Collaborative Infrastructure for Test-Driven Scientific Model Validation

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

Richard C. Gerkin
Arizona State University, USA
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 it with *unit testing* to get a more complete picture of the status of a project, particularly when it is complex and involves many contributors.

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

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Documentation; measurement; standardization

Keywords

unit testing; model validation; cyberinfrastructure

1 Introduction

Scientific theories are increasingly being organized around *quantitative models*: formal systems capable of generating predictions about observable quantities. A model can be characterized by its *scope*: the set of observable quantities that it attempts to predict, and by its *validity*: the extent to which its predictions agree with experimental observations of these quantities.

Quantitative models are today validated by *peer review*. For a model to be accepted by a scientific community, its advocates must submit a paper that describes how it works and provides evidence that it makes more accurate predictions than previous models (or that it makes a desirable tradeoff between accuracy and complexity) [2]. Other members of the relevant community are then tasked with ensuring that validity was measured properly and that all 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 [3].

Quantitative scientific modeling shares much in common with software development. Indeed, quantitative models are increasingly being implemented in software and in some cases, the software *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 mistakes, enforce style and architectural guidelines and check that the code is *valid* (i.e. that it achieves its intended goal) before permitting it to be committed to the primary source code repository.

Code review can be quite effective [8], but this requires that developers expend considerable effort [5]. Most large development teams thus supplement code reviews with more automated approaches to verification and validation, the most widely-used of which is *unit testing* [1]. In brief, unit tests are functions that check that the behavior of a single component satisfies a single functional criterion. A suite of such tests complements code review by making it easier to answer questions like these:

1. What functionality should each component have?
2. What functionality has been adequately implemented? What remains to be done?
3. Does a candidate code contribution cause *regressions* in other parts of a program?

Scientists ask analagous questions:

1. What are the contemporary community standards for measuring goodness-of-fit?
2. Which observations are already explained by existing models? What are the best models for a particular quantity? What are the open modeling problems of interest? What data has yet to be explained?
3. How do newly-made experimental observations impact the validity of previously published models? Can new models explain previously published data?

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

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

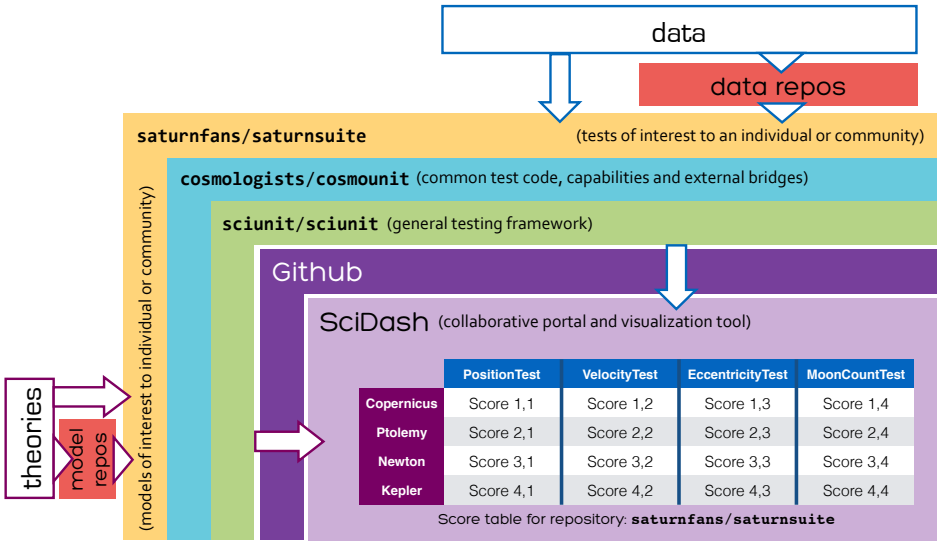

Figure 1: Tests are derived from data and models are derived from scientific theories, either directly or through existing data and model repositories. The score table summarizes the performance of a collection of models against a suite of tests. Tests and models are initialized, executed and visualized using `sciunit` in an IPython notebook stored inside a *suite repository* (e.g. `saturnsuite`) hosted on a social coding website (e.g. Github). SciDash discovers and organizes suite repositories and provides interactive views of the notebooks they contain. Common testing code, capabilities and bridges to external model and data repositories are also collaboratively developed (e.g. `cosmounit`).

precisely characterizing and comparing models to discover the state of the art and find open modeling problems can require an encyclopedic knowledge of the literature, as can finding all data relevant to a model. Senior scientists often attempt to fill this need by publishing review papers, but in many areas, the number of publications generated every year can be overwhelming [4], and comprehensive reviews of a particular area are published infrequently. Statisticians often complain that scientists are not following best practices and that community standards evolve too slowly because a canonical paper or popular review used outdated methods. Furthermore, if the literature simply doesn't address an important question of validity, because new data has been gathered since the model paper was published for example, a researcher might need to reimplement an existing model in order to evaluate it.

