daps

Programmer and Modeler Documentation for "downstream" prescient classes and

Documentation for "downstream" prescient classes and out-of-the-box functions

David L. Woodruff
Graduate School of Management, University of California.
Davis, CA 95616-8609, USA
dlwoodruff@ucdavis.edu

February 21, 2017

1 Introduction

This document is intended to be used by modelers who must produce *scenario tree tem*plates that specify stages, Var assignments to stages, and cost expressions; and programmers who will extend the daps classes, and implement *upstream* classes for stochastic process models and raw scenario creation.

I view the processing as starting with data and ending with inputs for PySP. In this document I sometimes refer to the software for doing this as *daps*; I am referring to the subject of this design work.

My design goal for December 2016 is to have good classes for the *later* steps in the processing so that folks working on the earlier steps (e.g. regression or error distribution fitting, sampling, etc.) have something to plug into that others can pick up. I view these later steps as beginning with what I call raw scenario tree node data.

Here are the goals for this part of the design work (the so-called "downstream" classes for "later steps."):

- 1. Support multi-stage fully from the start.
- 2. Support all known concrete and abstract use-cases.
- 3. Support "raw" scenario data that could be read by non-Python programs.
- 4. Provide a lot of flexibility.
- 5. Allow for easy use of data directly by deterministic models; particularly concrete models.

2 Files and Examples

I realize that some programmers and modellers would prefer not to read this document, but would rather look at examples. I now describe some of the files, including those with examples; where appropriate, I have tried to provide hints about what directory they might be in. Note that many of my examples use JSON, which is simply because it is easy to supply structured data that way. There is no particular reason to think that other programmers and modellers will favor that.

2.1 testdaps.py

This python script provides calls to "out-of-the-box" functions to illustrate their use. In a real application, typically only one of the calls in this little script would be used. The calls to runef are added to verify that the script worked. This is intended to be run from the top directory of daps (i.e., the directory that contains the directories named farmer and concrete_farmer.)

2.2 basicclasses.py

This has the furthest downstream class definitions and at the bottom of the file there are out-of-the-box functions to illustrate their use (and provide some out-of-the-box capabilities). These classes are where the "standard" daps dictionary is used to define the data for a "raw" node (particularly in Raw_Node_Data, which is the most important class in this file) and a raw scenario. These dictionaries sort of mimic Param. The outer indexes could be Param names. For singleton Params, the value is the value. For indexed Params, the value is a dictionary indexed by the full index values of the Param. Note that is hoped that concrete models can make use of these dictionaries directly, but when not, then the programmer must derive classes with writer methods to do the right thing.

Cryptic notes concerning the word "raw"

We use the term "raw" to distinguish data that may not be fully ready for use by Pyomo or PySP; however, some concrete models may, in fact, be able to process this data directly. Usually, though, at the very least raw scenario needs to be processed by something so that the scenario tree data structures know about it.

2.3 distrs.py

These are classes to support probability distributions and at the bottom of the file there are out-of-the-box functions to illustrate their use (and provide some out-of-the-box capabilities). In addition to what you might expect in the form of distribution classes, it includes support for the special notion of a DictDistr where distribution objects are the values in a "standard" daps node dictionary as introduced in basicclasses.py. It has a derived class of this type for the scipy distributions.

2.4 distr2pysp.py

This is the furthest "upstream" code provided and illustrate how one might get from models or data to the downstream classes for some particular cases: namely two stage farmer models sampling from scipy distributions. The parameters of the distributions can be supplied either directly (from a json file) or from data (with files names specified in json files). The classes in this file could be used directly, of course, but mainly they are intended to illustrate the use of the distr classes and those in basicclasses.py

3 Modeler Tasks for out-of-the-box functions

The modelers must do a little less with this system than they ordinarily would do to use PySP, but there may be some file naming requirements in order to use out-of-the-box functions supplied with prescient.

3.1 The Model

As usual, the modeler must provide a model file that is often called ReferenceModel.py and it must contain cost Expressions for the stages.

3.2 Tree Template

Although there are other ways to specify a scenario tree, this software presently makes use of the AMPL format data file that is often called ScenarioStructure.dat but the entire file need not be supplied by the modeler. The modeler must supply the stage names, the assignment of Vars to stages and the names of the cost expressions for each stage. If more data is supplied it will be left in the final ScenarioStructure.dat file and the new data computed by prescient will be appended. This is less safe than providing only the minimum and relies on the fact that AMPL format data overwrites as it reads through the file.

