Norwegian University
of Life Sciences

Master's Thesis 2020 30 ECTS
Faculty of Science and Technology

Metamodelling of simulation results from Brunel's Neural Network model using Local Multivariate Regression (HC-PLSR)

Anja Stene
Data Science (M.Sc)

Acknowledgements

This thesis is written as a part of the Master Program in Data Science at the Faculty of Science and Technology (REALTEK), at the Norwegian University of Life Sciences (NMBU) the spring of 2020. The thesis marks the conclusion of a two-year master's degree in data science, and I am now ready and well prepared for the next phase of my life.

I would like to thank my main supervisor, Professor Kristin Tøndel, for introducing me to metamodelling, and for providing excellent guidance and support during this project.
I would also like to thank my co-supervisor, Professor Gaute Tomas Einevoll, for inspiring me to be curious about computational neuroscience, and for the encouragement and interesting conversations during my projects involving neuroscience.

My dearest thanks and deepest gratitude to family and friends, near and far, for your feedback, encouragement and support during the last 5 years. I would like to recognize the invaluable assistance and inspiration that you all provided during my study.

Ås, August the 15th 2020

_____

Anja Stene

1. Table of Contents

1. TABLE OF CONTENTS 3

2. SUMMARY 6

2.1. CONTEXTUAL 6

2.2. GOALS 6

2.3. SUMMARY OF RESULTS 7

3. INTRODUCTION 8

3.1. MOTIVATION 8

3.2. BACKGROUND FOR THE STUDY 9

4. THEORY 12

4.1. MODEL 12

4.1.1. Meta modelling 12

4.1.2. HC-PLSR 14

4.1.3. Artificial neurons 16

4.1.4. Neural networks 20

4.1.5. PLSR regression 23

4.1.6. Clustering methods 25

4.1.7. K-Means Clustering 26

4.1.8. Fuzzy C-Means Clustering 27

4.1.9. Linear vs non-linear models 27

4.1.10. Collinearity vs Interaction terms 29

4.1.11. Latin Hypercube Sampling 30

4.2. OPTIMIZATION 32

4.2.1. Measures of model performance 32

4.2.2. Feature engineering 33

4.2.3. Feature selection 34

5. MATERIALS 35

5.1. WORKING ENVIRONMENT 35

5.2. MODULES IN PROJECT 36

5.3. DESCRIPTION OF PACKAGES (EXTERNAL) 36

5.3.1. NEST – Network simulation tool 36

5.3.2. Featuretools (Deep Feature Synthesis) 37

5.3.3. Scikit Learn – PLSR Regression 38

5.3.4. SkFuzz – Fuzzy clustering and prediction 38

5.3.5. PyPet – Hierarchical organization of work ... *39*
5.3.6. Latin Hypercube Sampling from SMT ... *40*
5.3.7. Elephant – Adding Statistics ... *40*
5.4. Implementation design choices ... 41
6. METHODS ... 41
6.1. Overall metamodelling pipeline ... 41
6.2. Data generation implementation. (network creation, simulation, storing) ... 42
6.2.1. Simulations using NEST ... *42*
6.2.2. Sampled feature space ... *44*
6.3. Data Handling ... 44
6.4. Data preprocessing ... 45
6.4.1. Transforming Features ... *45*
6.4.2. Adding statistics /spike train summaries using elephant ... *46*
6.4.3. Standardizing/scaling ... *47*
6.5. Data inspection ... 47
6.6. Metamodelling Procedures ... 49
7. RESULTS ... 52
7.1. Data preparation ... 53
7.2. Combined Model Variations ... 54
7.2.1. PLSR original terms ... *54*
7.2.2. PLSR Polynomial ... *54*
7.2.3. Variation a ... *55*
7.2.4. Variation b ... *58*
7.2.5. Variation c ... *61*
7.2.6. Variation d ... *64*
7.3. Overview of performance results ... 67
7.3.1. Total overview of model variations ... *67*
7.3.2. FCP Inspection ... *69*
7.3.3. Number of clusters and fuzzifier inspection ... *70*
8. DISCUSSION ... 73
9. FURTHER WORK ... 76
10. CONCLUSION ... 77
11. REFERENCES ... 79
12. FIGURES AND TABLES ... 82
12.1. List of figures ... 82

12.2. LIST OF TABLES 83

13. APPENDIX 84

2. Summary

2.1.Contextual

In efforts of explaining biological system behavior, a common mean has been to use mathematical models. To model intricate biological systems does often require complex, non-linear and high-dimensional differential equation systems. This is especially the case in computational neuroscience, where models of the human brain and nervous system is at center for the mathematical models and theoretical analysis. The human brain consists of 100 billion neurons and 100 trillion synaptic connections, and the electrical activity in these neural networks (interconnected neurons) is determined by a wide range of factors [1] , thus modelling of such networks require a large number of parameters and state variables. In turn, highly complex models result in increased computational costs.

