INL/EXT-15-36632
Revision 0

Developing and Implementing the Data Mining Algorithms in RAVEN

Sonat Sen,
Daniel Maljovec,
Andrea Alfonsi,
Cristian Rabiti

September 2015

The INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance

INL/EXT-15-36632
Revision 0

Developing and Implementing the Data Mining Algorithms in RAVEN

Sonat Sen,
Daniel Maljovec,
Andrea Alfonsi,
Cristian Rabiti

September 2015

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Developing and Implementing the Data Mining Algorithms in RAVEN

INL/EXT-15-36632
Revision 0

September 2015

Approved by:

Name	Date
Title [optional]	
Name	Date
Title [optional]	
Name	Date
Title [optional]	
Name	Date
Title [optional]	

SUMMARY

The RAVEN code is becoming a comprehensive tool to perform probabilistic risk assessment, uncertainty quantification, and verification and validation. The RAVEN code is being developed to support many programs and to provide a set of methodologies and algorithms for the analysis of data sets, simulation of physical phenomenon, etc.

Scientific computer codes can generate enormous amounts of data. To post-process and analyze such data might, in some cases, take longer than the initial software runtime. Data mining algorithms/methods help in recognizing and understanding patterns in the data, and thus discover knowledge in databases.

The methodologies used in dynamic probabilistic risk assessment or in uncertainty and error quantification analysis couple system/physics codes with simulation controller codes, such as RAVEN. RAVEN introduces both deterministic and stochastic elements into the simulation while the system/physics code model the dynamics deterministically. A typical analysis is performed by sampling values of a set of parameter attributes. A major challenge in using dynamic probabilistic risk assessment or uncertainty and error quantification analysis for a complex system is to analyze the large number of scenarios generated. Data mining techniques are typically used to better organize and understand data, i.e. recognizing patterns in the data.

This report focuses on development and implementation of a post processing infrastructure and Application Programming Interfaces (APIs) for different data mining algorithms within RAVEN to extend its capability. The application of these algorithms to different databases will be also presented.

CONTENTS

FIGURES

TABLES

ACRONYMS

ADS – Automatic Depressurization System

AMSC – Approximate Morse-Smale Complex

API – Application Programming Interface

BIC – Bayesian Information Criterion

BWR – Boiling Water Reactor

CDF – Cumulative Distribution Function

CPU – Central Processing Unit

CST – Condensate Storage Tank

DBSCAN – Density-Based Spatial Clustering of Applications with Noise

DET – Dynamic Event Tree

DG – Diesel Generator

DPGMM – Dirichlet Process Gaussian Mixture Model

DPRA – Dynamic Probabilistic Risk Assessment

EM – Expectation-Maximization

FOM – Figure of Merit

GMM – Gaussian Mixture Model

HCTL – Heat Capacity Temperature Limit

HPCI – High Pressure Coolant Injection

ICA – Independent Component Analysis

KDD – Knowledge Discovery in Databases

KDE – Kernel Density Estimation

LHS – Latin Hypercube Sampling

LLE – Locally Linear Embedding

LOOP – Loss Of Offsite Power

LS – Limit Surface

LSA – Latent Semantic Analysis

MC – Monte Carlo

MDS – Multi-Dimensional Scaling

MOOSE – Multi-physics Object-Oriented Simulation Environment

NPP – Nuclear Power Plant

PCA – Principal Component Analysis

PDF – Probability Density Function

PHISICS – Parallel and Highly Innovative Simulation for INL Code System

PP – Post-Processor

PRA – Probabilistic Risk Assessment

PSP – Pressurized Suppression Pool

RAVEN – Reactor Analysis and Virtual Control Environment

RCIC – Reactor Core Isolation Cooling

RELAP – Reactor Excursion and Leak Analysis Program

RHR – Residual Heat Removal

ROM – Reduced Order Model

RPV – Reactor Pressure Vessel

SBO – Station Black Out

SGD – Stochastic Gradient Descent

SRV – Safety Relief Valve

SVD – Singular Value Decomposition

SVM – Support Vector Machine

VBGMM – Variational Bayesian Gaussian Mixture Model

Developing and Implementing the Data Mining Algorithms in RAVEN

1. INTRODUCTION

RAVEN [1 - 6] is advancing its capability to perform statistical analyses of stochastic dynamic systems, putting a big effort in the identification and development of methodologies able to identify the region of interest in the uncertain/parametric space allowing for subsequent optimization of the available computational resources.

The simulation codes used in nuclear engineering analysis implement computationally intensive methods to perform safety analysis of nuclear power plants. The new generation of nuclear engineering codes couple several different physics; they include nuclear phenomena, thermal-hydraulic phenomena, structural behaviors and system dynamics, etc. The Dynamic Probabilistic Risk Assessment (DPRA) methodologies couple these multi-physics codes with stochastic analysis tools, such as RAVEN, to perform probabilistic risk analysis, uncertainty quantification, and sensitivity analysis. This type of analysis is typically performed by sampling values for a set of parameters from the space of interest with uncertainty. The system behavior is then simulated for that specific set of parameter values.

Investigation of the probabilistic evolution of accident scenarios for a complex system such as a nuclear power plant (NPP) is not a trivial challenge. The complexity of the system to be modeled leads to demanding computational requirements to simulate even just one of the many possible evolutions of an accident scenario (tens of CPU hours). At the same time, the probabilistic analysis requires thousands of runs (simulation of one of the possible scenario evolutions) to investigate outcomes characterized by low probability and severe consequence. The final product, the data generated, can be too complex and large that even more time is required for analyzing and understanding the outcome as a whole. Therefore, most of the time the input space of such analysis usually does not consider all of the input parameters and the output space usually is limited to a few figures of merit (FOMs).

Large and complex datasets can be analyzed with data mining techniques. Data mining techniques are used to find and interpret patterns in these large datasets. Generally, the data mining process includes the extraction of information from a dataset and the subsequent transformation of it into an understandable structure for further use. The ultimate task is the automatic or semi-automatic analysis of large quantities of data to extract previously unknown information.

A few of the many algorithms being incorporated in data mining include:

- **Clustering:** maps a data item into one of several categorical classes (or clusters) in which the classes must be determined from the data. Clusters are defined by finding natural groupings of data items based on similarity metrics or probability density models. If the clusters are predefined then the process is called “classification.”
- **Regression:** a learning function, which maps a data item to a real-valued prediction variable

This report is structured as follows:

- Section 1 gives a brief introduction/overview of the data mining approach
- Section 2 gives an overview of the RAVEN code with its main components, especially data mining Post Processors (PP)

- Section 3 introduces the algorithms in the SciKit learn library and the Topology PP
- Section 4 presents a series of test cases to show the possibilities of using the data mining algorithms
- Section 5 highlights the possible future development paths
- Section 6 presents conclusions.

2. RAVEN FRAMEWORK

2.1 Introduction

As inferred from the initial introduction, the data mining algorithms from Scikit Learn [7], which is an open source machine-learning library developed in Python, and the topological post processor, were implemented and assessed within the probabilistic and uncertainty quantification framework, RAVEN. Hence, it is helpful to provide a brief overview of the code and its main capabilities and internal structure.

RAVEN was developed in a highly modular and pluggable way to enable easy integration of different programming languages (i.e., C++ and Python) and coupling with any system/physics code. Its main goal is to provide a tool to allow exploration of the uncertain domain, dispatching several different capabilities in an integrated environment.

2.2 Software Infrastructure Overview

The main idea behind the design of the RAVEN software package is the creation of a multi-purpose framework characterized by high flexibility with respect to the possible set of analyses that a user might request. To obtain this result, the code infrastructure must be capable of constructing the analysis/calculation flow at run-time, interpreting the user-defined instructions, and assembling the different analysis tasks following a user-specified scheme.

The need to achieve such flexibility, combined with reasonably fast development, pushed toward the programming language that is naturally suitable for this kind of approach: Python.

Hence, RAVEN is coded in Python and characterized by a highly object-oriented design. The core of the analysis available through RAVEN is represented by a set of basic components (entities) the user can combine, to create a custom analysis flow. A list of these components and summary of their most important functionalities are as follows:

Distribution: The probability of a specific system outcome is related to the probability of the set of input parameters and initial conditions that led to such an outcome. Moreover, some sampling techniques (e.g., Monte-Carlo [MC]) explore the input space according to the probabilistic distribution associated to the input variables. Consequently, RAVEN possesses a large library of PDFs.

Sampler: A proper approach to sample the input space is fundamental for optimizing the computational time. In RAVEN, a "sampler" determines a unique exploration strategy that is applied to the input space of a system. The association of uncertain variables and their corresponding probability distributions constitute the probabilistic input space on which the sampler operates.

Model: A model is the representation of a physical system (e.g., NPP); it is therefore capable of predicting the evolution of a system given a coordinate set in the input space (i.e., the initial condition of

the system phase space). A model, usually, does not belong to RAVEN but it is made available to RAVEN by the user either by the available APIs or by coding it directly inside RAVEN as external model.

ROM: The evaluation of the system response, as a function of the coordinates in the uncertain domain (also known as input space), is very computationally expensive, which makes brute-force approaches (e.g., MC methods) impractical. ROMs are used to lower this cost by reducing the number of needed points and prioritizing the area of the uncertain domain that needs to be explored. They are a pure mathematical representation of the link between the input and output spaces for a particular system.

Post-Processors: The post-processors are used to process the datasets resulting from a simulation of a system either in RAVEN or via an external code. Post-processors can be used to obtain basic statistical information of the data, compare datasets statistically, or discover patterns in the datasets, i.e. data mining.

The list above is not comprehensive of all the RAVEN framework components, which also include visualization and storage infrastructure.

2.2.1 Distributions

As already mentioned, the perturbation of the input space (initial conditions/parameters affected by uncertainties) needs to be performed to account for their probabilistic distributions. RAVEN provides, through an interface to the Boost library, the following univariate distributions (with optional truncation):

- Bernoulli
- Binomial
- Exponential
- Logistic
- Lognormal
- Normal
- Poisson
- Triangular
- Uniform
- Weibull
- Gamma
- Beta
- Categorical.

The use of univariate distributions for sampling initial conditions is based on the assumption that the uncertain parameters are not correlated with each other. Quite often uncertain parameters are subject to correlations and thus the univariate approach is not applicable. This happens when a generic outcome depends on multiple variables or vice versa. In such cases, the outcome dependency description cannot be collapsed to a function of a single variable. RAVEN currently supports N-dimensional (N-D) PDFs both in the form of a multivariate normal distribution and user-provided PDFs. The user can provide files containing the distribution values on either a Cartesian or sparse grid. Depending on the grid structure

used to provide the distribution values, RAVEN determines the interpolation algorithm used in the evaluation of the imported cumulative distribution function (CDF)/PDF:

- N-D spline [8] for Cartesian grids
- Inverse weight [9] for sparse grids.

Internally, RAVEN provides the needed N-D differentiation and integration algorithms to compute the PDF from the CDF and vice versa. This is needed to cover both cases where the user provides the PDF or CDF.

As already mentioned, the sampling methods use distributions to perform a probability-weighted exploration of the input space. For example, in the MC approach, a random number $\in [\mathbf{0,1}]$ is generated (probability threshold) and the CDF, corresponding to that probability, is inverted to retrieve the parameter value used in the simulation. The existence of the inverse for univariate distributions is guaranteed by the monotonicity of the CDF. For N-D distributions, this condition is not sufficient since the $\boldsymbol{CDF(X) \rightarrow [0,1], x \in R^N}$ and therefore, it is not guaranteed to be a bijective function. From an application point of view, this means the inverse of an N-D CDF is not unique.

As an example, Figure 1 shows a multivariate normal distribution for a pipe failure as a function of the pressure and temperature. The plane identifies an iso-probability surface (in this case, a line) that represents a probability threshold of 50% in this example. Hence, the inverse of this CDF is an infinite number of points.

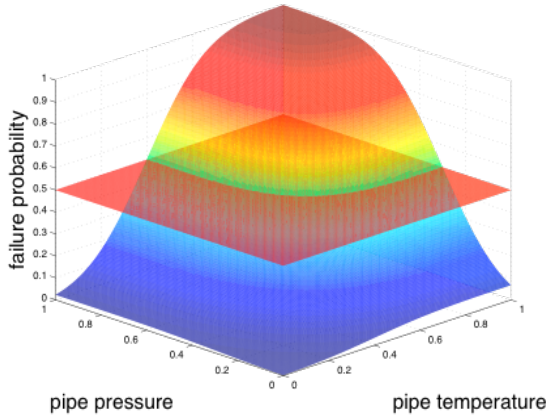

Figure 1: Example of two-dimensional multivariate probability distribution function

As easily inferable, the standard sampling approach cannot directly be employed. When multivariate distributions are used, RAVEN implements a surface search algorithm to identify the iso-probability surface location. Once the location of the surface is found, RAVEN chooses, randomly, one point on it [10].

2.2.2 Samplers

As already mentioned, the sampler is a key entity in the RAVEN framework needed to employ many of its capabilities of analysis. Indeed, it performs the driving of the specific sampling strategy and, hence, determines the effectiveness of the analysis, from both an accuracy and computational point of view. The samplers, that are available in RAVEN, are categorized into three main classes:

1. Forward
2. Dynamic event tree (DET)
3. Adaptive.

The following subsections briefly introduce the forward and DET samplers. It is also worth mentioning that besides the adaptive samplers there is a current effort internally founded at Idaho National Laboratory aimed to construct an adaptive sampler for the full representation of the system response by polynomial interpolation. Unfortunately the only reference present at this time on this work is the RAVEN manual. [3]

2.2.2.1 Forward Samplers

The forward sampler category collects all the strategies that perform the sampling of the input space without exploiting, through dynamic learning approaches (active learning), the information made available from the outcomes of calculation previously performed (adaptive sampling) and the common system evolution (patterns) that different sampled calculations can generate in the phase space (DET).

In the RAVEN framework, several different forward samplers are available:

- MC
- Stratified (if equally spaced in probability ->LHS)
- Grid-based
- Factorial designs:
  - Full factorial
  - Two-level fractional-factorial
  - Plackett-Burman
- Response surface designs:
  - Box-Behnken
  - Central composite
- Stochastic collocation.

Since most of the forward sampling strategies previously listed are well known, they are not fully described in this report. More details regarding the MC, stratified, and grid sampling strategies are found in Ref. [11]; details regarding the factorial and response surface designs are found in Ref. [4]. In conclusion, detailed information about the stochastic collocation method is found in Ref. [12].

2.2.2.2 Dynamic Event Tree Sampler

To clarify the idea behind the DET sampler currently available in RAVEN, a brief overview is needed. In technologically complex systems, such as NPPs, an accident scenario begins with an initiating event and then evolves over time through the interaction of deterministic and stochastic events. This mutual action leads to the production of infinitely many different outcomes. When for the same point in the input space the system might generate more than one final state the input output correspondence is not anymore bijective. At any time along the trajectory of the system in the phase space, the system might take a different path that is determined by a multivariate PDF. Since the continuous problem is almost intractable, an approximate approach is needed to perform the PRA analysis. An approximation alternative is offered by the DET approach.

In PRA analysis, in the conventional event tree [13] approaches, branches are used to differentiate among different statuses of the system and they do not have a temporal meaning (e.g., auxiliary generator working/not working). This approach lacks the capability to evaluate the impact of timing of the transition between different states of the system (in reality some treatment is possible but in a very costly fashion). To overcome these limitations, a "dynamic" approach is needed. The DET [13] technique brings several advantages, among which is the fact that it simulates probabilistic system evolution in a way that is consistent with its deterministic time evolution. This is done by taking the timing of events explicitly into account, leading to a more realistic and mechanistically consistent analysis of the possible evolution of the system. This feature of the DET is very important, for example, when the complexity of the system leads to strong non-linear responses that characteristically evolve over time (the non-linear structure of the system strongly changes during the time evolution of the scenario). This result is obtained by "letting" the system code determine the pathway of an accident scenario within a probabilistic "environment."

This strategy requires a tight interaction between the system code and sampling driver (i.e., RAVEN framework). In addition, the system code must have a control logic capability (i.e., trigger system). For these reasons, the application of this sampling approach to a generic code needs more effort when compared to the other samplers available in RAVEN. Currently, the DET is fully available for the thermal-hydraulic codes RELAP-7 and RELAP5-3D (still in beta version). [14]

2.2.3 Models

The model entity, in the RAVEN environment, represents a "connection pipeline" between the input and output spaces. The RAVEN framework does not own any physical model (i.e., it does not possess the equations needed to simulate any physical system), but implements application program interfaces (APIs) to these models, which are supplied by the user. The RAVEN framework provides APIs for three different model categories:

- Codes
- Externals
- ROMs.

The code model represents the communication pipe between the RAVEN and any external software. Currently, RAVEN has implemented APIs for RELAP5-3D, RELAP-7, any Multi-Physics Object-Oriented Simulation Environment (MOOSE)-based [15] application, and the PHISICS code [16].

The external model allows the user to create, in a Python file (imported, at run-time, in the RAVEN framework), her/his own model (e.g., set of equations representing a physical model, connection to another code, or control logic.). This model is interpreted/used by the framework and, at run-time, becomes part of RAVEN itself.

Direct software interfaces or files perform the data exchange between RAVEN and the system code. If the system code is parallelized, data exchange by files is generally the preferred method since it is more optimized for large clusters.

2.2.3.1 Post-Processors

A Post-Processor (PP) in RAVEN 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" to the post-processor. RAVEN supports several different types of PPs.

Currently, the following PPs are available in RAVEN:

- Basic Statistics: computes the basic statistical characteristics of a given dataset. It contains algorithms to compute many of the most common statistic quantities.
- Comparison Statistics: computes statistics for comparing two different datasets.
- Safest Point: finds the farthest point from a given limit surface, thus "Safest Point".
- Limit Surface: identifies the transition zones that determine a change in the status of the system, e.g. fail or pass.
- Limit Surface Integral: computes the probability of an event, whose boundaries are represented by a Limit Surface, given as an input either from "Limit Surface" PP or "Adaptive Sampling".
- External Post-Processor: executes any arbitrary python function defined externally using the "Functions" interface.
- Topological Decomposition: computes approximated hierarchical Morse-Smale complex (AMSC) and performs linear regression on each component.
- Data Mining: identifies the patterns in the given dataset by employing the unsupervised learning algorithms of choice found in the SciKit-Learn library [7].

The PPs allow the users to understand the correlation between the input space and output space as well as to identify the importance of the parameters in the input space with respect to an FOM, which allows the user to clearly identify the operational bounding box of the system.

2.2.4 Simulation Environment

Figure 2 shows a schematic representation of the whole framework, highlighting the communication pipes among the different modules and engines. As seen in Figure 2, all the components discussed so far are addressed. In addition, the data management, mining, and processing modules are shown.

