UrbanSim Templates Documentation

Release 0.2.dev9

UDST

May 16, 2020

Contents

UrbanSim Templates provides building blocks for Orca-based simulation models. It's part of the Urban Data Science Toolkit (UDST).

The library contains templates for common types of model steps, plus a tool called ModelManager that runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions.

v0.2.dev9, released July 22, 2019

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Intro

UrbanSim Templates is a Python library that provides building blocks for Orca-based simulation models. It's part of the Urban Data Science Toolkit (UDST).

The library contains templates for common types of model steps, plus a tool called ModelManager that runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions. The package was developed to make it easier to set up new simulation models — model step templates reduce the need for custom code and make settings more portable between models.

UrbanSim Templates is hosted on Github with a BSD 3-Clause open source license. The code repository includes some material not found in this documentation: a change log, a contributor's guide, and instructions for running the tests, updating the documentation, and creating a new release.

Another useful resource is the issues and pull requests on Github, which include detailed feature proposals and other discussions.

UrbanSim Templates was created in 2018 by Sam Maurer (maurer@urbansim.com), who remains the lead developer, with contributions from Paul Waddell, Max Gardner, Eddie Janowicz, Arezoo Besharati Zadeh, Xavier Gitiaux, and others.

1.1.2 Installation

UrbanSim Templates is tested with Python versions 2.7, 3.5, 3.6, and 3.7.

As of Feb. 2019, there is an installation problem in Python 3.7 when using Pip (because of an issue with Orca's PyTables dependency). Conda should work.

Note: It can be helpful to set up a dedicated Python environment for each project you work on. This lets you use a stable and replicable set of libraries that won't be affected by other projects. Here are some good environment settings for UrbanSim Templates projects.

Production releases

UrbanSim Templates can be installed using the Pip or Conda package managers. With Conda, you (currently) need to install UrbanSim separately; Pip will handle this automatically.

```
pip install urbansim_templates
```

```
conda install urbansim_templates --channel conda-forge
conda install urbansim --channel udst
```

Dependencies include NumPy, Pandas, and Statsmodels, plus two other UDST libraries: Orca and ChoiceModels. These will be included automatically when you install UrbanSim Templates.

Certain less-commonly-used templates require additional packages: currently, PyLogit and Scikit-learn. You'll need to install these manually to use the associated templates.

When new production releases of UrbanSim Templates come out, you can upgrade like this:

```
pip install urbansim_templates --upgrade
```

```
conda update urbansim_templates --channel conda-forge
```

Developer pre-releases

Developer pre-releases of UrbanSim Templates can be installed using the Github URL. These versions sometimes require having a developer release of ChoiceModels as well. Information about the developer releases can be found in Github pull requests.

```
pip install git+git://github.com/udst/choicemodels.git
pip install git+git://github.com/udst/urbansim_templates.git
```

You can use the same command to upgrade.

Cloning the repository

If you'll be modifying the code, you can install UrbanSim Templates by cloning the Github repository:

```
git clone https://github.com/udst/urbansim_templates.git
cd urbansim_templates
python setup.py develop
```

Update it with `git pull`.

1.1.3 Basic usage

Initializing ModelManager

To get started, import and initialize ModelManager. This makes sure there's a directory set up to store any template-based model steps that are generated within the script or notebook.

```
from urbansim_templates import modelmanager

modelmanager.initialize()
```

The default file location is a `configs` folder located in the current working directory; you can provide an alternate path if needed. If ModelManager finds existing saved objects in the directory, it will load them and register them with Orca.

Note: It can be helpful to add a cell to your notebook that reports which version of UrbanSim Templates is installed, particularly if you're using development releases!

```
In [2]: import urbansim_templates
        print(urbansim_templates.__version__)

Out[2]: '0.2.dev0'
```

Creating a model step

Now we can choose a template and use it to build a model step. The templates are Python classes that contain logic for setting up and running different kinds of model logic — currently focusing on OLS regressions and discrete choice models.