Here is a sample tree template for the farmer example:

3.3 Scenario Data

There is a very general and flexible system for dealing with scenario data, but in the subsection we begin by looking at some specific examples and discuss out-of-the-box treatment. See testdaps.py for some examples of calls to the out-of-the-box functions.

3.3.1 Two-stage Concrete json

To use an out-of-the box function, supply the data for each scenario in a file with a special name that is composed from some literal tokens, dashes, the scenario name, and the scenario

probability. The filename extension should reflect the data type, e.g. json. If the scenario name happens to be 1 (not very creative) and the scenario has a probability 0.25, then the filename for a json file should be NODE-1-PARENT-ROOT-PROB-0.25.json and the python model file (e.g., ReferenceModel.py) should have a callback that uses either this filename or a simpler filename that the prescient function copies the data into as a side-effect. This filename would 1.json (so one maybe just using integers as scenario or node names is not a great idea, but that is your choice).

Consider the farmer example that has three scenarios with names Average, AboveAverage and BelowAverage each of which sets the value of the mutable Param called Yield. Here is a reasonable callback function for ReferenceModel.py (or whatever you name your deterministic Pyomo model file).

```
def pysp_instance_creation_callback(scenario_name, node_names):
    """ Read the Yield data from a JSON file with a file
    that is scenario_name + ".json"
    The object should be named Yield (to play by the rules of prescient,
    which wants a dictionary, perhaps of dictionaries).
    """
    # dec 2016: uncool literal directory name...
    fname = "concrete_farmer/" + scenario_name + ".json"
    print ("in callback fname=",fname)
    with open(fname, "r") as scenfile:
        YieldIn = json.load(scenfile)

instance = model.clone()
    instance.Yield.store_values(YieldIn['Yield'])

return instance
```

Here are the contents of one of the three raw scenario data files (namely NODE-AboveAverage-PARENT-RO

```
{"Yield": {"CORN": 3.6, "WHEAT": 3.0, "SUGAR_BEETS": 24.0}}
```

The idea is that these raw scenario node data files should provide data for a dictionary with indexes that are Param names (or whatever might be needed by a concrete model) and values that either the data value or dictionaries (as in this case) where the inner indexes are the indexes for the Param (or whatever the concrete model needs). Note that in the example of farmer, there is only one outer dictionary index because only one Param is stochastic.

3.3.2 Two Stage AMPL with Scenario Template without tokens

The tree template file must be supplied, but a scenario template is optional. The template file functions more-or-less like a root node data file because it is the starting point for all scenarios. Here is the file farmer/ScenTemplate.dat

```
set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;
```

The raw node data file for a particular farmer scenario is very short, such as this ssv file:

```
Yield; WHEAT; 3.0
Yield; CORN; 3.6
Yield; SUGAR_BEETS; 24
```

The raw scenario data file could also be a json file as shown for the concrete model. The use of AMPL format for the raw node data could be supported, but it is not supported as of Dec 2016 and there does not seem to be a good reason to support it (unlike for the template).

3.3.3 Two Stage AMPL with Scenario Template without tokens

The scenario template could also be a full scenario file with tokens that mark the part that should be removed an replaced for each scenarios. Here is an example of a file with tokens that are used in the testdaps.py script.

```
set CROPS := WHEAT CORN SUGAR_BEETS ;
param TOTAL_ACREAGE := 500 ;
# no quotas on wheat or corn
param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;
param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;
param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;
```

```
param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;
# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;
#STARTSCEN
param Yield:= CORN 3 ;
SUGAR_BEETS 20 ;
WHEAT 2.5 ;
#ENDSCEN
```

4 Implementation of downstream classes

This section is directly from the design document and discuses the classes that underlie the out-of-the-box examples above. Programmers may wish to extend these classes and/or use them to create new out-of-the-box functions.

The basic design is coded in basic lasses.py as "base" classes. To provide some out-of-the-box capabilities, these classes have some constructors and services implemented. Also, there are some functions at the bottom of the py file that support various common, yet special, cases.

4.1 Key Design Feature

At the center of the design are the concepts of raw scenario tree node data and scenario template data. There is also the notion of a scenario tree template that names the stages (given in order), assigns Vars to stages, and names the cost expressions for the stages.