Existing techniques for parameter estimation and sensitivity analysis is often more suitable for low dimensional output space and does typically focus on one output variable at the time. Statistical representation of models is an increasingly explored technique for prediction of input-output relations. Statistical emulations (also called *metamodels*) has shown ability to act as a parameter reduction technique, and thus reducing the computational costs. In addition to improving computational efficiency, it has also shown beneficial for serving as a basis for sensitivity analysis (the study of how the system input variations influence the output).

2.2.Goals

The aim of this paper is to explore the possibilities in using metamodelling on data generated by realistic deterministic dynamic models of complex biological systems, and to implement a specific strategy that has proven useful in other studies [2]. As a part of this, it is also a sub goal to contribute to the development of a robust metamodelling methodology capable of producing accurate predictive mappings which allows for extensive automation. This can then also serve as a tool for exploring of dataset by other scientists.
The content of this study can be summarized to contain an overview of some benefits of metamodelling, and a reflection upon modelling strategies for cultivating interdisciplinary understanding and collaboration across scientific fields. It will introduce a framework for applying regression methods to non-linear data. Specifically, a reasonably new regression method called Hierarchical Cluster based Partial Least Squares Regression (HC-PLSR) [2] is implemented and demonstrated. The hypothesis herein, is whether a strategy of local

modelling (by separating data using clustering techniques) can account for non-linearities in the dataset, that a single regression (PLSR) model cannot. The HC-PLSR method has proven useful in other cases by improving performance due to local modelling strategies, and this has been demonstrated on different kinds of datasets. Thus, it was of interest to contribute to the exploration of this methodology by using a dataset generated from model simulation of a neuroscientific neural network design described by Brunel [3].

2.3. Summary of results

After this project, resulting content of the work consists of; 1) a framework for implementing and simulation of Brunel's Model A [3] with parameters sampled from the AI-space (Asynchronous Irregular firing), 2) a single, global PLSR model implemented for performance comparing, 3) Implementation of HC-PLSR model variations for exploration of method performance on non-linear data. When inspecting the resulting dataset prepared for modeling experimentation, the relations between regressors (X) and responses (Y) was indeed non-linear, but not what could be described strong non-linearities.

The HC-PLSR local modelling did not outperform the PLSR regression method in all cases, however, it did have a general higher prediction accuracy (R^2, MSE and MAE) when compared to a linear PLSR model. This was especially apparent when no interaction/higher order terms were included. This might indicate that the HC-PLSR does account for non-linearities in the data, but when the data is not strongly non-linear, it might be sufficient to use a polynomial PLSR model (by adding higher order terms and cross terms).

The work process has also demonstrated the need for a modular and generalized framework, because the HC-PLSR method can be optimized and tuned by a number of regression model parameters. Thus, a resulting framework (using python programming language) is shared [4], containing different model variations/combinations, with the purpose of making metamodelling accessible by utilizing a modular strategy.

3. Introduction

3.1. Motivation

Working with modelling of complex systems occur in **several fields of sciences**. Data generated from deterministic modelling, which describes realistic biological systems, are often characterized by a **large range of attributes, properties** and **relationships**. Advanced modelling in biology and neuroscience (to mention some fields) can consist of computationally demanding simulations. Such dynamic models with intricate properties can be difficult to **assess**, and a tool to facilitate the understanding/descriptions of such representations might give **insight to important** characteristics (for sensitivity analysis etc.). Metamodelling can meet both these challenges; firstly, by using a metamodel generated mapping for model reduction, and also as a technique to produce accurate predictive mappings explaining the input-output relationship of models.