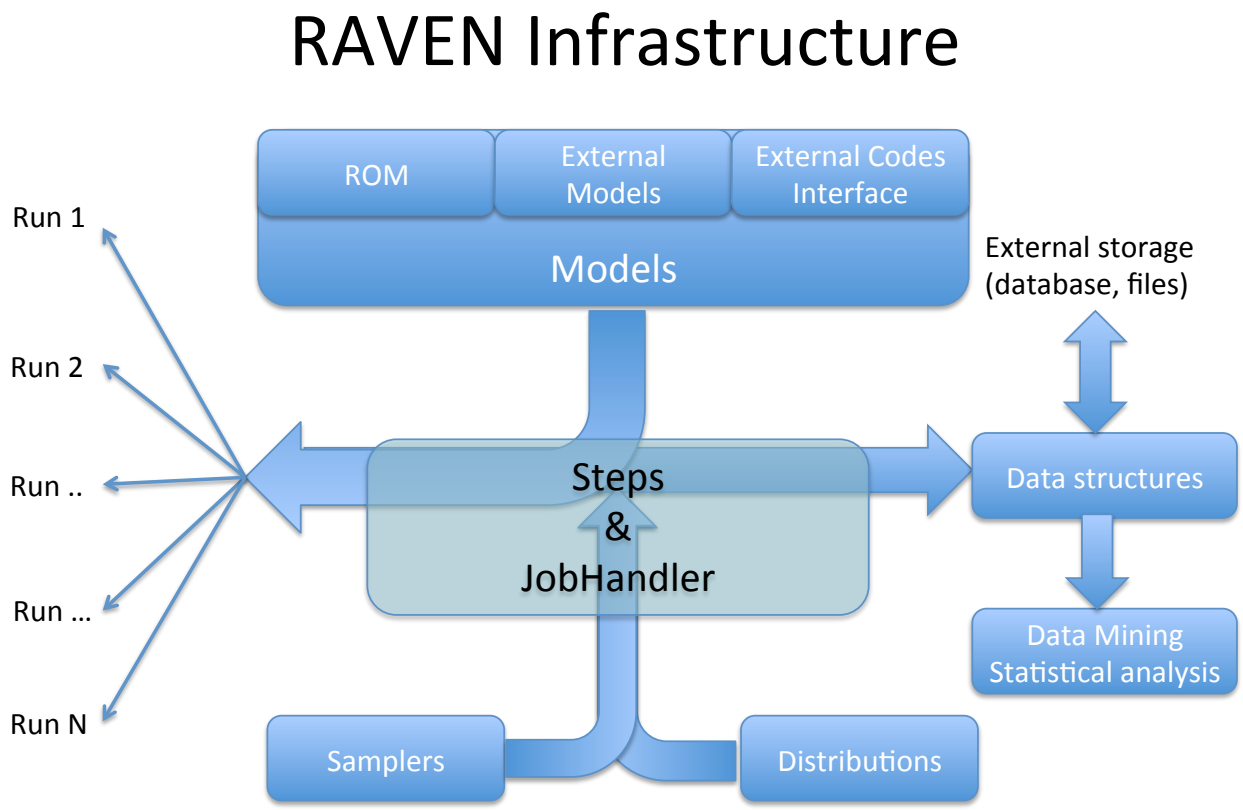

Figure 2: RAVEN schematic module interaction

From a user's standpoint, RAVEN is perceived as a pool of tools and data. Any action in which the tools are applied to the data is considered a 'step' in the RAVEN environment. Since this report is focused on post-processing algorithms, that require processing of the output data, only the "step" that is designed for this task (PostProcess) is mentioned.

The "PostProcess" step is designed to post-process data or to manipulate RAVEN entities. It is aimed at performing a single action that is employed by a "Model" of type "PostProcessor". The post-processor step requires at least one input entity, which is generally a "DataObjects", and an output entity. The type of the output entity depends on the type of the PostProcessor used as a "Model".

The job handler, at the center of Figure 2 is currently capable of running different instances of a code in parallel and can also handle codes that are internally multithreaded or use any form of message passing interface in a parallel implementation.

RAVEN is capable of plotting the simulation outcomes. The plotting can be performed either at the end of sampling or simultaneously with the sampling process. RAVEN is also capable of storing the data in a text file in CSV format or in an HDF5 format database file for later recovery. To illustrate the plotting capabilities of RAVEN *all the plots in this report were generated directly through RAVEN*.

3. DATA MINING ALGORITHMS

Knowledge discovery in databases (KDD) is the process of discovering useful/hidden knowledge from a collection of data. Several methods are available in extracting patterns that provide valuable, previously unknown, insight into the data. This information can be predictive or descriptive in nature. Data mining is the pattern extraction phase of KDD. Many algorithms can be used for data mining; the choice depends on the desired results. Major data mining application areas include marketing, fraud detection, telecommunication, and manufacturing.

The overall goal 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).

This usually involves using database techniques such as spatial indices. These patterns can then be seen as a kind of summary of the input data, and may be used in further analysis or, for example, in machine learning and predictive analytics. For example, data mining might identify multiple groups in the data, which can then be used to obtain more accurate prediction results by a decision support system. Neither the data collection (data preparation) nor resulting interpretation and reporting are part of the data-mining step, but do belong to the overall KDD process as additional steps. Although the KDD process involves many steps, this report focuses mainly on the implementation of the available data mining methods, especially focusing on cluster analysis, to RAVEN and application of these algorithms to sample datasets.

3.1 SciKit-Learn Algorithms

SciKit-learn is an open source, commercially usable, community-effort machine-learning library developed in Python. It contains simple and efficient tools for data mining and data analysis. Scikit-learn is built-on “NumPy”, “SciPy” and “matplotlib”, which are already a part of RAVEN code. It contains several machine-learning algorithms, supervised or unsupervised. The available unsupervised machine-learning algorithms are described in the following sections.

3.1.1 Gaussian Mixture Models

A Gaussian Mixture Model (GMM) 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. The GMMs are like kernel density estimates, but with a small number of components. Mixture models can be seen as a soft version of “k-means” clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians. Scikit-learn implements different classes to estimate GMMs, which correspond to different estimation strategies. The details are shown in the following sections.

Figure 3 illustrates how a RAVEN plot would look like when a GMM classifier algorithm with a “spherical” covariance type is applied to a dataset with 6 well-defined clusters. The dataset here is created with the “blobs” method found in the scikit-learn library (`sklearn.datasets.blobs`), which generates isotropic Gaussian blobs for clustering and thus should be well suited for analyzing with a GMM method.

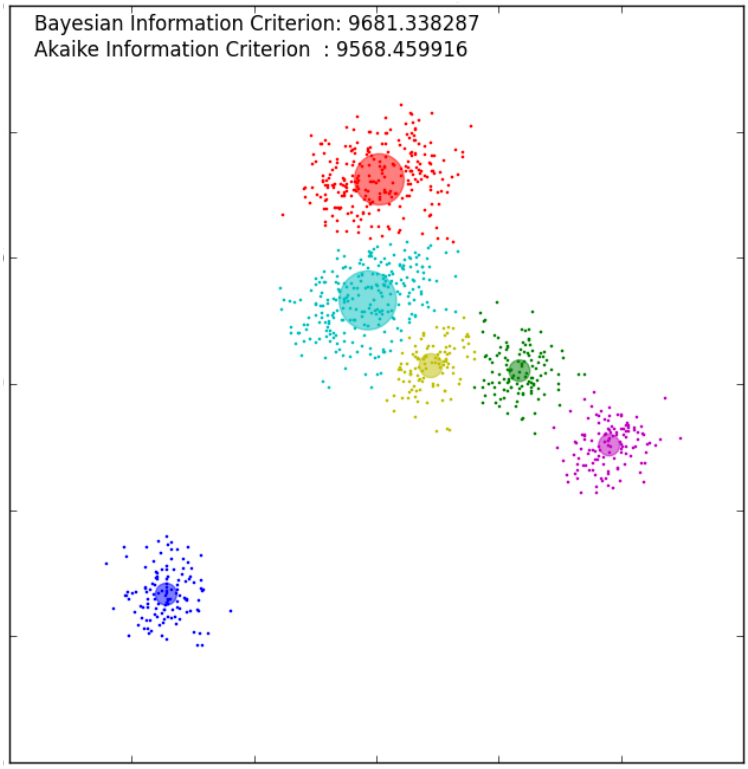

Figure 3: Gaussian Mixture Model classifier applied to an arbitrary dataset generated with the Scikit-learn Blobs module.

3.1.1.1 GMM classifier

The GMM object implements the expectation-maximization (EM) algorithm for fitting a mixture of Gaussian models. It can also draw confidence ellipsoids for multivariate models, and compute the Bayesian Information Criterion (BIC) to assess the number of clusters in the data as shown in Figure 3.

The GMM comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.

GMM is the fastest algorithm for learning a mixture model. This algorithm does not bias the means towards zero; it maximizes only the likelihood. However, when one has insufficient points per mixture, the algorithm is known to diverge and find solutions with infinite likelihood unless the covariances are artificially regularized. This algorithm always uses all of the components to which it has access. In the absence of external cues it needs held-out data or information theoretical criteria to decide how many components to use.

The Bayesian Information Criterion (BIC), which is a criterion for model selection among a finite set of models, can be used to select the number of components in a classical GMM. In theory, it recovers the true number of components only in the asymptotic regime (i.e. if a large amount of data is available).

In learning GMMs from unlabeled data, the information of which point comes from which latent component is usually not known. If this information is available, it is easy to fit a separate Gaussian distribution to each set of points. Expectation-maximization (EM) is a well-established statistical

algorithm to get around this problem through an iterative process. First one assumes random components (randomly centered on data points, either learned from "k-means", or it can be just normally distributed around the origin) and computes for each point a probability of being generated by each component of the model. Then, one tweaks the parameters to maximize the likelihood of the data given those assignments. Repeating this process is guaranteed to always converge to a local optimum.

3.1.1.2 Dirichlet Process GMM Classifier (DPGMM)

The DPGMM implements a variant of the Gaussian mixture model with a variable (but bounded) number of components using the Dirichlet process, which is a probability distribution whose domain is itself a set of probability distributions. It is often used to describe the prior knowledge about the distribution of random variables. Although it increases the computational time, this algorithm doesn't require the user to choose the number of components. It only requires the user to specify an upper bound for the number of components and a concentration parameter. Concentration parameter is a special kind of parameter of a probability distribution. It can be thought as a parameter for determining how "concentrated" the probability mass of a sample from a Dirichlet distribution is likely to be.

3.1.1.3 Variational GMM Classifier (VBGMM)

The VBGMM implements a variant of the Gaussian mixture model with variational inference algorithms. It has some of the Dirichlet process properties; therefore it can be seen as a middle ground between GMM and DPGMM.

The variational solutions have less special cases than EM solutions because of the incorporation of prior information. It is an extension of EM that maximizes a lower bound on model evidence (including priors) instead of data likelihood. Both variational methods and EM are iterative algorithms that alternate between finding the probabilities for each point to be generated by each mixture and fitting the mixtures to these assigned points. The integration of prior information avoids the singularities often found in EM. However, this introduces biases to the model. It will bias all means towards the origin and it will bias the covariance to be more spherical. Due to its Bayesian nature, the variational algorithm needs more hyper-parameters than expectation-maximization, the most important of these being the concentration parameter alpha. Specifying a high value of `alpha` leads more often to uniformly-sized mixture components, while specifying small (between 0 and 1) values will lead to some mixture components getting almost all the points while most mixture components will be centered on just a few of the remaining points.

3.1.2 Cluster Analysis

Cluster analysis is the task of grouping the dataset in such a way that each data in the same group ("cluster") shows more similar characteristics than the data in the other groups ("clusters"). Cluster analysis is not an algorithm but the task to be performed. Several algorithms are used for clustering analysis; these algorithms can be significantly different in terms of what they consider a cluster and how efficiently they find them.

An overview of the different clustering algorithms found in Scikit-Learn is given in Table 1. Figure 4 illustrates the application of several clustering algorithms to the same datasets; note that all three datasets are 2 dimensional. Algorithms might show different behavior on very high dimensional datasets. The advantage of using the non-flat geometry clustering algorithms, such as DBSCAN, when the clusters have a certain shape, i.e. the Euclidian distance is not the right metric, is also illustrated in Figure 4: in the bottom two rows.

Non-flat geometry clustering is useful when the clusters have a specific shape, i.e. a non-flat manifold, and the standard Euclidean distance is not the right metric. Gaussian mixture models, useful for clustering, are described in previous section [3.1.1]. K-Means can be seen as a special case of Gaussian mixture model with equal covariance per component.

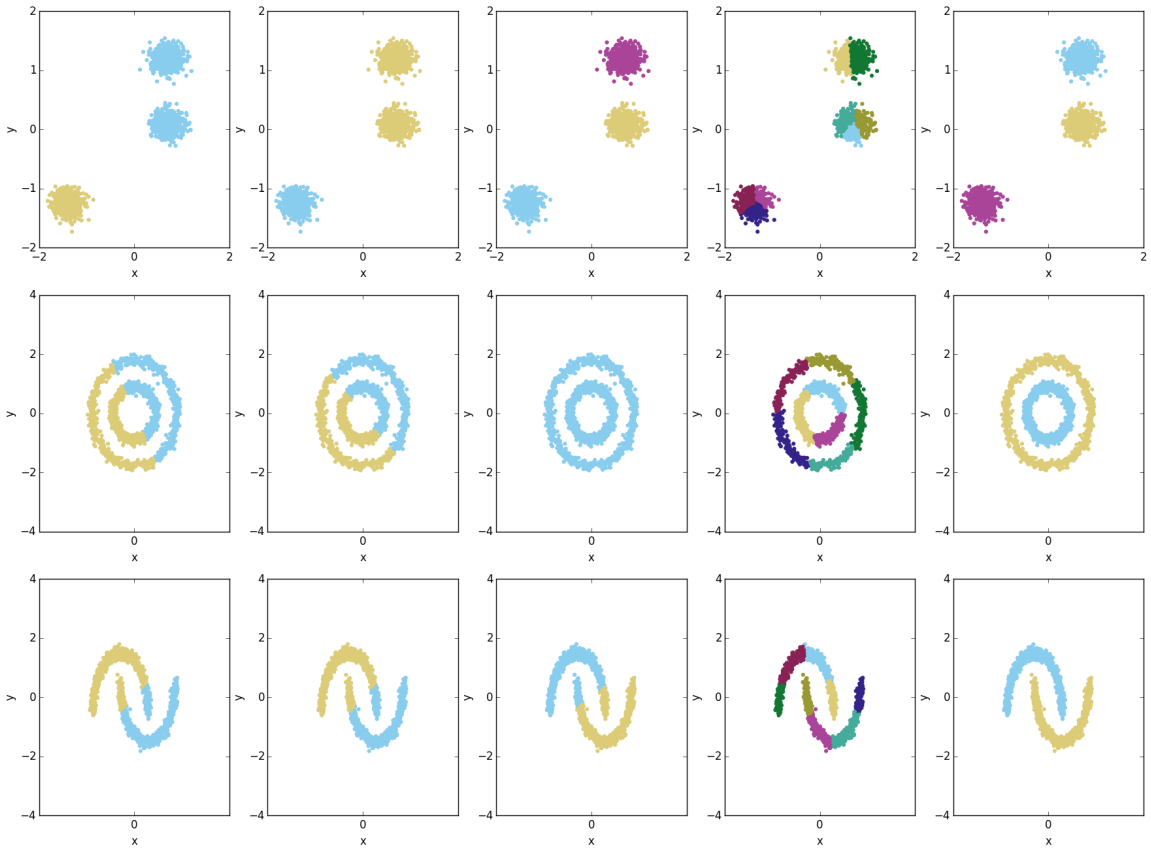

Figure 4: Clustering algorithms applied to sample datasets (left-to-right: K-Means, Mini Batch K-Means, Mean Shift, Spectral Clustering and DBSCAN).

Table 1: Overview of Clustering Algorithms [7]

Clustering Algorithm name	**Parameters**	**Scalability**	**Use case**	**Geometry (metric used)**
K-Means	Number of clusters	Very large n samples, medium n clusters with Mini Batch 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
Agglomerative 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

3.1.2.1 K-Means Algorithm

K-means algorithm is often referred to as Lloyd's algorithm. It separates the samples in n groups of equal variance, minimizing the within-cluster sum-of-squares, which is also known as the "inertia". 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.

The k-means algorithm divides a set of N samples X into K separate clusters C, each described by the mean μ_j of the samples in the cluster. The means are called the "cluster centroids"; the cluster centroids are not always points from X, although they are in the same space. The K-means algorithm aims to choose centroids that minimize the inertia:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} \left(\|x_j - \mu_i\|^2 \right)$$

Inertia, or the within-cluster sum of squares criterion, can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- It assumes that clusters are convex and isotropic. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- It is not a normalized metric: the lower values are better and zero is optimal. However, in very high-dimensional spaces, Euclidean distances tend to become inflated (so-called "curse of dimensionality"). Applying a dimensionality reduction algorithm prior to "k-means" clustering can alleviate this problem and speed up the computations.

The k-means algorithm has three steps: (1) choose the initial centroids; with the most basic method being to choose k samples from the dataset X, (2) assign each sample to its nearest centroid. And (3) create new centroids by taking the mean value of all of the samples assigned to each previous centroid. The algorithm iterates between the steps (2) and (3). The iteration repeats until the difference between the

old and the new centroids is less than a threshold. In other words, it iterates until the centroids do not move significantly.

K-means will always converge if given enough time, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. In practice, the analysis is often done several times, with different initializations of the centroids. The "k-means++" initialization scheme, which has been implemented in Scikit-learn, can aid in addressing this issue, as it initializes the centroids to be distant from each other, which leads to better results than random initialization, in such cases.

3.1.2.2 Mini Batch K-Means Algorithm

The Mini Batch K-Means algorithm is a variant of the K-Means algorithm. It uses mini-batches to reduce the computation time. Mini-batches are subsets of the input data, which are randomly sampled in each training iteration.

The algorithm iterates between two major steps: (1) b samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. (2) The centroids are updated, as in the k-means algorithm. However, in contrast to k-means, this is done on a per-sample basis. For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the sample and all previous samples assigned to that centroid. The algorithm iterate between these steps until convergence or a predetermined number of iterations is reached. Mini Batch K-Means converges faster than K-Means, but the quality of the results is reduced. In practice this difference in quality can be quite small.

3.1.2.3 Affinity Propagation Algorithm

Affinity Propagation creates clusters by sending messages between pairs of samples, i.e. it is based on the concept of "message passing". Unlike other clustering algorithms such as "k-means" affinity propagation does not require the number of clusters to be determined before running the algorithm.

Affinity propagation describes the dataset using a small number of exemplars, which are the member of the input set representative of the clusters. 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.

Although Affinity Propagation does not require the determination of the number of clusters prior to running because it chooses the number of clusters, the main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order $O(N^2T)$, where N is the number of samples and T is the number of iterations until convergence. Furthermore, the memory complexity is of the order $O(N^2)$ if a dense similarity matrix is used, but reducible if a sparse similarity matrix is used. Therefore the Affinity Propagation is most appropriate for small to medium sized datasets.