One might compare this to a large software development team answering the questions listed above based largely on carefully reviewed, but rarely updated, API documentation and annual high-level project summaries. While certainly a caricature, this motivates our suggestion that the scientific process could be improved by the adoption of test-driven methodologies alongside traditional peer review. However, the scientific community presents several unique challenges that must be addressed by any such methodology:

1. Unit tests are typically pass/fail, while goodness-of-fit between a model and data is typically measured by a continuous metric.
2. Unit tests often test a *particular* component, whereas a *validation testing* system must be able to handle and compare many models with the same scope, developed by teams who only loosely coordinate their efforts.
3. Scientists use a variety of programming languages.
4. Professional software developers are typically trained in testing practices and tools, while scientists rarely have training or experience with testing practices [7].
5. Different communities, groups and individuals prefer different goodness-of-fit metrics and focus on different sets of observable quantities. In contrast, there is typically more pressure to agree upon requirements and priorities in a software development project.

To begin to address these challenges, we will introduce a lightweight and flexible validation testing and framework called *SciUnit* in Sec. 2. Many of these challenges have to do with coordination between scientists. To begin to address this, we then describe a community workflow based on widely-adopted social coding tools (here, Github) and a lightweight community portal called *SciDash*, in Sec. 3. The overall goal of this work is to help scientists generate and examine tables like the one central to Figure 1, where the relative validity of a set of models having a common scope can be determined by examining scores produced by a suite of validation tests constructed from experimental data. We discuss adoption strategies and directions for future research into scientific model validation practices in Sec. 4.

2 Validation Testing with SciUnit

As a motivating example, we will begin by considering a community of early cosmologists recording and attempting to model observations of the planets visible in the sky, such as their position, velocity, orbital eccentricity and so on. One simple validation test might ask a model to predict planetary position on night $n+1$ given observations of its position on n previous nights. Figure 2 shows how to implement a test, using SciUnit, that captures this logic.

Before explaining the details of this example, we point out that SciUnit is implemented in Python. Python is one of the most widely used languages in science today [6], and is increasingly regarded as the *de facto* language of open source scientific computing. It is easy to read and has a simple object system that we use to specify the interface between tests and models (challenge 2). It supports calling into many other popular languages, including R, MAT-

```
1  class PositionTest(sciunit.Test):
2    """Tests a planetary position model based on
         positions observed on night n given the
         positions in the n-1 previous nights.
3    Observation: {
         'histories':list[list[Position]],
         'positions':list[Position]]
6    }"""
7    required_capabilities = [PredictsPlanetaryPosition]
8    def generate_prediction(self, model):
9      return [model.predict_next_pos(obs_history)
10       for obs_history
11       in self.observation['histories']]
12
13   def compute_score(self, observation, prediction):
14     obs_positions = observation['positions']
15     return pooled_p_val([abs_distance(obs, pred)
16       for (obs, pred)
17       in zip(obs_positions, prediction)])
```

Figure 2: An example test class in `cosmounit`.

LAB, C and Java, often more cleanly than they support calling into Python (challenge 3). The IPython notebook is a powerful web-based "read-eval-visualize loop" that we use to support interactive table visualization, and it also permits using other languages on a per-cell basis [**?**]. Together, this makes writing wrappers around tests and models written in other languages relatively simple. We anticipate leveraging this infrastructure to make generating such wrappers nearly automatic as future work.

A *SciUnit* validation test is an instance of a Python class implementing the `sciunit.Test` interface (line 1). A test takes one required parameter in its constructor (inherited from the base class) containing the observations against which the test will validate models. For the class `PositionTest`, this observation parameter is a dictionary containing two types of observations (documented per Python conventions on lines 2-6). To create a *particular* position test, we instantiate this class with particular observations. For example, the subset of cosmologists interested specifically in Saturn might instantiate a test by randomly chunking observations made about Saturn's nightly position over time as follows:

```
h, p = randomly_chunk(saturn_obs_positions)
saturn_position_test = PositionTest(
  {'histories': h, 'positions': p})
```

The class `PositionTest` defines logic that is not specific to any particular planet, so it is contained in a repository shared by all cosmologists called `cosmounit`. The particular test above would be constructed in a separate repository focused specifically on Saturn called `saturnsuite`. Both would be collaboratively developed by these (overlapping) research communities in source code repositories on Github.

