while the somatic membrane potential of each stimulated cell (in mV) is recorded and stored. A model claiming to capture this cell type's membrane potential dynamics must be able to accurately predict a variety of features observed in these data.

One simple validation test would ask candidate models to predict the number of action potentials (a.k.a. spikes) generated in response to a stimulus (e.g. white noise), and compare these *spike count* predictions to the distribution observed in repeated experimental trials using the same stimulus. For data of this type, goodness-of-fit can be measured by first calculating a p-value from a chi-squared statistic for each prediction and then combining these p-values using Fisher's method [?].

Alongside this *spike count test*, we might also specify a number of other tests capturing different features of the data to produce a more comprehensive suite. For data of this sort, the QSNMC defined 17 other validation criteria in addition to one based on the overall spike count, capturing features like spike latencies (SL), mean subthreshold voltage (SV), interspike intervals (ISI) and interspike minima (ISM) that can be extracted from the data [?]. They then defined a combined metric favoring models that broadly succeeded at meeting these criteria, to produce an overall ranking. Such combined criteria are simply validation tests that invoke other tests to produce a result.

2.2 Implementing a Validation Test in *SciUnit*

Fig. 2 shows how a scientist can implement spike count tests such as the one described above using *SciUnit*. A *SciUnit* validation test is an instance (i.e. an object) of a Python class implementing the sciunit.Test interface (cf. line 1). Here, we show a class SpikeCountTest taking three *parameters* in its constructor (constructors are named __init__ in Python, lines 9-10). The meaning of each parameter along with a description of the goodness-of-fit metric used by the test is documented on lines 4-7. To create a *particular* spike count test, we instantiate this class with particular experimental observations. For example, given observations from hippocampal CA1 cells (not shown), we can instantiate a test as follows:

```
1  CA1_sc_test = SpikeCountTest(CA1_inputs, CA1_means,
       CA1_stds)
```

We emphasize the crucial distinction between the *class* SpikeCountTest, which defines a *parameterized family* of validation tests, and the particular *instance* CA1_sc_test, which is an individual validation test because the necessary parameters, derived from data, have been provided. As we will describe below, we expect communities to build repositories of such families capturing the criteria used in their subfields of neuroscience so that test generation for a particular system of interest will often require simply instantiating a previously-developed family with particular experimental parameters and data. For single-neuron test families like SpikeCountTest, we have developed such a library, called *NeuronUnit* (*http://github.com/scidash/neuronunit*) (Sec. ??).

Classes that implement the sciunit.Test interface must contain a _judge method that receives a candidate *model* as input and produces a *score* as output. To specify the interface between the test and the model (that is, to specify an appropriate scope), the test author provides a list of *capabilities* in the required_capabilities attribute, seen on line 12 of Fig. 2. Capabilities are simply collections of methods that a test will need to invoke in order to receive relevant data, and are analogous to *interfaces* in e.g. Java (*http://docs.oracle.com/javase/tutorial/java/concepts/interface.html*). In Python, capabilities are written as classes with unimplemented members. The capability required by the test in Fig. 2 is shown in Fig. 3. In *SciUnit*, classes defining capabilities are tagged

```
1  class SpikeCountFromCurrent(sciunit.Capability):
2    def spike_count_from_current(self, input):
3      """Takes a numpy array containing current stimulus
           (in nA) and
4      produces an integer spike count. Can be called
         multiple times."""
5      raise NotImplementedError("Model does not
         implement capability.")
```

Figure 3: An example capability specifying a single required method (used by the test in Figure 2).

```
1  class TrainSpikeCountFromCurrent(sciunit.Capability):
2    def train_with_currents(self, currents, counts):
3      """Takes a list of numpy arrays containing current
         stimulus (in nA) and
4      observed spike counts. Model parameters should be
         adjusted based on this
5      training data."""
6      raise NotImplementedError("Model does not
         implement capability.")
```

Figure 4: Another capability specifying a training protocol (not used by the test in Figure 2).

as such by inheriting from sciunit.Capability. The test in Figure 2 uses this capability on line 19 to produce a spike count prediction for each input current.

The remainder of the _judge method implements the goodness-of-fit metric described above, returning an instance of sciunit.PValue, a subclass of sciunit.Score that is included with *SciUnit*. In addition to the p-value itself, the returned score object also contains metadata, via the related_data parameter, for scientists who may wish to examine the result in more detail later. In this case we save the input currents, the model outputs and the observed means and standard deviations (line 24).

2.3 Models

Capabilities are *implemented* by models. In *SciUnit*, models are instances of Python classes that inherit from sciunit.Model. Like tests, the class itself represents a family of models, parameterized by the arguments of the constructor. A particular model is an instance of such a class.

Figure 5 shows how to write a simple family of models, LinearModel, that implement the capability in Fig. 3 as well as another capability shown in Fig. 4, which we will discuss below. Models in this family generate a spike count by applying a linear transformation to the mean of the provided input current. The family is parameterized by the scale factor and the offset of the transformation, both scalars. To create a *particular* linear model, a modeler can provide particular parameter values, just as with test families:

```
1  CA1_linear_model_heuristic = LinearModel(3.0, 1.0)
```

Here, the parameters to the model were picked by the modeler heuristically, or based on externally-available knowledge. An alternative test design would add a training phase where these parameters were fit to data using the capability shown in Fig. 4. This test could thus only be used for those models for which parameters can be adjusted without human involvement. Whether to build a training phase into the test protocol is a choice left to each test development community.

Fig. 2 does not include a training phase. If training data is externally available, models that nevertheless do implement a training capability (like `LinearModel`) can simply be

```
1  class LinearModel(sciunit.Model, SpikeCountFromCurrent
       ,
2      TrainSpikeCountFromCurrent):
3    def __init__(self, scale=None, offset=None):
4      self.scale, self.offset = scale, offset
5
6    def spike_count_from_current(self, input):
7      return int(self.scale*numpy.mean(input) + self.
         offset)
8
9    def train_with_currents(self, currents, counts):
10     means = [numpy.mean(c) for c in currents]
11     [self.offset, self.scale] = numpy.polyfit(means,
         counts, deg=1)
```

Figure 5: A model that returns a spike count by applying a linear transformation to the mean input current. The parameters can be provided manually or learned from data provided by a test or user (see text).

trained explicitly by calling the capability method just like any other Python method:

```
1  CA1_linear_model_fit = LinearModel()
2  CA1_linear_model_fit.train_with_currents(
       CA1_training_in, CA1_training_out)
```

2.4 Executing Tests

A test is executed against a model using the judge method:

```
1  score = CA1_sc_test.judge(CA1_linear_model_heuristic)
```

This method proceeds by first checking that the provided model implements all required capabilities. It then calls the test's _judge method to produce a score. A reference to the test and model are added to the score for convenience (accessible via the test and model attributes, respectively), before it is returned.

2.5 Test Suites and Score Matrices

A collection of tests intended to be run on the same model can be put together to form a test suite. The following is a test suite that could be used for a simplified version of the QSNMC, as described above:

```
1  CA1_suite = sciunit.TestSuite([CA1_sc_test,
       CA1_sl_test, CA1_sv_test, CA1_isi_test,
       CA1_ism_test])
```

Like a single test, a test suite is capable of judging one or more models. The result is a score matrix much like the one diagramed in Fig. 1.

```
1  CA1_matrix = CA1_suite.judge([
       CA1_linear_model_heuristic, CA1_linear_model_fit
       ])
```

A simple summary of the scores in a score matrix can be printed to the console or visualized by other tools, such as the web application *SciDash* described in Sec. **??**.