The messages sent between points belong to one of two categories. The first is the responsibility $r(i,k)$, which is the accumulated evidence that sample k should be the exemplar for sample i. The second is the availability $a(i,k)$ that is the accumulated evidence that sample i should choose sample k to be its exemplar, and considers the values for all other samples that k should be an exemplar. In this way, exemplars are chosen by samples if they are (1) similar enough to many samples and (2) chosen by many samples to be representative of themselves.

More formally, the responsibility of a sample *k* to be the exemplar of sample *i* is given by:

$$r(i,k) \leftarrow s(i,k) \; - \; max\left[a\left(i,\acute{k}\right) + s\left(i,\acute{k}\right) \forall \acute{k} \neq k\right]$$

Where $s(i,k)$is the similarity between samples *i* and *k*. The availability of sample *k* to be the exemplar of sample *i* is given by:

$$a(i,k) \leftarrow min\left[0, r(k,k) + r(\acute{\imath},k) + \sum_{\acute{\imath}\, s.t. \acute{\imath} \notin \{i,k\}} r(\acute{\imath},k)\right]$$

To begin with, all values for *r* and *a* are set to zero, and the calculation of each iterates until convergence.

3.1.2.4 Mean Shift Algorithm

Mean Shift clustering is a nonparametric clustering technique, which does not require prior knowledge of the number of clusters in the dataset. It is a centroid-based algorithm, which aims to discover blobs in a smooth density of samples. It 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. Given a candidate centroid x_i for iteration t, the candidate is updated according to the following equation:

$$x_i^{t+1} = x_i^t + m\left(x_i^t\right)$$

Where $N(x_i)$ is the neighborhood of samples within a given distance around x_i and m is the mean shift vector. The mean shift vector is computed for each centroid. It always points toward the direction of the maximum increase in the density of points. Mean shift vector is computed using the following equation, effectively updating a centroid to be the mean of the samples within its neighborhood:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K\left(x_j - x_i\right) x_j}{\sum_{x_j \in N(x_i)} K\left(x_j - x_i\right)}$$

The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during the execution. However, it is guaranteed to converge. Because the mean shift procedure, obtained by successive:

- computation of the mean shift vector $m\left(x_i^t\right)$,
- translation of the window $x_i^{t+1} = x_i^t + m\left(x_i^t\right)$,

is guaranteed to converge to a point where the gradient of density function is zero. The algorithm will stop iterating when the change in centroids is small.

3.1.2.5 Spectral Clustering Algorithm

Spectral Clustering does a low-dimension embedding of the affinity matrix between samples, followed by a K-means in the low dimensional space. Spectral Clustering requires the number of clusters

to be specified. It works well for a small number of clusters and it is not advised when using many clusters. It solves a convex relaxation of the normalized cuts problem on the similarity graph: for two clusters, it cuts the graph in two so that the weight of the edges cut is small compared to the weights of the edges inside each cluster.

If the values of the affinity matrix are not well distributed, e.g. with negative values or with a distance matrix rather than a similarity, the spectral problem will be singular and the problem is not solvable. In which case a transformation should be applied to the entries of the matrix. For instance, in the case of a signed distance matrix, is common to apply a heat kernel:

$$similarity = exp\left(-\beta\frac{distance}{\sigma_{distance}}\right)$$

Spectral Clustering basically applies clustering to a projection to the normalized Laplacian. In practice, Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance, when clusters are nested circles on the 2D plan (Figure 4: middle row).

3.1.2.6 DBSCAN Clustering Algorithm

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm views clusters as areas of high density of data points. The data points in the low-density areas are seen as noise and border points, which are actually separating the clusters. Clusters found by DBSCAN can be any shape because of this approach.

The main element of the DBSCAN algorithm is the concept of core samples, which are samples that are in areas of high density. Therefore, a cluster is a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). There are two parameters to the algorithm: `min_samples` and `eps`. Higher `min_samples` or lower `eps` indicate higher density necessary to form a cluster.

A cluster is a set of core samples, that can be built by recursively by taking a core sample, finding all of its neighbors that are core samples, finding all of their neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples; these are on the borders of a cluster.

The DBSCAN algorithm finds core samples of high density and expands clusters from them. It is good for data, which contains clusters of similar density.

3.1.2.7 Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not a trivial task as counting the number of errors or the precision. Especially, any evaluation metric should take into account if it defines the separations of the data to some ground truth set of classes or if it satisfies that the members belong to the same class are more similar than 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 cluster (a).
2. The mean distance between a sample and all other points in the next nearest cluster (b).

The Silhouette Coefficient *s* for a single sample in terms of a and b is then given as:

$$s = \frac{b - a}{max(a, b)}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample.

The Silhouette coefficient 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. The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

3.1.3 Dimensionality Reduction Techniques

The dimensionality reduction techniques are usually performed to high-dimensional datasets before applying the data mining algorithms in order to avoid the effects of the "curse of dimensionality". The dimension reduction reduces the time and storage required while it makes it easier to visualize the data reduced to very low dimension, such as 2D or 3D. The removal of multi-collinearity improves the performance of the machine learning.

3.1.3.1 Principal Component Analysis (PCA)

PCA converts a set of correlated or uncorrelated variables into a set of linearly uncorrelated variables called principal components using an orthogonal transformation. The number of principal components cannot be larger than the number of original variables. The first principal component has the largest possible variance, and each succeeding component has the highest variance possible under the constraint that it is orthogonal to the preceding component(s). The resulting vectors are an uncorrelated orthogonal basis set.

Application of PCA algorithm to Iris dataset from scikit-learn library is illustrated in Figure 5. Iris dataset consists of 3 different types of irises' petal and sepal length and it has 4 features. Figure 5 shows the dataset when projected to 2 most important dimensions.

Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, PCA is implemented as a `transformer` object that learns `n` components in its `fit` method, and can be used on new data to project it on these components.

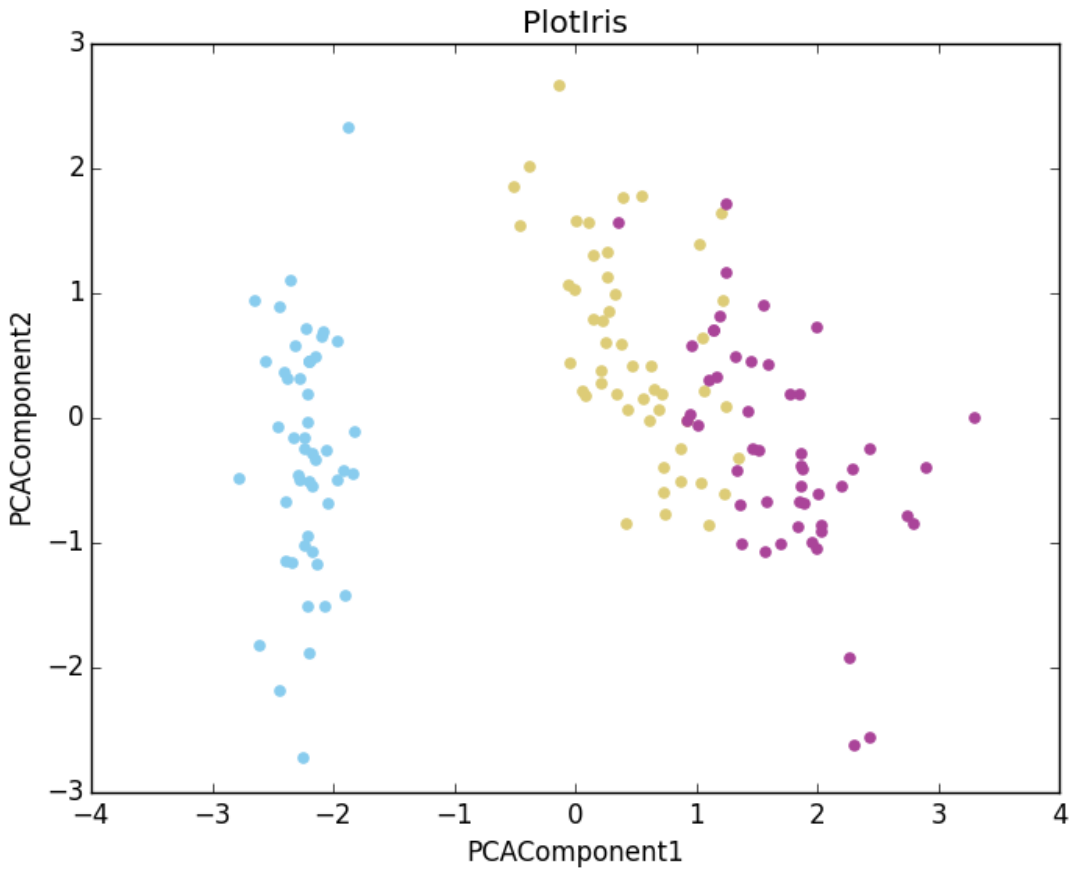

Figure 5: PCA implementation

Approximate PCA (or Randomized PCA)

The approximate PCA is a linear dimensionality reduction using an approximated Singular Value Decomposition (SVD) of the data. It keeps only the most significant singular vectors to project the data to a lower dimensional space. The PCA algorithm can be used to linearly transform the data while both reducing the dimensionality and preserving most of the explained variance at the same time.

Kernel PCA

Kernel PCA is an extension of PCA, which achieves non-linear dimensionality reduction through the use of kernels. It has many applications including denoising, compression, and structured prediction (kernel dependency estimation).

Sparse PCA and Mini Batch Sparse PCA

Sparse PCA has the goal of extracting the set of sparse components that best reconstruct the data. Mini-batch sparse PCA is a variant of Sparse PCA that is faster but less accurate. Iterating over small chunks of the set of features increases the speed.

3.1.3.2 Truncated SVD and Latent Semantic Analysis (LSA)

Truncated SVD implements a variant of (SVD) that only computes the user specified *k* largest singular values.

LSA transforms term-document matrices to a "semantic" space of low dimensionality if the truncated SVD is applied to such matrices. LSA is mentioned here because it is available in the Scikit Library, and the API is present in RAVEN. However, LSA is more geared for natural language processing.

3.1.3.3 Independent Component Analysis (ICA)

Independent component analysis separates a multivariate signal into additive subcomponents that are maximally independent. ICA is typically used for separating superimposed signals, not for reducing dimensionality. Whitening, which is simply a linear change of coordinate of the mixed data, must be applied in ICA model for the model to be correct, because it does not include a noise term. This can be done internally using the `whiten` argument or manually using one of the PCA variants.

3.1.4 Manifold Learning Algorithms

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.

Manifold learning can be thought of as an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data. A typical manifold learning is unsupervised, even though supervised variants do exist. It learns the high-dimensional structure of the data from the data itself and does not require the use of predetermined classifications.

The manifold learning implementations available in scikit-learn are summarized in the sections below.

3.1.4.1 Isometric Mapping Learning

Isometric Mapping (Isomap) is one of the earliest approaches to manifold learning. It can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap assumes that the pair-wise distances are only known between neighboring points, and uses the Floyd–Warshall algorithm to compute the pair-wise distances between all other points. Isomap estimates a lower-dimensional embedding which maintains geodesic distances between all points.

The Isomap algorithm comprises three stages:

1. **Nearest neighbor search**. Isomap uses `sklearn.neighbors.BallTree` for an efficient neighbor search. The cost is approximately $O[D \log(k)\, N \log(N)]$, for k nearest neighbors of N points in D dimensions.
2. **Shortest-path graph search**. The most efficient known algorithms for this are Dijkstra's Algorithm, which is approximately $O[N^2(k + \log(N))]$, or the Floyd-Warshall algorithm, which is $O(N^3)$ The algorithm can be selected by the user with the `path_method` keyword of Isomap. If unspecified, the code attempts to choose the best algorithm for the input data.
3. **Partial eigenvalue decomposition**. The embedding is encoded in the eigenvectors corresponding to the d largest eigenvalues of the $N \times N$ isomap kernel. For a dense solver, the cost is approximately $O(dN^2)$. This cost can often be improved using the `ARPACK` solver. The eigensolver can be specified by the user with the `path_method` keyword of Isomap. If unspecified, the code attempts to choose the best algorithm for the input data.

The overall complexity of Isomap is $O[D \log(k)\, N \log(N)] + O[N^2(k + \log(N))] + O[dN^2]$

N: number of training data points

D: input dimension

k: number of nearest neighbors

d: output dimension

3.1.4.2 Locally Linear Embedding

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data, which preserves distances within local neighborhoods. LLE begins by finding a set of the nearest neighbors of each point. It then computes a set of weights for each point that best describe the point as a linear combination of its neighbors. Finally, it uses an eigenvector-based optimization technique to find the low-dimensional embedding of points, such that each point is still described with the same linear combination of its neighbors.

The standard LLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. See discussion under Isomap above.
2. **Weight Matrix Construction**. $O[DNk^3]$ The construction of the LLE weight matrix involves the solution of a k×k linear equation for each of the N local neighborhoods
3. **Partial Eigenvalue Decomposition**. See discussion under Isomap above.

The overall complexity of standard LLE is $O[D \log(k)\, N \log(N)] + O[DNk^3] + O[dN^2]$.

N : number of training data points

D : input dimension

k : number of nearest neighbors

d : output dimension

3.1.4.3 Spectral Embedding

Spectral Embedding (also known as Laplacian eigenmaps) calculates a non-linear embedding. It relies on the assumption that the data lies on a low-dimensional manifold in a high-dimensional space. It finds a low-dimensional representation of the data using a spectral decomposition. Laplacian eigenmaps builds a graph from neighborhood information of the data set. The graph generated can be considered as a discrete approximation of the low dimensional manifold in the high dimensional space. Minimization of a cost function based on the graph ensures that points close to each other on the manifold are mapped close to each other in the low dimensional space, preserving local distances.

The Spectral Embedding algorithm comprises three stages:

1. **Weighted Graph Construction**. Transform the raw input data into a graph representation using an affinity (adjacency) matrix representation.
2. **Graph Laplacian Construction**. An unnormalized graph Laplacian is constructed as $L = D - A$ and normalized one as $L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}}$.
3. **Partial Eigenvalue Decomposition**. Eigenvalue decomposition is done on the graph Laplacian

The overall complexity of spectral embedding is $[D \log(k)\, N \log(N)] + O[DNk^3] + O[dN^2]$.

N : number of training data points

D : input dimension

k : number of nearest neighbors

d : output dimension

3.1.4.4 Multi-dimensional Scaling (MDS)

MDS represents the data in low dimensional space in a way that the distances respect well the distances in the original high-dimensional space. MDS is used for analyzing similarity or dissimilarity data.

There are two MDS algorithms: metric and non-metric. In Metric MDS, the input similarity matrix arises from a metric; the distances between two output points are then set to be as close as possible to the similarity or dissimilarity data. In the non-metric version, the algorithms try to preserve the order of the distances, and hence seek for a monotonic relationship between the distances in the embedded space and the similarities/dissimilarities.

3.2 Topology Post Processor

A new post-processor has been added to the RAVEN framework that is capable of decomposing arbitrary dimension input data mapped into a scalar function, that is a single output variable of interest modeled as a function of one or more parameters. The decomposition separates the input data space into monotonic regions as well as identifies the presence of local extrema. Such information can be useful for understanding the data from a structural point of view, but also these monotonic regions can be used to fit local low order models (linear) in order to better understand the main drivers over the separate regions of the input domain. In this way, RAVEN is able to create a structured sensitivity analysis that gives more useful information about input landscapes that are not monotonically increasing as compared to a global sensitivity analysis. In addition, being able to identify the location of local maxima and minima of an input dataset can be useful for determining areas of interest when further exploring the input domain in optimization problems or understanding the occurrence of important physical phenomena of a simulation.

Section 3.2.1 details the theory and algorithm used to approximate the aforementioned decomposition known as the Morse-Smale complex on a finite number of data points and explain an optional user interface used to augment the analysis of the algorithm's results. In Section 3.2.2, further use of the topological decomposition in RAVEN is investigated by using it as a ROM [17].

3.2.1 Implementation Details

The decomposition employed is derived from a branch of topology called Morse theory, and assumes the input represents a differentiable manifold space and the output is given as a smooth, real-valued function defined over the input space. Furthermore, this function should not have any degenerate critical points (witnessed by a singular Hessian matrix). Note, that since this structure is approximated on finite data, these restrictions only constrain the use to datasets that can be *modeled* as such. For example, a discrete step function can still be modeled as a smooth function. In some instances where a phenomenon under study is non-deterministic, it may make sense to obtain an average or nominal value per location in the input space, as the algorithm cannot handle duplicate input points.

Given a smooth function defined over a differentiable input space, any location in the input space can be traced from a source local minimum to a sink local maximum using the gradient direction of the function. An ascending (or unstable) manifold is composed of all locations that originate at the same local minimum, and the collection of ascending manifolds partition the input domain. Likewise, all locations

that terminate at the same local maximum comprise a descending (or stable) manifold. The collection of all descending manifolds over an input space is known as the Morse complex. The intersection of the ascending and descending manifolds creates the Morse-Smale complex and the relationship of these structures is demonstrated in Figure 6

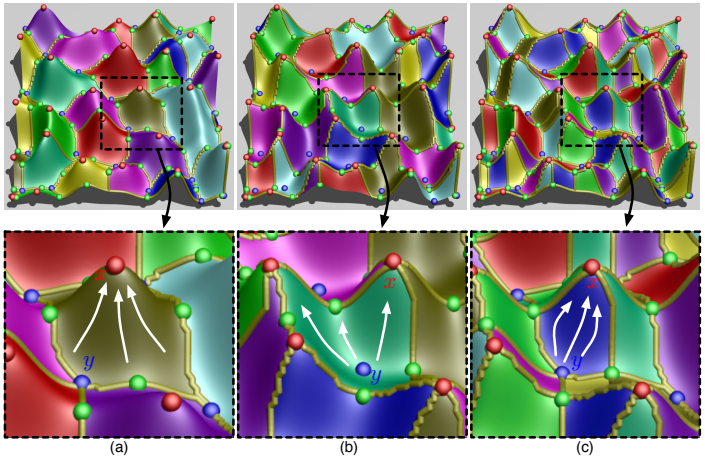

Figure 6: An example function decomposed into: (a) descending manifolds, (b) ascending manifolds, and (c) the Morse-Smale complex. Red dots represent local maxima, blue represent local minima, and green points represent saddle points.

In order to approximate this complex on finite data, a connectivity structure is imposed on the data such that the gradient direction can be estimated for each sample point. RAVEN provides three different types of neighborhood graphs that can be applied to the data before the topology is extracted. The first is the k-nearest neighbor graph that has a free parameter, k, specifying the number of neighbors, sorted by Euclidean distance from the query point, that are to be used to specify a neighborhood around a query point. Having a single parameter used everywhere in the input domain can be problematic if the sampling is not uniformly dense, as areas of high-density may favor one direction over another, and areas of low density may tend to be overly aggressive and connect distant neighbors. A more suitable option is to use an empty region graph that will tend to explore all directions around a point more uniformly.