A template takes a variety of arguments, which can either be passed as parameters or set as object properties after an instance of the template is created.

```
from urbansim_templates.models import OLSRegressionStep

m = OLSRegressionStep()
m.name = 'price-prediction'
m.tables = 'buildings'
m.model_expression = 'sale_price ~ residential_sqft'
```

This sets up `m` as an instance of the OLS regression template. The `tables` and `model_expression` arguments refer to data that needs to be registered separately with Orca. So let's load the data before trying to estimate the model:

```
import orca
import pandas as pd

url = "https://www.dropbox.com/s/vxg5pdfzxrh6osz/buildings-demo.csv?dl=1"
df = pd.read_csv(url).dropna()
orca.add_table('buildings', df)
```

Fitting the statistical model

Now we can fit the building price model:

```
m.fit()
```

This will print a summary table describing the estimation results.

Now that we have a fitted model, we can use it to predict sale prices for other buildings. UrbanSim forecasting models consist of many interconnected steps like this, iteratively predicting real estate prices, household moves, construction, and other urban dynamics.

Registering the step

Now we can register the model step:

```
modelmanager.register(m)
```

ModelManager parses the step, saves a copy to disk, and registers a runnable version of it as a standard Orca step, so that it can be invoked as part of a sequence of other steps:

```
orca.run(['price-prediction', 'household-moves', 'residential-development'])
```

In real usage, some additional parameters would be set to specify which data to use for prediction, and where to store the output.

Making changes

ModelManager also includes some interactive functionality. Previously registered steps can be retrieved as template objects, which can be modified and re-registered as needed. This also works with model steps loaded from disk.

```
modelmanager.list_steps()

m2 = modelmanager.get_step('price-prediction')
...

m2.name = 'better-price-prediction'
modelmanager.register(m2)
modelmanager.remove_step('price-prediction')
```

If you take a look in the `configs` folder, you'll see a yaml file representing the saved model step. It includes the settings we provided, plus the fitted coefficients and anything else generated by the internal logic of the template.

1.2 ModelManager API

ModelManager runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions.

The recommended way to load ModelManager is like this:

```
from urbansim_templates import modelmanager

modelmanager.initialize()
```

1.2.1 Core operations

1.2.2 Internal functionality

These functions are the building blocks of ModelManager. You probably won't need to use them directly, but they could be useful for debugging or for extending ModelManager's functionality.

1.3 Model step template APIs

The following templates are included in the core package. ModelManager can also work with templates defined elsewhere, as long as they follow the specifications described in the design guidelines.

1.3.1 OLS Regression

1.3.2 Binary Logit

1.3.3 Small Multinomial Logit

1.3.4 Large Multinomial Logit

1.3.5 Segmented Large Multinomial Logit

1.3.6 Template Step parent class

1.4 Data management templates

1.4.1 Usage

Data templates help you load tables into Orca, create columns of derived data, or save tables or subsets of tables to disk.

```
from urbansim_templates.data import LoadTable

t = LoadTable()
t.table = 'buildings'  # a name for the Orca table
t.source_type = 'csv'
t.path = 'buildings.csv'
t.csv_index_cols = 'building_id'
t.name = 'load_buildings'  # a name for the model step that sets up the table
```

You can run this directly using `t.run()`, or register the configured template to be part of a larger workflow:

```
from urbansim_templates import modelmanager

modelmanager.register(t)
```

Registration does two things: (a) it saves the configured template to disk as a yaml file, and (b) it creates a model step with logic for loading the table. Running the model step is equivalent to running the configured template object:

```
t.run()

# equivalent:
import orca
orca.run(['load_buildings'])
```

Strictly speaking, running the model step doesn't load the data, it just sets up an Orca table with instructions for loading the data when it's needed. (This is called lazy evaluation.)

```
orca.run(['load_buildings'])  # now an Orca table named 'buildings' is registered

orca.get_table('buildings').to_frame()  # now the data is read from disk
```

Because "running" the table-loading step is costless, it's done automatically when you register a configured template. It's also done automatically when you initialize a ModelManager session and table-loading configs are read from yaml. (If you'd like to disable this for a particular table, you can set `t.autorun == False`.)

