ROM Framework Documentation

Release 0.1.0

Nicholas Long

Sep 18, 2019

CONTENTS:

This repo will remain for the time being to maintain history, but it is highly recommended to use the newly name repository.

CHAPTER

ONE

REDUCED ORDER MODELING FRAMEWORK

The reduced order model (ROM) framework was created to build models to use for estimating commercial building energy loads. The framework currently supports linear models, random forests, and support vector regressions. The framework handles the building, evalulation, and validation of the models. During each set of the process, the framework exports diagnostic data for the user to evaluate the performance of the reduced order models. In addition to building, evaluating, and validating the reduced order models, the framework is able to load previously persisted model to be used in third-party applications (e.g. Modelica).

The project was initially developed focusing on evaluating ambient loop district heating and cooling systems. As a result, there are several hard coded methods designed to evaluate and validate building energy modeling data. These are planned to be removed and made more generic in the coming months.

This documentation will discuss how to inspect, build, evaluate, and validate a simple dataset focused on commercial building energy consumption. The documentation will also demonstrate how to load and run an already built ROM to be used to approximate building energy loads.

1.1 Instructions

The ROM Framework requires Python 3. After installing Python and configuring Python 3, the ROM Framework can be installed from source code (recommended) or from PyPI.

CHAPTER

TWO

INSTALLATION FROM SOURCE

1) Install Python and pip
2) Clone this repository
3) Install the Python dependencies

```
pip install -r requirements.txt
```

4) (Optional) install graphviz to visualize decision trees
   - OSX: `brew install graphviz`

CHAPTER

THREE

BUILDING EXAMPLE MODELS

A small office example has been included with the source code under the rom/tests directory. The small office includes 3,300 hourly samples of building energy consumption with several characteristics for each sample. The example shown here is only the basics, for further instructions view the complete documentation on readthedocs.

```
./rom-runner build -f rom/tests/smoff_test/metamodels.json -a smoff_test
./rom-runner evaluate -f rom/tests/smoff_test/metamodels.json -a smoff_test
./rom-runner validate -f rom/tests/smoff_test/metamodels.json -a smoff_test
```

CHAPTER

FOUR

INSTALLATION FROM PYPI

Not yet complete.

CHAPTER

FIVE

EXAMPLE REPOSITORY

An example repository was developed using the ROM Framework to evaluate the results of OpenStudio using PAT. There are several repositories to generate the datasets; however, the first link below contains a basic dataset in order to demonstrate the functionality of the ROM Framework.

- Ambient Loop Metamodels

The two repositories below were used to generate the OpenStudio/EnergyPlus models used for the ROM FRamework.