To combat this effect RAVEN offers a family of empty region graphs as an alternative. Empty region graphs impose a restriction that any valid edge is associated with a specific geometric shape containing no data points besides the endpoints of the edge. In this way, edges emanating from a query point tend to fill the space of possible directions more evenly. Specifically, RAVEN uses the lune-based β-skeleton and its relaxed version made available through the NGL library [23]. The β-skeleton uses an empty region defined by a free parameter, β. The required empty region is the intersection of two balls with a diameter equal to d*β where β is in the range (0,2] and d is the edge length. In the setting where $\beta \leq 1$, the balls are aligned such that the edge is a secant line of both balls. In the setting where beta > 1, each ball is anchored to one of the endpoints and its center point is collinear with the potential edge. See Figure 7 for an illustration.

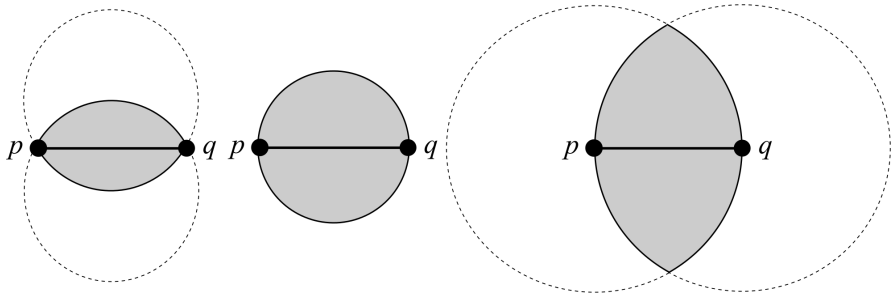

Figure 7: Examples of the empty region specified by various β-skeletons. From left to right, β < 1, β = 1, and β > 1.

Normally, the beta skeleton graph will reject an edge if any point intersects the lune-shaped intersection of the balls above, however this property can be relaxed to only reject edges if the intersecting point is already a neighbor of either endpoint of the edge in question. To do this appropriately, potential edges for a given query point have to be sorted and tested in order of increasing length to ensure correctness.

Once a neighborhood graph is placed on the data, the gradient flow at each point in the dataset can be estimated. The estimation is performed by following the steepest descending or ascending edges iteratively until a local extremum is encountered. Local extrema are identified by the absence of lower or higher valued neighbors.

Due to the discretization discussed above, possible noise in the input data, or the desired resolution of the data, the resulting topology may be over segmented. By utilizing the notion of persistence simplification, a hierarchical partitioning can be created whereby smaller features can be merged into larger features. The standard setting is shown in Figure 8where each local minimum or maximum is associated with a persistence value, the difference in function value between the extremum and its closest valued neighboring critical point, generally a saddle point. Due to the discretization all saddles are assumed to be simple 1-saddles, that is they lie between either two maxima or two minima. In this way, a saddle point and a local extremum can be canceled, and thus, the simplification simulates that flow is redirected from this extrema to the other higher persistence extrema on the other side of the saddle point.

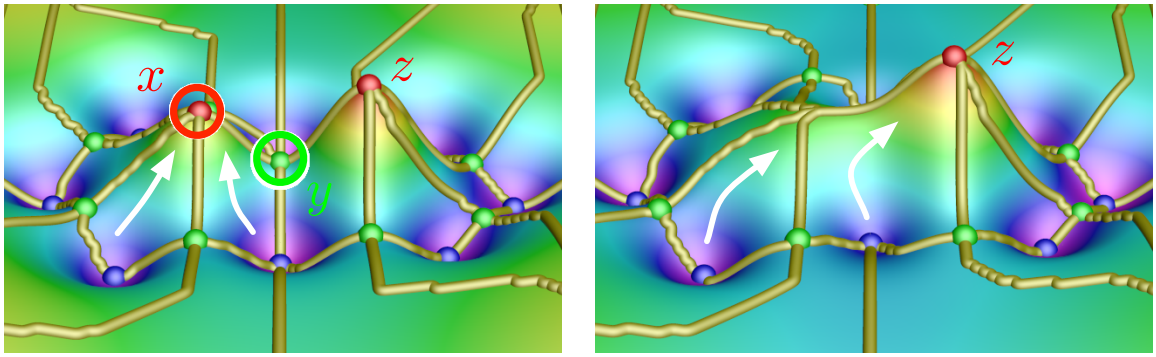

Figure 8: An example 2D function illustrating the effect after canceling the local maximum x and the circled green saddle point.

As a result of approximating a saddle for each pair of neighboring minima or maxima, the order bywhich extrema are simplified can be modified and still maintain a valid Morse-Smale complex. Due to this, other criteria than function value difference can be investigated in order to simplify the domain, such as size of the segment, where the smaller segment (fewer points) will merge into its neighboring larger segment. Also, total probability of a segment can be used, thus a more refined topology can be generated in the high probability regions and smoothing/flattening can be simulated in the areas of lower probability where less fidelity is required. Examples of these different simplification strategies are shown in Figure 9.

Once the data has been decomposed, each partition of the data is locally fit using a local linear regression. The linear regression can be weighted to fit the data more closely where the input probability is higher. This involves minimizing the least square error of the linear coefficients for each input parameter:

$$\hat{\beta} = arg\min_{\beta} s(\beta) = (X^T W X)^{-1} X^T W y$$

Where $\hat{\beta}$ is the set of coefficients, X is the set of input data, W is the set of associated probabilities, and y is the associated output data. The entire data set and the known local model associated to each partition of the data can be used to determine the quality of the linear regression by computing the coefficient of determination (R^2):

$$R^2 = 1 - \frac{\sum_{i=1}^{n} w_i \, (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} w_i \, (y_i - \bar{y}_i)^2}$$

Where $\bar{y}_i$ represents the mean of the observed output value, y_i is the true response value and $\hat{y}_i$ is the predicted value. In essence, this equation determines the prediction ability of the regression compared to that of a simple mean value predictor.

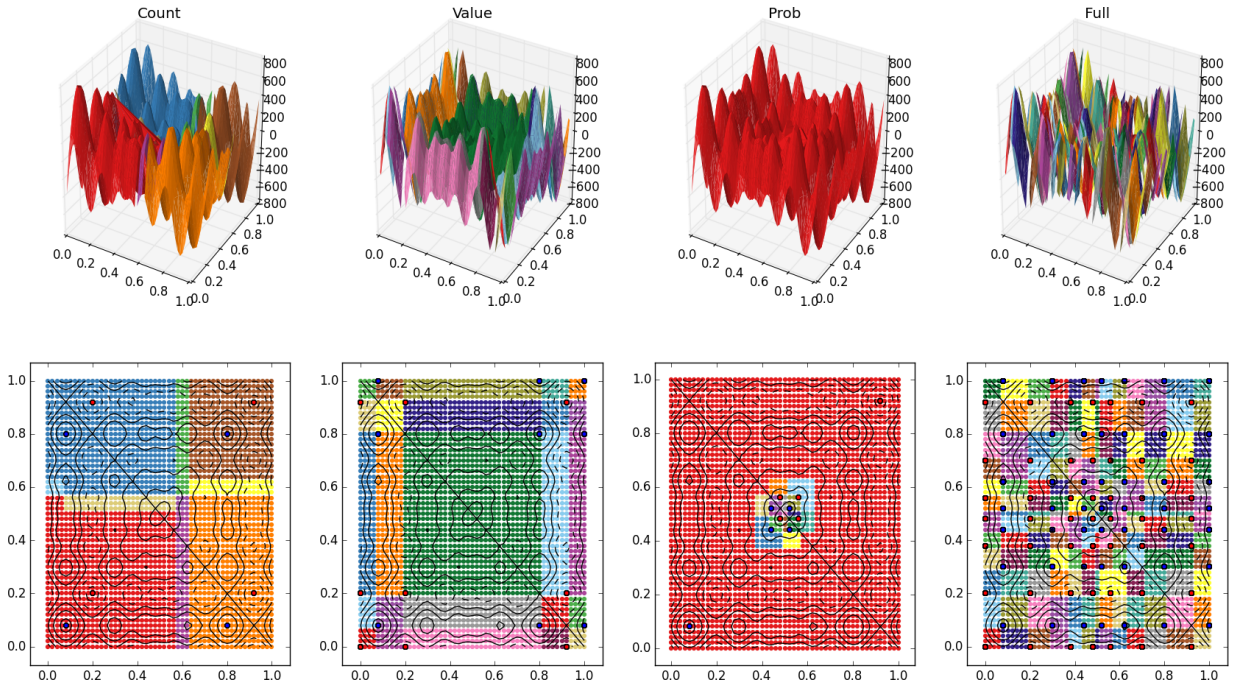

Figure 9: An example 2D function simplified using (from left): a point count metric where larger segments subsume smaller segments, the default persistence simplification metric based on function value difference, a probability metric (where each dimension consists of a normal distribution with a mean at x_i=0.5), and finally the full segmentation with no simplification performed.

The results of this algorithm can require some exploration of various levels of the hierarchy, and so an optional user interface for understanding the results of the topological decomposition and subsequent regression results is provided. An example of this interface on a synthetic dataset is provided in Figure 10.

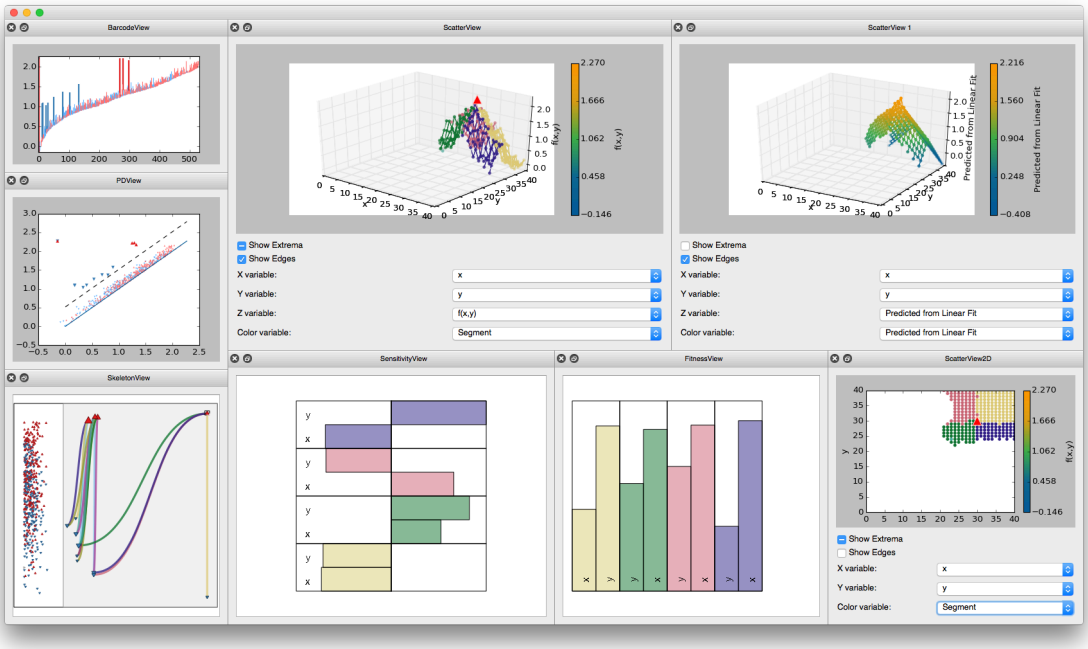

Figure 10: Optional user interface for interactively adjusting parameters and visualizing the effects on the sensitivity and fitness of the data.

The results include scatterplot projections of the data into 2 or 3 dimensions. Also, two standard topological summary plots including persistence barcodes (Figure 11 center) and a persistence diagram (Figure 11 right) help identify appropriate cutoff points for the simplification setting. The persistence plots show the relative persistence values of each extremum-saddle pairing and can identify areas of low density that represent a clean separation between signal and noise. A newly developed topology map (Figure 11 left) shows how the various extrema are connected via Morse-Smale cells as well as the relative persistence value of each extremum.

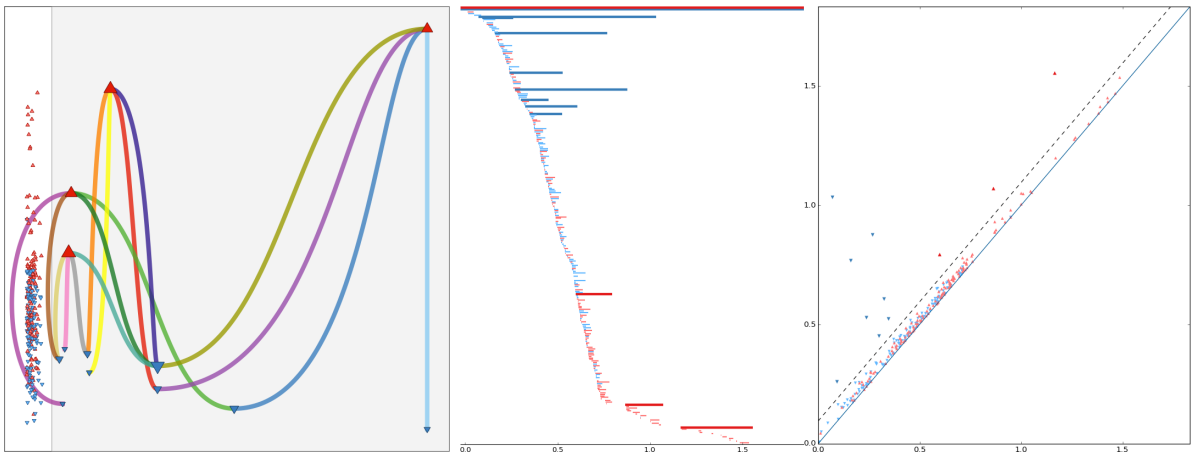

Figure 11: Persistence-based topological views of the data allow the user to select an appropriate simplification level in the topological hierarchy.

The signed sensitivity coefficients are plotted so that the relative importance of each input parameter on the output variable can be compared (Figure 12 center). This view makes it clearly apparent what the main drivers for each segment of the domain are by using uniformly scaled rectangles to allow direct

comparison of the sensitivity of any subset of parameters. In order to better understand the trustworthiness of the local linear models, the R^2 fitness plot (Figure 12 right) is used to build progressively more complex linear models by first using only the most important dimension and then adding dimensions in order of decreasing sensitivity magnitude. This is akin to stepwise linear regression. In this way, one is able to see at what point adding more dimensions does not increase the fidelity of the model. Lastly, this interface is all presented in a linked view fashion. That is, for example, selecting particular clusters in the topology map view will update all other views to only show the requested data.

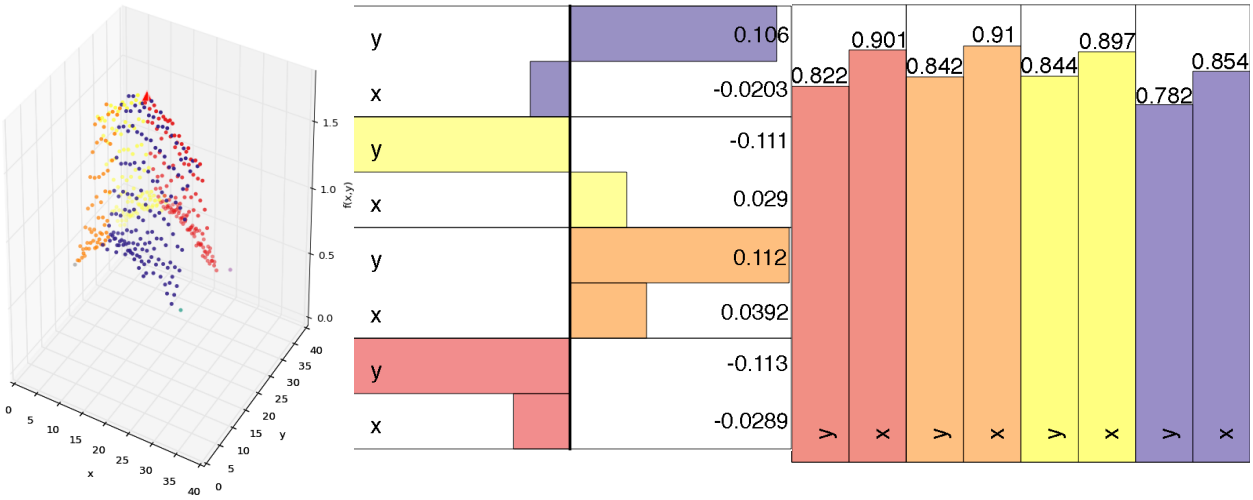

Figure 12: A scatterplot showing a set of partitions in the data (left), the associated sensitivity coefficients of each segment in the data (center), and the R^2 fitness plot demonstrating the quality of fit when performing stepwise regression.

3.2.2 Morse-Smale Regression ROM

In order to use the Morse-Smale complex as a predictor, the incoming query data needs to be associated with one or more of the partitions and then should be able to report either the most probable partition's prediction or intelligently blend the estimates from each locally constructed linear model. In order to do this in a sensible way, RAVEN provides both a kernel density estimator (KDE) and a support vector machine (SVM) that can assign weights to each local model for a given query location. With these weights, a user can either choose to use the highest weighted model or to blend the models based on their weights.

For the KDE method, a kernel function is used that accepts the distance between a query point and one training point. The kernel function has a maximal value at zero and falls off as the distance between the two locations increases. This ensures that the models composed of training data closest to the query location influence it the most. Thus, the weight of a linear model (w_i) is the average of the kernel function evaluated for each training data point:

$$w_i(x) = \frac{\sum_{j=0}^{n_i} K(\|x - x_j\|_2)}{n_i}$$

Several standard kernel methods are provided, however the examples reported in this report all use the Gaussian kernel. The full list of the provided kernel functions is shown in Figure 13. In addition, the KDE method provides the user with a bandwidth parameter that allows the user to specify the degree of falloff for the kernel function. A smaller bandwidth parameter will create steeper plots in Figure 13, whereas a larger bandwidth will stretch the plots along the x-axis.

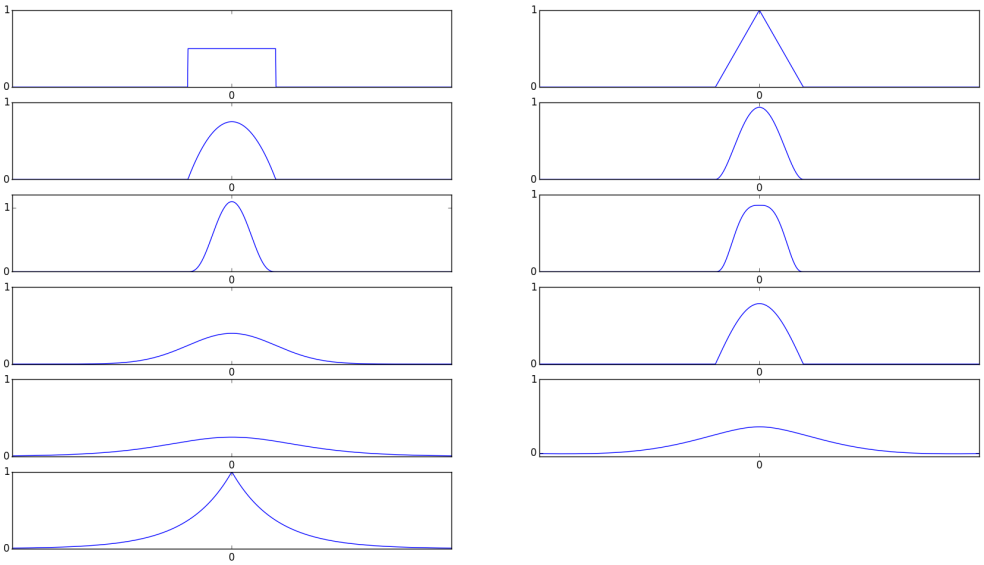

Figure 13: Kernels available for use in the KDE method. Left column (from top to bottom): uniform, Epanechnikov, triweight, Gaussian, logistic, and exponential. Right column (from top to bottom): triangular, biweight, tricube, cosine, and Silverman.