Metamodelling has a wide range of use and can be helpful for many different fields. Despite the fact that metamodels have been developed for various different fields and sciences already, it is lacking a standardized methodology that can encourage and motivate for cross disciplinary work and collaboration. Collaboration is especially useful in areas like modelling and mathematical analysis of complex systems, because the methods and theory behind is often quite similar even though the data might differ. The motivation for this study is to utilize cross disciplinary model work, data scientific methods for metamodelling and neuroscience models for simulation of the data to regress.

There has been some work done already that aims for standardizing the modelling of different data, for example the Surrogate Modelling Toolbox (SMT). As the name implies it is an (open-source python) package consisting of libraries for surrogate modelling methods. The focus in SMT, however, is mainly on derivatives and the use of gradient-based optimization. Even so, the ideas of standardizing and generalizing is fitting well with the viewpoints in this paper, and the developed tools herein will follow the same basic principles.

The same modelling strategies can be applied to different types of data, so a “cookbook”-approach for modelling data can provide an efficient tool for scientists, disregarding the type of dataset at hand. Another advantage of this is that a lot of cumbersome work due to implementation and the “trial and error”-way of working (for model optimizing) might be reduced. If there exist a framework that takes input data of a given structure and produces (regression) prediction results of from a selection of metamodel architectures, it can pinpoint

the direction of further work and model improvement. This can be used as a convenient tactic of getting more acquainted with the dataset at hand, and also provide a time-efficient approach to the testing of model design that almost always is required. The “trial and error” way is a widely used approach when creating models in data science, even if the dataset is quite well known in advance. It is hard to know beforehand exactly what strategy in the model design that will yield the best results, and modelling experience is very important. Scientists with less experience in modelling might still (also) benefit from the metamodelling results, and a standardized methodology will grant valuable information. The work for increasing model performance can be time consuming and inefficient, so can a framework be designed to help with this challenge?

3.2.Background for the study

Deterministic dynamic models of complex biological systems contain large numbers parameters and relationship attributes, often connected using (non-linear) differential equations. This model can be described using a metamodel; a statistical approximation, that effectively maps variations in input parameters to variation in the resulting output state variables (for the entire feature space). The input-output relations of realistic dynamic models can be extremely complex, and the use of metamodels can be helpful in regards of representing these complex/high dimensional models. It has also been useful in handling some of the challenges high-dimensional models brings, by increasing speed of numerical solvers and serve as a tool for automated model simplification. Metamodels can also serve as a basis for sensitivity analysis (the study of how the system input variations influence the output) [2] .

Computational neuroscience and computational biology are both evolving and fast-growing fields but making use of metamodelling is currently not a widely used approach here. Still it can be expected [2] that metamodel generated mappings can become useful as model reduction techniques for speeding up simulations, for performing global high-dimensional sensitivity analysis for several purposes, and for comparing high-dimensional prediction spaces of competing models etc. Input parameters and initial conditions can also be predicted from the model, providing opportunities for identifications of relevant parameter ranges [2]. However, for any of these tasks to be fully solved, there is a need for defining a methodology

that can quantify prediction accuracies and yield indications of modelling robustness. Additionally, extensive work and automation of the practice (and existing tools) is expected and the approach must take this into account and facilitate future work. There is a need for substantial development to make modelling strategies generally applicable in several fields (e.g. in biology and neuroscience), and to be feasible for applications by creating open-source and accessible tools.

Regression based analysis is a widely used technique, and mild nonlinearities can to some degree be modelled using polynomial regression (square and interaction terms). However, a robust modelling methodology must be capable of handling data with strong nonlinearities, in particular non-monotone input-output relationships. A candidate approach for this [2] is the HC-PLSR, which makes use of locally linear or locally polynomial regression modelling of selected subspaces of the original complex model. It has proven to be a successful strategy to split complex data into blocks for local modelling, which implies that non-linear and non-monotone response surfaces can be modelled locally by designated polynomial models. The HC-PLSR does also handle linear dependencies between regressors and the inter correlations between the responses, by using Partial Least Squares Regression (PLSR) instead of Ordinary Least Squares Regression (OLS) for the local modelling. PLSR maximizes the explained covariance between the regressors (X) and the responses (Y), and it also makes use of the intercorrelations between the response variables for model stabilization [2]. Consequently, it does not depend on linearly independent regressor variables. PLS Regression is a way of compressing data into its most relevant subspace (spanned by the estimated latent variables, also called principal components (PCs)), and hence provides a versatile means for data compression by reducing the rank of both regressors (X) and responses (Y). This can also be used to identify important features in a complex system. It should also be noted that if the rank of the data is not reduced (i.e. all PLS components are included in the regression model) the PLSR model is equivalent to OLS.