- OpenStudio's Parametric Analysis Tool Projects
- OpenStudio Measures

CHAPTER

SIX

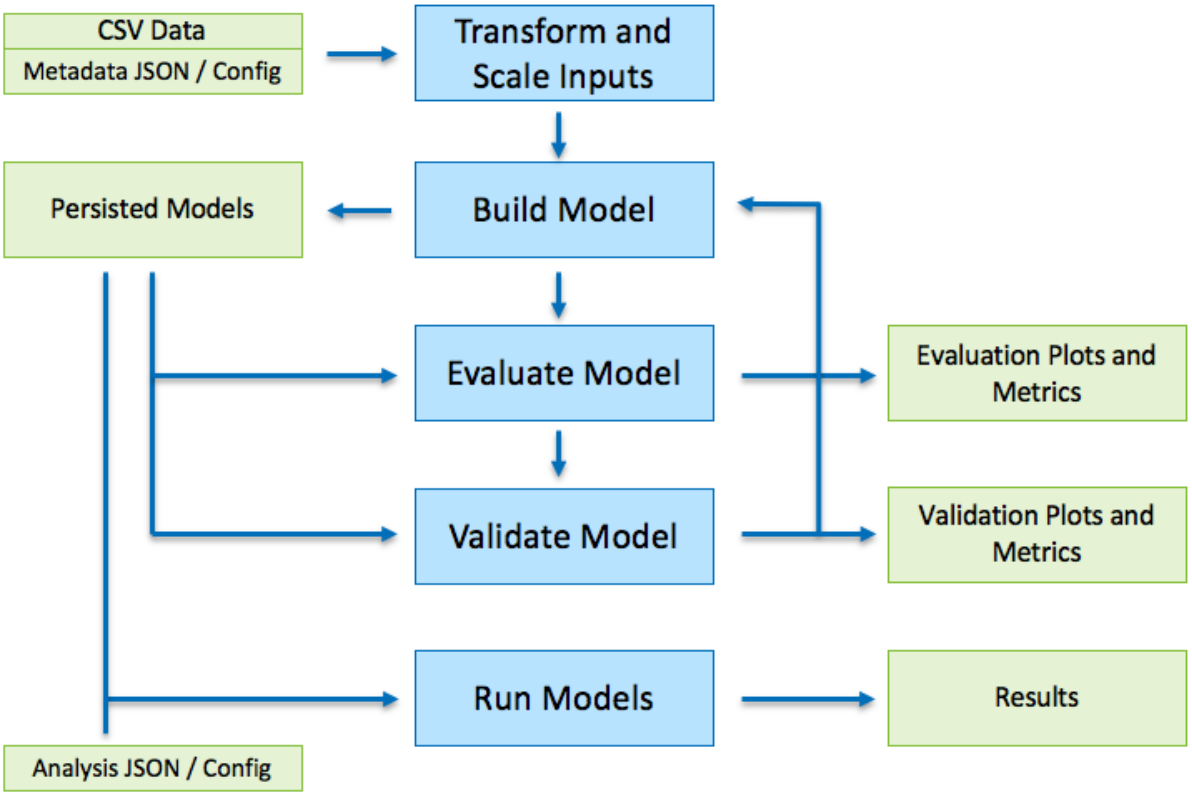
TO DOS

- Configure better CLI
- Allow for CLI to save results in specific location
- Remove downloaded simulation data from repository
- Write test for running the analysis_definition (currently untested!)

6.1 Getting Started

The ROM Framework is designed to help users build, evaluate, validate, and run reduced order models. The image below shows the typical workflow and the required data. Each of the blue boxes represent a process and the green boxes represent either an input dataset or a output data.

In order to run the build method, the user must supply the data in CSV format with an accompanying JSON file which describes a) the build options, b) the response variables, and c) the covariates. An explanation and example of how the metadata JSON config file looks is shown in *example metadata json file*.



The four main functions of the rom-runner.py file includes:

1) Inspect

Load the results dataframe and create a resulting dataframe (and CSV) describing the data. This is useful when determining what is in the dataframe and what the covariates and responses should be. This also calculates the means for all the variables which can be used to set as the default values for when running a parametric sweep with the resulting metamodel that is generated.

Each model that is generated will create the statistics summary in the data directory.

```
-f FILE, --file FILE  Metadata file to use
-a ANALYSIS_MONIKER, --analysis-moniker ANALYSIS_MONIKER
                      Name of the Analysis Model
-m [{LinearModel,RandomForest,SVR}], --model-type [{LinearModel,
↪RandomForest,SVR}]
                      Type of model to build
-d DOWNSAMPLE, --downsample DOWNSAMPLE
                      Specific down sample value
```

```
./rom-runner inspect -a smoff_test
```

2) Build

Use the build positional argument to build a new reduced order model as defined in the metamodels.json file. There are several arguments that can be passed with the build command including:

```
-f FILE, --file FILE  Metadata file to use
-a ANALYSIS_MONIKER, --analysis-moniker ANALYSIS_MONIKER
```

(continues on next page)

(continued from previous page)

```
                        Name of the Analysis Model
-m [{LinearModel,RandomForest,SVR}], --model-type [{LinearModel,
↪RandomForest,SVR}]
                        Type of model to build
-d DOWNSAMPLE, --downsample DOWNSAMPLE
                        Specific down sample value
```

```
./rom-runner build -a smoff_test
```