For the SVM version, a one-versus-one support vector classifier is constructed using scikit-learn's implementation of the support vector classifier. The class provided by scikit-learn directly provides a function (`SVC.predict_proba`) for predicting the probability that a query location belongs to a particular partition of the data. It is possible to allow the user to tune the SVM, however the results reported in this report use the default settings for scikit-learn's SVM. Results of these different types of model prediction are reported in Figure 14.

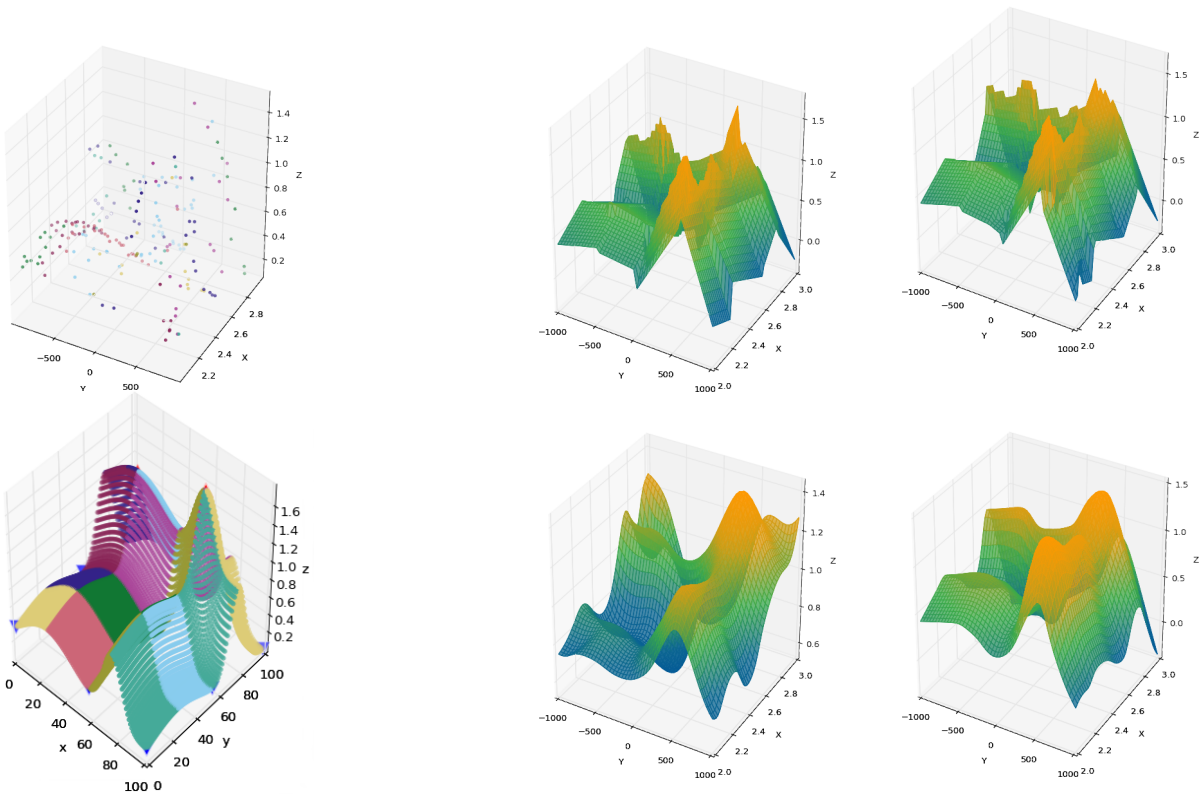

Figure 14: Prediction results using various versions of the Morse-Smale regression ROM. Clockwise from top left: The initial training data (colored by decomposed segmentation), the hard cutoff SVM, the hard cutoff KDE using a Gaussian kernel, the smooth KDE with a Gaussian kernel, the smooth SVM, and the true response being modeled (colored by the true topological decomposition).

4. TEST CASES AND RESULTS

The testing of the data mining algorithms has already been performed with sample datasets (section 3.1 and 3.2); this section presents the application of the data mining and topology post-processors to real physical simulations.

Section 4.1 applies the topology post-processor to a BISON [21] nuclear fuel simulation where RAVEN is used to perturb the parameters to generate an input space for analysis.

Section 4.2 applies the data mining post-processor to the same dataset as in section 4.1 and a few other datasets, which are described in the section.

4.1 Test Cases

4.1.1 BISON Nuclear Fuel Performance Simulation

For the testing the data mining algorithms a BISON fuel rodlet performance simulation was performed where RAVEN used to perturb the parameters to generate an input space for the analysis.

The rodlet under consideration is axis-symmetric and composed of ten stacked UO_2 pellets and surrounded by a zirconium alloy cladding. The BISON mesh used in the simulation is shown in Figure

15, note that, 10 pellets are simulated as a single, long pellet sandwiched between two insulator pellets. In this example, the maximum Von Mises stress occurring in the midplane of the cladding and 12 other FOM (shown in Table 2) were calculated over the course of several simulations where three different input parameters were modified. Too much stress in the cladding can cause the cladding to crack and allow radioactive gas to leak into the plant environment and this is the reason to chose middle plane maximum Von Mises stress as explicative FOM. The three parameters under study are the linear power scaling factor of the reactor (power_scalef), the grain radius scaling factor of the UO_2 fuel pellets considered (grainradius_scalef), and the thermal expansion coefficient of the reactor fuel (thermal_expansion).

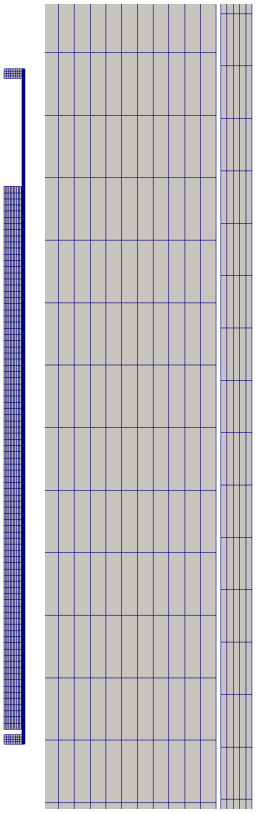

Figure 15: The BISON mesh used in the simulation: smeared pellet stack with hafnium insulator end pellets.

In this simulation, the linear power is ramped up from 0 W/m to 25000 W/m over the first 10000 seconds of the simulation before leveling off. Note that the power_scalef will augment the final power in each simulation. The goal is to attain a better understanding of the contact that occurs when the fuel pellets expand to the point of touching the cladding. The stress on the cladding at t=0 is due to a compressive force from the water pressure outside of the cladding. As fission occurs, the fuel rod expands as it is heated due to thermal expansion and swelling from the release of fission gas within the microstructure of the fuel. These same factors cause a force that counteracts the compressive force from the external water pressure, therefore the stress in the cladding decreases for a time until these forces reach equilibrium. After the equilibrium point, the expansive forces on the cladding begin to dominate, thus causing more stress on the cladding as it expands.

In the above scenario, contact is generally not indicative of a failure state, and is actually expected in these types of environments. The described problems arise only when the stress in the cladding becomes too high. In this study, RAVEN was used to stochastically sample the three parameters: power_scalef,

grainradius_scalef, and thermal_expansion of a BISON input file in order to generate a sample set of 17390 simulations.

Table 2: Parameters calculated in the Bison fuel analysis (output space)

No	Parameter	Description
1	Max_stress	Maximum Stress
2	Max_vonmises_stress	Maximum Von Mises Stress
3	Max_hoop_stress	Maximum Hoop Stress
4	Avg_clad_temp	Average Clad Temperature
5	Ave_temp_interior	Average Interior Temperature
6	Fis_gas_released	Released fission gas
7	Centerline_temp	Fuel Centerline Temperature
8	Midplane_hoop_stress_clad	Maximum Hoop Stress occurring at Clad midplane
9	midplane_hoop_strain_clad	Maximum Hoop Strain occurring at Clad midplane
10	midplane_hoop_strain_fuel	Maximum Hoop Strain occurring at Fuel midplane
11	Midplane_vonmises_stress	Maximum Von Mises Stress occurring at Clad midplane
12	Rod_input_power	Input power per rod
13	Rod_total_power	Total rod power

4.1.2 Station Black-Out Case

A Boiling Water Reactor (BWR) Station Black-Out (SBO) simulation was performed in [17] using RELAP5-3D and RAVEN. The system considered is a generic BWR power plant with a Mark I containment. The details of the model (shown in Figure 16) and the accident scenario can be found in [17], however it will be shortly mentioned here for completeness:

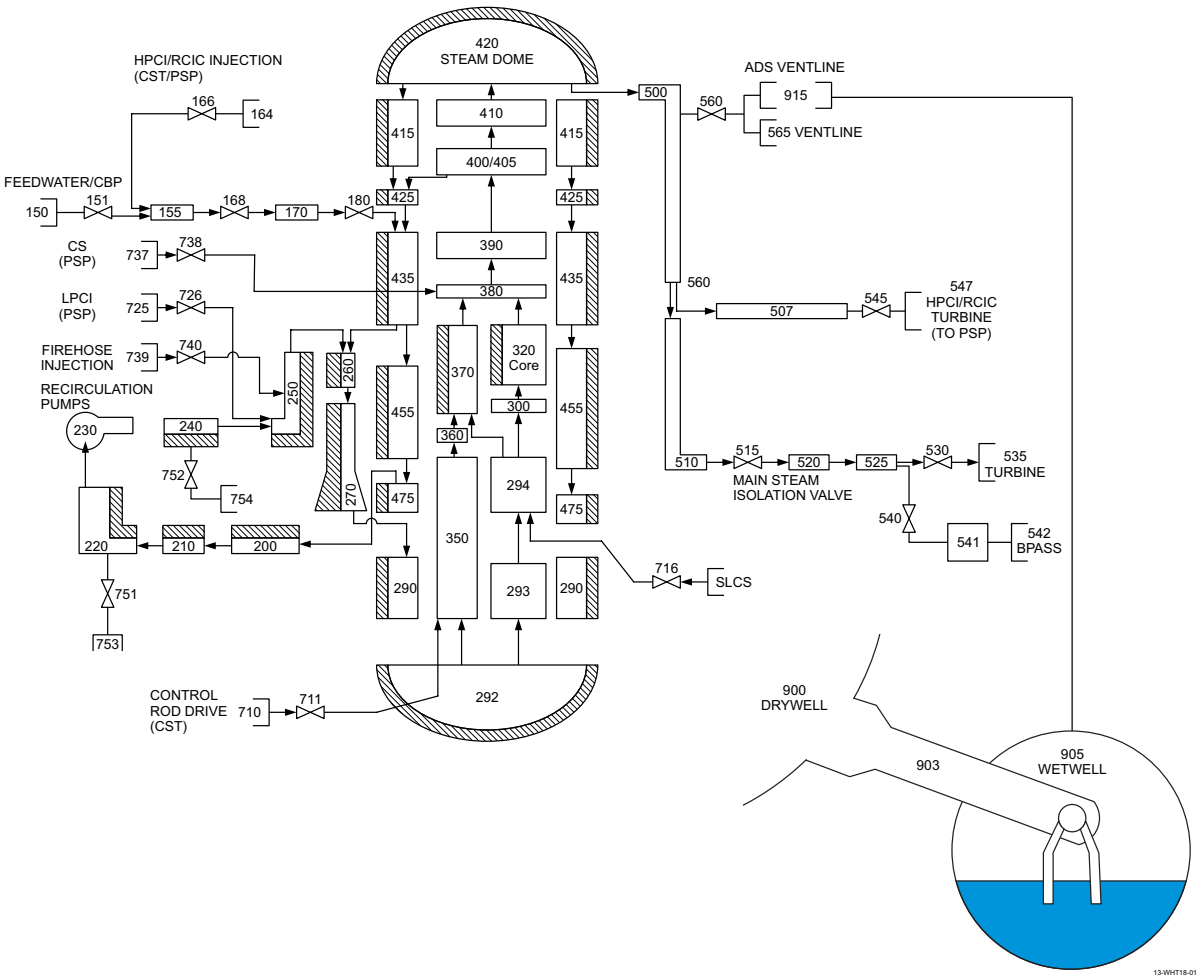

Figure 16: Schematic of the model used in the SBO Case [17]

- Reactor Pressure Vessel (RPV) level control systems provide manual/automatic control of the RPV water level. It consists of:
    1. Reactor Core Isolation Cooling (RCIC) provides high-pressure injection of water from the Condensate Storage Tank (CST) to the RPV.
    2. High Pressure Core Injection (HPCI) is similar to RCIC, but it allows greater water flow rates
- Safety Relief Valves (SRVs) are DC-powered valves that control and limit the RPV pressure.
- Automatic Depressurization System (ADS) is a separate set of relief valves that are employed in order to depressurize the RPV.
- Cooling water inventory:
    1. CST contains fresh water that can be used to cool the reactor core.
    2. Pressure Suppression Pool (PSP) water contains a large amount of fresh water that is used to provide the ultimate heat sink when AC power is lost.
    3. Firewater system can be injected into the RPV when other water injection systems are disabled and when RPV is depressurized.
- Power systems consists of two power grids, emergency diesel generators (DGs) and battery systems for the instrumentation and control systems.

The accident scenario considered is a loss of off-site power (LOOP) followed by loss of the DGs, i.e. SBO initiating event:

- At time $t = 0$: the following events occur:
  - LOOP condition occurs due to external events (i.e., power grid related)
  - LOOP alarm triggers the following actions:
    - Operators successfully scrams the reactor
    - Emergency DGs starts successfully
    - Core decay heat is removed from the RPV through the RHR system
    - DC systems are functional
- SBO condition may occur: due to internal failure, the set of DGs fails, thus removal of decay heat is impeded. Reactor operators start the SBO emergency operating procedures and perform:
  - RPV level control using RCIC or HPCI
  - RPV pressure control using SRVs
  - Containment monitoring (both dry well and PSP)
- Plant operators start recovery operations to bring back on-line the DGs while the recovery of the power grid is underway by the grid owner emergency staff
- Due to the limited life of the battery system and depending on the use of DC power, battery power can deplete. When this happens, all remaining control systems are offline causing the reactor core to heat until clad failure temperature is reached, i.e., core damage (CD)
- If DC power is still available and one of these conditions is reached:
  - Failure of both RCIC and HPCI
  - HCTL limits reached
  - Low RPV water level

  then the reactor operators will activate the ADS system in order to depressurize the RPV
- Firewater injection: as an emergency action, when RPV pressure is below 100 psi plant staff can connect the firewater system to the RPV in order to cool the core and maintain an adequate water level. Such task is, however, hard to complete since the physical connection between the firewater system and the RPV inlet has to made manually
- When AC power is recovered, through successful re-start/repair of DGs or off-site power, RHR can be now employed to keep the reactor core cool.

The twelve parameters are sampled in 20000 simulations using Monte Carlo Sampling. These parameters are summarized in Table 3. More details on the sampling and simulation strategies can be found in [17].

The simulations are evaluated in terms of core damage probability and the main focus was on performing a limit surface analysis.

Table 3: Summary of the stochastic parameters and their associated distributions

No.	**Stochastic Variable***	**Distribution Type**
1	Failure time of DGs (h)	Exponential
2	Recovery time of DGs (h)	Weibull
3	Battery life (h)	Triangular
4	SRV 1 fails to open (h)	Bernoulli
5	Offsite AC power recovery (h)	Lognormal
6	Clad Fail Temperature (F)	Triangular

7	HPCI fails to run (h)	Exponential
8	RCIC fails to run (h)	Exponential
9	Battery failure time (h)	Exponential
10	*Battery recovery time (min)*	Lognormal
11	*Firewater availability time (min)*	Lognormal
12	Fire water flow rate (gpm)	Uniform
* - Parameters related to human operations are in *italics*		

4.2 Applications of Data Mining Post-Processor

4.2.1 BISON Fuel Simulation

The data mining post-processor implemented in RAVEN has been tested on a BISON fuel rodlet performance simulation. The findings of the data mining postprocessor available in RAVEN are discussed in the following paragraphs.

Figure 17 shows the cluster labels when the k-means algorithm is applied to the full output space and the projection of the cluster labels is made on the input space. The cluster labels show that the variation of the "power scaling factor" dominates the transition between the clusters, which means that the output space is more sensitive to the "power scaling factor" than the other two input parameters.

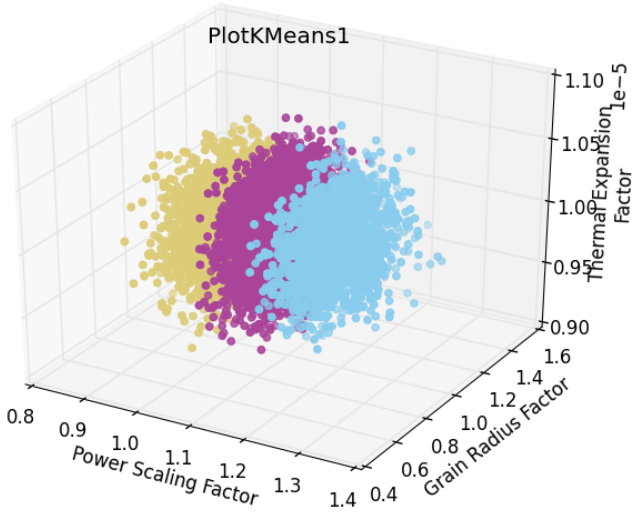

Figure 17: Input space with cluster labels obtained with k-means algorithm applied to output

When the clustering is applied to the whole output space the clusters show some noise as seen in Figure 17 and Figure 18, especially it is better visualized in the middle and bottom plots of

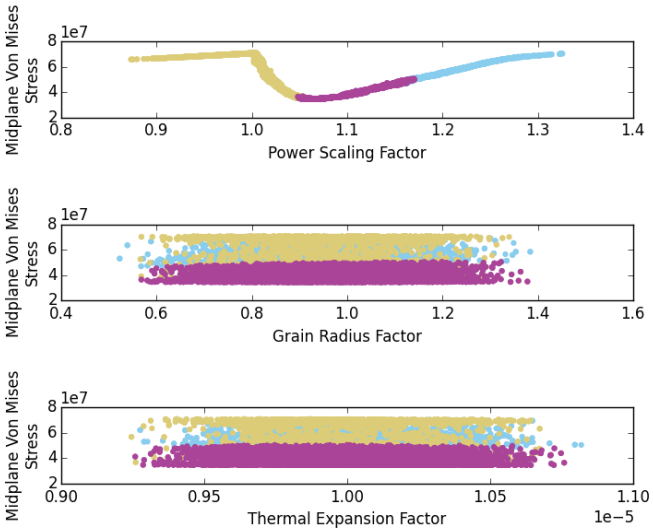

Figure 19. The noise is the effect of other parameters with non-negligible importance to the output parameters; in this case the "Power Scale Factor" shows much more significant importance compared to other parameters.