Classes that implement the `sciunit.Test` interface must contain a `generate_prediction` method that takes a *model* as input and extracts a prediction. To specify the interface between the test and the model, the test author specifies a list of *required capabilities* in the `required_capabilities`, seen on line 7. A capability specifies a collection of methods that a test will need to invoke in order to receive relevant data, and is analogous to an *interface* in a language like Java. In Python, interfaces are written as classes with unimplemented members. The capability required in Figure 2 is shown in Figure 3. Classes defining capabilities are tagged as such by inheriting from `sciunit.Capability`. The test in Figure 2 repeatedly uses this capability on lines 9-11 to return a list of predicted positions for each observed history (using a Python list comprehension for brevity). A planet's position is represented using a standard celestial coordinate

```
1  class PredictsPlanetaryPosition(sciunit.Capability):
2    def predict_next_pos(self, history):
3      """Takes a list of previous Positions and produces
           the next Position."""
4      raise NotImplementedError()
5
6  class LinearPlanetModel(sciunit.Model,
7      PredictsPlanetaryPosition):
8    def predict_next_pos(self, history):
9      return linear_prediction(history)
```

Figure 3: An example capability and a model class that implements it in `cosmounit`.

system specified within `cosmounit` by the class `Position`. A model class can implement a capability by simply inheriting from it and implementing the required methods (Figure 3). A particular model is an instance of such a class. The scope of a model is identified by the capabilities it implements.

```
lin_saturn_model = LinearPlanetModel()
```

Once a prediction has been extracted from a model, the test class must compute a *score*. The framework invokes the `compute_score` method with the observation provided upon test instantiation and the prediction just generated (line 14 of Fig. 2). Here, the test class constructs a list of distances between the observed and predicted positions, then determines a pooled p-value to determine the goodness-of-fit (the details are omitted for concision). A p-value is represented as an instance of `sciunit.PValue`, a subclass of `sciunit.Score` that has been included with SciUnit due to its wide use across science.

This illustrates a key difference between unit testing, which would simply produce a boolean result, and our conception of scientific validation testing (challenge 1). A score class must induce an ordering, so that a table like that shown in Figure 1 can be sorted along its columns, and it can specify a normalization scheme so the cells can be color-coded. Predictions are extracted by a separate method to make it easier for statisticians to write new tests for which only the goodness-of-fit metric differs, not the method by which the prediction is extracted (challenge 5).

Te `judge` method of a test can be invoked to compute a score for a single model:

```
score = saturn_position_test.judge(lin_saturn_model)
```

This method proceeds by first checking that the provided model implements all required capabilities before calling the `generate_prediction` method followed immediately by the `compute_score` method. A reference to the test, model, observation, prediction and other related data the test provides (not shown) are available as attributes of the score.

To produce a more comprehensive test suite, the contributors to `saturnsuite` would instantiate a number of other tests and then create an instance of the `TestSuite` class:

```
saturn_motion_suite = sciunit.TestSuite([
  saturn_position_test, saturn_velocity_test, ...])
```

They would also instantiate a number of models. A test suite can be used to judge multiple models at once (challenge 2) if they satisfy the union of the capabilities required by the constituent tests. The result is a *score matrix*:

```
sm_matrix = saturn_motion.judge([lin_copernicus_sat,
    ptolemy_sat, newton_sat, kepler_sat])
```

If constructed inside an iPython notebook, a score matrix can then be visualized as an interactive table, much like the one shown in Figure 1. Scientists can sort by column and click on tests, models and scores to get more information on each (not shown, implementation in progress).

3 Collaborative Workflow

A model that performs well across tests in such a suite could reasonably claim (e.g. to reviewers) to be a coherent, valid model of Saturn's motion. As new data is collected, new tests can be added to the suite or existing tests can be refined. As long as the interface between the test and the model remains the same, the table can be updated entirely automatically. Similarly, when a new model is developed, it can immediately be evaluated against all known data that has been encoded as a test by simply exposing its predictions via the capabilities the test requires. If better statistical tests are developed, they too can be used to regenerate the table automatically, as discussed above.

