17 Models

In RAVEN, **Models** are important entities. A model is an object that employs a mathematical representation of a phenomenon, either of a physical or other nature (e.g. statistical operators, etc.). From a practical point of view, it can be seen, as a "black box" that, given an input, returns an output.

RAVEN has a strict classification of the different types of models. Each "class" of models is represented by the definition reported above, but it can be further classified based on its particular functionalities:

- **<Code>** represents an external system code that employs a high fidelity physical model.
- **<Dummy>** acts as "transfer" tool. The only action it performs is transferring the the information in the input space (inputs) into the output space (outputs). For example, it can be used to check the effect of a sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested.
- **<ROM>**, or reduced order model, is a mathematical model trained to predict a response of interest of a physical system. Typically, ROMs trade speed for accuracy representing a faster, rough estimate of the underlying phenomenon. The "training" process is performed by sampling the response of a physical model with respect to variation of its parameters subject to probabilistic behavior. The results (outcomes of the physical model) of the sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results.
- **<ExternalModel>**, as its name suggests, is an entity existing outside the RAVEN framework that is embedded in the RAVEN code at run time. This object allows the user to create a Python module that will be treated as a predefined internal model object.
- **<EnsembleModel>** is model that is able to combine **Code**, **ExternalModel** and **ROM** models. It is aimed to create a chain of Models (whose execution order is determined by the Input/Output relationships among them). If the relationships among the models evolve in a non-linear system, a Picard's Iteration scheme is employed.
- **<PostProcessor>** is a container of all the actions that can manipulate and process a data object in order to extract key information, such as statistical quantities, clustering, etc.

Before analyzing each model in detail, it is important to mention that each type needs to be contained in the main XML node **<Models>**, as reported below:

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <WhatEverModel name='whatever'>
      ...
    </WhatEverModel>
    ...
  </Models>
  ...
</Simulation>
```

In the following sub-sections each **Model** type is fully analyzed and described.

17.1 Code

As already mentioned, the **Code** model represents an external system software employing a high fidelity physical model. The link between RAVEN and the driven code is performed at run-time, through coded interfaces that are the responsible for transferring information from the code to RAVEN and vice versa. In Section 21, all of the available interfaces are reported and, for advanced users, Section 22 explains how to couple a new code.

The specifications of this model must be defined within a `<Code>` XML block. This XML node accepts the following attributes:

- `name`, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- `subType`, ***required string attribute***, specifies the code that needs to be associated to this Model. **Note:** See Section 21 for a list of currently supported codes.

This model can be initialized with the following children:

- `<executable>` ***string, required field*** specifies the path of the executable to be used. **Note:** Either an absolute or relative path can be used.
- `<alias>` ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Code. These aliases can be used anywhere in the RAVEN input to refer to the Code variables. In the body of this node the user specifies the name of the variable that the model is going to use (during its execution). The actual alias, usable throughout the RAVEN input, is instead defined in the `variable` attribute of this tag.

The user can specify aliases for both the input and the output space. As sanity check, RAVEN requires an additional required attribute `type`. This attribute can be either "input" or "output". **Note:** The user can specify as many aliases as needed.
Default: None

- `<clargs>` ***string, optional field*** allows addition of command-line arguments to the execution command. If the code interface specified in `<Code>` `subType` does not specify how to determine the input file(s), this node must be used to specify them. There are several types of `<clargs>`, based on the `type`:

  - `type` ***string, required field*** specifies the type of command-line argument to add. Options include `'input'`, `'output'`, `'prepend'`, `'postpend'`, and `'text'`.
  - `arg` ***string, optional field*** specifies the flag to be used before the entry. For example, `arg='-i'` would place a `-i` before the entry in the execution command. Required for the `'output'` `type`.
  - `extension` ***string, optional field*** specifies the type of file extension to use (for example, `-i` or `-o`). This links the `<Input>` file in the `<Step>` to this location in the execution command. Required for `'input'` `type`.

  The execution command is combined in the order `'prepend'`, `<executable>`, `'input'`, `'output'`, `'text'`, `'postpend'`.

- `<fileargs>` ***string, optional field*** like `<clargs>`, but allows editing of input files to specify the output filename and/or auxiliary file names. The location in the input files to edit using these arguments are identified in the input file using the prefix-postfix notation, which defaults to `$RAVEN-var$` for variable keyword *var*. The variable keyword is then listed in the `<fileargs>` node in the attribute `arg` to couple it in Raven. If the code interface specified in `<Code>` `subType` does not specify how to name the output file, that must be specified either through `<clargs>` or `<filargs>`, with `type` `'output'`. The attributes required for `<fileargs>` are as follows:

  - `type` ***string, required field*** specifies the type of entry to replace in the file. Possible values for `<fileargs>` `type` are `'input'` and `'output'`.
  - `arg` ***string, required field*** specifies the Raven variable with which to replace the file of interest. This should match the entry in the template input file; that is, if `$RAVEN-auxinp$` is in the input file, the arg for the corresponding input file should be `'auxinp'`.
  - `extension` ***string, optional field*** specifies the extension of the input file that should replace the Raven variable in the input file. This attribute is required for the `'input'` `type` and ignored for the `'output'` `type`. **Note:** Currently, there can only be a one-to-one pairing between input files and extensions; that is, multiple Raven-editable input files cannot have the same extension.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <Code name='aUserDefinedName' subType='RAVEN_Driven_code'>
      <executable>path_to_executable</executable>
      <alias variable='internal_RAVEN_input_variable_name1'
         type="input">
         External_Code_input_Variable_Name_1
      </alias>
      <alias variable='internal_RAVEN_input_variable_name2'
         type='input'>
         External_Code_input_Variable_Name_2
      </alias>
      <alias variable='internal_RAVEN__output_variable_name'
         type='output'>
         External_Code_output_Variable_Name_2
      </alias>
      <clargs type='prepend' arg='python'/>
      <clargs type='input' arg='-i' extension='.i'/>
      <fileargs type='input' arg='aux' extension='.two'
      <fileargs type='output' arg='out' />
    </Code>
    ...
  </Models>
  ...
</Simulation>
```

17.2 Dummy

The **Dummy** model is an object that acts as a pass-through tool. The only action it performs is transferring the information in the input space (inputs) to the output space (outputs). For example, it can be used to check the effect of a particular sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested.

The specifications of this model must be defined within a **<Dummy>** XML block. . This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other

objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

- `subType`, ***required string attribute***, this attribute must be kept empty.

This model can be initialized with the following children:

- `<alias>` ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Dummy. These aliases can be used anywhere in the RAVEN input to refer to the Dummy variables. In the body of this node the user specifies the name of the variable that the model is going to use (during its execution). The actual alias, usable throughout the RAVEN input, is instead defined in the `variable` attribute of this tag.
  The user can specify aliases for both the input and the output space. As sanity check, RAVEN requires an additional required attribute `type`. This attribute can be either "input" or "output". **Note:** The user can specify as many aliases as needed.
  Default: None
  Since the **Dummy** model represents a transfer function only, the usage of the alias is relatively meaningless.

Given a particular *Step* using this model, if this model is linked to a *Data* with the role of **Output**, it expects one of the output parameters will be identified by the keyword "OutputPlaceHolder" (see Section 20).

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <Dummy name='aUserDefinedName1' subType=''/>

    <Dummy name='aUserDefinedName2' subType=''>
      <alias variable="a_RAVEN_input_variable" type="input">
      another_name_for_this_variable_in_the_model
      </alias>
    </Dummy>
    ...
  </Models>
  ...
</Simulation>
```

17.3 ROM

A Reduced Order Model (ROM) is a mathematical model consisting of a fast solution trained to predict a response of interest of a physical system. The "training" process is performed by sampling the response of a physical model with respect to variations of its parameters subject, for example, to probabilistic behavior. The results (outcomes of the physical model) of the sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results. RAVEN supports several different types of ROMs, both internally developed and imported through an external library called "scikit-learn" [5].

Currently in RAVEN, the ROMs are classified into several sub-types that, once chosen, provide access to several different algorithms. These sub-types are specified in the **subType** attribute and should be one of the following:

- **'NDspline'**
- **'GaussPolynomialRom'**
- **'HDMRRom'**
- **'NDinvDistWeight'**
- **'SciKitLearn'**
- **'MSR'**
- **'ARMA'**

The specifications of this model must be defined within a **<ROM>** XML block. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, defines which of the sub-types should be used, choosing among the previously reported types. This choice conditions the subsequent the required and/or optional **<ROM>** sub-nodes.

In the **<ROM>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<Features>**, ***comma separated string, required field***, specifies the names of the features of this ROM. **Note:** These parameters are going to be requested for the training of this object (see Section 20.4);

- **<Target>**, ***comma separated string, required field***, contains a comma separated list of the targets of this ROM. These parameters are the Figures of Merit (FOMs) this ROM is supposed to predict. **Note:** These parameters are going to be requested for the training of this object (see Section 20.4).

In addition, if the user wants to use the alias system, the following XML block can be inputted:

- **<alias>** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the ROM. These aliases can be used anywhere in the RAVEN input to refer to the ROM variables. In the body of this node the user specifies the name of the variable that the model is going to use (during its execution). The actual alias, usable throughout the RAVEN input, is instead defined in the **variable** attribute of this tag.
  The user can specify aliases for both the input and the output space. As sanity check, RAVEN requires an additional required attribute **type**. This attribute can be either "input" or "output". **Note:** The user can specify as many aliases as needed.
  Default: None

The types and meaning of the remaining sub-nodes depend on the sub-type specified in the attribute **subType**.

Note that if an HistorySet is provided in the training step then a temporal ROM is created, i.e. a ROM that generates not a single value prediction of each element indicated in the **<Target>** block but its full temporal profile.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing most of the Reduced Order Models (e.g. most of the SciKitLearn-based ROMs):

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{3}$$

In the following sections the specifications of each ROM type are reported, highlighting when a **Z-score normalization** is performed by RAVEN before constructing the ROM or when it is not performed.

17.3.1 NDspline

The NDspline sub-type contains a single ROM type, based on an N-dimensional spline interpolation/extrapolation scheme. In spline interpolation, the interpolant is a special type of piecewise polynomial called a spline. The interpolation error can be made small even when using low degree polynomials for the spline. Spline interpolation avoids the problem of Runge's phenomenon, in which oscillation can occur between points when interpolating using higher degree polynomials.

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'NDspline'** (see the example below). No further XML sub-nodes are required. **Note:** This ROM type must be trained from a regular Cartesian grid. Thus, it can only be trained from the outcomes of a grid sampling strategy.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *NDspline* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{4}$$

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='NDspline'>
       <Features>var1,var2,var3</Features>
       <Target>result1,result2</Target>
     </ROM>
    ...
  </Models>
  ...
</Simulation>
```

17.3.2 GaussPolynomialRom

The GaussPolynomialRom sub-type contains a single ROM type, based on a characteristic Gaussian polynomial fitting scheme: generalized polynomial chaos expansion (gPC). In gPC, sets of polynomials orthogonal with respect to the distribution of uncertainty are used to represent the original model. The method converges moments of the original model faster than Monte Carlo for small-dimension uncertainty spaces ($N < 15$). In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'GaussPolynomialRom'** (see the example below). The GaussPolynomialRom is dependent on specific sampling; thus, this ROM cannot be trained unless a SparseGridCollocation or similar Sampler specifies this ROM in its input and is sampled in a MultiRun step. In addition to the common **<Target>** and **<Features>**, this ROM requires two more nodes and can accept multiple entries of a third optional node.

- **<IndexSet>**, ***string, required field***, specifies the rules by which to construct multidimensional polynomials. The options are **'TensorProduct'**, **'TotalDegree'**,

`'HyperbolicCross'`, and `'Custom'`. Total degree is efficient for uncertain inputs with a large degree of regularity, while hyperbolic cross is more efficient for low-regularity input spaces. If `'Custom'` is chosen, the `<IndexPoints>` is required.

- `<PolynomialOrder>`, ***integer, required field***, indicates the maximum polynomial order in any one dimension to use in the polynomial chaos expansion. **Note:** If non-equal importance weights are supplied in the optional `<Interpolation>` node, the actual polynomial order in dimensions with high importance might exceed this value; however, this value is still used to limit the relative overall order.

- `<SparseGrid>`,***string, optional field***, allows specification of the multidimensional quadrature construction strategy. Options are `'smolyak'` and `'tensor'`. Default is `'smolyak'`.

- `<IndexPoints>`, ***list of tuples, required field***, used to specify the index set points in a `'Custom'` index set. The tuples are entered as comma-seprated values between parenthesis, with each tuple separated by a comma. Any amount of whitespace is acceptable. For example, `<IndexPoints>(0,1),(0,2),(1,1),(4,0)</IndexPoints>` **Note:** Using custom index sets does not guarantee accurate convergence.

- `<Interpolation>`, ***string, optional field***, offers the option to specify quadrature, polynomials, and importance weights for the given variable name. The ROM accepts any number of `<Interpolation>` nodes up to the dimensionality of the input space. This node accepts several attributes, all of which are optional and default to the code-defined optimal choices based on the input dimension uncertainty distribution:

  - `quad`, ***string, optional field***, specifies the quadrature type to use for collocation in this dimension. The default options depend on the uncertainty distribution of the input dimension, as shown in Table 2. Additionally, Clenshaw Curtis quadrature can be used for any distribution that doesn't include an infinite bound.
    Default: see Table 2. **Note:** For an uncertain distribution aside from the four listed on Table 2, this ROM makes use of the uniform-like range of the distribution's CDF to apply quadrature that is suited uniform uncertainty (Legendre). It converges more slowly than the four listed, but are viable choices. Choosing polynomial type Legendre for any non-uniform distribution will enable this formulation automatically.

  - `poly`, ***string,optional field***, specifies the interpolating polynomial family to use for the polynomial expansion in this dimension. The default options depend on the quadrature type chosen, as shown in Table 2. Currently, no polynomials are available outside the default.
    Default: see Table 2.

  - `weight`, ***float, optional field***, delineates the importance weighting of this dimension. A larger importance weight will result in increased resolution for this dimension at the cost of resolution in lower-weighted dimensions. The algorithm normalizes weights at run-time.
    Default: 1.

Unc. Distribution	Default Quadrature	Default Polynomials
Uniform	Legendre	Legendre
Normal	Hermite	Hermite
Gamma	Laguerre	Laguerre
Beta	Jacobi	Jacobi
Other	Legendre*	Legendre*

Table 2. GaussPolynomialRom defaults

Note: This ROM type must be trained from a collocation quadrature set. Thus, it can only be trained from the outcomes of a SparseGridCollocation sampler. Also, this ROM must be referenced in the SparseGridCollocation sampler in order to accurately produce the necessary sparse grid points to train this ROM.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *GaussPolynomialRom* ROM.

Example:

```
<Simulation>
  ...
  <Samplers>
    ...
    <SparseGridCollocation name="mySG" parallel="0">
      <variable name="x1">
        <distribution>myDist1</distribution>
      </variable>
      <variable name="x2">
        <distribution>myDist2</distribution>
      </variable>
      <ROM class = 'Models' type = 'ROM' >myROM</ROM>
    </SparseGridCollocation>
    ...
  </Samplers>
  ...
  <Models>
    ...
    <ROM name='myRom' subType='GaussPolynomialRom'>
      <Target>ans</Target>
      <Features>x1,x2</Features>
      <IndexSet>TotalDegree</IndexSet>
      <PolynomialOrder>4</PolynomialOrder>
      <Interpolation quad='Legendre' poly='Legendre'
         weight='1'>x1</Interpolation>
      <Interpolation quad='ClenshawCurtis' poly='Jacobi'
         weight='2'>x2</Interpolation>
```

```
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

When Printing this ROM via a Print OutStream (see 16.1), the available metrics are:

- **'mean'**, the mean value of the ROM output within the input space it was trained,
- **'variance'**, the variance of the ROM output within the input space it was trained,
- **'samples'**, the number of distinct model runs required to construct the ROM,
- **'indices'**, the Sobol sensitivity indices (in percent), Sobol total indices, and partial variances,
- **'polyCoeffs'**, the polynomial expansion coefficients (PCE moments) of the ROM. These are listed by each polynomial combination, with the polynomial order tags listed in the order of the variables shown in the XML print.

17.3.3 HDMRRom

The HDMRRom sub-type contains a single ROM type, based on a Sobol decomposition scheme. In Sobol decomposition, also known as high-density model reduction (HDMR, specifically Cut-HDMR), a model is approximated as as the sum of increasing-complexity interactions. At its lowest level (order 1), it treats the function as a sum of the reference case plus a functional of each input dimesion separately. At order 2, it adds functionals to consider the pairing of each dimension with each other dimension. The benefit to this approach is considering several functions of small input cardinality instead of a single function with large input cardinality. This allows reduced order models like generalized polynomial chaos (see 17.3.2) to approximate the functionals accurately with few computations runs. In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'HDMRRom'** (see the example below). The HDMRRom is dependent on specific sampling; thus, this ROM cannot be trained unless a Sobol or similar Sampler specifies this ROM in its input and is sampled in a MultiRun step. In addition to the common **<Target>** and **<Features>**, this ROM requires the same nodes as the GaussPolynomialRom (see 17.3.2. Additionally, this ROM requires the **<SobolOrder>** node.

- **<SobolOrder>**, ***integer, required field***, indicates the maximum cardinality of the input space used in the subset functionals. For example, order 1 includes only functionals of each independent dimension separately, while order 2 considers pair-wise interactions.

Note: This ROM type must be trained from a Sobol decomposition training set. Thus, it can only be trained from the outcomes of a Sobol sampler. Also, this ROM must be referenced in the Sobol sampler in order to accurately produce the necessary sparse grid points to train this ROM. Experience has shown order 2 Sobol decompositions to include the great majority of uncertainty in most models.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *HDMRRom* ROM.

Example:

```
<Samplers>
  ...
  <Sobol name="mySobol" parallel="0">
    <variable name="x1">
      <distribution>myDist1</distribution>
    </variable>
    <variable name="x2">
      <distribution>myDist2</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM' >myHDMR</ROM>
  </Sobol>
  ...
</Samplers>
...
<Models>
  ...
  <ROM name='myHDMR' subType='HDMRRom'>
    <Target>ans</Target>
    <Features>x1,x2</Features>
    <SobolOrder>2</SobolOrder>
    <IndexSet>TotalDegree</IndexSet>
    <PolynomialOrder>4</PolynomialOrder>
    <Interpolation quad='Legendre' poly='Legendre'
       weight='1'>x1</Interpolation>
    <Interpolation quad='ClenshawCurtis' poly='Jacobi'
       weight='2'>x2</Interpolation>
  </ROM>
  ...
</Models>
```

When Printing this ROM via an OutStream (see 16.1), the available metrics are:

- **'mean'**, the mean value of the ROM output within the input space it was trained,
- **'variance'**, the ANOVA-calculated variance of the ROM output within the input space it was trained.

- **'samples'**, the number of distinct model runs required to construct the ROM,
- **'indices'**, the Sobol sensitivity indices (in percent), Sobol total indices, and partial variances.

17.3.4 MSR

The MSR sub-type contains a class of ROMs that perform a topological decomposition of the data into approximately monotonic regions and fits weighted linear patches to the identified monotonic regions of the input space. Query points have estimated probabilities that they belong to each cluster. These probabilities can eitehr be used to give a smooth, weighted prediction based on the associated linear models, or a hard classification to a particular local linear model which is then used for prediction. Currently, the probability prediction can be done using kernel density estimation (KDE) or through a one-versus-one support vector machine (SVM).

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'MSR'** (see the associated example). This model can be initialized with the following children:

- **<persistence>**, ***string, optional field***, specifies how to define the hierarchical simplification by assigning a value to each local minimum and maximum according to the one of the strategy options below:
  - `difference` - The function value difference between the extremum and its closest-valued neighboring saddle.
  - `probability` - The probability integral computed as the sum of the probability of each point in a cluster divided by the count of the cluster.
  - `count` - The count of points that flow to or from the extremum.

  Default: `difference`
- **<gradient>**, ***string, optional field***, specifies the method used for estimating the gradient, available options are:
  - `steepest`

  Default: `steepest`
- **<simplification>**, ***float, optional field***, specifies the amount of noise reduction to apply before returning labels.
  Default: 0

- **<graph> , *string, optional field***, specifies the type of neighborhood graph used in the algorithm, available options are:
  - `beta skeleton`
  - `relaxed beta skeleton`
  - `approximate knn`

  Default: `beta skeleton`
- **<beta>**, ***float in the range: (0,2], optional field***, is only used when the **<graph>** is set to `beta skeleton` or `relaxed beta skeleton`.
  Default: 1.0
- **<knn>**, ***integer, optional field***, is the number of neighbors when using the `approximate knn` for the **<graph>** sub-node and used to speed up the computation of other graphs by using the approximate knn graph as a starting point for pruning. -1 means use a fully connected graph.
  Default: -1
- **<partitionPredictor>**, ***string, optional***, a flag that specifies how the predictions for query point classification should be performed. Available options are:
  - `kde`
  - `svm`

  Default: kde
- **<smooth>**, if this node is present, the ROM will blend the estimates of all of the local linear models weighted by the probability the query point is classified as belonging to that partition of the input space.
- **<kernel>**, ***string, optional field***, this option is only used when the **<partitionPredictor>** is set to `kde` and specifies the type of kernel to use in the kernel density estimation. Available options are:
  - `uniform`
  - `triangular`
  - `gaussian`
  - `epanechnikov`
  - `biweight` or `quartic`
  - `triweight`

- `tricube`
- `cosine`
- `logistic`
- `silverman`
- `exponential`

Default: gaussian

- **<bandwidth>**, ***float or string, optional field***, this option is only used when the **<partitionPredicto** is set to `kde` and specifies the scale of the fall-off. A higher bandwidth implies a smooother blending. If set to `variable`, then the bandwidth will be set to the distance of the k-nearest neighbor of the query point where k is set by the **<knn>** parameter.
Default: 1.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MSR* ROM.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    </ROM>
    <ROM name='aUserDefinedName' subType='MSR'>
       <Features>var1,var2,var3</Features>
       <Target>result1,result2</Target>
       <!-- <weighted>true</weighted> -->
       <simplification>0.0</simplification>
       <persistence>difference</persistence>
       <gradient>steepest</gradient>
       <graph>beta skeleton</graph>
       <beta>1</beta>
       <knn>8</knn>
       <partitionPredictor>kde</partitionPredictor>
       <kernel>gaussian</kernel>
       <smooth/>
       <bandwidth>0.2</bandwidth>
     </ROM>
    ...
  </Models>
```

```
  ...
</Simulation>
```

17.3.5 NDinvDistWeight

The NDinvDistWeight sub-type contains a single ROM type, based on an N-dimensional inverse distance weighting formulation. Inverse distance weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to unknown points are calculated via a weighted average of the values available at the known points.