For example the noise in the middle and bottom plots of Figure 19 is therefore due to the "grain radius scaling factor" which lead the transition yellow to purple. Figure 20 illustrates when a dimensionality reduction is applied to the parameters in Figure 18. When the clusters appear to be horizontal it indicates not relevance of the parameter toward the FOM used to discriminate the cluster.

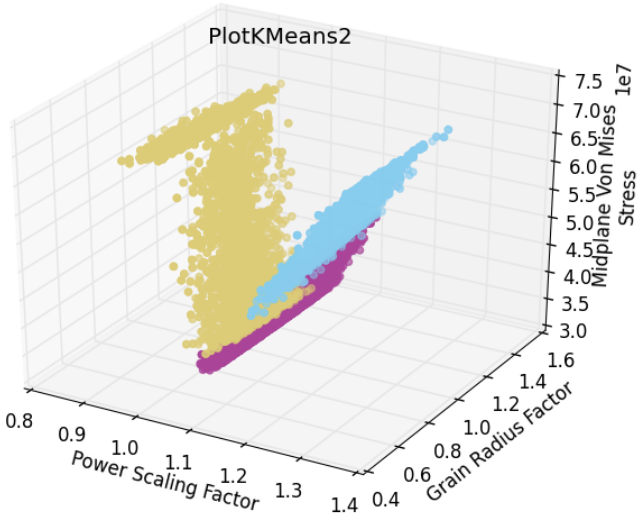

Figure 18: Midplane Von Mises Stress vs. Power Scale Factor and Grain Radius Factor: Clustering (k-means) applied to full output space

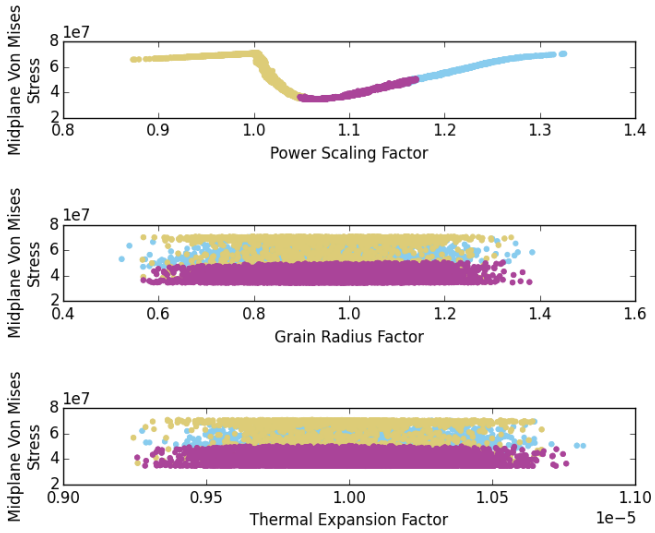

Figure 19: Coloring is based on a clustering (k-means) applied to the full output space: Midplane Von Mises Stress vs. Power Scale Factor (top), Grain Radius Factor (middle) and Thermal Expansion Factor (bottom).

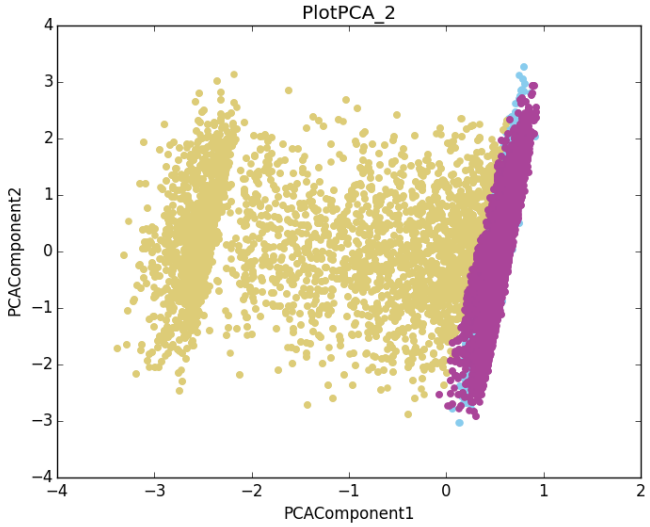

Figure 20: Clustering (k-means) applied to output space and the dimensionality reduction (PCA) applied to the 3-plotted parameters in Figure 18 plotted with the cluster labels

The dataset obtained with the BISON fuel performance simulation has 13 dimensions in the output space, the clustering applied to this space seems to give meaningful results. It is also possible to apply dimensionality reduction to the output space prior to the clustering analysis in case the data mining algorithms are effected by the "curse of dimensionality", results of such an application applied to BISON fuel performance simulation is illustrated in Figure 21 and Figure 22. The output space is reduced to 5 dimensions before applying the k-means clustering algorithm; the first 3 components are plotted in Figure 21 with the cluster labels obtained from the 5 dimensional reduced output space.

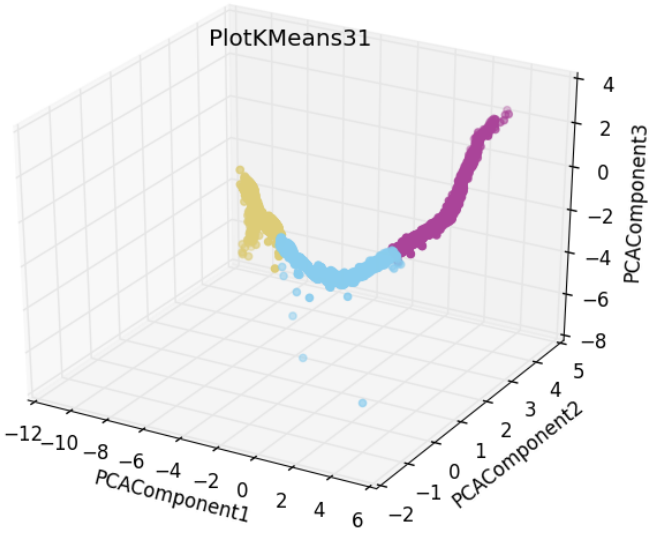

Figure 21: The first 3 components of the reduced output space

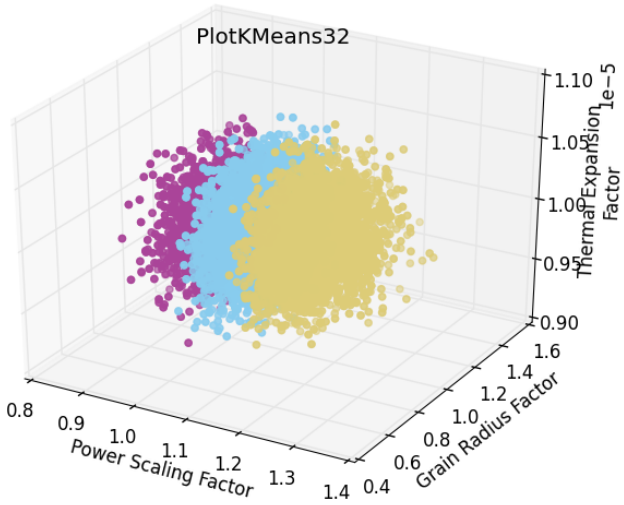

Figure 22: The input space with the cluster labels obtained from the reduced output space

The input space plotted in Figure 22 shows the cluster labels from the reduced output space, which does not show any significant difference from Figure 17, where the cluster labels are obtained from the original high dimensional output space. The variation of the "power scaling factor" dominates the transition between the clusters, which means that the reduced output space is again more sensitive to the "power scaling factor" than the other two input parameters. Even though the data mining process shown here in Figure 21 and Figure 22 do not show significant difference when applied to this dataset, it would be beneficial if the dataset shows the "curse of dimensionality".

4.2.2 BWR SBO Case

The data mining post-processor implemented in RAVEN has been tested on a BWR SBO scenario analysis. The data mining is applied to a dataset without the knowledge of the physical meaning of the

data in input and output space. The findings of the data mining post-processor available in RAVEN are discussed in the following paragraphs.

Data mining is applied to the 1-dimensional output space; two clusters are identified in the data sets. Then, the next process is typically to apply (project) the cluster labels to the input space to identify the patterns. However, the large dimensionality of the input space (12D) makes it difficult to visualize the input space. The input space dimension can either be reduced to low-dimensions (2D or 3D) to visualize or the visualization can be performed only in 2D or 3D. However, this will show noise in the visualization, which is illustrated in Figure 23 through Figure 25.

The cluster labels in Figure 23 through Figure 25 shows almost a horizontal band, which could be indicated as that there is no effect from any of the parameters at all. However, this is expected since the clad failure temperature is one in the input space and its distribution had a minimum value, which is 1800 ^{o}F in the analysis. The dataset has 19996 points over 12 dimensions, i.e. each dimension has 1666 points. The so-called noise in Figure 23 through Figure 25 is effect of other 11 dimensions not shown in the figures.

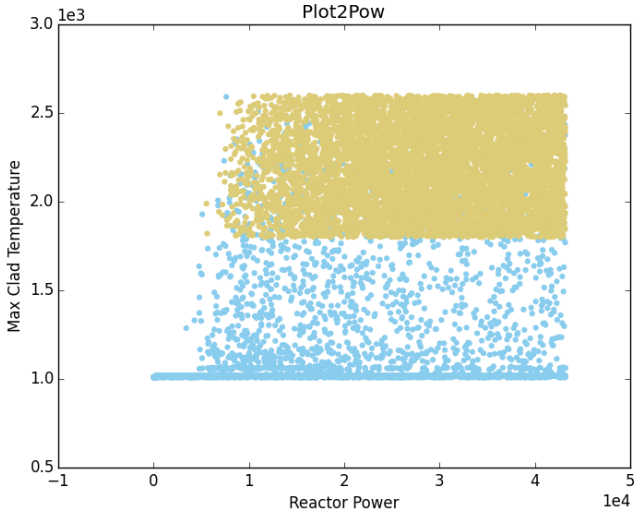

Figure 23: Maximum Clad Temperature as a function of Reactor Power

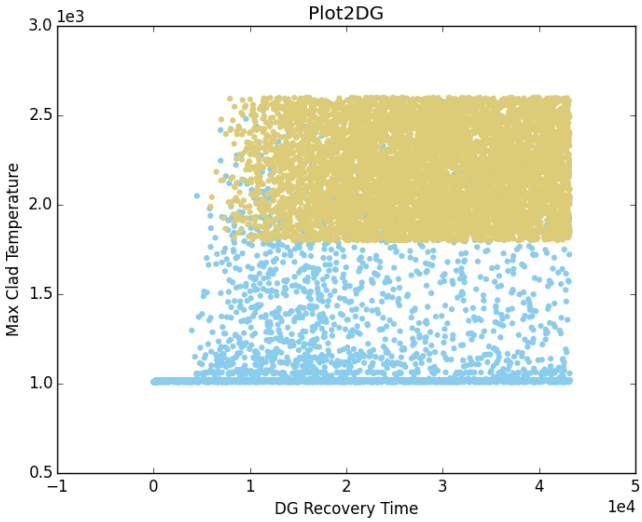

Figure 24:Maximum Clad temperature as a function of DG recovery time

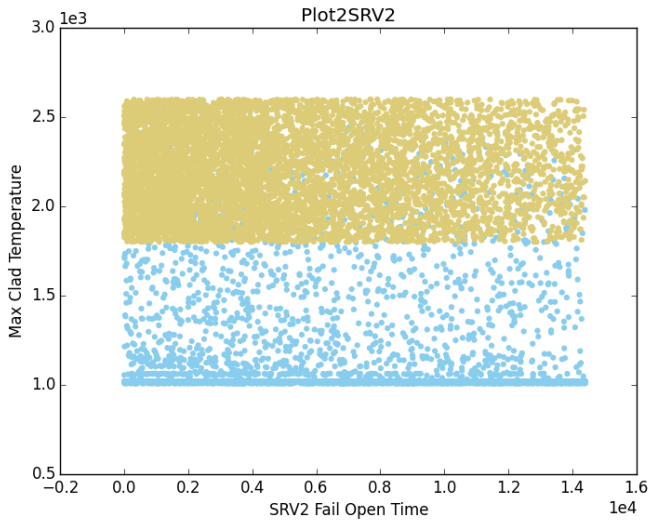

Figure 25: Maximum Clad Temperature as a function of SRV2 Fail Open time.

The noise is removed from the dataset by:

- a covariance analysis is performed with the "BasicStatistics" post-processor to rank the input space
- The cluster labels obtained with the previous cluster analysis is used as a target to train an stochastic gradient descent (SGD) Classifier ROM with the 3 most important input parameters used as features.
- 25x25x25 data points on a grid is sampled and applied to already trained SGD Classifier ROM.

The results showed similar clustering without the noise from the other less important input parameters. The three most important variables turned out to be:

- Variable 1: Reactor Power
- Variable 2: Diesel Generator Recovery Time
- Variable 3: Safety Release Valve (2) fails to open Time

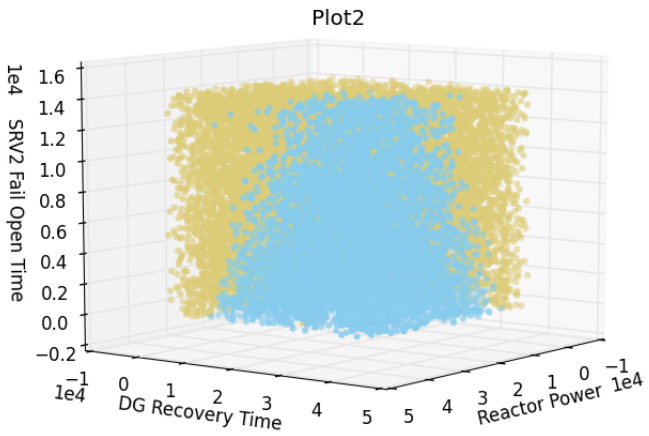

Figure 26: The most important 3 input parameters with the cluster labels from output space.

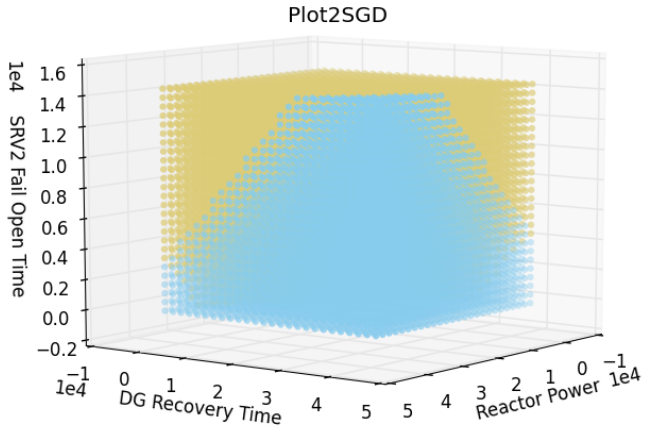

Figure 27: Results of SGD Classifier ROM.

The separate dependencies of the output to the 3 most important input parameters are illustrated in Figure 28. The noise due to the other input parameters are removed by training a Support Vector Machine (SVM) based Support Vector Regressor (SVR) ROM with using each input parameter separately as the single feature and a using the output as the target. Then each parameter is sampled with a grid sampler and the trained ROM is used with these sampled values. A complete input that applies such an analysis can be found in Appendices.

The filtering applied to each input variable separately, which smoothens the dependence of the output to each variable. The kernel coefficient for 'rbf' kernel, and the penalty parameter of the error term in the SVR based ROM are equal to 1 which means that the regression "function" will depend less on the variability of the data. If one needs to have more dependence on the variability, then these parameters should be chosen to be higher values.

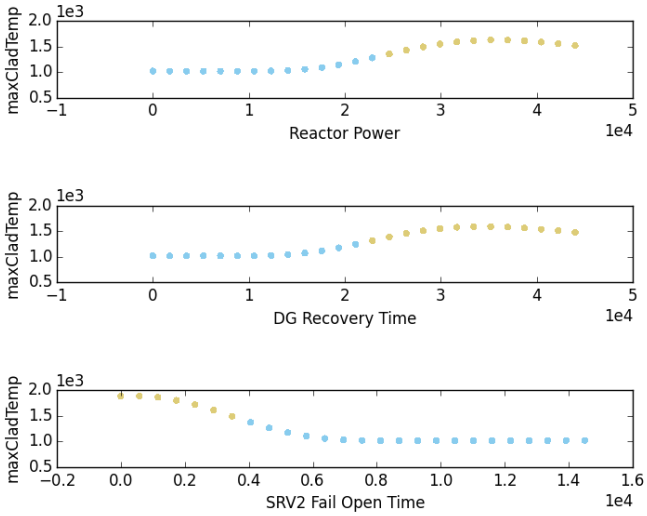

Figure 28: The separate dependence of the output variable to the most important 3 input parameters.

4.3 Application of Topology Post-Processor

The topology post-processor's capabilities have been tested on a BISON fuel rodlet performance simulation. In the following paragraphs, the findings when using the structured sensitivity analysis offered by the topological postprocessor available in RAVEN are explained.

Figure 29 shows several visualizations of the dataset. The leftmost image is the topology map. The red, upward-pointing triangles represent identified local maxima in the dataset, and the blue, downward-pointing triangles represent the local minima in the dataset. The horizontal positioning of the extrema represents each extrema's persistence value, meaning that extrema on the left can be considered more noisy features, and the extrema on the right are the more persistent and significant extrema. For this example, the default persistence implementation that is based on output value difference is used. The vertical positioning specifies the relative output value, thus local maxima tend toward the top of the plot and local minima tend toward the bottom. As one can see, there is a large gap in the horizontal layout of the data, which highlights a natural place to start when setting the simplification level. In this example, a level in the topological hierarchy is selected that highlights two distinct partitions. These partitions are shown in the topology map as Bezier curves connecting a local minimum to a local maximum that uniquely identify each Morse-Smale cell in the data.

The next plot shows the linear coefficients ($\hat{\beta}$) of performing linear fits on each Morse-Smale cell of the data. The plot in Figure 29 shows that the power_scalef is by far the most sensitive parameter for this problem. . After sorting these dimensions in order of decreasing sensitivity, stepwise regression is performed and the coefficients of determination (R^2) are plotted in order to better understand the value added for each dimension. This is shown in the third plot of Figure 29. One can again see that most of the linearity is captured in the power_scalef and little information is gained by incorporating the remaining two dimensions. . This information can be used to select an appropriate scatterplot projection, which is shown in the right image of Figure 29.