The design just described has been purposefully left as simple as possible, in pursuit of challenge 4. Despite its simple design, it captures the essential elements of the scientific model validation process and satisfies the four challenges we laid out. SciUnit can be used by individual scientists to organize their workflows, but because science, and in particular, model validation is a collaborative process, we have also described the intended use of SciUnit within a collaborative workflow, mediated by a social coding tool like Github.

More directly: we anticipate common testing logic (e.g. `PositionTest`), modeling logic and capabilities being collaboratively developed by larger communities in less specialized repositories like `cosmounit`. Individuals and small groups will then parameterize these tests and models with data they have gathered, as well as data known from the literature and data contained in existing data repositories to create test suites in repositories like `saturnsuite`. The interface between data collection software and existing data representation standards and tests will be mediated by bridge logic also contained in repositories like `cosmounit`.

Statisticians wishing to promote new validation metrics can simply for a *X*`unit` repository and implement new test logic. By reusing the same interfaces as previous tests used, these new metrics can immediately be used to validate or invalidate a large body of existing models, providing valuable data for use in convincing the community to adopt these into the mainstream repositories. Similarly, investigators who wish to de-emphasize or emphasize different quantities can fork *X*`suite` repository and add or remove tests.

To help organize these repositories, a simple collaborative portal sitting above Github called *SciDash* is currently under development (*http://scidash.org/*). SciDash consists of an organized, collaboratively filtered listing of *X*`unit` and *X*`suite` repositories. That is, it is a tool to help scientists find the most popular repositories in their research area, to facilitate the development of community standards. SciDash also supports extracting summaries of tests, models, scores and related data from suite repositories to generate hyperlinked score tables and documentation. This facilitates exploratory workflows. To modify a suite (e.g. by adding the model a scientist is developing to it), a scientist can simply fork the repository. SciDash automatically discovers public forks of repositories that it is already indexing.

4 Discussion

Academic peer review and code review are similar, but software developers typically supplement human review with test-driven methodologies. Such methodologies may benefit scientific communities as well, but several unique constraints make reusing existing testing frameworks and processes difficult. We describe a core testing framework that addresses these issues by using an existing, widely-adopted language and designing a flexible, simple framework that captures the domain-specific structure of scientific model validation. We then describe, by example, how the various components can be organized into software repositories and collaboratively maintained using social coding tools to support existing scientific practices. We end by outlining a collaboratively filtered portal that sits atop these tools and facilitates exploratory analyses and repository discovery.

While we discuss a toy example based on planetary movement here, we have applied this framework to more realistic problems in neurobiology in a project called NeuronUnit (http://github.com/scidash/neuronunit). We have used this as part of a collaboration with a large-scale neuroinformatics project (http://www.opensourcebrain.org), whose developers are enthusiastic about a practical pipeline for testing the dozens of models they currently maintain. Much future work remains to be done to investigate whether these tools are truly usable and useful to scientific communities, to further develop the community workflow and infrastructure and to develop a range of realistic case studies. Nevertheless, we believe that identifying the synergies between testing practices and model validation, and describing the basic tooling, represents a novel contribution to the literature on software testing. The core SciUnit framework has been fully developed and is available at http://sciunit.scidash.org/. SciDash is under active development, and is currently capable of basic forms of the operations described in Sec. 3.

We also plan to use SciUnit to implement competitions among classes of models or techniques in particular subfields. Typically such competitions use *ad hoc* infrastructure. A common framework could be used to make designing and continuously operating such contests more feasible.

Acknowledgements

We thank Sharon Crook, Shreejoy Tripathy and Padraig Gleeson for their helpful discussions. The work was supported in part by grant R01MH081905 from the National Institute of Mental Health. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

5 References

[1] K. Beck. *Test Driven Development: By Example.* Addison Wesley, 2003.

[2] G. E. Box and N. R. Draper. *Empirical model-building and response surfaces.* John Wiley & Sons, 1987.

[3] J. Howison and J. Herbsleb. Scientific software production: incentives and collaboration. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 513–522. ACM, 2011.

[4] A. E. Jinha. Article 50 million: an estimate of the number of scholarly articles in existence. *Learned Publishing*, 23(3):258–263, July 2010.

[5] C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, 2009.

[6] M. F. Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[7] J. Segal. Models of scientific software development. In *SECSE 08, First International Workshop on Software*

Engineering in Computational Science and Engineering, 13 May 2008, Leipzig, Germany., May 2008.

[8] H. Siy and L. Votta. Does the modern code inspection have value? In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 281–289, 2001.