In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be xml-StringNDinvDistWeight (see the example below). This model can be initialized with the following child:

- **<p>**, ***integer, required field***, must be greater than zero and represents the "power parameter". For the choice of value for **<p>**,it is necessary to consider the degree of smoothing desired in the interpolation/extrapolation, the density and distribution of samples being interpolated, and the maximum distance over which an individual sample is allowed to influence the surrounding ones (lower p means greater importance for points far away).

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *NDinvDistWeight* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{5}$$

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='NDinvDistWeight'>
      <Features>var1,var2,var3</Features>
      <Target>result1,result2</Target>
      <p>3</p>
     </ROM>
    ...
  </Models>
```

```
  ...
</Simulation>
```

17.3.6 SciKitLearn

The SciKitLearn sub-type represents the container of several ROMs available in RAVEN through the external library scikit-learn [5].

In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be **'SciKitLearn'** (i.e. **subType='SciKitLearn'**). The specifications of a **'SciKitLearn'** ROM depend on the value assumed by the following sub-node within the main **<ROM>** XML node:

- **<SKLtype>**, ***vertical bar (|) separated string, required field***, contains a string that represents the ROM type to be used. As mentioned, its format is:
  <SKLtype>`mainSKLclass|algorithm`**</SKLtype>** where the first word (before the "|" symbol) represents the main class of algorithms, and the second word (after the "|" symbol) represents the specific algorithm.

Based on the **<SKLtype>** several different algorithms are available. In the following paragraphs a brief explanation and the input requirements are reported for each of them.

17.3.6.1 Linear Models

The LinearModels' algorithms implement generalized linear models. They include Ridge regression, Bayesian regression, lasso, and elastic net estimators computed with least angle regression and coordinate descent. This class also implements stochastic gradient descent related algorithms. In the following, all of the linear models available in RAVEN are reported.

17.3.6.1.1 Linear Model: Automatic Relevance Determination Regression

The *Automatic Relevance Determination* (ARD) regressor is a hierarchical Bayesian approach where hyperparameters explicitly represent the relevance of different input features. These relevance hyperparameters determine the range of variation for the parameters relating to a particular input, usually by modelling the width of a zero-mean Gaussian prior on those parameters. If the width of the Gaussian is zero, then those parameters are constrained to be zero, and the corresponding input cannot have any effect on the predictions, therefore making it irrelevant. ARD optimizes these hyperparameters to discover which inputs are relevant. In order to use the *Automatic Relevance Determination regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|ARDRegression`**</SKLtype>**.

In addition to this XML node, several others are available: .

- **<n_iter>**, ***integer, optional field***, is the maximum number of iterations.
  Default: 300

- **<tol>**, ***float, optional field***, stop the algorithm if the convergence error felt below the tolerance specified here.
  Default: 1.e-3

- **<alpha_1>**, ***float, optional field***, is a shape hyperparameter for the Gamma distribution prior over the α parameter.
  Default: 1.e-6

- **<alpha_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the α parameter.
  Default: 1.e-6

- **<lambda_1>**, ***float, optional field***, shape hyperparameter for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6

- **<lambda_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6

- **<compute_score>**, ***boolean, optional field***, if True, compute the objective function at each step of the model.
  Default: False

- **<threshold_lambda>**, ***float, optional field***, specifies the threshold for removing (pruning) weights with high precision from the computation.
  Default: 1.e+4

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ARDRegression* ROM.

17.3.6.1.2 Linear Model: Bayesian ridge regression

The *Bayesian ridge regression* estimates a probabilistic model of the regression problem as described above. The prior for the parameter w is given by a spherical Gaussian:

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p}) \tag{6}$$

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian. The resulting model is called Bayesian ridge regression, and is similar to the classical ridge regression. The parameters w, α, and λ are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be non-informative. The parameters are estimated by maximizing the marginal log likelihood. In order to use the *Bayesian ridge regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|BayesianRidge`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_iter>**, ***integer, optional field***, is the maximum number of iterations.
  Default: 300
- **<tol>**, ***float, optional field***, stop the algorithm if the convergence error felt below the tolerance specified here.
  Default: 1.e-3
- **<alpha_1>**, ***float, optional field***, is a shape hyperparameter for the Gamma distribution prior over the α parameter.
  Default: 1.e-6
- **<alpha_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the α parameter.
  Default: 1.e-6
- **<lambda_1>**, ***float, optional field***, shape hyperparameter for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6
- **<lambda_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6

- **<compute_score>**, ***boolean, optional field***, if True, compute the objective function at each step of the model.
  Default: False

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *BayesianRidge* ROM.

17.3.6.1.3 Linear Model: Elastic Net

The *Elastic Net* is a linear regression technique with combined L1 and L2 priors as regularizers. It minimizes the objective function:

$$1/(2*n_{samples})*||y-Xw||_2^2+alpha*l1\_ratio*||w||_1+0.5*alpha*(1-l1\_ratio)*||w||_2^2 \quad (7)$$

In order to use the *Elastic Net regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|ElasticNet`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies a constant that multiplies the penalty terms. $alpha = 0$ is equivalent to an ordinary least square, solved by the **LinearRegression** object.
  Default: 1.0

- **<l1_ratio>**, ***float, optional field***, specifies the ElasticNet mixing parameter, with $0 <= l1\_ratio <= 1$. For $l1\_ratio = 0$ the penalty is an L2 penalty. For $l1\_ratio = 1$ it is an L1 penalty. For $0 < l1\_ratio < 1$, the penalty is a combination of L1 and L2.
  Default: 0.5

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is

expected to be already centered).
Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: False

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
Default: False

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ElasticNet* ROM.

17.3.6.1.4 Linear Model: Elastic Net CV

The *Elastic Net CV* is a linear regression similar to the Elastic Net model but with an iterative fitting along a regularization path. The best model is selected by cross-validation.

In order to use the *Elastic Net CV regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|ElasticNetCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<l1_ratio>**, ***float, optional field***, Float flag between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties). For $l1\_ratio = 0$ the penalty is an L2 penalty. For $l1\_ratio = 1$ it is an L1 penalty. For $0 < l1\_ratio < 1$, the penalty is a combination of L1 and L2 This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for $l1\_ratio$ is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as

in [.1, .5, .7, .9, .95, .99, 1].
Default: 0.5

- **<eps>**, ***float, optional field***, specifies the length of the path. eps=1e-3 means that $alpha\_min/alpha\_max = 1e-3$.
Default: 0.001

- **<n_alphas>**, ***integer, optional field***, is the number of alphas along the regularization path used for each $l1\_ratio$.
Default: 100

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
Default: 1.e-4

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ElasticNetCV* ROM.

17.3.6.1.5 Linear Model: Least Angle Regression model

The *Least Angle Regression model* (LARS) is a regression algorithm for high-dimensional data. The LARS algorithm provides a means of producing an estimate of which variables to include, as well as their coefficients, when a response variable is determined by a linear combination of a subset of potential covariates.

In order to use the *Least Angle Regression model*, the user needs to set the sub-node:

<SKLtype>`linear_model|Lars`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_nonzero_coefs>**, ***integer, optional field***, represents the target number of non-zero coefficients.
  Default: 500

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<eps>**, ***float, optional field***, represents the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the **<tol>** parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
  Default: 2.2204460492503131e-16

- **<fit_path>**, ***boolean, optional field***, if True the full path is stored in the coef_path_attribute. If you compute the solution for a large problem or many targets, setting fit_path to False will lead to a speedup, especially with a small alpha.
  Default: True

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *Lars* ROM.

17.3.6.1.6 Linear Model: Cross-validated Least Angle Regression model

The *Cross-validated Least Angle Regression model* is a regression algorithm for high-dimensional data. It is similar to the LARS method, but the best model is selected by cross-validation. In order to use the *Cross-validated Least Angle Regression model*, the user needs to set the sub-node:

<SKLtype>`linear_model|LarsCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500

- **<max_n_alphas>**, ***integer, optional field***, specifies the maximum number of points on the path used to compute the residuals in the cross-validation.
  Default: 1000

- **<eps>**, ***float, optional field***, represents the machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the *tol* parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
  Default: 2.2204460492503131e-16

17.3.6.1.7 Linear Model trained with L1 prior as regularizer (aka the Lasso)

The *Linear Model trained with L1 prior as regularizer (Lasso)* is a shrinkage and selection method for linear regression. It minimizes the usual sum of squared errors, with a bound on the sum of the absolute values of the coefficients. In order to use the *Linear Model trained with L1 prior as regularizer (Lasso)*, the user needs to set the sub-node:

<SKLtype>`linear_model|Lasso`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, sets a constant multiplier for the L1 term. alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object. For numerical reasons, using alpha = 0 with the Lasso object is not advised and you should instead use the

LinearRegression object.
Default: 1.0

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: False

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
Default: False **Note:** For sparse input this option is always True to preserve sparsity.

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
Default: False

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LarsCV* ROM.

17.3.6.1.8 Lasso linear model with iterative fitting along a regularization path

The *Lasso linear model with iterative fitting along a regularization path* is an algorithm of the Lasso family, that computes the linear regressor weights, identifying the regularization path in an iterative fitting (see http://www.jstatsoft.org/v33/i01/paper)

In order to use the *Lasso linear model with iterative fitting along a regularization path regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<eps>**, ***float, optional field***, represents the length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.
  Default: 1.0e-3
- **<n_alphas>**, ***int, optional field***, sets the number of alphas along the regularization path.
  Default: 100
- **<alphas>**, ***numpy array, optional field***, lists the locations of the alphas used to compute the models.
  Default: None If None, alphas are set automatically.
- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000
- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False
- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoCV* ROM.

17.3.6.1.9 Lasso model fit with Least Angle Regression

Lasso model fit with Least Angle Regression (aka Lars) It is a Linear Model trained with an L1 prior as regularizer. In order to use the *Least Angle Regression model regressor*, the user needs to set the sub-node In order to use the *Least Angle Regression model regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLars`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies a constant that multiplies the penalty terms. $alpha = 0$ is equivalent to an ordinary least square, solved by the **LinearRegression** object.
  Default: 1.0
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True
- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500
- **<eps>**, ***float, optional field***, sets the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLars* ROM.

17.3.6.1.10 Cross-validated Lasso, using the LARS algorithm

The *Cross-validated Lasso, using the LARS algorithm* is a cross-validated Lasso, using the LARS algorithm.

In order to use the *Cross-validated Lasso, using the LARS algorithm regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLarsCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True
- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500
- **<max_n_alphas>**, ***integer, optional field***, specifies the maximum number of points on the path used to compute the residuals in the cross-validation.
  Default: 1000
- **<eps>**, ***float, optional field***, specifies the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLarsCV* ROM.

17.3.6.1.11 Lasso model fit with Lars using BIC or AIC for model selection

The *Lasso model fit with Lars using BIC or AIC for model selection* is a Lasso model fit with Lars using BIC or AIC for model selection. The optimization objective for Lasso is: $(1/(2 * n\_samples)) * ||y - Xw||_2^2 + alpha * ||w||_1$ AIC is the Akaike information criterion and BIC is the Bayes information criterion. Such criteria are useful in selecting the value of the regularization parameter by making a trade-off between the goodness of fit and the complexity of the model. A good model explains the data well while maintaining simplicity. In order to use the *Lasso model fit with Lars using BIC or AIC for model selection regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLarsIC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***'bic' — 'aic'*** , specifies the type of criterion to use.
  Default: 'aic'
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True
- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500
- **<eps>**,***float, optional field***, represents the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the tol parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLarsIC* ROM.

17.3.6.1.12 Ordinary least squares Linear Regression

The *Ordinary least squares Linear Regression* is a method for estimating the unknown parameters in a linear regression model, with the goal of minimizing the differences between the observed responses in some arbitrary dataset and the responses predicted by the linear approximation of the data. In order to use the *Ordinary least squares Linear Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LinearRegression`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LinearRegression* ROM.

17.3.6.1.13 Logistic Regression

The *Logistic Regression* implements L1 and L2 regularized logistic regression using the liblinear library. It can handle both dense and sparse input. This regressor uses C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied). In order to use the *Logistic Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LogisticRegression`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<penalty>**, ***string, 'l1' or 'l2'***, specifies the norm used in the penalization.
  Default: 'l2'

- **<dual>**, ***boolean***, specifies the dual or primal formulation. Dual formulation is only implemented for the l2 penalty. Prefer dual=False when n_samples $>$ n_features.
  Default: False

- **<C>**, ***float, optional field***, is the inverse of the regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
  Default: 1.0

- **<fit_intercept>**, ***boolean***, specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
  Default: True

- **<intercept_scaling>**, ***float, optional field***, when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight. **Note:** The synthetic feature weight is subject to l1/l2 regularization as are all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.
  Default: 1.0

- **<class_weight>**, ***dict, or 'balanced', optional*** Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)) Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified. New in version 0.17: class_weight='balanced' instead of deprecated class_weight='auto'.
  Default: None

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 0.0001

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *LogisticRegression* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{8}$$

17.3.6.1.14 Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer

The *Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer* is a regressor where the optimization objective for Lasso is: $(1/(2*n\_samples))*||Y-XW||^2_{Fro}+alpha*||W||_{21}$ Where: $||W||_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$ i.e. the sum of norm of each row. In order to use the *Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer regressor*, the user needs to set the sub-node:

`<SKLtype>linear_model|MultiTaskLasso</SKLtype>`.

In addition to this XML node, several others are available:

- `<alpha>`, ***float, optional field***, sets the constant multiplier for the L1/L2 term.
  Default: 1.0
- `<fit_intercept>`, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- `<normalize>`, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False
- `<max_iter>`, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000
- `<tol>`, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4
- `<warm_start>`, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultiTaskLasso* ROM.

17.3.6.1.15 Multi-task Elastic Net model trained with L1/L2 mixed-norm as regularizer

The *Multi-task Elastic Net model trained with L1/L2 mixed-norm as regularizer* is a regressor where the optimization objective for MultiTaskElasticNet is: $(1/(2 * n\_samples)) * ||Y -$

$XW||_2^{Fro} + alpha * l1\_ratio * ||W||_{21} + 0.5 * alpha * (1 - l1\_ratio) * ||W||_{Fro}^2$ Where: $||W||_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$ i.e. the sum of norm of each row. In order to use the *Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|MultiTaskElasticNet`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, represents a constant multiplier for the L1/L2 term.
  Default: 1.0
- **<l1_ratio>**, ***float***, represents the Elastic Net mixing parameter, with $0 < l1\_ratio \leq 1$. For $l1\_ratio = 0$ the penalty is an L1/L2 penalty. For $l1\_ratio = 1$ it is an L1 penalty. For $0 < l1\_ratio < 1$, the penalty is a combination of L1/L2 and L2.
  Default: 0.5
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4
- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultiTaskElasticNet* ROM.

17.3.6.1.16 Orthogonal Mathching Pursuit model (OMP)

The *Orthogonal Mathching Pursuit model (OMP)* is a type of sparse approximation which involves finding the "best matching" projections of multidimensional data onto an over-complete

dictionary, D. In order to use the *Orthogonal Mathching Pursuit model (OMP) regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|OrthogonalMatchingPursuit`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_nonzero_coefs>**, ***int, optional field***, represents the desired number of non-zero entries in the solution. If None, this value is set to 10% of n_features.
  Default: None
- **<tol>**, ***float, optional field***, specifies the maximum norm of the residual. If not None, overrides n_nonzero_coefs.
  Default: None
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True
- **<precompute>**, {***True, False, 'auto'***}, specifies whether to use a precomputed Gram and Xy matrix to speed up calculations. Improves performance when n_targets or n_samples is very large. **Note:** If you already have such matrices, you can pass them directly to the fit method.
  Default: 'auto'

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *OrthogonalMatchingPursuit* ROM.

17.3.6.1.17 Cross-validated Orthogonal Mathching Pursuit model (OMP)

The *Cross-validated Orthogonal Mathching Pursuit model (OMP)* is a regressor similar to OMP which has good performance in sparse recovery. In order to use the *Cross-validated Orthogonal Mathching Pursuit model (OMP) regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|OrthogonalMatchingPursuitCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: None Maximum number of iterations to perform, therefore maximum features to include 10% of n_features but at least 5 if available.

- **<cv>**, ***cross-validation generator, optional***, see sklearn.cross_validation.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *OrthogonalMatchingPursuitCV* ROM.

17.3.6.1.18 Passive Aggressive Classifier

The *Passive Aggressive Classifier* is a principled approach to linear classification that advocates minimal weight updates i.e., the least required to correctly classify the current training instance. In order to use the *Passive Aggressive Classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|PassiveAggressiveClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float***, specifies the maximum step size (regularization).
  Default: 1.0

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<loss>**, ***string, optional field***, the loss function to be used:

  – hinge: equivalent to PA-I (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf)

  – squared_hinge: equivalent to PA-II (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06

  Default: 'hinge'

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveClassifier* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{9}$$

17.3.6.1.19 Passive Aggressive Regressor

The *Passive Aggressive Regressor* is similar to the Perceptron in that it does not require a learning rate. However, contrary to the Perceptron, this regressor includes a regularization parameter, C.

In order to use the *Passive Aggressive Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|PassiveAggressiveRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float***, sets the maximum step size (regularization).
  Default: 1.0

- **<epsilon>**, ***float***, if the difference between the current prediction and the correct label is below this threshold, the model is not updated.
  Default: 0.1

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<loss>**, ***string, optional field***, specifies the loss function to be used:

  - epsilon_insensitive: equivalent to PA-I in the reference paper (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf).
  - squared_epsilon_insensitive: equivalent to PA-II in the reference paper (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf).

  Default: 'epsilon_insensitive'

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveRegressor* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{10}$$

17.3.6.1.20 Perceptron

The *Perceptron* method is an algorithm for supervised classification of an input into one of several possible non-binary outputs. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time. In order to use the *Perceptron classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|Perceptron`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<penalty>**, ***None, 'l2' or 'l1' or 'elasticnet'***, defines the penalty (aka regularization term) to be used.
  Default: None
- **<alpha>**, ***float***, sets the constant multiplier for the regularization term if regularization is used.
  Default: 0.0001
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5
- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True
- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: 0
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0
- **<eta0>**, ***double, optional field***, defines the constant multiplier for the updates.
  Default: 1.0
- **<class_weight>**, ***dict,*** $\{$***class_label: weight***$\}$ ***or "balanced" or None, optional*** Preset for the class_weight fit parameter. Weights associated with classes. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically

adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveRegressor* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{11}$$

17.3.6.1.21 Linear least squares with l2 regularization

The *Linear least squares with l2 regularization* solves a regression model where the loss function is the linear least squares function and the regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multivariate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]). In order to use the *Linear least squares with l2 regularization*, the user needs to set the sub-node:

<SKLtype>`linear_model|Ridge`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, array-like***, shape = [n_targets] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-}1$ in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.
  Default: 1.0

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: determined by scipy.sparse.linalg.

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<solver>**, {***'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'***}, specifies the solver to use in the computational routines:
  - 'auto' chooses the solver automatically based on the type of data.
  - 'svd' uses a singular value decomposition of X to compute the ridge coefficients. More stable for singular matrices than 'cholesky.'
  - 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
  - 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set tol and max_iter).
  - 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fatest but may not be available in old scipy versions. It also uses an iterative procedure.

  All three solvers support both dense and sparse data.
  Default: 'auto'

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *Ridge* ROM.

17.3.6.1.22 Classifier using Ridge regression

The *Classifier using Ridge regression* is a classifier based on linear least squares with l2 regularization. In order to use the *Classifier using Ridge regression*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float***, small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-}1$ in other linear models such as LogisticRegression or LinearSVC.
  Default: 1.0
- **<class_weight>**, ***dict, optional field***, specifies weights associated with classes in the form class_label: weight. If not given, all classes are assumed to have weight one.
  Default: None
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is

expected to be already centered).
Default: True

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: determined by scipy.sparse.linalg.

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: False

- **<solver>**, {***'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'***}, specifies the solver to use in the computational routines:

  – 'auto' chooses the solver automatically based on the type of data.
  – 'svd' uses a singular value decomposition of X to compute the ridge coefficients. More stable for singular matrices than 'cholesky.'
  – 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
  – 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set tol and max_iter).
  – 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fatest but may not be available in old scipy versions. It also uses an iterative procedure.

  All three solvers support both dense and sparse data.
  Default: 'auto'

- **<tol>**, ***float***, defines the required precision of the solution.
Default: 0.001

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeClassifier* ROM.

17.3.6.1.23 Ridge classifier with built-in cross-validation

The *Ridge classifier with built-in cross-validation* performs Generalized Cross-Validation, which is a form of efficient leave-one-out cross-validation. Currently, only the n_features $>$ n_samples case is handled efficiently. In order to use the *Ridge classifier with built-in cross-validation classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeClassifierCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alphas>**, ***numpy array of shape [n_alphas]***, is an array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-1}$ in other linear models such as LogisticRegression or LinearSVC.
  Default: (0.1, 1.0, 10.0)

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<scoring>**, ***string, callable or None, optional***, is a string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y).
  Default: None

- **<cv>**, ***cross-validation generator, optional***, If None, Generalized Cross-Validation (efficient leave-one-out) will be used.
  Default: None

- **<class_weight>**, ***dic, optional field***, specifies weights associated with classes in the form class_label:weight. If not given, all classes are supposed to have weight one.
  Default: None

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeClassifierCV* ROM.

17.3.6.1.24 Ridge regression with built-in cross-validation

The *Ridge regression with built-in cross-validation* performs Generalized Cross-Validation, which is a form of efficient leave-one-out cross-validation. In order to use the *Ridge regression with built-in cross-validation regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alphas>**, ***numpy array of shape [n_alphas]***, specifies an array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the vari-

ance of the estimates. Alpha corresponds to $(2 * C)^{-1}$ in other linear models such as LogisticRegression or LinearSVC.
Default: (0.1, 1.0, 10.0)

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: False

- **<scoring>**, ***string, callable or None, optional***, is a string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y).
Default: None

- **<cv>**, ***cross-validation generator, optional field***, if None, Generalized Cross-Validation (efficient leave-one-out) will be used.
Default: None

- **<gcv_mode>**, {***None, 'auto,' 'svd,' 'eigen'***}***, optional field***, is a flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

  - 'auto:' use svd if n_samples ¿ n_features or when X is a sparse matrix, otherwise use eigen
  - 'svd:' force computation via singular value decomposition of X (does not work for sparse matrices)
  - 'eigen:' force computation via eigendecomposition of $X^T X$

  The 'auto' mode is the default and is intended to pick the cheaper option of the two depending upon the shape and format of the training data.
  Default: None

- **<store_cv_values>**, ***boolean***, is a flag indicating if the cross-validation values corresponding to each alpha should be stored in the cv_values_attribute (see below). This flag is only compatible with cv=None (i.e. using Generalized Cross-Validation).
Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeCV* ROM.

17.3.6.1.25 Linear classifiers (SVM, logistic regression, a.o.) with SGD training

The *Linear classifiers (SVM, logistic regression, a.o.) with SGD training* implements regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated for each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). SGD allows minibatch (online/out-of-core) learning, see the partial_fit method. This implementation works with data represented as dense or sparse arrays of floating point values for the features. The model it fits can be controlled with the loss parameter; by default, it fits a linear support vector machine (SVM). The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared Euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieves online feature selection. In order to use the *Linear classifiers (SVM, logistic regression, a.o.) with SGD training*, the user needs to set the sub-node:

<SKLtype>`linear_model|SGDClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<loss>**, ***str, 'hinge,' 'log,' 'modified_huber,' 'squared_hinge,' 'perceptron,' or a regression loss: 'squared_loss,' 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive'***, dictates the loss function to be used. The available options are:
  - 'hinge' gives a linear SVM.
  - 'log' loss gives logistic regression, a probabilistic classifier.
  - 'modified_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates.
  - 'squared_hinge' is like hinge but is quadratically penalized.
  - 'perceptron' is the linear loss used by the perceptron algorithm.

  The other losses are designed for regression but can be useful in classification as well; see SGDRegressor for a description.
  Default: 'hinge'
- **<penalty>**, ***str, 'l2' or 'l1' or 'elasticnet'***, defines the penalty (aka regularization term) to be used. 'l2' is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2.'
  Default: 'l2'
- **<alpha>**, ***float***, is the constant multiplier for the regularization term.
  Default: 0.0001
- **<l1_ratio>**, ***float***, is the Elastic Net mixing parameter, with 0 ¡= l1_ratio ¡= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1.
  Default: 0.15