3) Evaluate

Use the build positional argument to build a new reduced order model as defined in the metamodels.json file. There are several arguments that can be passed with the build command including:

```
-f FILE, --file FILE  Metadata file to use
-a ANALYSIS_MONIKER, --analysis-moniker ANALYSIS_MONIKER
                        Name of the Analysis Model
-m [{LinearModel,RandomForest,SVR}], --model-type [{LinearModel,
↪RandomForest,SVR}]
                        Type of model to build
-d DOWNSAMPLE, --downsample DOWNSAMPLE
                        Specific down sample value
```

```
./rom-runner evaluate -a smoff_test
```

4) Validate

Use the build positional argument to build a new reduced order model as defined in the metamodels.json file. There are several arguments that can be passed with the build command including:

```
-f FILE, --file FILE  Metadata file to use
-a ANALYSIS_MONIKER, --analysis-moniker ANALYSIS_MONIKER
                        Name of the Analysis Model
-m [{LinearModel,RandomForest,SVR}], --model-type [{LinearModel,
↪RandomForest,SVR}]
                        Type of model to build
-d DOWNSAMPLE, --downsample DOWNSAMPLE
                        Specific down sample value
```

```
./rom-runner validate -a smoff_test
```

5) Run

Use the build positional argument to build a new reduced order model as defined in the metamodels.json file. There are several arguments that can be passed with the build command including:

```
-ad ANALYSIS_DEFINITION, --analysis-definition ANALYSIS_DEFINITION
                          Definition of an analysis to run using the ROMs
-w WEATHER, --weather WEATHER
                          Weather file to run analysis-definition
-o OUTPUT, --output OUTPUT
                          File to save the results to
```

```
./rom-runner.py run -a smoff_parametric_sweep -m RandomForest -ad␣
↪examples/smoff-one-year.json -w examples/lib/USA_CO_Golden-NREL.724666_
↪TMY3.epw -d 0.15 -o output.csv
```

6.2 Metadata Definition File

The JSON file shown below is meant to serve as documentation by example for the metamodel definition JSON file.

6.3 Analysis Definition

6.3.1 Static

This example configuration file shows setting all the covariates to a single value.

6.3.2 EPW Data Source

This example configuration file shows reading data from an EPW file for the ROM covariates.

6.3.3 Single

This example configuration file shows sweeping over a single variable range.

6.3.4 Multiple

This example configuration file shows sweeping over multiple variable ranges. In this case the total number of samples will result in the full combinatorial of all the values in this file.

6.3.5 Code

Analysis Definition

Parser for analysis definition JSON files.

class rom.analysis_definition.analysis_definition.**AnalysisDefinition**(*definition_file*)
Bases: object

Pass in a definition file and a weather file to generate distributions of models

as_dataframe()
Return the dataframe with all the data needed to run the analysis defined in the json file.

Note that the first field in the analysis definition json file must be a value or an EPW.

Returns pandas dataframe

load_files(*definition_file*)

load_weather_file(*weather_file*)
Load in the weather file and convert the field names to what is expected in the JSON file :return:

EPW File

Process an EPW file

class rom.analysis_definition.epw_file.**EpwFile**(*filepath*)
Bases: object

as_dataframe()
Return the EPW file as a dataframe. This drops the data_source column for brevity.

Returns pandas DataFrame

post_process_data()
Add in derived columns

Returns

The analysis definition module is used for loading an already generated reduced order and running a subsequent analysis. The input is a JSON file that defines each of the covariates of interest. The analysis can take of

- Single value analysis, see *example*
- Sweep values over a year (as defined by an EPW file), see *example*
- Sweep values over specified ranges for single variable, see *example*
- Sweep values over specified ranges for multiple variable, see *example*

To run an analysis with a JSON file, first load a metamodel, then load the analysis defintion.

```
from rom.analysis_definition.analysis_definition import AnalysisDefinition
from rom.metamodels import Metamodels
```

6.4 Using the ROMs

6.4.1 CLI Example

This example CLI file shows how a simple application can be built that reads in all the values from the command line and reports the values of the responses passed.

```
# Single response - with only setting the inlet temperature
python analysis_cli_ex1.py -f smoff/metamodels.json -i 18

# Multiple responses - with only setting the inlet temperature
python analysis_cli_ex1.py -f smoff/metamodels.json -i 18 -r HeatingElectricity␣
↪DistrictHeatingHotWaterEnergy
```