4.1.1 Scenario template data

This is a flexible concept that might contain a full scenario, might contain root node data only, or might be empty. It is the starting point for every scenario and when the scenario is created, data for the scenario will be added to the template and/or replace template data. The template could be in a variety of formats, such as an AMPL format data file, a JSON file or a csv file, etc. The details of how it gets used depends on the format.

4.1.2 Raw Scenario Tree Node Data

These data need to find their way into a dictionary whose elements are either data or dictionaries. For abstract models using AMPL format input files, the dictionary names are intended to be Param names. For concrete models (and for use by abstract model build actions) they are intended to be whatever they should be.

If the elements are a dictionary, the indexes in the inner dictionary are intended to be the entire index string for a Param or else whatever they should be.

When these data are supplied in a file, I want a lot of flexibility in the format and I want to be sure that packages written in other languages would have a shot at using them if they wanted to. Furthermore, I want to be sure the for two-stage concrete models they can be used directly as deterministic input. Hence, I propose coding the scenario tree in the file names:

NODE-name-PARENT-name-PROB-prob.ext

where the upper case tokens are literals and must be present and ext signals the type of data in the file (e.g., json). The special node name ROOT is assumed to exist unless a file is present with the special parent name NONE, which then defines the root node (I think it will be unusual for a raw root node data file to be created, but that will be an option). IMPORTANT: prob is the conditional probability.

There should be almost nothing stopping this thing from having a full scenario, but it is only responsible for being correct up to ts node. When a scenario is assembled it will be done in stage order.

4.2 Use Cases (Upstream)

This section envisions how developers would create upstream software to feed the nodal raw scenario data to the downstream system that ultimately produces a ScenarioStructure.dat file and provides scenario data.

4.2.1 Two-stage stochastics for IDAES Flow Sheets

Bottom line

Supply the scenarios (perhaps in files with special names) and a ScenarioStructure.dat template that has the stage names, Var assignments to stages and the cost expression names.

Discussion

These are concrete models, so the callback in ReferenceModel.py needs to pick up whatever data it needs. Let's assume it is going get that data from a JSON file, just for fun.

Since this is two-stage, all nodes are leaf nodes so the "raw" scenario data (which, in this case, is the scenario data) will be in files with names like

NODE-1-PARENT-ROOT-PROB-0.27.json

assuming the scenarios have uncreative names that are integers (1 in the this example).

The daps system does not need to do anything to these files other than look a their names (collectively) so it can append the tree structure information to the ScenarioStructure.dat file. I suppose it might need to rename them if the callback cannot live with these names.

It would be necessary for the modeler to supply a template for the ScenarioStructure.dat file, which could be just a file for a test instance (it needs to name the stages, assign Vars to stages, and name the cost expressions). No template would be needed for the scenario data, unless the modeler wanted to supply one for some reason.

So the upstream programmer needs to write code to create the scenario files and name them appropriately. The modeler needs to create the scenario template.

4.2.2 Two-stage stochastics for UC experiments

These are abstract models that use AMPL format data for scenarios, a template for the scenario data files must be provided. It could just be a deterministic data file or an arbitrary scenario data file. This supplies the root node such as generator characteristics.

A template is also needed for the ScenarioStructure.dat file, which needs to indicate how the Vars are assigned to stages (it could be a complete file).

Currently, our code produces special format csv files, so we could modify it to move the probability data to the file name and leave the scenario data in the file in the univariate case.

In the multivariate case, this is a bit of change for us. Currently we produce univariate data files from marginals and then combine them in pyspgen.py with a straight cross. One of the points of this whole exercise is to make that go away. Now the upstream programmer needs to cross the sources of uncertainty and/or draw scenario data from a multi-variate distribution.

4.2.3 Multi-stage

The tree template (e.g., a template ScenarioStructure.dat file) defines the stage names, assigns Vars to stages and assigns cost expressions to stages. Whatever writes the raw node data files needs to know what tree it is trying to make because it needs to define the file names so as to create the tree. The prescient software then figures out the tree from the collection of file names.

4.3 Note about root node data files

There is no requirement to write a root node file, unless the modeler wanted to do things that way (one could use a root node raw data file, or one could specify template scenario data that has the root node data, or root node data could be "hard-coded" in the ReferenceModel.py file or read from another source by the ReferenceModel.py file).

5 Notes

6 UML

See daps_UML.pdf for simple UML-like figures that can help clarify some of the class relationships.