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<epsilon>**, ***float, optional field***, varies meaning depending on the value of **<loss>**. If loss is 'huber', 'epsilon_insensitive' or 'squared_epsilon_insensitive' then this is the epsilon in the epsilon-insensitive loss functions. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For 'epsilon_insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.
  Default: 0.1

- **<learning_rate>**, ***string, optional field***, specifies the learning rate:

  - 'constant:' eta = eta0
  - 'optimal:' eta = 1.0 / (t + t0)
  - 'invscaling:' eta = eta0 / pow(t, power_t)

  Default: 'optimal'

- **<eta0>**, ***double***, specifies the initial learning rate for the 'constant' or 'invscaling' schedules. The default value is 0.0 as eta0 is not used by the default schedule 'optimal.'
  Default: 0.0

- **<power_t>**, ***double***, represents the exponent for the inverse scaling learning rate.
  Default: 0.5

- **<class_weight>**, ***dict, class_label***, is the preset for the class_weight fit parameter. Weights associated with classes. If not given, all classes are assumed to have weight one. The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
  Default: None

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SGDClassifier* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{12}$$

17.3.6.1.26 Linear model fitted by minimizing a regularized empirical loss with SGD

The *Linear model fitted by minimizing a regularized empirical loss with SGD* is a model where SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieving online feature selection. This implementation works with data represented as dense numpy arrays of floating point values for the features. In order to use the *Linear model fitted by minimizing a regularized empirical loss with SGD*, the user needs to set the sub-node:

<SKLtype>`linear_model|SGDRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<loss>**, ***str, 'squared_loss,' 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive'***, specifies the loss function to be used. Defaults to 'squared_loss' which refers to the ordinary least squares fit. 'huber' modifies 'squared_loss' to focus less on getting outliers correct by switching from squared to linear loss past a distance of epsilon. 'epsilon_insensitive' ignores errors less than epsilon and is linear past that; this is the loss function used in SVR. 'squared_epsilon_insensitive' is the same but becomes squared loss past a tolerance of epsilon.
  Default: 'squared_loss'

- **<penalty>**, ***str, 'l2' or 'l1' or 'elasticnet'***, sets the penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.
  Default: 'l2'

- **<alpha>**, ***float***, Constant that multiplies the regularization term. Defaults to 0.0001

- **<l1_ratio>**, ***float***, is the Elastic Net mixing parameter, with $0 \leq l1\_ratio \leq 1$. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1.
  Default: 0.15

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True

- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<epsilon>**, ***float***, sets the epsilon in the epsilon-insensitive loss functions; only if loss is 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive.' For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.
  Default: 0.1

- **<learning_rate>**, ***string, optional field***, Learning rate:

  - constant: eta = eta0
  - optimal: eta = 1.0/(t+t0)
  - invscaling: eta= eta0 / pow(t, power_t)

  Default: invscaling

- **<eta0>**, ***double***, specifies the initial learning rate.
  Default: 0.01

- **<power_t>**, ***double, optional field***, specifies the exponent for inverse scaling learning rate.

  Default: 0.25

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SGDRegressor* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{13}$$

17.3.6.2 Support Vector Machines

In machine learning, **Support Vector Machines** (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.
It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SVM-based* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{14}$$

In the following, all the SVM models available in RAVEN are reported.

17.3.6.2.1 Linear Support Vector Classifier

The *Linear Support Vector Classifier* is similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better (to large numbers of samples). This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme. In order to use the *Linear Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|LinearSVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term.
  Default: 1.0
- **<loss>**, ***string, 'hinge' or 'squared_hinge'***, specifies the loss function. 'hinge' is the hinge loss (standard SVM) while 'squared_hinge' is the squared hinge loss.
  Default: 'squared_hinge'
- **<penalty>**, ***string, 'l1' or 'l2'***, specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to coef_vectors that are sparse.
  Default: 'l2'
- **<dual>**, ***boolean***, selects the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples $>$ n_features.
  Default: True
- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-4
- **<multi_class>**, ***string, 'ovr' or 'crammer_singer'***, Determines the multi-class strategy if y contains more than two classes. ovr trains n_classes one-vs-rest classifiers, while crammer_singer optimizes a joint objective over all classes. While crammer_singer is interesting from a theoretical perspective as it is consistent, it is seldom used in practice and rarely leads to better accuracy and is more expensive to compute. If crammer_singer is chosen, the options loss, penalty and dual will be ignored.
  Default: 'ovr'
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<intercept_scaling>**, ***float, optional field***, when True, the instance vector x becomes [x,self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling

is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight. **Note:** The synthetic feature weight is subject to l1/l2 regularization as are all other features. To lessen the effect of regularization on the synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.
Default: 1

- **<class_weight>**, ***dict, 'auto', optional***, sets the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are assumed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
Default: 0 **Note:** This setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
Default: None

17.3.6.2.2 C-Support Vector Classification

The *C-Support Vector Classification* is a based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples. The multiclass support is handled according to a one-vs-one scheme. In order to use the *C-Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|SVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term.
Default: 1.0

- **<kernel>**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:
  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'
  - 'precomputed'
  - a callable object

If a callable is given it is used to pre-compute the kernel matrix.
Default: 'rbf'

- **<degree>**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
Default: 3.0

- **<gamma>**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead.
Default: 'auto'

- **<coef0>**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
Default: 0.0

- **<probability>**, ***boolean, optional field***, determines whether or not to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.
Default: False

- **<shrinking>**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
Default: True

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
Default: 1e-3

- **<cache_size>**, ***float, optional field***, specifies the size of the kernel cache (in MB).

- **<class_weight>**, ***dict, 'auto', optional***, sets the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are assumed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
Default: None

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: -1

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
Default: None

17.3.6.2.3 Nu-Support Vector Classification

The *Nu-Support Vector Classification* is similar to SVC but uses a parameter to control the number of support vectors. The implementation is based on libsvm. In order to use the *Nu-Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|NuSVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<nu>**, ***float, optional field***, is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].
  Default: 0.5
- **<kernel>**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:
  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'
  - 'precomputed'
  - a callable object

  If a callable is given it is used to pre-compute the kernel matrix.
  Default: 'rbf'
- **<degree>**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
  Default: 3
- **<gamma>**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead.
  Default: 'auto'
- **<coef0>**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
  Default: 0.0
- **<probability>**, ***boolean, optional field***, determines whether or not to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.
  Default: False

- **<shrinking>**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
  Default: True

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-3

- **<cache_size>**, ***float, optional field***, specifies the size of the kernel cache (in MB).

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: -1

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
  Default: None

17.3.6.2.4 Support Vector Regression

The *Support Vector Regression* is an epsilon-Support Vector Regression. The free parameters in this model are C and epsilon. The implementations is a based on libsvm. In order to use the *Support Vector Regressor*, the user needs to set the sub-node:

<SKLtype>`svm|SVR`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term.
  Default: 1.0

- **<epsilon>**, ***float, optional field***, specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.
  Default: 0.1

- **<kernel>**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:

  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'

- 'precomputed'
- a callable object

If a callable is given it is used to pre-compute the kernel matrix.
Default: 'rbf'

- **<degree>**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
Default: 3.0
- **<gamma>**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead.
Default: 'auto'
- **<coef0>**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
Default: 0.0
- **<shrinking>**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
Default: True
- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
Default: 1e-3
- **<cache_size>**, ***float, optional field***, specifies the size of the kernel cache (in MB).
- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
Default: -1

17.3.6.3 Multi Class

Multiclass classification means a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *multi-class-based* ROM.

In the following, all the multi-class models available in RAVEN are reported.

17.3.6.3.1 One-vs-the-rest (OvR) multiclass/multilabel strategy

The *One-vs-the-rest (OvR) multiclass/multilabel strategy*, also known as one-vs-all, consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only n_classes classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy and is a fair default choice.

In order to use the *One-vs-the-rest (OvR) multiclass/multilabel classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OneVsRestClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<estimator>**, ***boolean, required field***, An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the following attribute:
  - **estimatorType**, ***required string attribute***, this attribute is another reduced order mode type that needs to be used for the construction of the multi-class algorithms. Each sub-sequential node depends on the chosen ROM.

17.3.6.3.2 One-vs-one multiclass strategy

The *One-vs-one multiclass strategy* consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit n_classes * (n_classes - 1) / 2 classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which do not scale well with n_samples. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used n_classes times.

In order to use the *One-vs-one multiclass classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OneVsOneClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<estimator>**, ***boolean, required field***, An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the following attribute:
  - **estimatorType**, ***required string attribute***, this attribute is another reduced order mode type that needs to be used for the construction of the multi-class algorithms.

Each sub-sequential node depends on the chosen ROM.

17.3.6.3.3 Error-Correcting Output-Code multiclass strategy

The *Error-Correcting Output-Code multiclass strategy* consists in representing each class with a binary code (an array of 0s and 1s). At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen. The main advantage of these strategies is that the number of classifiers used can be controlled by the user, either for compressing the model ($0 < code\_size < 1$) or for making the model more robust to errors ($code\_size > 1$).

In order to use the *Error-Correcting Output-Code multiclass classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OutputCodeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<estimator>**, ***boolean, required field***, An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the following attribute:
  - **estimatorType**, ***required string attribute***, this attribute is another reduced order mode type that needs to be used for the construction of the multi-class algorithms. Each sub-sequential node depends on the chosen ROM.
- **<code_size>**, ***float, required field***, represents the percentage of the number of classes to be used to create the code book. A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. A number greater than 1 will require more classifiers than one-vs-the-rest.

17.3.6.4 Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)} \tag{15}$$

Using the naive independence assumption that

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y), \tag{16}$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)} \tag{17}$$

Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \Downarrow \tag{18}$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y), \tag{19}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set. The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.) Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from predict_proba are not to be taken too seriously. In the following, all the Naive Bayes available in RAVEN are reported.

17.3.6.4.1 Gaussian Naive Bayes

The *Gaussian Naive Bayes strategy* implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{20}$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

In order to use the *Gaussian Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>`naiveBayes|GaussianNB`**</SKLtype>**.

There are no additional sub-nodes available for this method.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *GaussianNB* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{21}$$

17.3.6.4.2 Multinomial Naive Bayes

The *Multinomial Naive Bayes* implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data is typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y. The parameters θ_y are estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n} \tag{22}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y. The smoothing priors $\alpha \geq 0$ account for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing. In order to use the *Multinomial Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>`naiveBayes|MultinomialNB`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies an additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
  Default: 1.0
- **<fit_prior>**, ***boolean, required field***, determines whether to learn class prior probabilities or not. If false, a uniform prior will be used.
  Default: True
- **<class_prior>**, ***array-like float (n_classes), optional field***, specifies prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
  Default: None

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultinomialNB* ROM.

17.3.6.4.3 Bernoulli Naive Bayes

The *Bernoulli Naive Bayes* implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore,

this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a *Bernoulli Naive Bayes* instance may binarize its input (depending on the binarize parameter). The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i) \tag{23}$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature i that is an indicator for class y, where the multinomial variant would simply ignore a non-occurring feature. In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. *Bernoulli Naive Bayes* might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits. In order to use the *Bernoulli Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>naiveBayes|BernoulliNB**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies an additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
  Default: 1.0
- **<binarize>**, ***float, optional field***, Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.
  Default: 0.0
- **<fit_prior>**, ***boolean, required field***, determines whether to learn class prior probabilities or not. If false, a uniform prior will be used.
  Default: True
- **<class_prior>**, ***array-like float (n_classes), optional field***, specifies prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
  Default: None

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *BernoulliNB* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{24}$$

17.3.6.5 Neighbors

The *Neighbors* class provides functionality for unsupervised and supervised neighbor-based learning methods. The unsupervised nearest neighbors method is the foundation of many other learning

methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbor-based methods are known as non-generalizing machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree.).

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *Neighbors-based* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{25}$$

In the following, all the Neighbors' models available in RAVEN are reported.

17.3.6.5.1 K Neighbors Classifier

The *K Neighbors Classifier* is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. It implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user.

In order to use the *K Neighbors Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|KNeighborsClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_neighbors>**, ***integer, optional field***, specifies the number of neighbors to use by default for 'k_neighbors' queries.
  Default: 5
- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;

- *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

Default: uniform

- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto

- **<leaf_size>**, ***integer, optional field***, sets the leaf size passed to the BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
  Default: 30

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

- **<p>**, ***integer, optional field***, is a parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used.
  Default: 2

17.3.6.5.2 Radius Neighbors Classifier

The *Radius Neighbors Classifier* is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. It implements learning based on the number of neighbors within a fixed radius r of each training point, where r is a floating-point value specified by the user.

In order to use the *Radius Neighbors Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|RadiusNeighbors`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<radius>**, ***float, optional field***, specifies the range of parameter space to use by default for 'radius_neighbors' queries.
Default: 1.0

- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:

  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default: uniform

- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:

  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto

- **<leaf_size>**, ***integer, optional field***, sets the leaf size passed to the BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
Default: 30

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
Default: minkowski

- **<p>**, ***integer, optional field***, is a parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used.
Default: 2

- **<outlier_label>**, ***integer, optional field***, is a label, which is given for outlier samples (samples with no neighbors on a given radius). If set to None, ValueError is raised, when an outlier is detected.
Default: None

17.3.6.5.3 K Neighbors Regressor

The *K Neighbors Regressor* can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors. It implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user.

In order to use the *K Neighbors Regressor*, the user needs to set the sub-node:

<SKLtype>`neighbors|KNeighborsRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_neighbors>**, ***integer, optional field***, specifies the number of neighbors to use by default for 'k_neighbors' queries.
  Default: 5
- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default: uniform
- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto
- **<leaf_size>**, ***integer, optional field***, sets the leaf size passed to the BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
  Default: 30

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

- **<p>**, ***integer, optional field***, is a parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used.
  Default: 2

17.3.6.5.4 Radius Neighbors Regressor

The *Radius Neighbors Regressor* can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors. It implements learning based on the neighbors within a fixed radius r of the query point, where r is a floating-point value specified by the user.

In order to use the *Radius Neighbors Regressor*, the user needs to set the sub-node:

<SKLtype>`neighbors|RadiusNeighborsRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<radius>**, ***float, optional field***, specifies the range of parameter space to use by default for 'radius_neighbors' queries.
  Default: 1.0

- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:

  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default: uniform

- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:

  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

Note: Fitting on sparse input will override the setting of this parameter, using brute force.
Default: auto

- **<leaf_size>**, ***integer, optional field***, sets the leaf size passed to the BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
Default: 30

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
Default: minkowski

- **<p>**, ***integer, optional field***, is a parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used.
Default: 2

17.3.6.5.5 Nearest Centroid Classifier

The *Nearest Centroid classifier* is a simple algorithm that represents each class by the centroid of its members. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed.

In order to use the *Nearest Centroid Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|NearestCentroid`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<shrink_threshold>**, ***float, optional field***, defines the threshold for shrinking centroids to remove features.
Default: None

The *Quadratic Discriminant Analysis* is a classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class.

In order to use the *Quadratic Discriminant Analysis Classifier*, the user needs to set the sub-node:

<SKLtype>`qda|QDA`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<priors>**, ***array-like (n_classes), optional field***, specifies the priors on the classes.
  Default: None
- **<reg_param>**, ***float, optional field***, regularizes the covariance estimate as (1-reg_param)*Sigma + reg_param*Identity(n_features).
  Default: 0.0

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *QDA* ROM.

17.3.6.6 Tree

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

- Some advantages of decision trees are:
- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however, that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialized in analyzing datasets that have only one type of variable.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *tree-based* ROM:

$$\mathbf{X} = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{26}$$

In the following, all the tree-based algorithms available in RAVEN are reported.

17.3.6.6.1 Decision Tree Classifier

The *Decision Tree Classifier* is a classifier that is based on the decision tree logic.

In order to use the *Decision Tree Classifier*, the user needs to set the sub-node:

<SKLtype>`tree|DecisionTreeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***string, optional field***, specifies the function used to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.
  Default: gini

- **<splitter>**, ***string, optional field***, specifies the strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
  Default: best

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto," then max_features=sqrt(n_features).
  - If "sqrt," then max_features=sqrt(n_features).
  - If "log2," then max_features=log2(n_features).
  - If None, then max_features=n_features.

  Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default: None

- **<max_depth>**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **<min_samples_split>**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **<min_samples_leaf>**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **<max_leaf_nodes>**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

17.3.6.6.2 Decision Tree Regressor

The *Decision Tree Regressor* is a Regressor that is based on the decision tree logic. In order to use the *Decision Tree Regressor*, the user needs to set the sub-node:

<SKLtype>`tree|DecisionTreeRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **`<criterion>`**, ***string, optional field***, specifies the function used to measure the quality of a split. The only supported criterion is "mse" for mean squared error.
  Default: mse

- **`<splitter>`**, ***string, optional field***, specifies the strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
  Default: best

- **`<max_features>`**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto," then max_features=sqrt(n_features).
  - If "sqrt," then max_features=sqrt(n_features).
  - If "log2," then max_features=log2(n_features).
  - If None, then max_features=n_features.

  Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default: None

- **`<max_depth>`**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **`<min_samples_split>`**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **`<min_samples_leaf>`**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **`<max_leaf_nodes>`**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

17.3.6.6.3 Extra Tree Classifier

The *Extra Tree Classifier* is an extremely randomized tree classifier. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.

In order to use the *Extra Tree Classifier*, the user needs to set the sub-node:

<SKLtype>`tree|ExtraTreeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***string, optional field***, specifies the function used to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.
  Default: gini

- **<splitter>**, ***string, optional field***, specifies the strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
  Default: random

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto," then max_features=sqrt(n_features).
  - If "sqrt," then max_features=sqrt(n_features).
  - If "log2," then max_features=log2(n_features).
  - If None, then max_features=n_features.

  Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default: auto

- **<max_depth>**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **<min_samples_split>**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **<min_samples_leaf>**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **<max_leaf_nodes>**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

17.3.6.6.4 Extra Tree Regressor

The *Extra Tree Regressor* is an extremely randomized tree regressor. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.

In order to use the *Extra Tree Regressor*, the user needs to set the sub-node:

<SKLtype>`tree|ExtraTreeRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***string, optional field***, specifies the function used to measure the quality of a split. The only supported criterion is "mse" for mean squared error.
  Default: mse

- **<splitter>**, ***string, optional field***, specifies the strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
  Default: random

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto," then max_features=sqrt(n_features).

  - If "sqrt," then max_features=sqrt(n_features).
  - If "log2," then max_features=log2(n_features).
  - If None, then max_features=n_features.

  Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default: auto

- **<max_depth>**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **<min_samples_split>**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **<min_samples_leaf>**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **<max_leaf_nodes>**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

17.3.6.7 Gaussian Process

Gaussian Processes for Machine Learning (GPML) is a generic supervised learning method primarily designed to solve regression problems. The advantages of Gaussian Processes for Machine Learning are:

- The prediction interpolates the observations (at least for regular correlation models).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and exceedance probabilities that might be used to refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different linear regression models and correlation models can be specified. Common models are provided, but it is also possible to specify custom models provided they are stationary.

The disadvantages of Gaussian Processes for Machine Learning include:

- It is not sparse. It uses the whole samples/features information to perform the prediction.
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency.
- Classification is only a post-processing, meaning that one first needs to solve a regression problem by providing the complete scalar float precision output y of the experiment one is attempting to model.

In order to use the *Gaussian Process Regressor*, the user needs to set the sub-node:

<SKLtype>`GaussianProcess|GaussianProcess`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<regr>**, ***string, optional field***, is a regression function returning an array of outputs of the linear regression functional basis. The number of observations n_samples should be greater than the size p of this basis. Available built-in regression models are 'constant,' 'linear,' and 'quadratic.'
  Default: constant
- **<corr>**, ***string, optional field***, is a stationary autocorrelation function returning the autocorrelation between two points x and x'. Default assumes a squared-exponential autocorrelation model. Built-in correlation models are 'absolute_exponential,' 'squared_exponential,' 'generalized_exponential,' 'cubic,' and 'linear.'
  Default: squared_exponential
- **<beta0>**, ***float, array-like, optional field***, specifies the regression weight vector to perform Ordinary Kriging (OK).
  Default: None
- **<storage_mode>**, ***string, optional field***, specifies whether the Cholesky decomposition of the correlation matrix should be stored in the class (storage_mode = 'full') or not (storage_mode = 'light').
  Default: full
- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False
- **<theta0>**, ***float, array-like, optional field***, is an array with shape (n_features,) or (1,). This represents the parameters in the autocorrelation model. If thetaL and thetaU are also specified, theta0 is considered as the starting point for the maximum likelihood estimation of the best set of parameters.
  Default: [1e-1]

- **<thetaL>**, ***float, array-like, optional field***, is an array with shape matching that defined by **<theta0>**. Lower bound on the autocorrelation parameters for maximum likelihood estimation.
  Default: None

- **<thetaU>**, ***float, array-like, optional field***, is an array with shape matching that defined by **<theta0>**. Upper bound on the autocorrelation parameters for maximum likelihood estimation.
  Default: None

- **<normalize>**, ***boolean, optional field***, if True, the input X and observations y are centered and reduced w.r.t. means and standard deviations estimated from the n_samples observations provided.
  Default: True

- **<nugget>**, ***float, optional field***,specifies a nugget effect to allow smooth predictions from noisy data. The nugget is added to the diagonal of the assumed training covariance. In this way it acts as a Tikhonov regularization in the problem. In the special case of the squared exponential correlation function, the nugget mathematically represents the variance of the input values.
  *Default: 10 * MACHINE_EPSILON*

- **<optimizer>**, ***string, optional field***, specifies the optimization algorithm to be used. Available optimizers are: 'fmin_cobyla', 'Welch'.
  Default: fmin_cobyla

- **<random_start>**, ***integer, optional field***, sets the number of times the Maximum Likelihood Estimation should be performed from a random starting point. The first MLE always uses the specified starting point (theta0), the next starting points are picked at random according to an exponential distribution (log-uniform on [thetaL, thetaU]).
  Default: 1

- **<random_state>**, ***integer, optional field***, is the seed of the internal random number generator.
  Default: None

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *GaussianProcess* ROM.

Example:

```
<Simulation>
  ...
  <Models>
```

```
    ...
    <ROM name='aUserDefinedName' subType='SciKitLearn'>
      <Features>var1,var2,var3</Features>
      <Target>result1</Target>
      <SKLtype>linear_model|LinearRegression</SKLtype>
      <fit_intercept>True</fit_intercept>
      <normalize>False</normalize>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

17.3.7 ARMA

The ARMA sub-type contains a single ROM type, based on an autoregressive moving average time series model. ARMA is a type of time dependent model that characterizes the autocorrelation between time series data. The mathematic description of ARMA is given as

$$x_t = \sum_{i=1}^{p} \phi_i x_{t-i} + \alpha_t + \sum_{j=1}^{q} \theta_j \alpha_{t-j},$$

where x is a vector of dimension n, and ϕ_i and θ_j are both n by n matrices. When $q = 0$, the above is autoregressive (AR); when $p = 0$, the above is moving average (MA). The user is allowed to provide upper and lower limits for p and q (see below), and the training process will choose the optimal p and q that fall into the user-specified range. When training ARMA, the input needs to be a synchronized HistorySet. For unsynchronized data, use PostProcessor methods to synchronize the data before training ARMA.

The ARMA model implemented allows an option to use Fourier series to detrend the time series before fitting to ARMA model to train. The Fourier trend will be stored in the trained ARMA model for data generation. The following equation describes the detrend process.

$$x_t = y_t - \sum_{m} \left\{ \sum_{k=1}^{K_m} a_k \sin(2\pi k f_m t) + \sum_{k=1}^{K_m} b_k \cos(2\pi k f_m t) \right\},$$

where K_m and f_m are user-defined parameters.

In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be **'ARMA'** (see the example below). This model can be initialized with the following child:

- **<pivotParameter>**, ***string, optional field***, defines the pivot variable (e.g., time) that is non-decreasing in the input HistorySet.
  Default: Time

- **<Features>**, ***string, required field***, defines the features (e.g., scaling). Note that only one feature is allowed for **'ARMA'** and in current implementation this is used for evaluation only.

- **<Target>**, ***string, required field***, defines the variables of the time series.

- **<Pmax>**, ***integer, optional field***, defines the maximum value of p.
  Default: 3

- **<Pmin>**, ***integer, optional field***, defines the minimum value of p.
  Default: 0

- **<Qmax>**, ***integer, optional field***, defines the maximum value of q.
  Default: 3

- **<Qmin>**, ***integer, optional field***, defines the minimum value of q.
  Default: 0

- **<Fourier>**, ***integers, optional field***, must be positive integers. This defines the based period (with unit of second) that would be used for Fourier detrending, i.e., this field defines $1/f_m$ in the above equation. When this filed is not specified, the ARMA considers no Fourier detrend.

- **<FourierOrder>**, ***integers, optional field***, must be positive integers. The number of integers specified in this field should be exactly same as the number of base periods specified in the node **<Fourier>**. This field defines K_m in the above equation.

- **<outTruncation>**, ***string, optional field***, defines whether and how the output time series is truncated. Currently available options are: positive, negative.
  Default: None

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='ARMA'>
      <pivotParameter>Time</pivotParameter>
      <Features>scaling</Features>
      <Target>Speed1,Speed2</Target>
      <Pmax>5</Pmax>
```

```
      <Pmin>1</Pmin>
      <Qmax>4</Qmax>
      <Qmin>1</Qmin>
      <Fourier>604800,86400</Fourier>
      <FourierOrder>2, 4</FourierOrder>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

17.4 External Model

As the name suggests, an external model is an entity that is embedded in the RAVEN code at run time. This object allows the user to create a python module that is going to be treated as a predefined internal model object. In other words, the **External Model** is going to be treated by RAVEN as a normal external Code (e.g. it is going to be called in order to compute an arbitrary quantity based on arbitrary input).

The specifications of an External Model must be defined within the XML block **<ExternalModel>**. This XML node needs to contain the attributes:

- **name**, ***required string attribute***, user-defined name of this External Model. **Note:** As with the other objects, this is the name that can be used to refer to this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, must be kept empty.
- **ModuleToLoad**, ***required string attribute***, file name with its absolute or relative path. **Note:** If a relative path is specified, the code first checks relative to the working directory, then it checks with respect to where the user runs the code. Using the relative path with respect to where the code is run is not recommended.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in her/his own python Module, the variables need to be specified in the **<ExternalModel>** input block. The user needs to input, within this block, only the variables that RAVEN needs to be aware of (i.e. the variables are going to directly be used by the code) and not the local variables that the user does not want to, for example, store in a RAVEN internal object. These variables are specified within a **<variables>** block:

- **<variables>**, ***string, required parameter***. Comma-separated list of variable names. Each variable name needs to match a variable used/defined in the external python model.

In addition, if the user wants to use the alias system, the following XML block can be inputted:

- **<alias>** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the ExternalModel. These aliases can be used anywhere in the RAVEN input to refer to the ExternalModel variables. In the body of this node the user specifies the name of the variable that the model is going to use (during its execution). The actual alias, usable throughout the RAVEN input, is instead defined in the **variable** attribute of this tag.
  The user can specify aliases for both the input and the output space. As sanity check, RAVEN requires an additional required attribute **type**. This attribute can be either "input" or "output". **Note:** The user can specify as many aliases as needed.
  Default: None

When the external function variables are defined, at run time, RAVEN initializes them and tracks their values during the simulation. Each variable defined in the **<ExternalModel>** block is available in the module (each method implemented) as a python "self."

In the External Python module, the user can implement all the methods that are needed for the functionality of the model, but only the following methods, if present, are called by the framework:

- **def _readMoreXML**, ***OPTIONAL METHOD***, can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in "self" in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method).
- **def initialize**, ***OPTIONAL METHOD***, can implement all the actions need to be performed at the initialization stage.
- **def createNewInput**, ***OPTIONAL METHOD***, creates a new input with the information coming from the RAVEN framework. In this function the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method.
- **def run**, ***REQUIRED METHOD***, is the actual location where the user needs to implement the model action (e.g. resolution of a set of equations, etc.). This function is going to receive the Input (or Inputs) generated either by the External Model "createNewInput" method or the internal RAVEN one.

In the following sub-sections, all the methods are going to be analyzed in detail.

17.4.1 Method: def _readMoreXML

As already mentioned, the **readMoreXML** method can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in "self" in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method). If this method is implemented in the **External Model**, RAVEN is going to call it when the node **<ExternalModel>** is found parsing the XML input file. The method receives from RAVEN an attribute of type "xml.etree.ElementTree", containing all the sub-nodes and attribute of the XML block **<ExternalModel>**.

Example XML:

```
<Simulation>
  ...
  <Models>
     ...
    <ExternalModel name='AnExtModule' subType=''
       ModuleToLoad='path_to_external_module'>
       <variables>sigma,rho,outcome</variables>
       <!--
          here we define other XML nodes RAVEN does not read
             automatically.
          We need to implement, in the external module
             'AnExtModule' the readMoreXML method
        -->
        <newNodeWeNeedToRead>
            whatNeedsToBeRead
        </newNodeWeNeedToRead>
    </ExternalModel>
     ...
  </Models>
  ...
</Simulation>
```

Corresponding Python function:

```
def _readMoreXML(self,xmlNode):
  # the xmlNode is passed in by RAVEN framework
  # <newNodeWeNeedToRead> is unknown (in the RAVEN framework)
  # we have to read it on our own
  # get the node
  ourNode = xmlNode.find('newNodeWeNeedToRead')
  # get the information in the node
```

```
self.ourNewVariable = ourNode.text
# end function
```

17.4.2 Method: `def initialize`

The **initialize** method can be implemented in the **External Model** in order to initialize some variables needed by it. For example, it can be used to compute a quantity needed by the "run" method before performing the actual calculation). If this method is implemented in the **External Model**, RAVEN is going to call it at the initialization stage of each "Step" (see section 20. RAVEN will communicate, thorough a set of method attributes, all the information that are generally needed to perform a initialization:

- runInfo, a dictionary containing information regarding how the calculation is set up (e.g. number of processors, etc.). It contains the following attributes:
  - `DefaultInputFile` – default input file to use
  - `SimulationFiles` – the xml input file
  - `ScriptDir` – the location of the pbs script interfaces
  - `FrameworkDir` – the directory where the framework is located
  - `WorkingDir` – the directory where the framework should be running
  - `TempWorkingDir` – the temporary directory where a simulation step is run
  - `NumMPI` – the number of mpi process by run
  - `NumThreads` – number of threads by run
  - `numProcByRun` – total number of core used by one run (number of threads by number of mpi)
  - `batchSize` – number of contemporaneous runs
  - `ParallelCommand` – the command that should be used to submit jobs in parallel (mpi)
  - `numNode` – number of nodes
  - `procByNode` – number of processors by node
  - `totalNumCoresUsed` – total number of cores used by driver
  - `queueingSoftware` – queueing software name
  - `stepName` – the name of the step currently running
  - `precommand` – added to the front of the command that is run
  - `postcommand` – added after the command that is run

- `delSucLogFiles` – if a simulation (code run) has not failed, delete the relative log file (if True)
- `deleteOutExtension` – if a simulation (code run) has not failed, delete the relative output files with the listed extension (comma separated list, for example: 'e,r,txt')
- `mode` – running mode, curently the only mode supported is mpi (but custom modes can be created)
- *expectedTime* – how long the complete input is expected to run
- *logfileBuffer* – logfile buffer size in bytes

- inputs, a list of all the inputs that have been specified in the "Step" using this model.

In the following an example is reported:

```
def initialize(self,runInfo,inputs):
 # Let's suppose we just need to initialize some variables
  self.sigma = 10.0
  self.rho   = 28.0
  # end function
```

17.4.3 Method: `def createNewInput`

The **createNewInput** method can be implemented by the user to create a new input with the information coming from the RAVEN framework. In this function, the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method. The new input created needs to be returned to RAVEN (i.e. "return NewInput"). RAVEN communicates, thorough a set of method attributes, all the information that are generally needed to create a new input:

- `inputs`, ***python list***, a list of all the inputs that have been defined in the "Step" using this model.
- `samplerType`, ***string***, the type of Sampler, if a sampling strategy is employed; will be None otherwise.
- `Kwargs`, ***dictionary***, a dictionary containing several pieces of information (that can change based on the "Step" type). If a sampling strategy is employed, this dictionary contains another dictionary identified by the keyword "SampledVars", in which the variables perturbed by the sampler are reported.

Note: If the "Step" that is using this Model has as input(s) an object of main class type "DataObjects" (see Section 14), the internal "createNewInput" method is going to convert it in a dictionary of values.

Here we present an example:

```
def createNewInput(self,inputs,samplerType,**Kwargs):
  # in here the actual createNewInput of the
  # model is implemented
  if samplerType == 'MonteCarlo':
    avariable = inputs['something']*inputs['something2']
  else:
    avariable = inputs['something']/inputs['something2']
  return avariable*Kwargs['SampledVars']['aSampledVar']
```

17.4.4 Method: `def run`

As stated previously, the only method that *must* be present in an External Module is the **run** function. In this function, the user needs to implement the algorithm that RAVEN will execute. The **run** method is generally called after having inquired the "createNewInput" method (either the internal or the user-implemented one). The only attribute this method is going to receive is a Python list of inputs (the inputs coming from the `createNewInput` method). If the user wants RAVEN to collect the results of this method, the outcomes of interest need to be stored in "self." **Note:** RAVEN is trying to collect the values of the variables listed only in the **<ExternalModel>** XML block.

In the following an example is reported:

```
def run(self,Input):
  # in here the actual run of the
  # model is implemented
  input = Input[0]
  self.outcome = self.sigma*self.rho*input[``whatEver'']
```

17.5 PostProcessor

A Post-Processor (PP) can be considered as an action performed on a set of data or other type of objects. Most of the post-processors contained in RAVEN, employ a mathematical operation on the data given as "input". RAVEN supports several different types of PPs.

Currently, the following types are available in RAVEN:

- **BasicStatistics**
- **ComparisonStatistics**
- **ImportanceRank**
- **SafestPoint**
- **LimitSurface**
- **LimitSurfaceIntegral**
- **External**
- **TopologicalDecomposition**
- **RavenOutput**
- **DataMining**

The specifications of these types must be defined within the XML block **`<PostProcessor>`**. This XML node needs to contain the attributes:

- **`name`**, ***required string attribute***, user-defined identifier of this post-processor. **Note:** As with other objects, this is the name that can be used to refer to this specific entity from other input XML blocks.
- **`subType`**, ***required string attribute***, defines which of the post-processors needs to be used, choosing among the previously reported types. This choice conditions the subsequent required and/or optional **`<PostProcessor>`** sub nodes.

As already mentioned, all the types and meaning of the remaining sub-nodes depend on the post-processor type specified in the attribute **`subType`**. In the following sections the specifications of each type are reported.

17.5.1 BasicStatistics

The **BasicStatistics** post-processor is the container of the algorithms to compute many of the most important statistical quantities. It is important to notice that this post-processor can accept as input both ***PointSet*** and ***HistorySet*** data objects, depending on the type of statistics the user wants to compute:

- ***PointSet***: Static Statistics;
- ***HistorySet***: Dynamic Statistics. Depending on a "pivot parameter" (e.g. time) the post-processor is going to compute the statistics for each value of it (e.g. for each time step). In case an **HistorySet** is provided as Input, the Histories needs to be synchronized (use ***Interfaced*** post-processor of type **HistorySetSync**).

In order to use the *BasicStatistics post-processor* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor subType='BasicStatistics'/>.

Several sub-nodes are available:

- **<(metric)>**, ***comma separated string or node list, required field***, specifications for the metric to be calculated. The name of each node is the requested metric. There are two forms for specifying the requested parameters of the metric. For scalar values such as **<expectedValue>** and **<variance>**, the text of the node is a comma-separated list of the parameters for which the metric should be calculated. For matrix values such as **<sensitivty>** and **<covariance>**, the matrix node requires two sub-nodes, **<targets>** and **<features>**, each of which is a comma-separated list of the targets for which the metric should be calculated, and the features for which the metric should be calculated for that target. See the example below.

  Note: When defining the metrics to use, it is possible to have multiple nodes with the same name. For example, if a problem has inputs W, X, Y, and Z, and the responses are A, B, and C, it is possible that the desired metrics are the **<sensitivity>** of A and B to X and Y, as well as the **<sensitivity>** of C to W and Z, but not the sensitivity of A to W. In this event, two copies of the **<sensitivity>** node are added to the input. The first has targets A, B and features X, Y, while the second node has target C and features W, Z. This could reduce some computation effort in problems with many responses or inputs. An example of this is shown below.
  Currently the scalar quantities available for request are:

  - **expectedValue**: expected value or mean
  - **minimum**: The minimum value of the samples.
  - **maximum**: The maximum value of the samples.
  - **median**: median
  - **variance**: variance
  - **sigma**: standard deviation
  - **percentile**: the percentile. If this quantity is inputted as *percentile* the 5% and 95% percentile(s) are going to be computed. Otherwise the user can specify this quantity as *percentile*_$X\%$, where *X* represents the requested percentile (an integer value between 1 and 100)
  - **variationCoefficient**: coefficient of variation, i.e. **sigma/expectedValue**. **Note:** If the **expectedValue** is zero, the **variationCoefficient** will be **INF**.
  - **skewness**: skewness
  - **kurtosis**: excess kurtosis (also known as Fisher's kurtosis)

The matrix quantities available for request are:

- **sensitivity**: matrix of sensitivity coefficients, computed via linear regression method.
- **covariance**: covariance matrix
- **pearson**: matrix of correlation coefficients
- **NormalizedSensitivity**: matrix of normalized sensitivity coefficients. **Note:** It is the matrix of normalized VarianceDependentSensitivity
- **VarianceDependentSensitivity**: matrix of sensitivity coefficients dependent on the variance of the variables
- **samples**: the number of samples in the data set used to determine the statistics.

If all the quantities need to be computed, this can be done through the `<all>` node, which requires the `<targets>` and `<features>` sub-nodes.