6.4.2 Analysis Example

Example analysis script demonstrating how to programatically load and run already persisted reduced order models. This example loads two response variables (models) from the small office random forest reduced order models. The loaded models are then passed through the swee-temp-test.json analysis definition file. The analysis definition has few fixed covariates and a few covariates with multiple values to run.

```
python analysis_ex1.py
```

6.4.3 Sweep Example

Example analysis script demonstrating how to programatically load and run already persisted reduced order models using a weather file. This example is very similar to the analysis_ex1.py excpet for the analysis.load_weather_file method. This method and the smoff-one-year.json file together specify how to parse the weather file.

The second part of this script using seaborn to generate heatmaps of the two responses of interest. The plots are stored in a child directory. Run the example by calling the following:

```
python analysis_sweep_ex1.py
```

6.4.4 Modelica Example

This example file shows how to load the models using a method based approach for use in Modelica. The run_model takes only a list of numbers (int and floats). The categorical variables are converted as needed in order to correctly populate the list of covariates in the dataframe.

For use in Modelica ake sure that the python path is set, such as by running export PYTHONPATH=‘pwd‘

Call the following bash command shown below to run this example. This example runs as an entrypoint; however, when connected to modelica the def run_model will be called directory. Also, note that the run_model method loads the models every time it is called. This is non-ideal when using this code in a timestep by timestep simulation. Work needs to be done to determine how to load the reduced order models only once and call the reduced order model yhat methods each timestep.

```
python analysis_modelica_ex1.py
```

6.5 Developer Notes

6.5.1 Building Documentation

```
$ sphinx-apidoc -o docs/source/modules . rom
$ cd docs
$ make html
```

6.6 Code Documentation

6.6.1 Reduced Order Models

Metamodels

exception rom.metamodels.**DuplicateColumnName**
Bases: Exception

class rom.metamodels.**ETSModel**(*response_name*, *model_file*, *scaler_file=None*)
Bases: object

yhat(*data*)
Run predict on supplied data.

Parameters **data** – array, Values to predict on. The format is dependent on the model.

e.g. [[month, hour, dayofweek, t_outdoor, rh, inlet_temp]] :return: array, Model predictions.

class rom.metamodels.**Metamodels**(*filename*)

Bases: object

property algorithm_options

Return the algorithm options from the metamodels.json file that was passed in.

Returns dict, Algorithm options.

property analysis

Return the ROM analysis file.

Returns Parsed JSON ROM file.

property analysis_name

Return the analysis name from the metamodels.json file that was passed in.

Returns str, Analysis name.

available_response_names(*_model_type*)

Return a list of response names.

Parameters **_model_type** – str, The type of reduced order model (e.g. RandomForest).

Returns list, Response names.

covariate_names(*model_type*)

Return a list of covariate names. The order in the JSON file must be the order that is passed into the metamodel, otherwise the data will not make sense.

Parameters **model_type** – str, The type of reduced order model (e.g. RandomForest).

Returns list, Covariate names.

covariate_types(*model_type*)

Return dictionary of covariate types.

Parameters **model_type** – str, The type of reduced order model (e.g. RandomForest).

Returns dict, {'type':['covariate name']}.

covariates(*model_type*)

Return dictionary of covariates for specified model type.

Parameters **model_type** – str, The type of reduced order model (e.g. RandomForest).

Returns dict, Covariates.

downsamples(*model_name*)

Return the downsamples list from the metamodels.json file that was passed in.

Parameters **model_name** – str, name of the model to look for down samples

Returns list, Downsamples.

load_file(*filename*)

Parse the file that defines the ROMs that have been created.

Parameters **filename** – str, The JSON ROM file path.

load_models(*model_type*, *models_to_load=None*, *downsample=None*, *root_path=None*)

Load in the metamodels/generators.

Parameters

- **model_type** – str, The type of reduced order model (e.g. RandomForest).

- **models_to_load** – list, Name of responses to load.
- **downsample** – float, The downsample value to load. Defaults to None.