Recommended data schemas

The `LoadTable` template will work with any data that can be loaded into a Pandas DataFrame. But we highly recommend following stricter data schema rules:

1. Each table should include a unique, named index column (a.k.a. primary key) or set of columns (multi-index, a.k.a composite key).
2. If a column is meant to be a join key for another table, it should have the same name as the index of that table.
3. Duplication of column names across tables (except for the join keys) is discouraged, for clarity.

If you follow these rules, tables can be automatically merged on the fly, for example to assemble estimation data or calculate indicators.

You can use `validate_table()` or `validate_all_tables()` to check whether these expectations are met. When templates merge tables on the fly, they use `merge_tables()`.

These utility functions work with any Orca table that meets the schema expectations, whether or not it was created with a template.

Compatibility with Orca

From Orca's perspective, tables set up using the `LoadTable` template are equivalent to tables that are registered using `orca.add_table()` or the `@orca.table` decorator. Technically, they are `orca.TableFuncWrapper` objects.

Unlike the templates, Orca relies on user-specified "broadcast" relationships to perform automatic merging of tables. `LoadTable` does not register any broadcasts, because they're not needed if tables follow the schema rules above. So if you use these tables in non-template model steps, you may need to add broadcasts separately.

1.4.2 Data loading API

LoadTable

1.4.3 Column creation API

ColumnFromExpression
ExpressionSettings

1.4.4 Data output API

SaveTable

1.5 Shared utilities

The utilities are mainly helper functions for templates.

1.5.1 General template tools API

CoreTemplateSettings

1.5.2 Column output tools API

OutputColumnSettings
register_column

1.5.3 Table schemas and merging API

validate_table
validate_all_tables
merge_tables

1.5.4 Other helper functions API

all_cols
cols_in_expression
get_data
get_df
trim_cols
to_list
update_column
update_name

1.5.5 Spec validation API

validate_template

1.5.6 Version management API

parse_version
version_greater_or_equal

1.6 Development guide

Below are some strategies we've come up with for the templates. Technical contribution guidelines are in the Github repo.

1.6.1 Design patterns for templates

A ModelManager-compliant template is a Python class that conforms to the following spec:

1. can save itself to a dict using a method named `to_dict()`
2. can rebuild itself from a dict using a method named `from_dict()`
3. can execute a configured version of itself using a method named `run()`
4. accepts parameters `name` (str) and `tags` (list of str)
5. uses the `@modelmanager.template` decorator

Running a configured model step executes logic and typically saves output to Orca.

Templates should try to use parameter names that are consistent or harmonious with other templates.

Tables and columns of data should be input as named Orca objects. Other inputs that are hard to store as strings (like callables) should probably be input as Orca objects as well; we're still working on a solution for this.

All template inputs should be accepted either as constructor parameters or object properties, if feasible:

```
m1 = TemplateStep(foo='yes')
m2 = TemplateStep()
m2.foo = 'yes'
```

It's fine for templates to require interactive configuration, like fitting a statistical model. Also fine to require these actions to be completed before the model step can be saved or run.

Ideally, users should be able to edit object properties and re-run the interactive components whenever they like. Changes will not be saved until a an object is re-registered with ModelManager.

Lightweight intermediate outputs like summary tables and fitted parameters should be saved in an object's dictionary representation if feasible.

Bigger intermediate outputs, like pickled copies of full fitted models, can be automatically stored to disk by providing an entry named `supplemental_objects` in a model's dictionary representation. This should contain a list of dicts, each of which has parameters `name` (str), `content` (obj), and `content_type` (str, e.g. 'pickle').

To avoid dependency bloat, the default installation only includes the dependencies required for core model management and the most commonly used templates. Templates using additional libraries should check whether they're installed before fitting or running a model step, and provide helpful error messages if not.