The projected scatterplot reiterates and elucidates what the other visualizations have told the audience. First, the power_scalef is more or less driving the value of the max stress. For lower values of the power_scalef the stress is decreasing implying that the simulation is in a pre-contact state, that is the stress is decreasing due to the expansive forces counteracting the compressive forces, however once entering the yellow segment, a sharp spike in the stress is noted as the fuel has come into contact with the

cladding. Lastly, the reason for the blue segment's inferior linear fit is made clear as there is a small upward trend followed by a steep downward trend in the stress-power_scalef relationship. In order to better capture this portion of the data, the resolution of the simplification should be increased, which is investigated in Figure 30.

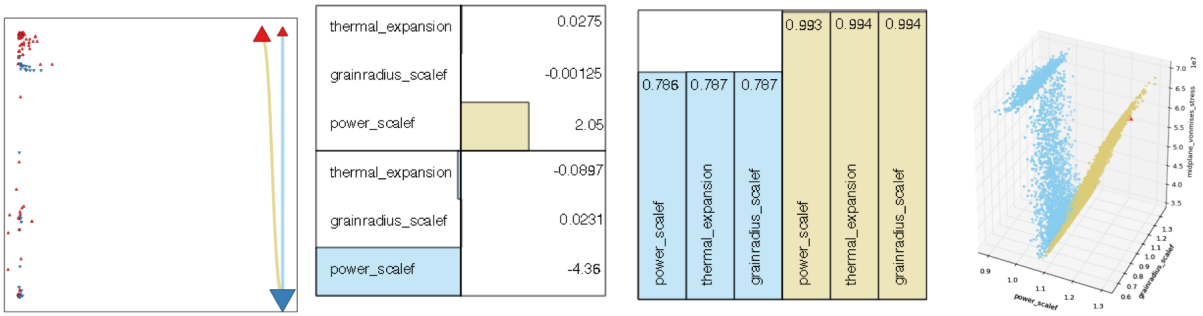

Figure 29: BISON test case results. Left: the topology map, middle left: the sensitivity values of each dimension on each partition of the data, middle right: the fitness values of performing stepwise regression where dimensions are added in order of decreasing sensitivity, and right: a 3D scatter plot where thermal_expansion is suppressed.

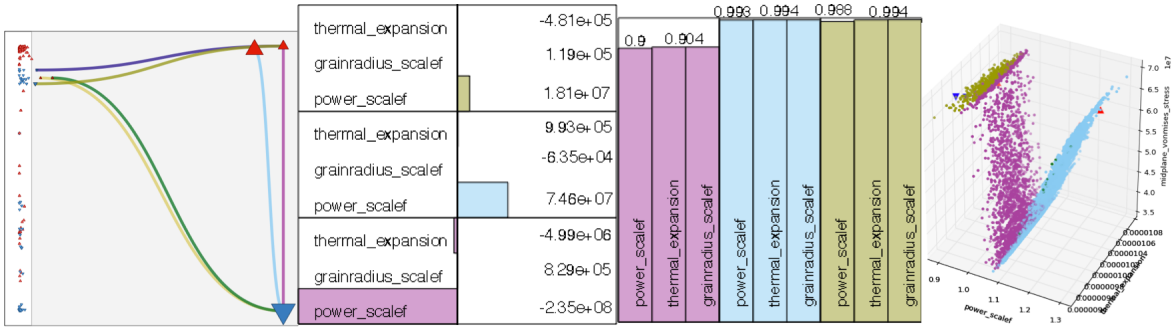

Figure 30: BISON test case results at a more resolved topological decomposition. Left: the topology map, middle left: the sensitivity values of each dimension on the most significant partitions of the data, middle right: the fitness values of performing stepwise regression where dimensions are added in order of decreasing sensitivity for the same partitions, and right: a 3D scatter plot where grainradius_scalef is suppressed.

For Figure 30, more refined partitions are included in the analysis in order to capture the initial upward trend in the data. Note, that in order to capture this segment of the data, several steps down in the resolution must be taken. These segments are shown in the left and right images of Figure 30, and notably in the right image several of these segments include very few data points and are less useful for analysis. For this reason, attention of the middle two images is focused on only the three main trends in the data, which includes the yellow partition of Figure 29 (now the blue partition), and the decomposed blue segment of Figure 29 (now the magenta and gold partitions). Now, these three partitions are able to fit these three partitions much more accurately with linear models. The end result remains the same for each of these three main segments, that is, the power_scalef is the most dominant parameter by several orders of magnitude, and between the other two, thermal_expansion is slightly more sensitive.

5. FUTURE DEVELOPMENTS

The new topology post-processor approximates the Morse-Smale complex given a finite sample set. Minor improvements/alternatives can be explored and instantiated that allow for variants of the algorithm

including variants of the gradient estimation, use of different graph structures, sampling or resampling the input data, extracting the topology of ROMs rather than the potentially sparse input data coming from computationally expensive simulations. Furthermore, the efficacy of the topology post-processor has been demonstrated on low dimensional examples, however the power of this method is its ability to accept higher dimensional data that cannot be directly visualized using scatterplot projections.

Already, the topological capabilities have been extended to be used as a ROM, however further extensions can be made by potentially using ascending/descending manifolds to fit local models around extrema. As these manifolds are no longer monotonic, a new type of local model would need to be used that takes full advantage of the information known about these manifolds (that is, they have a single sink or source). In addition, the topological decomposition can be in other parts of RAVEN, such as in the adaptive sampling search [19,20,21]. Research into such methods is being performed concurrently with the University of Utah.

The data mining techniques presented in this method are mainly used for the datasets obtained at a given time, or the final result of a transient analysis, so far the time dependence is not applicable. The algorithms need to be extended to find patterns in time dependent datasets.

6. CONCLUSIONS

In this report, we introduce the implementation of the data mining algorithms in the SciKit-Learn to RAVEN, the topology post-processor and the underlying Morse-Smale decomposition.

The application of data mining algorithms to find and identify the patterns in the large datasets is shown in two different physical simulation rather than artificial databases.

The reduction of the dimensionality and separation of the individual effects of input parameters using unsupervised and supervised learning algorithms are powerful tools for visualizing the datasets.

The topology post-processor and its associated visual interface is a useful technique for intelligently decomposing the domain to provide more accurate and localized sensitivity information. This can provide a useful first pass at a dataset to help identify what areas of the domain are interesting, and also what parameters are most sensitive. It is possible that a particular area of the domain is highly dependent on a subset of the parameters, but a separate subspace of the domain has a different set of drivers. This type of information will stand out when using the topology post-processor signaling the user to use different sampling strategies or to use different ROMs to model different subspaces of the input domain of interest.

Furthermore, the versatility of the Morse-Smale complex was hinted at by instantiating it in a post-processor, and also as a ROM. As mentioned in Section 5, the Morse-Smale complex can also be used to augment certain sampling strategies. As such, the addition of the Morse-Smale complex into the RAVEN code provides the potential for several new areas of exploration and is a good supplement to the more traditional data mining techniques provided by scikit-learn.

These capabilities will need to be extended to time dependent analysis to be capable to appreciate how different parameters change their impact depending on the evolution of the transient considered. The Morse-Smale based topological analysis will need to be extended to deal with intrinsically stochastic systems.

7. REFERENCES

1. Rabiti, C., A. Alfonsi, D. Mandelli, J. Cogliati, and R. Kinoshita, "RAVEN, a New Software for Dynamic Risk Analysis," PSAM 12 Probabilistic Safety Assessment and Management, Honolulu, Hawaii, June 2014.

2. Rabiti, C., A. Alfonsi, D. Mandelli, J. Cogliati, R. Martinueau, and C. Smith, Deployment and Overview of RAVEN Capabilities for a Probabilistic Risk Assessment Demo for a PWR Station Blackout, INL/EXT-13-29510, Idaho National Laboratory, 2013.

3. Rabiti, C., A. Alfonsi, D. Mandelli, J. Cogliati, R. Kinoshita, and S. Sen, RAVEN User Manual, INL/EXT-15-34123, Idaho National Laboratory, 2015.

4. Alfonsi, A., C. Rabiti, D. Mandelli, J. Cogliati, R. Kinoshita, and A. Naviglio, "RAVEN and dynamic probabilistic risk assessment: Software overview," in Proceedings of ESREL European Safety and Reliability Conference, 2014.

5. Alfonsi, A., C. Rabiti, D. Mandelli, J. Cogliati, and R. Kinoshita, "Raven as a tool for dynamic probabilistic risk assessment: Software overview," in Proceedings of M&C2013 International Topical Meeting on Mathematics and Computation, CD-ROM, American Nuclear Society, LaGrange Park, IL, 2013.

6. Rabiti, C., D. Mandelli, A. Alfonsi, J. Cogliati, and B. Kinoshita, "Mathematical framework for the analysis of dynamic stochastic systems with the raven code," in Proceedings of International Conference of Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013), Sun Valley, ID, pp. 320–332, 2013.

7. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

8. Habermann, C., and F. Kindermann, "Multidimensional Spline Interpolation: Theory and Applications," Computational Economics, Volume 30, Issue 2, pp. 153-169.

9. Gordon, W.J., and J.A.Wixom, "Shepard's Method of Metric Interpolation to Bivariate and Multivariate Interpolation," In Mathematics and Computation, Volume 32, Issue 141, pp. 253-264, 1978.

10. Andrea Alfonsi, Cristian Rabiti, Diego Mandelli, Joshua Cogliati, Sonat Sen, Curtis Smith,"Improving Limit Surface Search Algorithms in RAVEN Using Acceleration Schemes", INL/EXT-15-36100, 2015

11. Rabiti, C., A. Alfonsi, D. Mandelli, J. Cogliati, and B. Kinoshita, Advanced Probabilistic Risk Analysis Using RAVEN and RELAP-7, INL/EXT-14-32491, Idaho National Laboratory, 2014.

12. Rabiti, C., P. Talbot, A. Alfonsi, D. Mandelli, and J. Cogliati, Implementation of Stochastic Polynomials Approach in the RAVEN Code, INL/EXT-13-30611, Idaho National Laboratory, 2013.

13. Alfonsi, A., C. Rabiti, D. Mandelli, J. Cogliati, R. Kinoshita, and A. Naviglio, "Dynamic event tree analysis through Raven," in Proceedings of ANS PSA 2013 International Topical Meeting on Probabilistic Safety Assessment and Analysis, 2013.

14. RELAP5 Code Development Team, RELAP5-3D Code Manual, Idaho National Laboratory, 2012.

15. Gaston, D., C. Newman, G. Hansen, and D. Lebrun-Grandi, "MOOSE: A parallel computational framework for coupled systems of nonlinear equations," Nuclear Engineering Design, 239, pp. 1768-1778, 2009.

16. Alfonsi, A., C. Rabiti, A. S. Epiney, Y. Wang, and J. Cogliati, "PHISICS Toolkit: Multi-Reactor Transmutation Analysis Utility–MRTAU," in Proceedings of PHYSOR 2012 "Advances in Reactor Physics Linking Research, Industry, and Education," Knoxville, TN, April 15-20, 2012.

17. Mandelli, D., C. Smith, Z. Ma, T. Riley, J. Nielsen, A. Alfonsi, C. Rabiti, J. Cogliati, Risk-Informed Safety Margin Characterization Methods Development Work, INL/EXT-14-33191, Idaho National Laboratory, 2014.

18. Gerber, S. and Potter, K. "Data analysis with the Morse-Smale complex: The MSR package for R." Journal of Statistical Software, 50(2):1–22, 7 2012.

19. Maljovec, D., Wang, B., Kupresanin, A., Johannesson, G., Pascucci, V., and Bremer, P.-T., "Adaptive Sampling with Topological Scores," International Journal for Uncertainty Quantification (IJUQ), 3(2), pages 119-141, 2013.

20. Maljovec, D., Wang, B., Mandelli, D., Bremer, P.-T., and Pascucci, V. "Adaptive Sampling Algorithms for Probabilistic Risk Assessment of Nuclear Simulations," International Topical Meeting on Probabilistic Safety Assessment and Analysis (PSA), 2013.

21. Maljovec, D., Wang, B., Moeller, J., and Pascucci, V. "Topology-Based Active Learning." SCI Technical Report UUSCI-2014-00, 2014.

22. Hales, J., Novascone, S., Pastore G., Perez D., Spencer B., and Williamson, R. *BISON Theory Manual: The Equations Behind Nuclear Fuel Analysis*, 2013.

23. C. Correa and P. Lindstrom. Towards robust topology of sparsely sam- pled data. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):1852–1861, Dec 2011.

Appendix A: Sample Inputs

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation verbosity='debug'>
  <RunInfo>
    <WorkingDir>SBODataMining</WorkingDir>

<Sequence>read_in_hdf5,pp0,pp,trainRom,romMC,trainRom1,rom1MC,pp1,trainRom2,rom2MC,pp2,trainRom3,
rom3MC,pp3,outputAll</Sequence>
    <batchSize>100</batchSize>
  </RunInfo>
  <Files>
      <Input name="sboDataPointSet">./data20000.csv</Input>
      <Input name="outputBS.csv">./outputBS.csv</Input>
  </Files>
  <Steps>
    <IOStep name='read_in_hdf5' verbosity='debug'>
          <Input    class = 'Files'             type = ''              >sboDataPointSet</Input>
        <Output   class = 'DataObjects'       type = 'PointSet'      >sboData</Output>
     </IOStep>
    <PostProcess name='pp0' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
        <Model    class = 'Models'            type = 'PostProcessor' >BasicStatistics</Model>
        <Output   class = 'Files'             type = ""              >outputBS.csv</Output>
    </PostProcess>
    <IOStep name='LSplot' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
        <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2LS</Output>
    </IOStep>
    <RomTrainer name='trainRom1' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
        <Output   class = 'Models'            type = "ROM"           >PowerSVR</Output>
    </RomTrainer>
    <MultiRun name="rom1MC" re-seeding="200286" sleepTime='1E-3'>
        <Input   class="DataObjects"          type = "Point"         >romData</Input>
        <Model   class="Models"               type = "ROM"           >PowerSVR</Model>
        <Sampler class="Samplers"             type = "Grid"          >RAVENgrid</Sampler>
        <Output  class="DataObjects"          type = "PointSet"      >rom1OutputData</Output>
    </MultiRun>
    <PostProcess name='pp1' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >rom1OutputData</Input>
        <Model    class = 'Models'            type = 'PostProcessor' >ClusteringPP1</Model>
        <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2Power</Output>
    </PostProcess>
    <RomTrainer name='trainRom2' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
        <Output   class = 'Models'            type = "ROM"           >RecoveryTimeSVR</Output>
    </RomTrainer>
    <MultiRun name="rom2MC" re-seeding="200286" sleepTime='1E-3'>
        <Input   class="DataObjects"          type = "Point"         >romData</Input>
        <Model   class="Models"               type = "ROM"           >RecoveryTimeSVR</Model>
        <Sampler class="Samplers"             type = "Grid"          >RAVENgrid</Sampler>
        <Output  class="DataObjects"          type = "PointSet"      >rom2OutputData</Output>
    </MultiRun>
    <PostProcess name='pp2' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >rom2OutputData</Input>
        <Model    class = 'Models'            type = 'PostProcessor' >ClusteringPP2</Model>
        <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2RecoveryTime</Output>
    </PostProcess>
    <RomTrainer name='trainRom3' pauseAtEnd = 'True'>
        <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
        <Output   class = 'Models'            type = "ROM"           >StuckOpenTimeSVR</Output>
    </RomTrainer>
    <MultiRun name="rom3MC" re-seeding="200286" sleepTime='1E-3'>
        <Input   class="DataObjects"          type = "Point"         >romData</Input>
        <Model   class="Models"               type = "ROM"           >StuckOpenTimeSVR</Model>
        <Sampler class="Samplers"             type = "Grid"          >RAVENgrid</Sampler>
        <Output  class="DataObjects"          type = "PointSet"      >rom3OutputData</Output>
```

```
  </MultiRun>
  <PostProcess name='pp3' pauseAtEnd = 'True'>
      <Input    class = 'DataObjects'       type = 'PointSet'      >rom3OutputData</Input>
      <Model    class = 'Models'            type = 'PostProcessor' >ClusteringPP3</Model>
      <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2StuckOpenTime</Output>
  </PostProcess>
  <PostProcess name='pp' pauseAtEnd = 'True'>
      <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
      <Model    class = 'Models'            type = 'PostProcessor' >ClusteringPP4</Model>
      <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2</Output>
  </PostProcess>
  <RomTrainer name='trainRom' pauseAtEnd = 'True'>
      <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
      <Output   class = 'Models'            type = "ROM"           >SGDClassifier</Output>
  </RomTrainer>
  <MultiRun name="romMC" re-seeding="200286" sleepTime='1E-3'>
      <Input   class="DataObjects"          type = "Point"         >romData</Input>
      <Model   class="Models"               type = "ROM"           >SGDClassifier</Model>
      <Sampler class="Samplers"             type = "Grid"          >RAVENgrid</Sampler>
      <Output  class="DataObjects"          type = "PointSet"      >romOutputData</Output>
  </MultiRun>
  <IOStep name='output' pauseAtEnd = 'True'>
      <Input    class = 'DataObjects'       type = 'PointSet'      >romOutputData</Input>
      <Output   class = 'OutStreamManager'  type = 'Plot'          >Plot2SGD</Output>
  </IOStep>
  <IOStep name='outputAll' pauseAtEnd = 'True'>
      <Input    class = 'DataObjects'       type = 'PointSet'      >sboData</Input>
      <Input    class = 'DataObjects'       type = 'PointSet'      >romOutputData</Input>
      <Input    class = 'DataObjects'       type = 'PointSet'      >rom1OutputData</Input>
      <Input    class = 'DataObjects'       type = 'PointSet'      >rom2OutputData</Input>
      <Input    class = 'DataObjects'       type = 'PointSet'      >rom3OutputData</Input>
      <Output   class = 'OutStreamManager'  type = 'Plot'          >PlotAll</Output>
  </IOStep>
</Steps>
<Distributions>
    <Uniform name="FailureTimeDGDist">
        <lowerBound>1.0E0</lowerBound>
        <upperBound>3.0E4</upperBound>
    </Uniform>
    <Uniform name="RecoveryTimeDGdist">
        <lowerBound>0.0E0</lowerBound>
        <upperBound>4.4E4</upperBound>
    </Uniform>
    <Uniform name="OFFsitePowerRecoveryTimeDist">
        <lowerBound>3.2E0</lowerBound>
        <upperBound>4.4E4</upperBound>
    </Uniform>
    <Uniform name="SRV1stuckOpenTimeDist">
        <lowerBound>6.0E0</lowerBound>
        <upperBound>4.4E4</upperBound>
    </Uniform>
    <Uniform name="SRV2stuckOpenTimeDist">
        <lowerBound>0.0E0</lowerBound>
        <upperBound>1.5E4</upperBound>
    </Uniform>
    <Uniform name="cladFailureTemperatureDist">
        <lowerBound>1.8E3</lowerBound>
        <upperBound>2.6E3</upperBound>
    </Uniform>
    <Uniform name="HPCIFailToRunTimeDist">
        <lowerBound>2.9E0</lowerBound>
        <upperBound>2.9E4</upperBound>
    </Uniform>
    <Uniform name="RCICFailToRunTimeDist">
        <lowerBound>2.0E0</lowerBound>
        <upperBound>4.3E4</upperBound>
    </Uniform>
    <Uniform name="ReactorPowerDist">
        <lowerBound>1.0E1</lowerBound>
```