Returns dict, Metrics {response, model type, downsample, load time, disk size}.

property loaded_models
Return the list of available keys in the models dictionary.

Returns list, Responses.

model(*response_name*)
Return model for specific response.

Parameters **response_name** – str, Name of model response.

model_paths(*model_type*, *response*, *downsample=None*, *root_path=None*)
Return the paths to the model to be loaded. This includes the scaler value if the model requires the data to scale the input.

If the root path is provided, then that path will take precedent over the downsample and no values passed format.

Parameters

- **model_type** – str, The type of reduced order model (e.g. RandomForest).
- **response** – str, The response (or model) to load (e.g. ETSOutletTemperature).
- **downsample** – float, The downsample value to load. Defaults to None.
- **root_path** – If used, then it is the root path of the models. The models will be in subdirectories for each

of the model_types. :return: list, [model_path, scaler_path].

models_exist(*model_type*, *models_to_load=None*, *downsample=None*, *root_path=None*)
Check if the models exist, if not, then return false.

Parameters

- **model_type** – str, The type of reduced order model (e.g. RandomForest).
- **models_to_load** – list, Name of responses to load.
- **downsample** – float, The downsample value to load. Defaults to None.
- **root_path** – If used, then it is the root path of the models. The models will be in subdirectories for each

of the model_types. :return: bool

classmethod resolve_algorithm_options(*algorithm_options*)
Go through the algorithm options that are in the metamodel.json file and run 'eval' on the strings. This allows complex strings to exist in the json file that get expanded as necessary.

TODO: Add an example

Parameters **algorithm_options** – dict, the algorithm options to run eval on

Returns

property results_file
Path to the results file that is to be processed. This is a CSV file.

Returns str, path.

save_2d_csvs(*data*, *first_dimension*, *file_prepend*)
Generate 2D (time, first) CSVs based on the model loaded and the two dimensions.

The rows are the datetimes as defined in the data (DataFrame).

Parameters

- **data** – pandas DataFrame
- **first_dimension** – str, The column heading variable.
- **file_prepend** – str, Special variable to prepend to the file name.

Returns None

save_3d_csvs(*data*, *first_dimension*, *second_dimension*, *second_dimension_short_name*, *file_prepend*, *save_figure=False*)
Generate 3D (time, first, second) CSVs based on the model loaded and the two dimensions. The second dimension becomes individual files.

The rows are the datetimes as defined in the data (DataFrame)

Parameters

- **data** – pandas DataFrame
- **first_dimension** – str, The column heading variable.
- **second_dimension** – str, The values that will be reported in the table.
- **second_dimension_short_name** – str, Short display name for second variable (for filename).
- **file_prepend** – str, Special variable to prepend to the file name.

Returns None

save_csv(*data*, *csv_name*)
Save pandas DataFrame in CSV format.

Parameters

- **data** – pandas DataFrame, Data to be exported.
- **csv_name** – str, Name of the CSV file.

Returns

set_analysis(*moniker*)
Set the index of the analysis based on the ID or the name of the analysis.

Parameters **moniker** – str, Analysis ID or name.

Returns bool

property validation_id
Return the validation ID from the metamodels.json file that was passed in.

Returns str, Validation ID.

yhat(*response_name*, *data*)
Run predict on the selected model (response) with the supplied data.

Parameters

- **response_name** – str, Name of the model to evaluate.
- **data** – pandas DataFrame, Values to predict on.

Returns pandas DataFrame, Predictions.

Raises Exception: Model does not have the response.

yhats(*data*, *prepend_name*, *response_names=None*)
Run predict on multiple responses with the supplied data and store the results in the supplied DataFrame.

The prepend_name is needed in order to not overwrite the existing data in the dataframe after evaluation. For example, if the response name is HeatingElectricity, the supplied data may already have that field provided; therefore, this method adds the prepend_name to the newly predicted data. If prepend_name is set to 'abc', then the new column would be 'abc_HeatingElectricity'.

Parameters

- **data** – pandas DataFrame, Values to predict on.
- **prepend_name** – str, Name to prepend to the beginning of each of the response names.
- **response_names** – list, Responses to evaluate. If None, then defaults to all the available_response_names.