Note: If the weights are present in the system then weighted quantities are calculated automatically. In addition, if a matrix quantity is requested (e.g. Covariance matrix, etc.), only the weights in the output space are going to be used for both input and output space (the computation of the joint probability between input and output spaces is not implemented yet).
Note: Certain ROMs provide their own statistical information (e.g., those using the sparse grid collocation sampler such as: `'GaussPolynomialRom'` and `'HDMRRom'`) which can be obtained by printing the ROM to file (xml). For these ROMs, computing the basic statistics on data generated from one of these sampler/ROM combinations may not provide the information that the user expects.

- `<pivotParameter>`, ***string, optional field***, name of the parameter that needs to be used for the computation of the Dynamic BasicStatistics (e.g. time). This node needs to be inputted just in case an **HistorySet** is used as Input. It represents the reference monotonic variable based on which the statistics is going to be computed (e.g. time-dependent statistical moments).
  Default: None

- `<biased>`, ***string (boolean), optional field***, if *True* biased quantities are going to be calculated, if *False* unbiased.
  Default: False

- `<methodsToRun>`, ***comma separated string, optional field***, specifies the method names of an external Function that need to be run before computing any of the predefined quantities. If this XML node is specified, the `<Function>` node must be present.
  Default: None

- **Assembler Objects** This object is required in case the `<methodsToRun>` node is specified. The object must be listed with a rigorous syntax that, except for the xml node tag, is

common among all the objects. Each of these nodes must contain 2 attributes that are used to map those within the simulation framework:

- `class`, ***required string attribute***, it is the main "class" the listed object is from;
- `type`, ***required string attribute***, it is the object identifier or sub-type.

The **BasicStatistics** post-processor approach optionally accepts the following object type:

- `<Function>`, ***string, required field***, The body of this xml block needs to contain the name of an External Function defined within the `<Functions>` main block (see section 18). This object needs to contain the methods listed in the node `<methodsToRun>`.

Example (Static Statistics): This example demonstrates how to request the expected value of `'x01'` and `'x02'`, along with the sensitivity of both `'x01'` and `'x02'` to `'a'` and `'b'`.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue>x01,x02</expectedValue>
      <sensitivity>
        <targets>x01,x02</targets>
        <features>a,b</features>
      </sensitivity>
      <methodsToRun>failureProbability</methodsToRun>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Example (Static using `<all>`): This example is similar to the one above, but shows using the `<all>` node.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='BasicStatistics' verbosity='debug'>
      <all>
        <targets>x01,x02</targets>
```

```
        <features>a,b</features>
      </all>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Example (Static, multiple matrix nodes): This example shows how multiple nodes can specify particular metrics multiple times to include different target/feature combinations. This postprocessor calculates the expected value of A, B, and C, as well as the sensitivity of both A and B to X and Y as well as the sensitivity of C to W and Z.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue>A,B,C</expectedValue>
      <sensitivity>
        <targets>A,B</targets>
        <features>x,y</features>
      </sensitivity>
      <sensitivity>
        <targets>C</targets>
        <features>w,z</features>
      </sensitivity>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Example (Dynamic Statistics):

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedNameForDynamicPP'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue>x01,x02</expectedValue>
```

```
      <sensitivity>
        <targets>x01,x02</targets>
        <features>a,b</features>
      </sensitivity>
      <methodsToRun>failureProbability</methodsToRun>
      <pivotParameter>time</pivotParameter>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.2 ComparisonStatistics

The **ComparisonStatistics** post-processor computes statistics for comparing two different dataObjects. This is an experimental post-processor, and it will definitely change as it is further developed.

There are four nodes that are used in the post-processor.

- **<kind>**: specifies information to use for comparing the data that is provided. This takes either uniformBins which makes the bin width uniform or equalProbability which makes the number of counts in each bin equal. It can take the following attributes:
  - **numBins** which takes a number that directly specifies the number of bins
  - **binMethod** which takes a string that specifies the method used to calculate the number of bins. This can be either square-root or sturges.
- **<compare>**: specifies the data to use for comparison. This can either be a normal distribution or a dataObjects:
  - **<data>**: This will specify the data that is used. The different parts are separated by |'s.
  - **<reference>**: This specifies a reference distribution to be used. It takes distribution to use that is defined in the distributions block. A name parameter is used to tell which distribution is used.
- **<fz>**: If the text is true, then extra comparison statistics for using the f_z function are generated. These take extra time, so are not on by default.
- **<interpolation>**: This switches the interpolation used for the cdf and the pdf functions between the default of quadratic or linear.

The **ComparisonStatistics** post-processor generates a variety of data. First for each data provided, it calculates bin boundaries, and counts the numbers of data points in each bin. From the numbers in each bin, it creates a cdf function numerically, and from the cdf takes the derivative to generate a pdf. It also calculates statistics of the data such as mean and standard deviation. The post-processor can generate either a CSV file or a PointSet.

The post-processor uses the generated pdf and cdf function to calculate various statistics. The first is the cdf area difference which is:

$$cdf\_area\_difference = \int_{-\infty}^{\infty} \|CDF_a(x) - CDF_b(x)\| dx \tag{27}$$

This given an idea about how far apart the two pieces of data are, and it will have units of x.

The common area between the two pdfs is calculated. If there is perfect overlap, this will be 1.0, if there is no overlap, this will be 0.0. The formula used is:

$$pdf\_common\_area = \int_{-\infty}^{\infty} \min(PDF_a(x), PDF_b(x)) dx \tag{28}$$

The difference pdf between the two pdfs is calculated. This is calculated as:

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(x - z) dx \tag{29}$$

This produces a pdf that contains information about the difference between the two pdfs. The mean can be calculated as (and will be calculated only if fz is true):

$$\bar{z} = \int_{-\infty}^{\infty} z f_Z(z) dz \tag{30}$$

The mean can be used to get an signed difference between the pdfs, which shows how their means compare.

The variance of the difference pdf can be calculated as (and will be calculated only if fz is true):

$$var = \int_{-\infty}^{\infty} (z - \bar{z})^2 f_Z(z) dz \tag{31}$$

The sum of the difference function is calculated if fz is true, and is:

$$sum = \int_{-\infty}^{\infty} f_z(z) dz \tag{32}$$

This should be 1.0, and if it is different that points to approximations in the calculation.

Example:

```
<Simulation>
   ...
   <Models>
      ...
      <PostProcessor name="stat_stuff"
         subType="ComparisonStatistics">
      <kind binMethod='sturges'>uniformBins</kind>
      <compare>
        <data>OriData|Output|tsin_TEMPERATURE</data>
        <reference name='normal_410_2' />
      </compare>
      <compare>
        <data>OriData|Output|tsin_TEMPERATURE</data>
        <data>OriData|Output|tsout_TEMPERATURE</data>
      </compare>
      </PostProcessor>
      <PostProcessor name="stat_stuff2"
         subType="ComparisonStatistics">
        <kind numBins="6">equalProbability</kind>
        <compare>
          <data>OriData|Output|tsin_TEMPERATURE</data>
        </compare>
        <Distribution class='Distributions'
           type='Normal'>normal_410_2</Distribution>
      </PostProcessor>
      ...
   </Models>
   ...
   <Distributions>
      <Normal name='normal_410_2'>
         <mean>410.0</mean>
         <sigma>2.0</sigma>
      </Normal>
   </Distributions>
</Simulation>
```

17.5.3 ImportanceRank

The **ImportanceRank** post-processor is specifically used to compute sensitivity indices and importance indices with respect to input parameters associated with multivariate normal distributions.

In addition, the user can also request the transformation matrix and the inverse transformation matrix when the PCA reduction is used. In order to use the *ImportanceRank* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor subType='ImportanceRank'/>.

Several sub-nodes are available:

- **<what>**, ***comma separated string, required field***, List of quantities to be computed. Currently the quantities available are:
  - **'SensitivityIndex'**: used to measure the impact of sensitivities on the model.
  - **'ImportanceIndex'**: used to measure the impact of sensitivities and input uncertainties on the model.
  - **'PCAIndex'**: the indices of principal component directions, used to measure the impact of principal component directions on input covariance matrix. **Note:** **'PCAIndex'** can be only requested when subnode **<latent>** is defined in **<features>**.
  - **'transformation'**: the transformation matrix used to map the latent variables to the manifest variables in the original input space.
  - **'InverseTransformation'**: the inverse transformation matrix used to map the manifest variables to the latent variables in the transformed space.
  - **'ManifestSensitivity'**: the sensitivity coefficients of **<target>** with respect to **<manifest>** variables defined in **<features>**.

    Note: In order to request **'transformation'** matrix or **'InverseTransformation'** matrix or **'ManifestSensitivity'**, the subnodes **<latent>** and **<manifest>** under **<features>** are required (more details can be found in the following).

  If all the quantities need to be computed, the user can input in the body of **<what>** the string **'all'**. **Note:** **'all'** equivalent to **'SensitivityIndex, ImportanceIndex, PCAIndex'**.

  Since the transformation and InverseTransformation matrix can be very large, they are not printed with option **'all'**. In order to request the transformation matrix (or inverse transformation matrix) from this post processor, the user need to specify **'transformation'** or **'InverseTransformation'** in **<what>**. In addition, both **<manifest>** and **<latent>** subnodes are required and should be defined in node **<features>**. For example, let $\mathbf{L}, \mathbf{P}$ represent the transformation and inverse transformation matrices, respectively. We will define vectors $\mathbf{x}$ as manifest variables and vectors $\mathbf{y}$ as latent variables. If a absolute covariance matrix is used in given distribution, the following equation will be used:

  $\delta\mathbf{x} = \mathbf{L} * \mathbf{y}$

  $\mathbf{y} = \mathbf{P} * \delta\mathbf{x}$

If a relative covariance matrix is used in given distribution, the following equation will be used:

$\frac{\delta \mathbf{x}}{\mu} = \mathbf{L} * \mathbf{y}$

$\mathbf{y} = \mathbf{P} * \frac{\delta \mathbf{x}}{\mu}$

where $\delta \mathbf{x}$ denotes the changes in the input vector $\mathbf{x}$, and μ denotes the mean values of the input vector $\mathbf{x}$.

- **<features>**, ***XML node, required parameter***, used to specify the information for the input variables. In this xml-node, the following xml sub-nodes need to be specified:
  - **<manifest>**,***XML node, optional parameter***, used to indicate the input variables belongs to the original input space. It can accept the following child node:
    - **<variables>**,***comma separated string, required field***, lists manifest variables.
    - **<dimensions>**, ***comma separated integer, optional field***, lists the dimensions corresponding to the manifest variables. If not provided, the dimensions are determined by the order indices of given manifest variables.
  - **<latent>**,***XML node, optional parameter***, used to indicate the input variables belongs to the transformed space. It can accept the following child node:
    - **<variables>**,***comma separated string, required field***, lists latent variables.
    - **<dimensions>**, ***comma separated integer, optional field***, lists the dimensions corresponding to the latent variables. If not provided, the dimensions are determined by the order indices of given latent variables.

    Note: At least one of the subnodes, i.e. **<manifest>** and **<latent>** needs to be specified.
- **<targets>**, ***comma separated string, required field***, lists output responses.
- **<mvnDistribution>**, ***string, required field***, specifies the multivariate normal distribution name. The **<MultivariateNormal>** node must be present.

Here is an example to show the user how to request the transformation matrix, the inverse transformation matrix, the manifest sensitivities and other quantities.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='ImportanceRank'>
```

```
      <what>SensitivityIndex,ImportanceIndex,Transformation,
         InverseTransformation,ManifestSensitivity</what>
      <features>
        <manifest>
          <variables>x1,x2</variables>
          <dimensions>1,2</dimensions>
        </manifest>
        <latent>
          <variables>latent_1, latent_2</variables>
          <dimensions>1,2</dimensions>
        </latent>
      </features>
      <targets>y1,y2</targets>
      <mvnDistribution>MVN</mvnDistribution>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.4 SafestPoint

The **SafestPoint** post-processor provides the coordinates of the farthest point from the limit surface that is given as an input. The safest point coordinates are expected values of the coordinates of the farthest points from the limit surface in the space of the "controllable" variables based on the probability distributions of the "non-controllable" variables.

The term "controllable" identifies those variables that are under control during the system operation, while the "non-controllable" variables are stochastic parameters affecting the system behaviour randomly.

The "SafestPoint" post-processor requires the set of points belonging to the limit surface, which must be given as an input. The probability distributions as "Assembler Objects" are required in the "Distribution" section for both "controllable" and "non-controllable" variables.

The sampling method used by the "SafestPoint" is a "value" or "CDF" grid. At present only the "equal" grid type is available.

In order to use the *Safest Point* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor subType='SafestPoint'/>`.

Several sub-nodes are available:

- **<Distribution>**, ***Required***, represents the probability distributions of the "controllable" and "non-controllable" variables. These are **Assembler Objects**, each of these nodes must contain 2 attributes that are used to identify those within the simulation framework:
  - **class**, ***required string attribute***, is the main "class" the listed object is from.
  - **type**, ***required string attribute***, is the object identifier or sub-type.
- **<controllable>** lists the controllable variables. Each variable is associated with its name and the two items below:
  - **<distribution>** names the probability distribution associated with the controllable variable.
  - **<grid>** specifies the **type**, **steps**, and tolerance of the sampling grid.
- **<non-controllable>** lists the non-controllable variables. Each variable is associated with its name and the two items below:
  - **<distribution>** names the probability distribution associated with the non-controllable variable.
  - **<grid>** specifies the **type**, **steps**, and tolerance of the sampling grid.

Example:

```
<Simulation>
  ...
    <Models>
    ...
    <PostProcessor name='SP' subType='SafestPoint'>
      <Distribution  class='Distributions'
         type='Normal'>x1_dst</Distribution>
      <Distribution  class='Distributions'
         type='Normal'>x2_dst</Distribution>
      <Distribution  class='Distributions'
         type='Normal'>gammay_dst</Distribution>
      <controllable>
        <variable name='x1'>
          <distribution>x1_dst</distribution>
          <grid type='value' steps='20'>1</grid>
        </variable>
        <variable name='x2'>
          <distribution>x2_dst</distribution>
```

```
          <grid type='value' steps='20'>1</grid>
        </variable>
      </controllable>
      <non-controllable>
        <variable name='gammay'>
          <distribution>gammay_dst</distribution>
          <grid type='value' steps='20'>2</grid>
        </variable>
      </non-controllable>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.5 LimitSurface

The **LimitSurface** post-processor is aimed to identify the transition zones that determine a change in the status of the system (Limit Surface).

In order to use the *LimitSurface* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor subType='LimitSurface'/>`.

Several sub-nodes are available:

- `<parameters>`, ***comma separated string, required field***, lists the parameters that define the uncertain domain and from which the LS needs to be computed.
- `<tolerance>`, ***float, optional field***, sets the absolute value (in CDF) of the convergence tolerance. This value defines the coarseness of the evaluation grid.
  Default: 1.0e-4
- `<side>`, ***string, optional field***, in this node the user can specify which side of the limit surface needs to be computed. Three options are available:
  negative, Limit Surface corresponding to the goal function value of "-1";
  positive, Limit Surface corresponding to the goal function value of "1";
  both, either positive and negative Limit Surface is going to be computed.
  Default: negative
- **Assembler Objects** These objects are either required or optional depending on the functionality of the Adaptive Sampler. The objects must be listed with a rigorous syntax that, except

for the xml node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to map those within the simulation framework:

- `class`, ***required string attribute***, is the main "class" of the listed object. For example, it can be "Models," "Functions," etc.
- `type`, ***required string attribute***, is the object identifier or sub-type. For example, it can be "ROM," "External," etc.

The **LimitSurface** post-processor requires or optionally accepts the following objects' types:

- **<ROM>**, ***string, optional field***, body of this xml node must contain the name of a ROM defined in the **<Models>** block (see section 17.3).
- **<Function>**, ***string, required field***, the body of this xml block needs to contain the name of an External Function defined within the **<Functions>** main block (see section 18). This object represents the boolean function that defines the transition boundaries. This function must implement a method called *__residuumSign(self)*, that returns either -1 or 1, depending on the system conditions (see section 18).

Example:

```
<Simulation>
 ...
 <Models>
  ...
    <PostProcessor name="computeLimitSurface"
       subType='LimitSurface' verbosity='debug'>
      <parameters>x0,y0</parameters>
      <ROM class='Models' type='ROM'>Acc</ROM>
      <!-- Here, you can add a ROM defined in Models block.
           If it is not Present, a nearest neighbor algorithm
           will be used.
       -->
      <Function class='Functions' type='External'>
        goalFunctionForLimitSurface
      </Function>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.6 LimitSurfaceIntegral

The **LimitSurfaceIntegral** post-processor is aimed to compute the likelihood (probability) of the event, whose boundaries are represented by the Limit Surface (either from the LimitSurface post-processor or Adaptive sampling strategies). The inputted Limit Surface needs to be, in the **Post-Process** step, of type **PointSet** and needs to contain both boundary sides (-1.0, +1.0).
The **LimitSurfaceIntegral** post-processor accepts as outputs both files (CSV) and/or **PointSet**s.

In order to use the *LimitSurfaceIntegral* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor subType='LimitSurfaceIntegral'/>`.

Several sub-nodes are available:

- `<variable>`, ***XML node, required parameter*** will specify one attribute:
  - `name`, ***required string attribute***, user-defined name of this variable.

  This `<variable>` recognizes the following child node:
  - `<distribution>`, ***string, optional field***, name of the distribution that is associated to this variable. Its name needs to be contained in the `<Distributions>` block explained in Section 11. If this node is not present, the `<lowerBound>` and `<upperBound>` XML nodes must be inputted.
  - `<lowerBound>`, ***float, optional field***, lower limit of integration domain for this dimension (variable). If this node is not present, the `<distribution>` XML node must be inputted.
  - `<upperBound>`, ***float, optional field***, upper limit of integration domain for this dimension (variable). If this node is not present, the `<distribution>` XML node must be inputted.
- `<tolerance>`, ***float, optional field***, specifies the tolerance for numerical integration confidence.
  Default: 1.0e-4
- `<integralType>`, ***string, optional field***, specifies the type of integrations that need to be used. Currently only MonteCarlo integration is available
  Default: MonteCarlo
- `<seed>`, ***integer, optional field***, specifies the random number generator seed.
  Default: 20021986

- **<target>**, ***string, optional field***, specifies the target name that represents the $f(\bar{x})$ that needs to be integrated.
  Default: last output found in the inputted PointSet

Example:

```
<Simulation>
 ...
 <Models>
  ...
    <PostProcessor name="LimitSurfaceIntegralDistributions"
       subType='LimitSurfaceIntegral'>
        <tolerance>0.0001</tolerance>
        <integralType>MonteCarlo</integralType>
        <seed>20021986</seed>
        <target>goalFunctionOutput</target>
        <variable name='x0'>
          <distribution>x0_distrib</distribution>
        </variable>
        <variable name='y0'>
          <distribution>y0_distrib</distribution>
        </variable>
    </PostProcessor>
    <PostProcessor name="LimitSurfaceIntegralLowerUpperBounds"
       subType='LimitSurfaceIntegral'>
        <tolerance>0.0001</tolerance>
        <integralType>MonteCarlo</integralType>
        <seed>20021986</seed>
        <target>goalFunctionOutput</target>
        <variable name='x0'>
          <lowerBound>-2.0</lowerBound>
          <upperBound>12.0</upperBound>
        </variable>
        <variable name='y0'>
            <lowerBound>-1.0</lowerBound>
            <upperBound>11.0</upperBound>
        </variable>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.7 External

The **External** post-processor will execute an arbitrary python function defined externally using the *Functions* interface (see Section 18 for more details).

In order to use the *External* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor subType='External'/>.

Several sub-nodes are available:

- **<method>**, ***comma separated string, required field***, lists the method names of an external Function that will be computed (each returning a post-processing value). The name of the method represents a new variable that can be stored in a new ***DataObjects*** entity.
- **<Function>**, ***xml node, required string field***, specifies the name of a Function where the *methods* listed above are defined. **Note:** This name should match one of the Functions defined in the **<Functions>** block of the input file. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map them within the simulation framework:
  - **class**, ***required string attribute***, is the main "class" the listed object is from, the only acceptable class for this post-processor is **'Functions'**;
  - **type**, ***required string attribute***, is the object identifier or sub-type, the only acceptable type for this post-processor is **'External'**.

This Post-Processor accepts as Input/Output both **'PointSet'** and **'HistorySet'**:

- If a **'PointSet'** is used as Input, the parameters are passed in the external **'Function'** as numpy arrays. The methods' return type must be either a new array or a scalar. In the following it is reported an example with two methods, one that returns a scalar and the other one that returns an array:

```
import numpy as np
def sum(self):
  return np.sum(self.aParameterInPointSet)

def sumTwoArraysAndReturnAnotherone(self):
  return self.aParamInPointSet1+self.aParamInPointSet2
```

- If a **'HistorySet'** is used as Input, the parameters are passed in the external **'Function'** as a list of numpy arrays. The methods' return type must be either a new list of arrays (if the Output is another **'HistorySet'**), a scalar or a single array (if the Output is **'PointSet'** . In the following it is reported an example with two methods, one that returns a new list of arrays (Output = HistorySet) and the other one that returns an array (Output = PointSet):