```
            <upperBound>4.4E4</upperBound>
        </Uniform>
        <Uniform name="ADSactivationTimeDelayDist">
            <lowerBound>8.5E-1</lowerBound>
            <upperBound>7.2E3</upperBound>
        </Uniform>
        <Uniform name="firewaterTimeDist">
            <lowerBound>3.7E-1</lowerBound>
            <upperBound>2.9E4</upperBound>
        </Uniform>
        <Uniform name="ExtendedECCSoperation">
            <lowerBound>1.0E3</lowerBound>
            <upperBound>2.6E3</upperBound>
        </Uniform>
    </Distributions>
    <Samplers>
        <MonteCarlo name="RAVENmc3">
            <sampler_init>
                <limit>20000</limit>
            </sampler_init>
            <variable name="RecoveryTimeDG">
                <distribution>RecoveryTimeDGdist</distribution>
            </variable>
            <variable name="ReactorPower">
                <distribution>ReactorPowerDist</distribution>
            </variable>
            <variable name="SRV2stuckOpenTime">
                <distribution>SRV2stuckOpenTimeDist</distribution>
            </variable>
        </MonteCarlo>
        <Grid name="RAVENgrid">
            <variable name="RecoveryTimeDG">
                <distribution>RecoveryTimeDGdist</distribution>
                <grid type='value' construction='equal' steps='25'>0 4.4E4</grid>
            </variable>
            <variable name="ReactorPower">
                <distribution>ReactorPowerDist</distribution>
                <grid type='value' construction='equal' steps='25'>0 4.4E4</grid>
            </variable>
            <variable name="SRV2stuckOpenTime">
                <distribution>SRV2stuckOpenTimeDist</distribution>
                <grid type='value' construction='equal' steps='25'>0 1.45E4</grid>
            </variable>
        </Grid>
    </Samplers>
    <DataObjects>
        <PointSet name='sboData'>

<Input>FailureTimeDG,RecoveryTimeDG,OFFsitePowerRecoveryTime,SRV1stuckOpenTime,SRV2stuckOpenTime,
cladFailureTemperature,HPCIFailToRunTime,RCICFailToRunTime,ReactorPower,ADSactivationTimeDelay,fi
rewaterTime,ExtendedECCSoperation</Input>
            <Output>maxCladTemp,outcome</Output>
        </PointSet>
        <PointSet name='rom1OutputData'>
            <Input>RecoveryTimeDG,SRV2stuckOpenTime,ReactorPower</Input>
            <Output>maxCladTemp</Output>
        </PointSet>
        <PointSet name='rom2OutputData'>
            <Input>RecoveryTimeDG,SRV2stuckOpenTime,ReactorPower</Input>
            <Output>maxCladTemp</Output>
        </PointSet>
        <PointSet name='rom3OutputData'>
            <Input>RecoveryTimeDG,SRV2stuckOpenTime,ReactorPower</Input>
            <Output>maxCladTemp</Output>
        </PointSet>
        <PointSet name='romData'>
            <Input>RecoveryTimeDG,SRV2stuckOpenTime,ReactorPower</Input>
            <Output>OutputPlaceHolder</Output>
        </PointSet>
```

```
        <PointSet name='romOutputData'>
            <Input>RecoveryTimeDG,SRV2stuckOpenTime,ReactorPower</Input>
            <Output>maxCladTemp,labels</Output>
        </PointSet>
    </DataObjects>

    <Models>
        <ROM name='PowerSVR' subType='SciKitLearn'>
            <SKLtype>svm|SVR</SKLtype>
            <Features>ReactorPower</Features>
            <Target>maxCladTemp</Target>
        </ROM>
        <ROM name='RecoveryTimeSVR' subType='SciKitLearn'>
            <SKLtype>svm|SVR</SKLtype>
            <Features>RecoveryTimeDG</Features>
            <Target>maxCladTemp</Target>
        </ROM>
        <ROM name='StuckOpenTimeSVR' subType='SciKitLearn'>
            <SKLtype>svm|SVR</SKLtype>
            <Features>SRV2stuckOpenTime</Features>
            <Target>maxCladTemp</Target>
        </ROM>
        <ROM name='SGDClassifier' subType='SciKitLearn'>
            <SKLtype>linear_model|SGDClassifier</SKLtype>
            <Features>ReactorPower,RecoveryTimeDG,SRV2stuckOpenTime</Features>
            <Target>labels</Target>
        </ROM>
        <PostProcessor name="BasicStatistics" subType="BasicStatistics" verbosity="debug">
            <what>all</what>

<parameters>FailureTimeDG,RecoveryTimeDG,OFFsitePowerRecoveryTime,SRV1stuckOpenTime,SRV2stuckOpen
Time,cladFailureTemperature,HPCIFailToRunTime,RCICFailToRunTime,ReactorPower,ADSactivationTimeDel
ay,firewaterTime,ExtendedECCSoperation,maxCladTemp</parameters>
        </PostProcessor>
        <PostProcessor name='ClusteringPP1' subType='DataMiningPostProcessor' verbosity = 'quiet'>
            <KDD lib='SciKitLearn'>
                <Features>maxCladTemp</Features>
                <SKLtype>cluster|MiniBatchKMeans</SKLtype>
                <n_clusters>2</n_clusters>
                <tol>0.0001</tol>
                <init>random</init>
            </KDD>
            <DataObject    class = 'DataObjects' type = 'PointSet'>rom1OutputData</DataObject>
        </PostProcessor>
        <PostProcessor name='ClusteringPP2' subType='DataMiningPostProcessor' verbosity = 'quiet'>
            <KDD lib='SciKitLearn'>
                <Features>maxCladTemp</Features>
                <SKLtype>cluster|MiniBatchKMeans</SKLtype>
                <n_clusters>2</n_clusters>
                <tol>0.0001</tol>
                <init>random</init>
            </KDD>
            <DataObject    class = 'DataObjects' type = 'PointSet'>rom2OutputData</DataObject>
        </PostProcessor>
        <PostProcessor name='ClusteringPP3' subType='DataMiningPostProcessor' verbosity = 'quiet'>
            <KDD lib='SciKitLearn'>
                <Features>maxCladTemp</Features>
                <SKLtype>cluster|MiniBatchKMeans</SKLtype>
                <n_clusters>2</n_clusters>
                <tol>0.0001</tol>
                <init>random</init>
            </KDD>
            <DataObject    class = 'DataObjects' type = 'PointSet'>rom3OutputData</DataObject>
        </PostProcessor>
        <PostProcessor name='ClusteringPP4' subType='DataMiningPostProcessor' verbosity = 'quiet'>
            <KDD lib='SciKitLearn'>
                <Features>outcome</Features>
                <SKLtype>cluster|MiniBatchKMeans</SKLtype>
                <n_clusters>2</n_clusters>
```

```
            <tol>0.0001</tol>
            <init>random</init>
        </KDD>
        <DataObject    class = 'DataObjects' type = 'PointSet'>sboData</DataObject>
    </PostProcessor>
</Models>
<OutStreamManager>
    <Print name = 'cluster_dmp'>
        <type>csv</type>
        <source>romOutputData</source>
    </Print>
    <Plot dim="3" name="Plot2" overwrite="false" verbosity="debug">
        <plotSettings>
            <plot>
                <type>dataMining</type>
                <SKLtype>cluster</SKLtype>
                <x>sboData|Input|ReactorPower</x>
                <y>sboData|Input|RecoveryTimeDG</y>
                <z>sboData|Input|SRV2stuckOpenTime</z>
                <clusterLabels>sboData|Output|labels</clusterLabels>
                <kwargs>
                    <noClusters>2</noClusters>
                     <edgecolor>None</edgecolor>
                </kwargs>
            </plot>
            <xlabel>Variable1</xlabel>
            <ylabel>Variable2</ylabel>
            <zlabel>Variable3</zlabel>
        </plotSettings>
        <actions>
            <how>screen</how>
            <camera>
                <elevation>9</elevation>
                <azimuth>37</azimuth>
            </camera>
        </actions>
    </Plot>
    <Plot dim="2" name="Plot2Power" overwrite="false" verbosity="debug">
        <plotSettings>
            <plot>
                <type>dataMining</type>
                <SKLtype>cluster</SKLtype>
                <x>rom1OutputData|Input|ReactorPower</x>
                <y>rom1OutputData|Output|maxCladTemp</y>
                <clusterLabels>rom1OutputData|Output|labels</clusterLabels>
                <kwargs>
                    <noClusters>2</noClusters>
                     <edgecolor>None</edgecolor>
                </kwargs>
            </plot>
            <xlabel>Reactor Power</xlabel>
            <ylabel>maxCladTemp</ylabel>
        </plotSettings>
        <actions>
            <how>screen</how>
        </actions>
    </Plot>
    <Plot dim="2" name="Plot2RecoveryTime" overwrite="false" verbosity="debug">
        <plotSettings>
            <plot>
                <type>dataMining</type>
                <SKLtype>cluster</SKLtype>
                <x>rom2OutputData|Input|RecoveryTimeDG</x>
                <y>rom2OutputData|Output|maxCladTemp</y>
                <clusterLabels>rom2OutputData|Output|labels</clusterLabels>
                <kwargs>
                    <noClusters>2</noClusters>
                    <edgecolor>None</edgecolor>
                </kwargs>
```

```
        </plot>
        <xlabel>Recovery Time DG</xlabel>
        <ylabel>maxCladTemp</ylabel>
    </plotSettings>
    <actions>
        <how>screen</how>
    </actions>
</Plot>
<Plot dim="2" name="Plot2StuckOpenTime" overwrite="false" verbosity="debug">
    <plotSettings>
        <plot>
            <type>dataMining</type>
            <SKLtype>cluster</SKLtype>
            <x>rom3OutputData|Input|SRV2stuckOpenTime</x>
            <y>rom3OutputData|Output|maxCladTemp</y>
            <clusterLabels>rom3OutputData|Output|labels</clusterLabels>
            <kwargs>
                <noClusters>2</noClusters>
                 <edgecolor>None</edgecolor>
            </kwargs>
        </plot>
        <xlabel>SRV2 Stuck Open Time</xlabel>
        <ylabel>maxCladTemp</ylabel>
    </plotSettings>
    <actions>
        <how>screen</how>
    </actions>
</Plot>
<Plot dim="3" name="Plot2SGD" overwrite="false" verbosity="debug">
    <plotSettings>
        <plot>
            <type>dataMining</type>
            <SKLtype>cluster</SKLtype>
            <x>romOutputData|Input|ReactorPower</x>
            <y>romOutputData|Input|RecoveryTimeDG</y>
            <z>romOutputData|Input|SRV2stuckOpenTime</z>
            <clusterLabels>romOutputData|Output|labels</clusterLabels>
            <kwargs>
                <noClusters>2</noClusters>
                 <edgecolor>None</edgecolor>
            </kwargs>
        </plot>
        <xlabel>Variable1</xlabel>
        <ylabel>Variable2</ylabel>
        <zlabel>Variable3</zlabel>
    </plotSettings>
    <actions>
        <how>screen</how>
    </actions>
</Plot>
<Plot dim="3" name="Plot2LS" overwrite="false" verbosity="debug">
    <plotSettings>
        <plot>
            <type>scatter</type>
            <x>sboData|Input|ReactorPower</x>
            <y>sboData|Input|RecoveryTimeDG</y>
            <z>sboData|Input|SRV2stuckOpenTime</z>
            <colorMap>sboData|Output|outcome</colorMap>
            <cmap>summer</cmap>
            <kwargs>
                <edgecolor>None</edgecolor>
                <camera>
                    <elevation>9</elevation>
                    <azimuth>37</azimuth>
                </camera>
            </kwargs>
        </plot>
        <xlabel>Variable1</xlabel>
        <ylabel>Variable2</ylabel>
```

```
        <zlabel>Variable3</zlabel>
    </plotSettings>
    <actions>
        <how>screen</how>
        <camera>
            <elevation>9</elevation>
            <azimuth>37</azimuth>
        </camera>
    </actions>
</Plot>
<Plot dim="2" name="PlotAll" overwrite="false" verbosity="debug">
    <plotSettings>
        <gridSpace>5 1</gridSpace>
        <plot>
            <type>dataMining</type>
            <SKLtype>cluster</SKLtype>
            <x>rom1OutputData|Input|ReactorPower</x>
            <y>rom1OutputData|Output|maxCladTemp</y>
            <xlabel>Reactor Power</xlabel>
            <ylabel>maxCladTemp</ylabel>
            <gridLocation>
                <x>0</x>
                <y>0 1</y>
            </gridLocation>
            <clusterLabels>rom1OutputData|Output|labels</clusterLabels>
            <kwargs>
                <noClusters>2</noClusters>
                 <edgecolor>None</edgecolor>
            </kwargs>
        </plot>
        <plot>
            <type>dataMining</type>
            <SKLtype>cluster</SKLtype>
            <x>rom2OutputData|Input|RecoveryTimeDG</x>
            <y>rom2OutputData|Output|maxCladTemp</y>
            <xlabel>Recovery Time DG</xlabel>
            <ylabel>maxCladTemp</ylabel>
            <gridLocation>
                <x>2</x>
                <y>0 1</y>
            </gridLocation>
            <clusterLabels>rom2OutputData|Output|labels</clusterLabels>
            <kwargs>
                <noClusters>2</noClusters>
                 <edgecolor>None</edgecolor>
            </kwargs>
        </plot>
        <plot>
            <type>dataMining</type>
            <SKLtype>cluster</SKLtype>
            <x>rom3OutputData|Input|SRV2stuckOpenTime</x>
            <y>rom3OutputData|Output|maxCladTemp</y>
            <xlabel>SRV2 Stuck Open Time</xlabel>
            <ylabel>maxCladTemp</ylabel>
            <gridLocation>
                <x>4</x>
                <y>0 1</y>
            </gridLocation>
            <clusterLabels>rom3OutputData|Output|labels</clusterLabels>
            <kwargs>
                <noClusters>2</noClusters>
                 <edgecolor>None</edgecolor>
            </kwargs>
        </plot>
    </plotSettings>
    <actions>
        <how>png</how>
    </actions>
</Plot>
```

```
  </OutStreamManager>
</Simulation>
```

Appendix B: User Manual

7.1 TopologicalDecomposition

The TopologicalDecomposition post-processor will compute an approximated hierarchical Morse-Smale complex and perform linear regression on each component.

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.

7.2 Data Mining Post Processor

In order to use the DataMining post-processor, the user needs to set the attribute **subType**: **<PostProcessor subType= 'DataMiningPostProcessor'>**. 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'**
- AssemblerObjectsTheseobjectsareeitherrequiredoroptionaldependingonthefunctionalityofthedataMin- ing PostProcessor. The objects must be listed with a rigorous syntax that, except for the xml node tag, is com- mon 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 "DataObjects," "Models," etc.
  - **type**, *required string attribute*, is the object identifier or sub-type. For example, it can be "PointSet," "ROM," etc.

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

– **<DataObject>**,*string, required field*, body of this xml node must contain the name of a DataObject defined in the **<DataObjects>** block.

The algorithm for the dataMining is chosen by the subnode **<SKLType>** under the parent node **<KDD>.** The data that are used in the training of the DataMining postprocessor are supplied 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 seperately for each algorithm below.

All the available algorithms are described in the following sections.

7.2.1 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.

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 field,* 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'.

7.2.2 Dirichlet Process GMM Classifier (DPGMM)

The DPGMM implements a variant of the Gaussian mixture model with a variable (but bounded) number of components using the Dirichlet Process. The API is identical to GMM.

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

<SKLtype>`mixture|DPGMM`**</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.
  Default: 1.
- **<thresh>**, *float, optional field*, convergence threshold.
  Default: 0.01
- **<n_iter>**, *integer, optional field*, Number of EM iterations to perform.
  Default: 100
- **<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'.

7.2.3 Variational GMM Classifier (VBGMM)

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.
  Default: 1.

7.2.4 KMeans 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 KMeans 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: 10
- **<init>**, *string, optional field*, Method for initialization, k-means++', 'random' or an ndarray:
  Default: k-means++
  - 'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up conver-gence.
  - '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.

7.2.5 Mini Batch K-Means Clustering

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. 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 field*, 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: 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 field*, 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 conver-gence.
  - '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.

  Default: k-means++
- **<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 cen-ters. 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

7.2.6 Affinity Propagation

Affinity Propagation creates clusters by sending messages between pairs of samples until convergence.

In order to use the Affinity Propagation Clustering, the user needs to set the sub-node:

<SKLtype>`cluster|AffinityPropagation`**</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 esti-mated 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 exemplars. The number of exemplars, i.e. 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

7.2.7 Mean Shift

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 SciKit-Learn documentation for that function for hints on scalability.
  Default: `sklearn.cluster.estimate`
- **<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.
  Default: `clustering.get`
- **<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

7.2.8 Spectral clustering

Spectral Clustering does a low-dimension embedding of the affinity matrix between samples, followed by a KMeans in the low dimensional space

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.

  Default: 'rbf'
- **<gamma>**, *float, optional field*, Scaling factor of RBF, polynomial, exponential chi^2 and sigmoid affinity kernel. Ignored for affinity =^{0} nearest neighbors^0.
  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
  Default: None
- **<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> is '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

7.2.9 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 consid-ered 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: 'ecleudian'
- **<random_state>**, *numpy.RandomState, optional field*, The generator used to initialize the centers.
  Default: numpy.random.

7.2.10 Exact 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 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 (False by default) 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.

7.2.11 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>**, *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
- **<iterated_power>**, *int, optional field*, Number of iterations for the power method.
  Default: 3
- **<whiten>**, *boolean, optional field*, When True (False by default) 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

7.2.12 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>**, *integer, 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>**, *int, 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 ar:
  - 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

7.2.13 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,
  - cd

  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

7.2.14 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>**, *int, optional field*, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
  Default: 3
- **<method>**, *string, optional field*, method to use, options are:
  - lars,
  - cd

  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>**, *int or Random State, optional field*, Pseudo number generator state used for random sampling.
  Default: None

7.2.15 Truncated SVD

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

7.2.16 FastICA

A fast algorithm for Independent Component Analysis.

In order to use the FastICA, 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

7.2.17 Isometric Manifold Learning

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

7.2.18 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

7.2.19 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>**, *int 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

7.2.20 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>**, *int, 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>**, *int, optional field*, Maximum number of iterations of the SMACOF algorithm for a single run
  Default: 300
- **<verbose>**, *int, 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>**, *int, 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>**, *int 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'