Returns pandas DataFrame, Original data with added predictions.

Evaluate Helpers

rom.evaluate_helpers.**evaluate_process_all_model_results**(*data*, *validation_dir*)

rom.evaluate_helpers.**evaluate_process_cv_results**(*cv_result_file*, *response*, *output_dir*)

rom.evaluate_helpers.**evaluate_process_model_results**(*model_results_file*, *output_dir*)

Shared Methods

rom.shared.**apply_cyclic_transform**(*row*, *column_name*, *category_count*)

rom.shared.**convert**(*name*)

rom.shared.**is_int**(*value*)

rom.shared.**pickle_file**(*obj*, *filename*, *gzipfile=False*)

Parameters

- **obj** –
- **filename** – Filename, without the extension
- **gzipfile** –

rom.shared.**save_dict_to_csv**(*data*, *filename*)

rom.shared.**unpickle_file**(*filename*)

rom.shared.**zipdir**(*path*, *ziph*, *extension=None*)
Flattened zip directory :param path: :param ziph: :param extension:

Validation Helpers

rom.validation_helpers.**validate_dataframe**(*df*, *metadata*, *image_save_dir*)
Take the dataframe and perform various validations and create plots

Parameters **df** – Contains the actual and modeled data for various ROMs

Returns

rom.validation_helpers.**validation_plot_energy_temp**(*melted_df*, *filename*)

rom.validation_helpers.**validation_plot_timeseries**(*melted_df*, *filename*)

rom.validation_helpers.**validation_save_metrics**(*df*, *output_dir*)

6.6.2 ROM Generators

Model Generator Base

class rom.generators.model_generator_base.**ModelGeneratorBase**(*analysis_id*, *random_seed=None*, ***kwargs*)

Bases: object

anova_plots(*y_data*, *yhat*, *model_name*)

build(*metamodel*, ***kwargs*)

evaluate(*model*, *model_name*, *model_moniker*, *x_data*, *y_data*, *downsample*, *build_time*, *cv_time*, *covariates=None*, *scaler=None*)

Generic base function to evaluate the performance of the models.

Parameters

- **model** –
- **model_name** –
- **x_data** –
- **y_data** –
- **downsample** –
- **build_time** –

Returns Ordered dict

inspect()

Inspect the dataframe and return the statistics of the dataframe.

Returns

load_data(*datafile*)

Load the data into a dataframe. The data needs to be a CSV file at the moment.

Parameters **datafile** – str, path to the CSV file to load

Returns None

save_dataframe(*dataframe*, *path*)

train_test_validate_split(*dataset*, *metamodel*, *downsample=None*, *scale=False*)

Use the built in method to generate the train and test data. This adds an additional set of data for validation. This vaildation dataset is a unique ID that is pulled out of the dataset before the test_train method is called.

yy_plots(*y_data*, *yhat*, *model_name*)

Plot the yy-plots

Parameters

- **y_data** –

- **yhat** –
- **model_name** –

Returns

Linear Model

class rom.generators.linear_model.**LinearModel**(*analysis_id*, *random_seed=None*, ***kwargs*)

Bases: *rom.generators.model_generator_base.ModelGeneratorBase*

build(*metamodel*, ***kwargs*)

evaluate(*model*, *model_name*, *model_type*, *x_data*, *y_data*, *downsample*, *build_time*, *cv_time*, *covariates=None*, *scaler=None*)

Evaluate the performance of the forest based on known x_data and y_data. If the model was scaled, then the test data will already be scaled.

Random Forest Model

class rom.generators.random_forest.**RandomForest**(*analysis_id*, *random_seed=None*, ***kwargs*)

Bases: *rom.generators.model_generator_base.ModelGeneratorBase*

build(*metamodel*, ***kwargs*)

evaluate(*model*, *model_name*, *model_type*, *x_data*, *y_data*, *downsample*, *build_time*, *cv_time*, *covariates=None*, *scaler=None*)

Evaluate the performance of the forest based on known x_data and y_data.

