Using Pre-Oracled Data in Model-Based Testing

Harry Robinson
Microsoft

This document describes what is meant by “pre-oracled” data and how Semantic Test proposes to use it in testing search functions.

Some Background on Testing and Oracles

Simply stated, behavioral testing involves providing inputs to the application under test and observing the application’s outputs. We examine the outputs to verify that the application is behaving correctly. The term “test oracle,” or simply “oracle,” describes how we determine if the output we observed was correct. By “pre-oracled” data, we mean that our test engine already knows the answer we expect to receive based on the test data. (Note that “pre-oracled” is a term unique to our group and neither the term nor the concept appears in the literature.)

Some oracles are very simple. For instance, if the application crashes, that is almost always a bad thing. But checking for crashes is not enough. Most oracles need to be more sophisticated than just detecting crashes. We want some assurance that the application actually did what we expected.

When humans test, they are usually their own oracles; i.e., they typically have some idea of what they consider to be “good” application behavior. For instance, if I am editing a Word document that contains the string “spaghetti”, I will expect to find that word when I do a search of the document. Likewise, if I have deleted all instances of “spaghetti” from the document, I will expect the search to come up empty.

In short, people create their own mental model of how the application will behave. This model is based on

1. What they know how the application should behave, and
2. What they know about the data being processed.

In the Word example above, the tester knows (1) how the “Find” function of Word should behave, and (2) whether the string “spaghetti” is present in the document.

The Need for Models

One significant problem in testing is that people’s models tend to remain in their heads and are not written down. When people typically write their tests, they don’t write down the model that generated the tests, they only write down the actions to perform. This makes those tests static and hard to adapt to new situations.

For instance, if I have a file that contains the string “spaghetti”, my mental model tells me to expect a search for “spaghetti” to be successful. But if I only record my actions, my test will always search for the string “spaghetti”. To keep the result successful, I will need to ensure that “spaghetti” is always present in the file, usually by keeping the “spaghetti” document around.

Recording the actions rather than the model leads to a "hardening" of the tests. Because the oracle for this test has been preserved in the test actions and the data files, the test scripts become frozen into always testing for the same words in the same files. These tests become less useful as time goes on because they have already found the bugs they were intended to find. Also, the test framework is burdened with maintaining the data sets, such as the Word documents, that the tests were run on.

Modeling a Search Function

Our work in Semantic Test proposes using a model of the application's expected behavior to generate tests. For instance, suppose we are testing a search function's ability to locate documents that contain combinations of words. In the following example, we use a database to hold our simple model. The database associates document names with the words they contain. (Note that, for simplicity, this model does not care about the exact number of times nor where in the documents the words appear.)

Document	Spaghetti	Monkey	Klingon
File1	Yes	No	No
File2	Yes	Yes	No
File3	No	Yes	No
File4	No	No	No

If our model correctly portrays how these words are distributed throughout the documents, we can generate interesting tests. And querying the model in the database provides us with an automatic oracle for the output.

Example 1: Find documents that contain 'monkey'. The oracle for this test is provided by a query on the database model: "SELECT Document … WHERE Monkey = 'Yes' ". The model (in the database) and the search function should both answer with File2 and File3.

Example 2: Find documents that contain 'spaghetti' and 'monkey' is modeled by "SELECT Document … WHERE Spaghetti = 'Yes' AND Monkey = 'Yes' ". The model and the search function should both answer with File2.

Example 3: Find documents that contain "Klingon" is modeled by "SELECT Document … WHERE Klingon = 'Yes' ". The model and the search function should both reply that no such document was found.

And so on.

Growing a Model

One advantage of using models is that they can quickly provide rapid (though simple) testing of the application. And as the application and the tests evolve, it is straightforward to provide additional data to the model.

For instance, the model might eventually incorporate

- How many times the searched term appears in the file. This can affect ranking of results. For instance, many search engines rank documents higher when the search term appears more than once, but some search engines disqualify documents where the search term appears too many times.

- The position of search terms in the document. Some search engines index only the first several hundred words in a document.

- Whether multiple search terms are close to each other in the document. Close proximity of terms usually gives the document a high ranking.

- Whether the term is a significant part of the document, such as the title.

The Primacy of the Model

One of the most interesting facets of this modeling approach is that all we really care about is the model. We don't care how the data was put into the model. We care that the model is reasonably correct for our uses and that it contains the information we need to know about how the application will behave.

1. We can generate the model and the documents ourselves automatically,

2. We can fill in the model with the interesting characteristics of pre-existing documents, or

3. We can do some combination of the above.

All of these approaches offer benefits.

1. If we generate both model and documents ourselves, we are able to create test scenarios that might not exist in available documents. For instance, this would allow us to search for an extremely long word such as 'sesquipedalian' that might not be present in our corpus of documents. Further, these test scenarios can change and evolve as testing progresses, which is difficult with a real corpus.

2. If we gather data for the model from pre-existing documents, we have the assurance that we are testing against documents that resemble those the search function is likely to encounter in deployment. Note that this approach does not mean that we are re-implementing what development is already doing. Because the tests do not need to run in real-time, we can choose a simple and thorough algorithm for gathering our data. It doesn't matter if that algorithm is slow; we are more concerned with correctness than with speed.

3. Between these two extremes lie any number of interesting mixes that can give us different degrees of useful test scenarios and reasonable-looking documents. The Semantic Test team is currently investigating and implementing several of these approaches to discover which gives us the greatest return on our testing investment.