```
import numpy as np
def newHistorySetParameter(self):
  x = []*len(self.time)
  for history in range(len(self.time)):
    for ts in range(len(self.time[history])):
      if self.time[history][ts] >= 0.001: break
    x[history] = self.x[history][ts:]
  return x

def aNewPointSetParameter(self):
  x = []*len(self.time)
  for history in range(len(self.time)):
    x[history] = self.x[history][-1]
  return x
```

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="externalPP" subType='External'
       verbosity='debug'>
      <method>Delta,Sum</method>
      <Function class='Functions'
         type='External'>operators</Function>
        <!-- Here, you can add a Function defined in the
             Functions block. This should be present or
             else RAVEN will not know where to find the
             defined methods. -->
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.8 TopologicalDecomposition

The **TopologicalDecomposition** post-processor will compute an approximated hierarchical Morse-Smale complex which will add two columns to a dataset, namely `minLabel` and `maxLabel` that can be used to decompose a dataset.

The topological post-processor can also be run in 'interactive' mode, that is by passing the keyword `interactive` to the command line of RAVEN's driver. In this way, RAVEN will initiate an interactive UI that allows one to explore the topological hierarchy in real-time and adjust the simplification setting before adjusting a dataset. Use in interactive mode will replace the parameter **<simplification>** described below with whatever setting is set in the UI upon exiting it.

In order to use the **TopologicalDecomposition** post-processor, the user needs to set the attribute **subType**: **<PostProcessor subType='TopologicalDecomposition'>**. The following is a list of acceptable sub-nodes:

- **<graph>** , ***string, optional field***, specifies the type of neighborhood graph used in the algorithm, available options are:
  - `beta skeleton`
  - `relaxed beta skeleton`
  - `approximate knn`

  Default: `beta skeleton`
- **<gradient>**, ***string, optional field***, specifies the method used for estimating the gradient, available options are:
  - `steepest`

  Default: `steepest`
- **<beta>**, ***float in the range: (0,2], optional field***, is only used when the **<graph>** is set to `beta skeleton` or `relaxed beta skeleton`.
  Default: 1.0
- **<knn>**, ***integer, optional field***, is the number of neighbors when using the **'approximate knn'** for the **<graph>** sub-node and used to speed up the computation of other graphs by using the approximate knn graph as a starting point for pruning. -1 means use a fully connected graph.
  Default: -1

- **<weighted>**, ***boolean, optional***, a flag that specifies whether the regression models should be probability weighted.
  Default: False

- **<persistence>**, ***string, optional field***, specifies how to define the hierarchical simplification by assigning a value to each local minimum and maximum according to the one of the strategy options below:

  - `difference` - The function value difference between the extremum and its closest-valued neighboring saddle.
  - `probability` - The probability integral computed as the sum of the probability of each point in a cluster divided by the count of the cluster.
  - `count` - The count of points that flow to or from the extremum.

  Default: `difference`

- **<simplification>**, ***float, optional field***, specifies the amount of noise reduction to apply before returning labels.
  Default: 0

- **<parameters>**, ***comma separated string, required field***, lists the parameters defining the input space.

- **<response>**, ***string, required field***, is a single variable name defining the scalar output space.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="***" subType='TopologicalDecomposition'>
      <graph>beta skeleton</graph>
      <gradient>steepest</gradient>
      <beta>1</beta>
      <knn>8</knn>
      <normalization>None</normalization>
      <parameters>X,Y</parameters>
      <response>Z</response>
      <weighted>true</weighted>
      <simplification>0.3</simplification>
      <persistence>difference</persistence>
```

```
    </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

17.5.9 DataMining

Knowledge discovery in databases (KDD) is the process of discovering useful knowledge from a collection of data. This widely used data mining technique is a process that includes data preparation and selection, data cleansing, incorporating prior knowledge on data sets and interpreting accurate solutions from the observed results. Major KDD application areas include marketing, fraud detection, telecommunication and manufacturing.

DataMining is the analysis step of the KDD process. The overall of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. The actual data mining task is the automatic or semi-automatic analysis of large quantities of data to extract previously unknown, interesting patterns such as groups of data records (cluster analysis), unusual records (anomaly detection), and dependencies (association rule mining).
In order to use the **DataMining** post-processor, the user needs to set the attribute `subType`:

`<PostProcessor subType= 'DataMining'>`.

The following is a list of acceptable sub-nodes:

- `<KDD>` ***string,required field***, the subnodes specifies the necessary information for the algorithm to be used in the postprocessor. The `<KDD>` has the required attribute: `lib`, the name of the library the algorithm belongs to. Current algorithms applied in the KDD model is based on SciKit-Learn library. Thus currently there is only one library:

  - `'SciKitLearn'`

  The `<KDD>` has the optional attribute: `labelFeature`, the name associated to labels or dimensions generated by the **DataMining** post-processor. The default name depends on the type of algorithm employed. For clustering and mixture models it is the name of the PostProcessor followed by "Labels" (e.g., if the name of a clustering PostProcessor is "kMeans" then the default name associated to the labels is "kMeansLabels" if not specified in the attribute `labelFeature`). For decomposition and manifold models, the default names are the name of the PostProcessor followed by "Dimension" and an integer identifier beginning with 1. (e.g., if the name of a dimensionality reduction PostProcessor is "dr" and the user specifies 3 components, then the output dataObject will have three new outputs named "drDimension1," "drDimension2," and "drDimension3.").

17.5.9.1 SciKitLearn

'SciKitLearn' is based on algorithms in SciKit-Learn library, and it performs data mining over PointSet and HistorySet. Note that for HistorySet's **'SciKitLearn'** performs the task given in **<SKLType>** (see below) for each time step, and so only synchronized HistorySet can be used as input to this model. For unsynchronized HistorySet, use **'HistorySetSync'** method in **'Interfaced'** post-processor to synchronize the input data before using **'SciKitLearn'**. The rest of this subsection and following subsection is dedicated to the **'SciKitLearn'** library.

The temporal variable for a HistorySet **'SciKitLearn'** is specified in the **<pivotParameter>** node:

- **<pivotParameter>**, ***string, optional parameter*** specifies the pivot variable (e.g., time, etc) in the input HistorySet.
  Default: None.

The algorithm for the dataMining is chosen by the subnode **<SKLType>** under the parent node **<KDD>**. The format is same as in 17.3.6. However, for the completeness sake, it is repeated here.

The data that are used in the training of the **DataMining** postprocessor are suplied with subnode **<Features>** in the parent node **<KDD>**.

- **<SKLtype>**, ***vertical bar (|) separated string, required field***, contains a string that represents the data mining algorithm to be used. As mentioned, its format is:
  <SKLtype>`mainSKLclass|algorithm`**</SKLtype>** where the first word (before the "|" symbol) represents the main class of algorithms, and the second word (after the "|" symbol) represents the specific algorithm.
- **<Features>**, ***string, required field***, defines the data to be used for training the data mining algorithm. It can be:
  - the name of the variable in the defined dataObject entity
  - the location (i.e. input or output). In this case the data mining is applied to all the variables in the defined space.

The **<KDD>** node can have either optional or required subnodes depending on the dataMining algorithm used. The possible subnodes will be described separately for each algorithm below. The time dependent clustering data mining algorithms have a **<reOrderStep>** option that will try and keep the same labels on the clusters. The higher the number, the longer the history that the clustering algorithm will look through to maintain the same labeling between time steps.

All the available algorithms are described in the following sections.

17.5.9.2 Gaussian mixture models

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.
Scikit-learn implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

17.5.9.2.1 GMM classifier

The GMM object implements the expectation-maximization (EM) algorithm for fitting mixture-of-Gaussian models. The GMM comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.

In order to use the *Gaussian Mixture Model*, the user needs to set the sub-node:

<SKLtype>`mixture|GMM`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field*** Number of mixture components.
  Default: 1
- **<covariance_type>**, ***string, optional field***, describes the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.
  Default: diag
- **<random_state>**, ***integer seed or random number generator instance, optional field***, A random number generator instance
  Default: 0 or None
- **<min_covar>**, ***float, optional field***, Floor on the diagonal of the covariance matrix to prevent overfitting.
  Default: 1e-3.
- **<thresh>**, ***float, optional field***, convergence threshold.
  Default: 0.01
- **<n_iter>**, ***integer, optional field***, Number of EM iterations to perform.
  Default: 100
- **<n_init>**, ***integer, optional***, Number of initializations to perform. the best results is kept.
  Default: 1

- **<params>**, ***string, optional field***, Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

  Default: 'wmc'

- **<init_params>**, ***string, optional field***, Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.
  Default: 'wmc'.

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variableName</Features>
              <SKLtype>mixture|GMM</SKLtype>
              <n_components>2</n_components>
              <covariance_type>spherical</covariance_type>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

17.5.9.2.2 Variational GMM Classifier (VBGMM)

The VBGMM object implements a variant of the Gaussian mixture model with variational inference algorithms. The API is identical to GMM.

In order to use the *Variational Gaussian Mixture Model*, the user needs to set the sub-node:

<SKLtype>`mixture|VBGMM`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field*** Number of mixture components.
  Default: 1

- **<covariance_type>**, ***string, optional field***, describes the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.
  Default: diag

- **<alpha>**, ***float, optional field***, represents the concentration parameter of the dirichlet process. Intuitively, the Dirichlet Process is as likely to start a new cluster for a point as it is to add that point to a cluster with alpha elements. A higher alpha means more clusters, as the expected number of clusters is $\alpha * log(N)$.
  Default: 1.

17.5.9.3 Clustering

Clustering of unlabeled data can be performed with this subType of the DataMining PostProcessor.

An overwiev of the different clustering algorithms is given in Table3.

Table 3. Overview of Clustering Methods

Method name	**Parameters**	**Scalability**	**Usecase**	**Geometry (metric used)**
K-Means	number of clusters	Very large n_samples, medium n_clusters with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium n_samples, small n_clusters	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomera-tive clustering	number of clusters, linkage type, distance	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large n_samples, medium n_clusters	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers

17.5.9.3.1 K-Means Clustering

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares. This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields

In order to use the *K-Means Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|KMeans`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field*** The number of clusters to form as well as the number of centroids to generate.
  Default: 8
- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the k-means algorithm for a single run.
  Default: 300
- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 3
- **<init>**, ***string, optional***, Method for initialization, k-means++', 'random' or an ndarray:
  - 'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
  - 'random': choose k observations (rows) at random from data for the initial centroids.
  - If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.
- **<precompute_distances>**, ***boolean, optional field***, Precompute distances (if true faster but takes more memory).
  Default: true
- **<tol>**, ***float, optional field***, Relative tolerance with regards to inertia to declare convergence.
  Default: 1e-4
- **<n_jobs>**, ***integer, optional field***, The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n jobs even slices and computing them in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all,

which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.
Default: 1

- **<random_state>**, ***integer or numpy.RandomState, optional field*** The generator used to initialize the centers. If an integer is given, it fixes the seed.
Default: the global numpy random number generator.

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variableName</Features>
              <SKLtype>cluster|KMeans</SKLtype>
              <n_clusters>2</n_clusters>
              <tol>0.0001</tol>
              <init>random</init>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

17.5.9.3.2 Mini Batch K-Means

The MiniBatchKMeans is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration.

MiniBatchKMeans converges faster than KMeans, but the quality of the results is reduced. In practice this difference in quality can be quite small.

In order to use the *Mini Batch K-Means Clustering*, the user needs to set the sub-node:

<SKLtype>cluster|MiniBatchKMeans**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field*** The number of clusters to form as well as the number of centroids to generate.
  Default: 8

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the k-means algorithm for a single run.
  Default: 100

- **<max_no_improvement>**, ***integer, optional firld***, Control early stopping based on the consecutive number of mini batches that does not yield an improvement on the smoothed inertia. To disable convergence detection based on inertia, set max_no_improvement to None.

  Default: 10

- **<tol>**, ***float, optional field***, Control early stopping based on the relative center changes as measured by a smoothed, variance-normalized of the mean center squared position changes. This early stopping heuristics is closer to the one used for the batch variant of the algorithms but induces a slight computational and memory overhead over the inertia heuristic. To disable convergence detection based on normalized center change, set tol to 0.0 (default).
  Default: 0.0

- **<batch_size>**, ***integer, optional field***, Size of the mini batches.
  Default: 100

- init_size, ***integer, optional field***, Number of samples to randomly sample for speeding up the initialization (sometimes at the expense of accuracy): the only algorithm is initialized by running a batch KMeans on a random subset of the data. *This needs to be larger than k.*,
  *Default: 3 * **<batch_size>***

- **<init>**, ***string, optional***, Method for initialization, k-means++', 'random' or an ndarray:

  - 'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
  - 'random': choose k observations (rows) at random from data for the initial centroids.
  - If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

- **<precompute_distances>**, ***boolean, optional field***, Precompute distances (if true faster but takes more memory).
  Default: true

- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 3

- **<compute_labels>**, ***boolean, optional field***, Compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit.
  Default: True

- **<random_state>**, ***integer or numpy.RandomState, optional field*** The generator used to initialize the centers. If an integer is given, it fixes the seed.
  Default: the global numpy random number generator.

- reassignment_ratio, **<float, optional field>**, Control the fraction of the maximum number of counts for a center to be reassigned. A higher value means that low count centers are more easily reassigned, which means that the model will take longer to converge, but should converge in a better clustering.
  Default: 0.01

17.5.9.3.3 Affinity Propagation

AffinityPropagation creates clusters by sending messages between pairs of samples until convergence. A dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.

In order to use the *AffinityPropogation Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|AffinityPropogation`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<damping>**, ***float, optional field***, Damping factor between 0.5 and 1.
  Default: 0.5

- **<convergence_iter>**, ***integer, optional field***, Number of iterations with no change in the number of estimated clusters that stops the convergence.
  Default: 15

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations.
  Default: 200

- **<copy>**, ***boolean, optional field***, Make a copy of input data or not.
  Default: True

- **<preference>**, ***array-like, shape (n_samples,) or float, optional field***, Preferences for each point - points with larger values of preferences are more likely to be chosen as exem-

plars. The number of exemplars, ie of clusters, is influenced by the input preferences value.
Default: If the preferences are not passed as arguments, they will be set to the median of the input similarities.

- **<affinity>**, ***string, optional field***,Which affinity to use. At the moment precomputed and euclidean are supported. euclidean uses the negative squared euclidean distance between points.
Default: "euclidean"

- **<verbose>**, ***boolean, optional field***, Whether to be verbose.
Default: False

17.5.9.3.4 Mean Shift

MeanShift clustering aims to discover blobs in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

In order to use the *Mean Shift Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|MeanShift`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<bandwidth>**, ***float, optional field***, Bandwidth used in the RBF kernel. If not given, the bandwidth is estimated using *sklearn.cluster.estimate_bandwidth*; see the documentation for that function for hints on scalability.

- **<seeds>**, ***array, shape=[n_samples, n_features], optional field***, Seeds used to initialize kernels. If not set, the seeds are calculated by *clustering.get_bin_seeds* with bandwidth as the grid size and default values for other parameters.

- **<bin_seeding>**, ***boolean, optional field***, If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized.
Default: False Ignored if seeds argument is not None.

- **<min_bin_freq>**, ***integer, optional field***, To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds.
Default: 1.

- **<cluster_all>**, ***boolean, optional field***, If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false,

then orphans are given cluster label -1.
Default: True

17.5.9.3.5 Spectral clustering

SpectralClustering does a low-dimension embedding of the affinity matrix between samples, followed by a *KMeans* in the low dimensional space. It is especially efficient if the affinity matrix is sparse and the pyamg module is installed.

In order to use the *Spectral Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|Spectral`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field***, The dimension of the projection subspace.
  Default: 8
- **<affinity>**, ***string, array-like or callable, optional field***, If a string, this may be one of:
  - 'nearest_neighbors',
  - 'precomputed',
  - 'rbf' or
  - one of the kernels supported by *sklearn.metrics.pairwise_kernels*.

  Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.
  Default: 'rbf'
- **<gamma>**, ***float, optional field***, Scaling factor of RBF, polynomial, exponential chi^2 and sigmoid affinity kernel. Ignored for $affinity = 'nearest\_neighbors'$.
  Default: 1.0
- **<degree>**,***float, optional field***, Degree of the polynomial kernel. Ignored by other kernels.

  Default: 3
- **<coef0>**, ***float, optional field***, Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.
  Default: 1
- **<n_neighbors>**, ***integer, optional field***, Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for affinity='rbf'.
  Default: 10

- **<eigen_solver>** ***string, optional field***, The eigenvalue decomposition strategy to use:
  - None,
  - 'arpack',
  - 'lobpcg', or
  - 'amg'

  Note: AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities

- **<random_state>**, ***integer seed, RandomState instance, or None, optional field***, A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when $eigen_solver == 'amg'$ and by the K-Means initialization.
  Default: None

- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 10

- **<eigen_tol>**,***float, optional field***, Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.
  Default: 0.0

- **<assign_labels>**, ***string, optional field***, The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding:
  - 'kmeans',
  - 'discretize'

  k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization.
  Default: 'kmeans'

- **<kernel_params>**, ***dictionary of string to any, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
  Default: None

Notes

If you have an affinity matrix, such as a distance matrix, for which 0 means identical elements, and high values means very dissimilar elements, it can be transformed in a similarity matrix that is well suited for the algorithm by applying the Gaussian (RBF, heat) kernel:

$$np.exp(-X**2/(2.*delta**2)) \tag{33}$$

Another alternative is to take a symmetric version of the k nearest neighbors connectivity matrix of the points. If the *pyamg* package is installed, it is used: this greatly speeds up computation.

17.5.9.3.6 DBSCAN Clustering

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped.

In order to use the *DBSCAN Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|DBSCAN`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<eps>**, ***float, optional field***, The maximum distance between two samples for them to be considered as in the same neighborhood.
  Default: 0.5
- **<min_samples>**, ***integer, optional field***, The number of samples in a neighborhood for a point to be considered as a core point.
  Default: 5
- **<metric>**, ***string, or callable, optional field*** The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by *metrics.pairwise.calculate_distance* for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square.
  Default: 'euclidean'
- **<random_state>**, ***numpy.RandomState, optional field***, The generator used to initialize the centers.
  Default: numpy.random.

17.5.9.3.7 Agglomerative Clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all of the samples, the leaves being the clusters with only one sample. The AgglomerativeClustering object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

- Ward: it minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.

- Maximum or complete linkage: it minimizes the maximum distance between observations of pairs of clusters.
- Average linkage: it minimizes the average of the distances between all observations of pairs of clusters.

AgglomerativeClustering can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all of the possible merges.

In order to use the *Agglomerative Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|Agglomerative`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***int, optional field***, The number of clusters to find.
  Default: 2
- **<connectivity>**, ***array like or callable, optional field***, Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.
  Default: None
- **<affinity>**, ***string or callable, optional field***, Metric used to compute the linkage. Can be "euclidean", "$l1$", "$l2$", "manhattan","cosine", or "precomputed". If linkage is "ward", only "euclidean" is accepted.
  Default: euclidean
- **<n_components>**, ***int, optional field***, Number of connected components. If None the number of connected components is estimated from the connectivity matrix. NOTE: This parameter is now directly determined from the connectivity matrix and will be removed in 0.18.