Parameters

- **model** –
- **model_name** –
- **model_type** –
- **x_data** –
- **y_data** –
- **downsample** –
- **build_time** –
- **cv_time** –
- **covariates** –

Returns

export_tree_png(*tree*, *covariates*, *filename*)

save_cv_results(*cv_results*, *response*, *downsample*, *filename*)

Save the cv_results to a CSV file. Data in the cv_results file looks like the following.

The CV results are the results of the GridSearch k-fold cross validation. The form of the results take the following from:

```
{
    'param_kernel': masked_array(data=['poly', 'poly', 'rbf', 'rbf'],
                                 mask=[False False False False]...)
    'param_gamma': masked_array(data=[-- -- 0.1 0.2],
                                mask=[True  True False False]...),
    'param_degree': masked_array(data=[2.0 3.0 - - --],
                                 mask=[False False  True  True]...),
    'split0_test_score': [0.8, 0.7, 0.8, 0.9],
    'split1_test_score': [0.82, 0.5, 0.7, 0.78],
    'mean_test_score': [0.81, 0.60, 0.75, 0.82],
    'std_test_score': [0.02, 0.01, 0.03, 0.03],
    'rank_test_score': [2, 4, 3, 1],
    'split0_train_score': [0.8, 0.9, 0.7],
    'split1_train_score': [0.82, 0.5, 0.7],
    'mean_train_score': [0.81, 0.7, 0.7],
    'std_train_score': [0.03, 0.03, 0.04],
    'mean_fit_time': [0.73, 0.63, 0.43, 0.49],
    'std_fit_time': [0.01, 0.02, 0.01, 0.01],
    'mean_score_time': [0.007, 0.06, 0.04, 0.04],
    'std_score_time': [0.001, 0.002, 0.003, 0.005],
    'params': [{'kernel': 'poly', 'degree': 2}, ...],
}
```

Parameters

- **cv_results** –
- **filename** –

Returns

Support Vector Regression

class rom.generators.svr.**SVR**(*analysis_id*, *random_seed=None*, ***kwargs*)

Bases: *rom.generators.model_generator_base.ModelGeneratorBase*

build(*metamodel*, ***kwargs*)

evaluate(*model*, *model_name*, *model_moniker*, *x_data*, *y_data*, *downsample*, *build_time*, *cv_time*, *covariates=None*, *scaler=None*)

Evaluate the performance of the forest based on known x_data and y_data.

save_cv_results(*cv_results*, *response*, *downsample*, *filename*)

Save the cv_results to a CSV file. Data in the cv_results file looks like the following.

The CV results are the results of the GridSearch k-fold cross validation. The form of the results take the following from:

```
{
    'param_kernel': masked_array(data=['poly', 'poly', 'rbf', 'rbf'],
                                 mask=[False False False False]...)
    'param_gamma': masked_array(data=[-- -- 0.1 0.2],
                                mask=[True  True False False]...),
    'param_degree': masked_array(data=[2.0 3.0 - - --],
                                 mask=[False False  True  True]...),
    'split0_test_score': [0.8, 0.7, 0.8, 0.9],
    'split1_test_score': [0.82, 0.5, 0.7, 0.78],
    'mean_test_score': [0.81, 0.60, 0.75, 0.82],
```

(continues on next page)

(continued from previous page)

```
    'std_test_score': [0.02, 0.01, 0.03, 0.03],
    'rank_test_score': [2, 4, 3, 1],
    'split0_train_score': [0.8, 0.9, 0.7],
    'split1_train_score': [0.82, 0.5, 0.7],
    'mean_train_score': [0.81, 0.7, 0.7],
    'std_train_score': [0.03, 0.03, 0.04],
    'mean_fit_time': [0.73, 0.63, 0.43, 0.49],
    'std_fit_time': [0.01, 0.02, 0.01, 0.01],
    'mean_score_time': [0.007, 0.06, 0.04, 0.04],
    'std_score_time': [0.001, 0.002, 0.003, 0.005],
    'params': [{'kernel': 'poly', 'degree': 2}, ...],
}
```

Parameters

- **cv_results** –
- **filename** –

Returns

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

r

INDEX