- **<linkage>**, ***ward,complete,average, optional field***, Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion. Ward minimizes the variance of the clusters being merged. Average uses the average of the distances of each observation of the two sets. Complete or maximum linkage uses the maximum distances between all observations of the two sets..
  Default: ward

17.5.9.3.8 Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar that members of different classes according to some similarity metric.

If the ground truth labels are not known, evaluation must be performed using the model itself. The **Silhouette Coefficient** is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters. The Silhouette Coefficient is defined for each sample and is composed of two scores:

1. The mean distance between a sample and all other points in the same class.
2. The mean distance between a sample and all other points in the next nearest cluster.

The Silhoeutte Coefficient s for a single sample is then given as:

$$s = \frac{b - a}{max(a, b)} \tag{34}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample. In normal usage, the Silhouette Coefficient is applied to the results of a cluster analysis.

Advantages

- The score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.

Drawbacks

The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

17.5.9.4 Decomposing signals in components (matrix factorization problems)

17.5.9.4.1 Principal component analysis (PCA)

- **Exact PCA and probabilistic interpretation**
  Linear Dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space. In order to use the *Exact PCA*, the user needs to set the sub-node:

  <SKLtype>decomposition|PCA**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, None or String, optional field***, Number of components to keep. if
  - **<n_components>** is not set all components are kept,
    Default: all components
  - **<copy>**, ***boolean, optional field***, If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.
    Default: True
  - **<whiten>**, ***boolean, optional field***, When True the components_ vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances. Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.
    Default: False

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variable1,variable2,variable3,
                 variable4,variable5</Features>
              <SKLtype>decomposition|PCA</SKLtype>
              <n_components>2</n_components>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

- **Randomized (Approximate) PCA**
  Linear Dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space. In order to use the *Randomized PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|RandomizedPCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***interger, None or String, optional field***, Number of components to keep. if n_components is not set all components are kept.
    Default: all components
  - **<copy>**, ***boolean, optional field***, If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.
    Default: True
  - **<iterated_power>**, ***integer, optional field***, Number of iterations for the power method.
    Default: 3
  - **<whiten>**, ***boolean, optional field***, When True the components_ vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances. Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.
    Default: False
  - **<random_state>**, ***int, or Random State instance or None, optional field***, Pseudo Random Number generator seed control. If None, use the numpy.random singleton.
    Default: None

- **Kernel PCA**
  Non-linear dimensionality reduction through the use of kernels. In order to use the *Kernel PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|KernelPCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***interger, None or String, optional field***, Number of components to keep. if n_components is not set all components are kept.
    Default: all components
  - **<kernel>**, ***string, optional field***, name of the kernel to be used, options are:
    * linear
    * poly

    * rbf
    * sigmoid
    * cosine
    * precomputed

  Default: linear **<degree>**, ***integer, optional field***, Degree for poly kernels, ignored by other kernels.
  Default: 3 **<gamma>**, ***float, optional field***, Kernel coefficient for rbf and poly kernels, ignored by other kernels.
  Default: 1/n_features

- **<coef0>**, ***float, optional field***, independent term in poly and sigmoig kernels, ignored by other kernels.
- **<kernel_params>**, ***mapping of string to any, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
  Default: 3
- alpha, ***int, optional field***, Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True).
  Default: 1.0
- **<fit_inverse_transform>**, ***bool, optional field***, Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point)
  Default: False
- **<eigen_solver>**, ***string, optional field***, Select eigensolver to use. If n_components is much less than the number of training samples, arpack may be more efficient than the dense eigensolver. Options are:
    * auto
    * dense
    * arpack

  Default: False
- tol, ***float, optional field***, convergence tolerance for arpack.
  Default: 0 (optimal value will be chosen by arpack)
- max_iter, ***int, optional field***, maximum number of iterations for arpack.
  Default: None (optimal value will be chosen by arpack)
- **<remove_zero_eig>**, ***boolean, optional field***, If True, then all components with zero eigenvalues are removed, so that the number of components in the output may be ¡ n_components (and sometimes even zero due to numerical instability). When n_components is None, this parameter is ignored and components with zero eigenvalues are removed regardless.
  Default: True

- **Sparse PCA**
  Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha. In order to use the *Sparse PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|SparsePCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, optional field***, Number of sparse atoms to extract.
    Default: None
  - **<alpha>**, ***float, optional field***, Sparsity controlling parameter. Higher values lead to sparser components.
    Default: 1.0
  - **<ridge_alpha>**, ***float, optional field***, Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.
    Default: 0.01
  - **<max_iter>**, ***float, optional field***, maximum number of iterations to perform.
    Default: 1000
  - **<tol>**, ***float, optional field***, convergence tolerance.
    Default: 1E-08
  - **<method>**, ***string, optional field***, method to use, options are:
    * lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path)
    * cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso)

    Lars will be faster if the estimated components are sparse.
    Default: lars
  - **<n_jobs>**, ***int, optional field***, number of parallel runs to run.
    Default: 1
  - **<U_init>**, ***array of shape (n_samples, n_components) , optional field***, Initial values for the loadings for warm restart scenarios
    Default: None
  - **<V_init>**, ***array of shape (n_components, n_features), optional field***, Initial values for the components for warm restart scenarios
    Default: None
  - verbose, ***boolean, optional field***, Degree of verbosity of the printed output.
    Default: False
  - random_state, ***int or Random State, optional field***, Pseudo number generator state used for random sampling.
    Default: None

- **Mini Batch Sparse PCA**
  Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha. In order to use the *Mini Batch Sparse PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|MiniBatchSparsePCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, optional field***, Number of sparse atoms to extract.
    Default: None
  - **<alpha>**, ***float, optional field***, Sparsity controlling parameter. Higher values lead to sparser components.
    Default: 1.0
  - **<ridge_alpha>**, ***float, optional field***, Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.
    Default: 0.01
  - **<n_iter>**, ***float, optional field***, number of iterations to perform per mini batch.
    Default: 100
  - **<callback>**, ***callable, optional field***, callable that gets invoked every five iterations.

    Default: None
  - **<batch_size>**, ***int, optional field***, the number of features to take in each mini batch.

    Default: 3
  - **<verbose>**, ***boolean, optional field***, Degree of verbosity of the printed output.
    Default: False
  - **<shuffle>**, ***boolean, optional field***, whether to shuffle the data before splitting it in batches.
    Default: True
  - **<n_jobs>**, ***integer, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
    Default: 3
  - **<metho>**, ***string, optional field***, method to use, options are:
    * lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path),
    * cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso)

    Lars will be faster if the estimated components are sparse.
    Default: lars
  - **<random_state>**, ***integer or Random State, optional field***, Pseudo number generator state used for random sampling.
    Default: None

17.5.9.4.2 Truncated singular value decomposition

Dimensionality reduction using truncated SVD (aka LSA). In order to use the *Truncated SVD*, the user needs to set the sub-node:

<SKLtype>`decomposition|TruncatedSVD`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, Desired dimensionality of output data. Must be strictly less than the number of features. The default value is useful for visualisation. For LSA, a value of 100 is recommended.
  Default: 2
- **<algorithm>**, ***string, optional field***, SVD solver to use:
  - Randomized: randomized algorithm
  - Arpack: ARPACK wrapper in.

  Default: Randomized
- **<n_iter>**, ***float, optional field***, number of iterations andomized SVD solver. Not used by ARPACK.
  Default: 5
- **<random_state>**, ***int or Random State, optional field***, Pseudo number generator state used for random sampling. If not given, the numpy.random singleton is used.
  Default: None
- **<tol>**, ***float, optional field***, Tolerance for ARPACK. 0 means machine precision. Ignored by randomized SVD solver.
  Default: 0.0

17.5.9.4.3 Fast ICA

A fast algorithm for Independent Component Analysis. In order to use the *Fast ICA*, the user needs to set the sub-node:

<SKLtype>`decomposition|FastICA`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, Number of components to use. If none is passed, all are used.
  Default: None

- **<algorithm>**, ***string, optional field***, algorithm used in FastICA:
  - parallel,
  - deflation.

  Default: parallel
- **<fun>**, ***string or function, optional field***, The functional form of the G function used in the approximation to neg-entropy. Could be either:
  - logcosh,
  - exp, or
  - cube.

  One can also provide own function. It should return a tuple containing the value of the function, and of its derivative, in the point.
  Default: logcosh
- **<fun_args>**, ***dictionary, optional field***, Arguments to send to the functional form. If empty and if fun='logcosh', fun_args will take value 'alpha' : 1.0.
  Default: None
- **<max_iter>**, ***float, optional field***, maximum number of iterations during fit.
  Default: 200
- **<tol>**, ***float, optional field***, Tolerance on update at each iteration.
  Default: 0.0001
- **<w_init>**, ***None or an (n_components, n_components) ndarray, optional field***, The mixing matrix to be used to initialize the algorithm.
  Default: None
- **<randome_state>**, ***int or Random State, optional field***, Pseudo number generator state used for random sampling.
  Default: None

17.5.9.5 Manifold learning

A manifold is a topological space that resembles a Euclidean space locally at each point. Manifold learning is an approach to non-linear dimensionality reduction. It assumes that the data of interest lie on an embedded non-linear manifold within the higher-dimensional space. If this manifold is of low dimension, data can be visualized in the low-dimensional space. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

17.5.9.5.1 Isomap

Non-linear dimensionality reduction through Isometric Mapping (Isomap). In order to use the *Isometric Mapping*, the user needs to set the sub-node:

<SKLtype>`manifold|Isomap`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_neighbors>**, ***integer, optional field***, Number of neighbors to consider for each point.
  Default: 5
- **<n_components>**, ***integer, optional field***, Number of coordinates to manifold.
  Default: 2
- **<eigen_solver>**, ***string, optional field***, eigen solver to use:
  - auto: Attempt to choose the most efficient solver for the given problem,
  - arpack: Use Arnoldi decomposition to find the eigenvalues and eigenvectors
  - dense: Use a direct solver (i.e. LAPACK) for the eigenvalue decomposition

  Default: auto
- **<tol>**, ***float, optional field***, Convergence tolerance passed to arpack or lobpcg. not used if eigen_solver is 'dense'.
  Default: 0.0
- **<max_iter>**, ***float, optional field***, Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.
  Default: None
- **<path_method>**, ***string, optional field***, Method to use in finding shortest path. Could be either:
  - Auto: attempt to choose the best algorithm
  - FW: Floyd-Warshall algorithm
  - D: Dijkstra algorithm with Fibonacci Heaps

  Default: auto
- **<neighbors_algorithm>**, ***string, optional field***, Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.
  - auto,

- brute
- kd_tree
- ball_tree

Default: auto

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>input</Features>
              <SKLtype>manifold|Isomap</SKLtype>
              <n_neighbors>5</n_neighbors>
              <n_components>3</n_components>
              <eigen_solver>arpack</eigen_solver>
              <neighbors_algorithm>kd_tree</neighbors_algorithm>
            </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

17.5.9.5.2 Locally Linear Embedding

In order to use the *Locally Linear Embedding*, the user needs to set the sub-node:

`<SKLtype>manifold|LocallyLinearEmbedding</SKLtype>`.

In addition to this XML node, several others are available:

- **`<n_neighbors>`**, ***integer, optional field***, Number of neighbors to consider for each point.
  Default: 5
- **`<n_components>`**, ***integer, optional field***, Number of coordinates to manifold.
  Default: 2

- **<reg>**, ***float, optional field***, regularization constant, multiplies the trace of the local covariance matrix of the distances.
  Default: 0.01

- **<eigen_solver>**, ***string, optional field***, eigen solver to use:

  - auto: Attempt to choose the most efficient solver for the given problem,
  - arpack: use arnoldi iteration in shift-invert mode.
  - dense: use standard dense matrix operations for the eigenvalue

  Default: auto

- **<tol>**, ***float, optional field***, Convergence tolerance passed to arpack. not used if eigen_solver is 'dense'.
  Default: 1E-06

- **<max_iter>**, ***int, optional field***, Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.
  Default: 100

- **<method>**, ***string, optional field***, Method to use. Could be either:

  - Standard: use the standard locally linear embedding algorithm
  - hessian: use the Hessian eigenmap method
  - itsa: use local tangent space alignment algorithm

  Default: standard

- **<hessian_tol>**, ***float, optional field***, Tolerance for Hessian eigenmapping method. Only used if method == 'hessian'
  Default: 0.0001

- **<modified_tol>**, ***float, optional field***, Tolerance for modified LLE method. Only used if method == 'modified'
  Default: 0.0001

- **<neighbors_algorithm>**, ***string, optional field***, Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.

  - auto,
  - brute
  - kd_tree

- ball_tree

Default: auto

- **<random_state>**, ***int or numpy random state, optional field***, the generator or seed used to determine the starting vector for arpack iterations.
Default: None

17.5.9.5.3 Spectral Embedding

Spectral embedding for non-linear dimensionality reduction, it forms an affinity matrix given by the specified function and applies spectral decomposition to the corresponding graph laplacian. The resulting transformation is given by the value of the eigenvectors for each data point In order to use the *Spectral Embedding*, the user needs to set the sub-node:

<SKLtype>`manifold|SpectralEmbedding`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, the dimension of projected sub-space.
Default: 2
- **<eigen_solver>**, ***string, optional field***, the eigen value decomposition strategy to use:
  - none,
  - arpack.
  - lobpcg,
  - amg

Default: none

- **<random_state>**, ***integer or numpy random state, optional field***, A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg.
Default: None
- **<affinity>**, ***string or callable, optional field***, How to construct the affinity matrix:
  - *nearest_neighbors* : construct affinity matrix by knn graph
  - *rbf* : construct affinity matrix by rbf kernel
  - *precomputed* : interpret X as precomputed affinity matrix

– *callable* : use passed in function as affinity the function takes in data matrix (n_samples, n_features) and return affinity matrix (n_samples, n_samples).

Default: nearest_neighbor

- **<gamma>**, ***float, optional field***, Kernel coefficient for rbf kernel.
  Default: None

- **<n_neighbors>**, ***int, optional field***, Number of nearest neighbors for nearest_neighbors graph building.
  Default: None

17.5.9.5.4 Multi-dimensional Scaling (MDS)

In order to use the *Multi Dimensional Scaling*, the user needs to set the sub-node:

<SKLtype>`manifold|MDS`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<metric>**, ***boolean, optional field***, compute metric or nonmetric SMACOF (Scaling by Majorizing a Complicated Function) algorithm
  Default: True

- **<n_components>**, ***integer, optional field***, number of dimension in which to immerse the similarities overridden if initial array is provided.
  Default: 2

- **<n_init>**, ***integer, optional field***, Number of time the smacof algorithm will be run with different initialisation. The final results will be the best output of the n_init consecutive runs in terms of stress.
  Default: 4

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the SMACOF algorithm for a single run
  Default: 300

- **<verbose>**, ***integer, optional field***, level of verbosity
  Default: 0

- **<eps>**, ***float, optional field***, relative tolerance with respect to stress to declare converge
  Default: 1E-06

- **<n_jobs>**, ***integer, optional field***, The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.
Default: 1

- **<random_state>**, **<integer or numpy random state, optional field>**, The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.
Default: None

- **<dissimilarity>**, ***string, optional field***, Which dissimilarity measure to use. Supported are 'euclidean' and 'precomputed'.
Default: euclidean

17.5.9.6 Scipy

'Scipy' provides a Hierarchical clustering that performs clustering over PointSet and HistorySet. This algorithm also automatically generates a dendrogram in .pdf format (i.e., dendrogram.pdf).

- **<SCIPYtype>**, ***string, required field***, SCIPY algorithm to be employed.

- **<Features>**, ***string, required field***, defines the data to be used for training the data mining algorithm. It can be:

  - the name of the variable in the defined dataObject entity
  - the location (i.e. input or output). In this case the data mining is applied to all the variables in the defined space.

- **<method>**, ***string, required field***, The linkage algorithm to be used
Default: single, complete, weighted, centroids, median, ward.

- **<metric>**, ***string, required field***, The distance metric to be used
Default: 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.

- **<level>**, ***float, required field***, Clustering distance level where actual clusters are formed.

- **<criterion>**, ***string, required field***, The criterion to use in forming flat clusters. This can be any of the following values:

  - "inconsistent" : If a cluster node and all its descendants have an inconsistent value less than or equal to 't' then all its leaf descendants belong to the same flat cluster. When no non-singleton cluster meets this criterion, every node is assigned to its own cluster. (Default)
  - "distance" : Forms flat clusters so that the original observations in each flat cluster have no greater a cophenetic distance than t.
  - "maxclust" : Finds a minimum threshold "r" so that the cophenetic distance between any two original observations in the same flat cluster is no more than "r" and no more than t flat clusters are formed.
  - "monocrit" : Forms a flat cluster from a cluster node c with index i when $monocrit[j] <= t$.
  - "maxclust_monocrit" : Forms a flat cluster from a non-singleton cluster node "c" when $monocrit[i] <= r$ for all cluster indices "i" below and including "c". "r" is minimized such that no more than "t" flat clusters are formed. monocrit must be monotonic.

- **<dendrogram>**, ***boolean, required field***, If True the dendrogram is actually created.

- **<truncationMode>**, ***string, required field***, The dendrogram can be hard to read when the original observation matrix from which the linkage is derived is large. Truncation is used to condense the dendrogram. There are several modes:

  - "None": No truncation is performed (Default).
  - "lastp": The last p non-singleton formed in the linkage are the only non-leaf nodes in the linkage; they correspond to rows $Z[n - p - 2 : end]$ in Z. All other non-singleton clusters are contracted into leaf nodes.
  - "level"/"mtica": No more than p levels of the dendrogram tree are displayed. This corresponds to Mathematica behavior.

- **<p>**, ***int, required field***, The p parameter for truncationMode.

- **<leafCounts>**, ***boolean, required field***, When True the cardinality non singleton nodes contracted into a leaf node is indiacted in parenthesis.

- **<showContracted>**, ***boolean, required field***, When True the heights of non singleton nodes contracted into a leaf node are plotted as crosses along the link connecting that leaf node.

- **<annotatedAbove>**, ***float, required field***, Clustering level above which the branching level is annotated.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="hierarchical" subType="DataMining"
       verbosity="quiet">
      <KDD lib="Scipy" labelFeature='labels'>
        <SCIPYtype>cluster|Hierarchical</SCIPYtype>
        <Features>output</Features>
        <method>single</method>
        <metric>euclidean</metric>
        <level>75</level>
        <criterion>distance</criterion>
        <dendrogram>true</dendrogram>
        <truncationMode>lastp</truncationMode>
        <p>20</p>
        <leafCounts>True</leafCounts>
        <showContracted>True</showContracted>
        <annotatedAbove>10</annotatedAbove>
      </KDD>
    </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

17.5.10 Interfaced

The **Interfaced** post-processor is a Post-Processor that allows the user to create its own Post-Processor. While the External Post-Processor (see Section 17.5.7 allows the user to create case-dependent Post-Processors, with this new class the user can create new general purpose Post-Processors.

In order to use the *Interfaced* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor subType='Interfaced'/>`.

Several sub-nodes are available:

- `<method>`, ***comma separated string, required field***, lists the method names of a method

that will be computed (each returning a post-processing value). All available methods need to be included in the "/raven/framework/PostProcessorFunctions/" folder

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="example"
       subType='InterfacedPostProcessor'verbosity='debug'>
       <method>testInterfacedPP</method>
       <!--Here, the xml nodes required by the chosen method
          have to be
       included.
        -->
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

All the **Interfaced** post-processors need to be contained in the "/raven/framework/PostProcessorFunctions/" folder. In fact, once the **Interfaced** post-processor is defined in the RAVEN input file, RAVEN search that the method of the post-processor is located in such folder.

The class specified in the **Interfaced** post-processor has to inherit the PostProcessorInterfaceBase class and the user must specify this set of methods:

- initialize: in this method, the internal parameters of the post-processor are initialized. Mandatory variables that needs to be specified are the following:
  - self.inputFormat: type of dataObject expected in input
  - self.outputFormat: type of dataObject generated in output
- readMoreXML: this method is in charge of reading the PostProcessor xml node, parse it and fill the PostProcessor internal variables.
- run: this method performs the desired computation of the dataObject.

```
from PostProcessorInterfaceBaseClass import PostProcessorInterfaceBase
class testInterfacedPP(PostProcessorInterfaceBase):
  def initialize(self)
```

```
  def readMoreXML(self,xmlNode)
  def run(self,inputDic)
```

17.5.10.1 Data Format

The user is not allowed to modify directly the DataObjects, however the content of the DataObjects is available in the form of a python dictionary. Both the dictionary give in input and the one generated in the output of the PostProcessor are structured as follows:

```
inputDict = {'data':{}, 'metadata':{}}
```

where:

```
inputDict['data'] = {'input':{}, 'output':{}}
```

In the input dictonary, each input variable is listed as a dictionary that contains a numpy array with its own values as shown below for a simplified example

```
inputDict['input'] = {'inputVar1': array([ 1.,2.,3.]),
                      'inputVar2': array([4.,5.,6.])}
```

Similarly, if the dataObject is a PointSet then the output dictionary is structured as follows:

```
inputDict['output'] = {'outputVar1': array([ .1,.2,.3]),
                       'outputVar2':array([.4,.5,.6])}
```

Howevers, if the dataObject is a HistorySet then the output dictionary is structured as follows:

```
inputDict['output'] = {'hist1': {}, 'hist2':{}}
```

where

```
inputDict['output'][hist1] = {'time': array([ .1,.2,.3]),
                              'outputVar1':array([ .4,.5,.6])}
inputDict['output'][hist2] = {'time': array([ .1,.2,.3]),
                              'outputVar1':array([ .14,.15,.16])}
```

17.5.10.2 Method: HistorySetSampling

This Post-Processor performs the conversion from HistorySet to HistorySet The conversion is made so that each history H is re-sampled accordingly to a specific sampling strategy. It can be used to reduce the amount of space required by the HistorySet.

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<samplingType>**, ***string, required field***, specifies the type of sampling method to be used (uniform, firstDerivative secondDerivative, filteredFirstDerivative or filteredSecondDerivative).
- **<numberOfSamples>**, ***integer, optional field***, number of samples (required only for the following sampling types: uniform, firstDerivative secondDerivative)
- **<pivotParameter>**, ***string, required field***, ID of the temporal variable
- **<interpolation>**, ***string, optional field***, type of interpolation to be employed for the history recostruction (required only for the following sampling types: uniform, firstDerivative secondDerivative). Valid types of interpolation to specified: linear, nearest, zero, slinear, quadratic, cubic, intervalAverage;
- **<tolerance>**, ***string, optional field***, tolerance level (required only for the following sampling types: filteredFirstDerivative or filteredSecondDerivative)

17.5.10.3 Method: HistorySetSync

This Post-Processor performs the conversion from HistorySet to HistorySet The conversion is made so that all histories are synchronized in time. It can be used to allow the histories to be sampled at the same time instant.

There are two possible synchronization methods, specified through the **<syncMethod>** node. If the **<syncMethod>** is **'grid'**, a **<numberOfSamples>** node is specified, which yields an equally-spaced grid of time points. The output values for these points will be linearly derived using nearest sampled time points, and the new HistorySet will contain only the new grid points.

The other methods are used by specifying **<syncMethod>** as **'all'**, **'min'**, or **'max'**. For **'all'**, the postprocessor will iterate through the existing histories, collect all the time points used in any of them, and use these as the new grid on which to establish histories, retaining all the exact original values and interpolating linearly where necessary. In the event of **'min'** or **'max'**, the postprocessor will find the smallest or largest time history, respectively, and use those time values as nodes to interpolate between.

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<pivotParameter>**, ***string, required field***, ID of the temporal variable

- **<extension>**, ***string, required field***, type of extension when the sync process goes outside the boundaries of the history (zeroed or extended)
- **<syncMethod>**, ***string, required field***, synchronization strategy to employ (see description above). Options are **'grid'**, **'all'**, **'max'**, **'min'**.
- **<numberOfSamples>**, ***integer, optional field***, required if **<syncMethod>** is **'grid'**, number of new time samples

17.5.10.4 Method: HistorySetSnapShot

This Post-Processor performs the conversion from HistorySet to PointSet The conversion is made so that each history H is converted to a single point P. There are several methods that can be employed to choose the single point from the history:

- min: Take a time slice when the **<pivotVar>** is at its smallest value,
- max: Take a time slice when the **<pivotVar>** is at its largest value,
- average: Take a time slice when the **<pivotVar>** is at its time-weighted average value,
- value: Take a time slice when the **<pivotVar>** *first passes* its specified value,
- timeSlice: Take a time slice index from the sampled time instance space.

To demonstrate the timeSlice, assume that each history H is a dict of n output variables $x_1 = [...], x_n = [...]$, then the resulting point P is at time instant index t: $P = [x_1[t], ..., x_n[t]]$.

Choosing one the these methods for the **<type>** node will take a time slice for all the variables in the output space based on the provided parameters. Alternatively, a **'mixed'** type can be used, in which each output variable can use a different time slice parameter. In other words, you can take the max of one variable while taking the minimum of another, etc.

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<type>**, ***string, required field***, type of operation: **'min'**, **'max'**, **'average'**, **'value'**, **'timeSlice'**, or **'mixed'**
- **<extension>**, ***string, required field***, type of extension when the sync process goes outside the boundaries of the history (zeroed or extended)
- **<pivotParameter>**, ***string, optional field***, name of the temporal variable. Required for the **'average'** and **'timeSlice'** methods.

If a **'timeSlice'** type is in use, the following nodes also are required:

- **<timeInstant>**, ***integer, required field***, required and only used in the **'timeSlice'** type. Location of the time slice (integer index)
- **<numberOfSamples>**, ***integer, required field***, number of samples

If instead a **'min'**, **'max'**, **'average'**, or **'value'** is used, the following nodes are also required:

- **<pivotVar>**, ***string, required field***, Name of the chosen indexing variable (the variable whose min, max, average, or value is used to determine the time slice)
- **<pivotVal>**, ***float, optional field***, required for **'value'** type, the value for the chosen variable

Lastly, if a **'mixed'** approach is used, the following nodes apply:

- **<max>**, ***string, optional field***, the names of variables whose output should be their own maximum value within the history.
- **<min>**, ***string, optional field***, the names of variables whose output should be their own minimum value within the history.
- **<average>**, ***string, optional field***, the names of variables whose output should be their own average value within the history. Note that a **<pivotParameter>** node is required to perform averages.
- **<value>**, ***string, optional field***, the names of variables whose output should be taken at a time slice determined by another variable. As with the non-mixed **'value'** type, the first time the **pivotVar** crosses the specified **pivotVal** will be the time slice taken. This node requires two attributes, if used:
  - **pivotVar**, ***string, required field***, the name of the variable on which the time slice will be performed. That is, if we want the value of y when $t = 0.245$, this attribute would be **'t'**.
  - **pivotVal**, ***float, required field***, the value of the **pivotVar** on which the time slice will be performed. That is, if we want the value of y when $t = 0.245$, this attribute would be **'0.245'**.

  Note that all the outputs of the **<DataObject>** output of this postprocessor must be listed under one of the **'mixed'** node types in order for values to be returned.

Example (mixed): This example will output the average value of x for x, the value of y at time$= 0.245$ for y, and the value of z at $x = 4.0$ for z.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="mampp2"
       subType="InterfacedPostProcessor">
      <method>HistorySetSnapShot</method>
      <type>mixed</type>
      <average>x</average>
      <value pivotVar="time" pivotVal="0.245">y</value>
      <value pivotVar="x" pivotVal="4.0">z</value>
      <pivotParameter>time</pivotParameter>
      <extension>zeroed</extension>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.10.5 Method: HSPS

This Post-Processor performs the conversion from HistorySet to PointSet The conversion is made so that each history H is converted to a single point P. Assume that each history H is a dict of n output variables $x_1 = [...], x_n = [...]$, then the resulting point P is as follows; $P = [x_1, ..., x_n]$ Note: it is here assumed that all histories have been sync so that they have the same length, start point and end point. If you are not sure, do a pre-processing the the original history set.

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified (min, max, avg and value case):

- **<pivotParameter>**, ***string, optional field***, ID of the temporal variable (only for avg)

17.5.10.6 Method: TypicalHistoryFromHistorySet

This Post-Processor performs a simplified procedure of [6] to form a "typical" time series from multiple time series. The input should be a HistorySet, with each history in the HistorySet synchronized. For HistorySet that is not synchronized, use Post-Processor method **HistorySetSync** to synchronize the data before running this method.

Each history in input HistorySet is first converted to multiple histories each has maximum time specified in **<outputLen>** (see below). Each converted history H_i is divided into a set of subsequences $\{H_i^j\}$, and the division is guided by the **<subseqLen>** node specified in the input XML. The value of **<subseqLen>** should be a list of positive numbers that specify the length of each subsequence. If the number of subsequence for each history is more than the number of values given in **<subseqLen>**, the values in **<subseqLen>** would be reused.

For each variable x, the method first computes the empirical CDF (cumulative density function) by using all the data values of x in the HistorySet. This CDF is termed as long-term CDF for x. Then for each subsequence H_i^j, the method computes the empirical CDF by using all the data values of x in H_i^j. This CDF is termed as subsequential CDF. For the first interval window (i.e., $j = 1$), the method computes the Finkelstein-Schafer (FS) statistics [7] between the long term CDF and the subsequential CDF of H_i^1 for each i. The FS statistics is defined as following.

$$FS = \sum_x FS_x$$

$$FS_x = \frac{1}{N} \sum_{n=1}^{N} \delta_n$$

where N is the number of value reading in the empirical CDF and δ_n is the absolute difference between the long term CDF and the subsequential CDF at value x_n. The subsequence H_i^1 with minimal FS statistics will be selected as the typical subsequence for the interval window $j = 1$. Such process repeats for $j = 2, 3, \ldots$ until all subsequences have been processed. Then all the typical subsequences will be concatenated to form a complete history.

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<pivotParameter>**, ***string, optional field***, ID of the temporal variable
  Default: Time
- **<subseqLen>**, ***integers, required field***, length of the divided subsequence (see above)
- **<outputLen>**, ***integer, optional field***, maximum value of the temporal variable for the generated typical history
  Default: Maximum value of the variable with name of **<pivotParameter>**

17.5.10.7 Method: Discrete Risk Measures

This Post-Processor calculates a series of risk importance measures from a PointSet. This calculation if performed for a set of input paramteres given an output target.

The user is required to provide the following information:

- the set of input variables. For each variable the following need to be specified:
  - the set of values that imply a reliability value equal to 1 for the input variable
  - the set of values that imply a reliability value equal to 0 for the input variable
- the output target variable. For this variable it is needed to specify the values of the output target variable that defines the desired outcome.

The following variables are first determined for each input variable i:

- R_0 Probability of the outcome of the output target variable (nominal value)
- R_i^+ Probability of the outcome of the output target variable if reliability of the input variable is equal to 0
- R_i^- Probability of the outcome of the output target variable if reliability of the input variable is equal to 1

Available measures are:

- Risk Achievement Worth (RAW): $RAW = R_i^+/R_0$
- Risk Achievement Worth (RRW): $RRW = R_0/R_i^-$
- Fussell-Vesely (FV): $FV = (R_0 - R_i^-)/R_0$
- Birnbaum (B): $B = R_i^+ - R_i^-$

In the **<PostProcessor>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<measures>**, ***string, required field***, desired risk importance measures that have to be computed (RRW, RAW, FV, B)
- **<variable>**, ***string, required field***, ID of the input variable. This node is provided for each input variable. This nodes needs to contain also these attributes:
  - **R0values**, ***float, required field***, interval of values (comma separated values) that implies a reliability value equal to 0 for the input variable
  - **R1values**, ***float, required field***, interval of values (comma separated values) that implies a reliability value equal to 1 for the input variable

- **<target>**, ***string, required field***, ID of the output variable. This nodes needs to contain also the attribute **values**, ***string, required field***, interval of values of the output target variable that defines the desired outcome

Example: This example shows an example where it is desired to calculate all available risk importance measures for two input variables (i.e., pumpTime and valveTime) given an output target variable (i.e., Tmax). A value of the input variable pumpTime in the interval $[0, 240]$ implies a reliability value of the input variable pumpTime equal to 0. A value of the input variable valveTime in the interval $[0, 60]$ implies a reliability value of the input variable valveTime equal to 0. A value of the input variables valveTime and pumpTime in the interval $[1441, 2880]$ implies a reliability value of the input variables equal to 1. The desired outcome of the output variable Tmax occurs in the interval $[2200, 2500]$.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="riskMeasuresDiscrete"
       subType="InterfacedPostProcessor">
      <method>RiskMeasuresDiscrete</method>
      <measures>B,FV,RAW,RRW</measures>
      <variable R0values='0,240'
         R1values='1441,2880'>pumpTime</variable>
      <variable R0values='0,60'
         R1values='1441,2880'>valveTime</variable>
      <target   values='2200,2500'
         >Tmax</target>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

This Post-Processor allows the user to consider also multiple datasets (a data set for each initiating event) and calculate the global risk importance measures. This can be performed by:

- Including all datasets in the step

```
<Simulation>
  ...
  </Steps>
    ...
    <PostProcess name="PP">
```

```
      <Input   class="DataObjects"  type="PointSet"
         >outRun1</Input>
      <Input   class="DataObjects"  type="PointSet"
         >outRun2</Input>
      <Model   class="Models"       type="PostProcessor"
         >riskMeasuresDiscrete</Model>
      <Output  class="DataObjects"  type="PointSet"
         >outPPS</Output>
      <Output  class="OutStreams"   type="Print"
         >PrintPPS_dump</Output>
    </PostProcess>
  </Steps>
  ...
</Simulation>
```

- Adding in the Post-processor the frequency of the initiating event associated to each dataset

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="riskMeasuresDiscrete"
       subType="InterfacedPostProcessor">
      <method>riskMeasuresDiscrete</method>
      <measures>FV,RAW</measures>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Astatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Bstatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Cstatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Dstatus</variable>
      <target   values='0.9,1.1'>outcome</target>
      <data     freq='0.01'>outRun1</data>
      <data     freq='0.02'>outRun2</data>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

17.5.11 RavenOutput

The **RavenOutput** post-processor is specifically used to gather data from RAVEN output files and generate a PointSet suitable for plotting or other analysis. It can do this in two modes: static and dynamic. In static mode, the PostProcessor reads from from several static XML output files produced by RAVEN. In dynamic mode, the PostProcessor reads from a single dynamic XML output file and builds a PointSet where the pivot parameter (e.g. time) is the input and the requested values are returned for each of the pivot parameter values (e.g. points in time). The name for the pivot parameter will be taken directly from the XML structure. Note: by default the PostProcessor operates in static mode; to read a dynamic file, the **<dynamic>** node must be specified. In order to use the *RavenOutput* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor subType='RavenOutput'/>.

Several sub-nodes are available:

- **<dynamic>**, ***string, optional field***, if included will trigger reading a single dynamic file instead of multiple static files, unless the text of this field is **'false'**, in which case it will return to the default (multiple static files).
  *Default: (*False)
- **<File>**, ***XML Node, required field*** For each file to be read by this postprocessor, an entry in the **<Files>** node must be added, and a **<File>** node must be added to the postprocessor input block. The **<File>** requires two identifying attributes:
  - **name**, ***string, required field***, the RAVEN-assigned name of the file,
  - **ID**, ***float, optional field***, the floating point ID that will be unique to this file. This will appear as an entry in the output **<DataObject>** and the corresponding column are the values extracted from this file. If not specified, RAVEN will attempt to find a suitable integer ID to use, and a warning will be raised.

  Each value that needs to be extracted from the file needs to be specified by one of the following **<output>** nodes within the **<File>** node:
  - **<output>**, ***—-separated string, required field***, the specification of the output to extract from the file. The text of this node is a path separated by pipe characters ("—"), starting under the root; this means the root node should not be included in the path. See the example. The **<output>** node requires the following attribute:
    * **name**, ***string, required field***, specifies the entry in the Data Object that this value should be stored under.

Example (Static): Using an example, let us have two input files, named *in1.xml* and *in2.xml*. They appear as follows. Note that the name of the variables we want changes slightly between the XML; this is fine.

in1.xml

```
<BasicStatistics>
  <ans>
    <val1>6</val1>
    <val2>7</val2>
  </ans>
</BasicStatistics>
```

in2.xml

```
<ROM>
  <ans>
    <first>6.1</first>
    <second>7.1</second>
  </ans>
</BasicStatistics>
```

The RAVEN input to extract this information would appear as follows:

```
<Simulation>
 ...
 <Files>
   <Input name='in1'>inp1.xml</Input>
   <Input name='in2'>inp2.xml</Input>
 </Files>
 ...
 <Models>
   ...
   <PostProcessor name='pp' subType='RavenOut'>
     <File name='in1' ID='1'>
       <output name='first'>ans|val1</output>
       <output name='second'>ans|val2</output>
     </File>
     <File name='in2' ID='2'>
       <output name='first'>ans|first</output>
       <output name='second'>ans|second</output>
     </File>
   </PostProcessor>
   ...
 </Models>
 ...
</Simulation>
```

Example (Dynamic): For a dynamic example, consider this time-evolution of values example. *inFile.xml* is a RAVEN dynamic XML output.

in1.xml

```
<BasicStatistics type='Dynamic'>
  <time value='0.0'>
    <ans>
      <val1>6</val1>
      <val2>7</val2>
    </ans>
  <\time>
  <time value='1.0'>
    <ans>
      <val1>9</val1>
      <val2>10</val2>
    </ans>
  <\time>
</BasicStatistics>
```

The RAVEN input to extract this information would appear as follows:

```
<Simulation>
 ...
 <Files>
   <Input name='inFile'>inFile.xml</Input>
 </Files>
 ...
 <Models>
   ...
   <PostProcessor name='pp' subType='RavenOut'>
     <dynamic>true</dynamic>
     <File name='inFile'>
       <output name='first'>ans|val1</output>
     </File>
   </PostProcessor>
   ...
 </Models>
 ...
</Simulation>
```

The resulting PointSet has *time* as an input and *first* as an output.

17.5.11.1 Method: dataObjectLabelFilter

The **<HSPS>** Post-Processor performs a filtering of the dataObject. This particular filtering is based on the labels generated by any clustering algorithm. Given the selected label, this Post-Processor filters out all histories or points having a different label. In the **<PostProcessor>** input block, the following XML sub-nodes are required:

- **<dataType>**, ***string, required field***, type of dataObject (HistorySet or PointSet)
- **<label>**, ***string, required field***, varaiable which contains the cluster labels
- **<clusterIDs>**, ***int, required field***, cluster labels considered

17.6 EnsembleModel

As already mentioned, the **EnsembleModel** is able to combine **Code**(see 17.1), **ExternalModel**(see 17.4) and **ROM**(see 17.4) Models.
It is aimed to create a chain of Models (whose execution order is determined by the Input/Output relationships among them). If the relationships among the models evolve in a non-linear system, a Picard's Iteration scheme is employed.
Currently this model is able to share information (i.e. data) using **Point** and **PointSet** only; this means that only point information can be shared among the different Models (e.g. thermal conductivity of a certain medium, peak temperature, etc.).

The specifications of a EnsembleModel must be defined within the XML block **<EnsembleModel>**. This XML node needs to contain the attributes:

- **name**, ***required string attribute***, user-defined name of this EnsembleModel. **Note:** As with the other objects, this is the name that can be used to refer to this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, must be kept empty.

Within the **<EnsembleModel>** XML node, the multiple Models that constitute this EnsembleModel needs to be inputted. Each Model is specified within a **<Model>** block (**Note:** each model here specified need to be inputted in the**<Models>** main XML block) :

- **<Model>**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the Model
  This XML node needs to contain the attributes:
  - **class**, ***required string attribute***, the class of this sub-model (e.g. Models)
  - **type**, ***required string attribute***, the sub-type of this Model (e.g. ExternalModel, ROM, Code)

In addition the following XML sub-nodes need to be inputted:

- **<TargetEvaluation>**, ***string, required field***, represents the container where the output of this Model are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 14). Currently, the **<EnsembleModel>** accept "DataObjects" both of type "PointSet" and "HistorySet".
- **<Input>**, ***string, required field***, represents the input entities that need to be passed to this sub-model The user can specify as many **<Input>** as required by the sub-model. **Note:** All the inputs here specified need to be listed in the Steps where the EnsembleModel is used. Currently, the **<EnsembleModel>** accepts "DataObjects" of type "PointSet" only.

It is important to notice that when the EnsembleModel detects a chain of models that evolve in a non-linear system, a Picard's Iteration scheme is activated. In this case, an additional XML sub-node within the main **<EnsembleModel>** XML node needs to be specified:

- **<settings>**, ***XML node, required parameter (if Picard's activated)***. The body of this sub-node contains the following XML sub-nodes:
  - **<maxIterations>**, ***integer, optional field***, maximum number of Picard's iteration to be performed (in case the iteration scheme does not previously converge).
    Default: 30;
  - **<tolerance>**, ***float, optional field***, convergence criterion. It represents the L2 norm residue below which the Picard's iterative scheme is considered converged.
    Default: 0.001;
  - **<initialConditions>**, ***XML node, required parameter (if Picard's activated)***, Within this sub-node, the initial conditions for the input variables (that are part of a loop) need to be specified in sub-nodes named with the variable name (e.g. **<varName>**). The body of the **<varName>** contains the value of the initial conditions (scalar or arrays, depending of the type of variable). If an array needs to be inputted, the user can specify the attribute **repeat** and the code is going to repeat for **repeat**-times the value inputted in the body.

Example (Linear System):

```
<Simulation>
  ...
  <Models>
    ...
    <EnsembleModel name="heatTransferEnsembleModel" subType="">
      <Model class="Models" type="ExternalModel">
```

```
        thermalConductivityComputation
        <TargetEvaluation class="DataObjects" type="PointSet">
          thermalConductivityComputationContainer
        </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
      <Model class="Models" type="ExternalModel" >
          heatTransfer
          <TargetEvaluation class="DataObjects" type="PointSet">
            heatTransferContainer
          </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
    </EnsembleModel>
    ...
  </Models>
  ...
</Simulation>
```

Example (Non-Linear System):

```
<Simulation>
  ...
  <Models>
    ...
    <EnsembleModel name="heatTransferEnsembleModel" subType="">
      <settings>
        <maxIterations>8</maxIterations>
         <tolerance>0.01</tolerance>
         <initialConditions>
           <!-- the value 0.7 is going to be repeated 10 times
              in order to create  an array for var1 -->
           <var1 repeat="10">0.7</var1>
           <!-- an array for var2 has been inputted -->
           <var2> 0.5 0.3 0.4</var2>
           <!-- a scalar for var3 has been inputted -->
           <var3> 45.0</var3>
         </initialConditions>
      </settings>
```

```
      <Model class="Models" type="ExternalModel">
        thermalConductivityComputation
        <TargetEvaluation class="DataObjects" type="PointSet">
          thermalConductivityComputationContainer
        </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
      <Model class="Models" type="ExternalModel" >
          heatTransfer
          <TargetEvaluation class="DataObjects" type="PointSet">
            heatTransferContainer
          </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
    </EnsembleModel>
    ...
  </Models>
  ...
</Simulation>
```

18 Functions

The RAVEN code provides support for the usage of user-defined external functions. These functions are python modules, with a format that is automatically interpretable by the RAVEN framework. For example, users can define their own method to perform a particular post-processing activity and the code will be embedded and use the function as though it were an active part of the code itself. In this section, the XML input syntax and the format of the accepted functions are fully specified.

The specifications of an external function must be defined within the XML block **<External>**. This XML node requires the following attributes:

- **name**, ***required string attribute***, user-defined name of this function. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.
- **file**, ***required string attribute***, absolute or relative path specifying the code associated to this function. **Note:** If a relative path is specified, it must be relative with respect to where the user is running the instance of RAVEN.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in her/his own python function, the variables need to be specified in the **<External>** input block. The user needs to input, within this block, only the variables directly used by the external code and not the local variables that the user does not want, for example, those stored in a RAVEN internal object. These variables are named within consecutive **<variable>** XML nodes:

- **<variable>**, ***string, required parameter***, in the body of this XML node, the user needs to specify the name of the variable. This variable needs to match a variable used/defined in the external python function.

When the external function variables are defined, at runtime, RAVEN initializes them and keeps track of their values during the simulation. Each variable defined in the **<External>** block is available in the function as a python `self.` member. In the following, an example of a user-defined external function is reported (a python module and its related XML input specifications).

Example Python Function:

```
import numpy as np
def residuumSign(self):
  if self.var1 < self.var2 :
    return  1
  else:
    return -1
```

Example XML Input:

```
  ...
  <Functions>
    ...
    <External name='whatever' file='path_to_python_file'>
     ...
     <variable>var1</variable>
     <variable>var2</variable>
     ...
    </External>
    ...
  </Functions>
   ...
</Simulation>
```