The suitability of PLSR is emphasized when considering the importance of maintaining interpretability. Campbell [5] have shown that metamodels based on subspaces found by PLSR (compared to Legendre polynomials and PCA), gave the simplest and most predictive basis for sensitivity analysis for a set of computational models. As mentioned by Tøndel *et al.* [2], the suitability of PLSR for interpretation of complex biological systems and the use of PLSR in sensitivity analysis is demonstrated in [6]. This in turn was the motivation behind

the new technique of local modelling, by forming the method of HC-PLSR, and is followingly the method to be further tested and explored in this paper.

To investigate and test the performance of the HC-PLSR method of modelling, it was desirable to use a different kind of dataset than the ones already tested. This resulted in a dataset generated from a simulation tool, NEST (see section 5.3.1), which generates data based on spiking patterns from neurons in a neural network. The neural network design is described in detail by Brunel [3] and was developed for investigating spiking behavior in neural networks (excitatory and inhibitory neurons interconnected in a larger network of cells). The network is explored and commented to yield spiking patterns with different tendencies/behavior. These so-called "states" of spiking behavior might generate non-linearities in resulting datasets, which might be better modelled using a HC-PLSR approach. In this paper however, only one state/ form of spiking behavior (and the required parameter interval for generating this state) was included, the "Asynchronous Irregular firing" - state (AI-state). To expand the parameter space after the model has shown useful is a more natural way of developing the model; if the model cannot account for non-linearities in a subspace of the parameter space, then the performance based on the whole range might not be expected to be very good.

As mentioned above, strongly non-linear data can be hard to model well. There is a need for a methodology to tackle this, as well as to simplify computationally demanding simulations (i.e. make them more time and resource efficient). When there exists little or no prior knowledge of the data, it should still be possible to create sufficiently performing models, but this requires a standardized framework. The models could also in this case be used for getting a comprehension on how the data looks like (if no or relatively simple/non-monotone non-linearities are present in the data, the HC-PLSR would not outperform the PLSR model, and the utilization of the hierarchical approach would be unnecessary). It is also ideal if a model can perform/predict within a specified margin of error, that is defined by the model if necessary. All of the reasons mentioned above founded the motive for creating the modelling framework and testing paradigm in this paper. In summary it is aimed at resulting in a methodology that:

- Handles strong non-linearities
- Does not require pre-existing knowledge about the data
- Automizes the modelling process in a time and resource efficient way

- Can provide information about the structure of the given dataset
- Facilitates interdisciplinary work, and encourages collaboration on modelling theory (experimentation)
- Eases and assists the inclusion of domain knowledge into the model architecture (dataset specific)
- Offer modelling results where interpretability is not completely lost due to complexity
- Exploits modularity such that the use of only parts of the architecture is possible if desired

4. Theory

This section is meant to provide the necessary knowledge needed to create a common understanding/intuition of some relevant concepts that are used in creation of this project. It is split up into 2 parts; model related theory, optimization specific.

4.1.Model

4.1.1. Meta modelling

A metamodel (commonly called a surrogate model or an emulation model) is a model of models. The idea behind this concept is that a complex mathematical model can be substituted by a simpler model, in order to reduce computational costs and complexity. This is achieved when the input data (X) and the output (Y) of a simulation experiment (varying the input parameters) is used for calibration. The metamodel then learns to represent variations in the output data and map these changes to variations in the input data. The ideal surrogate model/metamodel will replace a complex model as accurate as possible.
In classical metamodelling, the outputs (from simulations) can be predicted based on simulation inputs, whereas in inverse metamodelling, it is the inputs that are predicted based on output data from the simulations. See Figure 1 - Inverse and Classical Metamodelling illustration for visualization.
These two metamodelling variations can in turn, alone or combined, be used to explain/describe behavior of the original model (used to run the simulations).
Different strategies (machine learning techniques, supervised and unsupervised) are used in construction of metamodels, completely depending on the purpose of development of the model. In this project, a strategy called HC-PLSR [2] is implemented and tested.

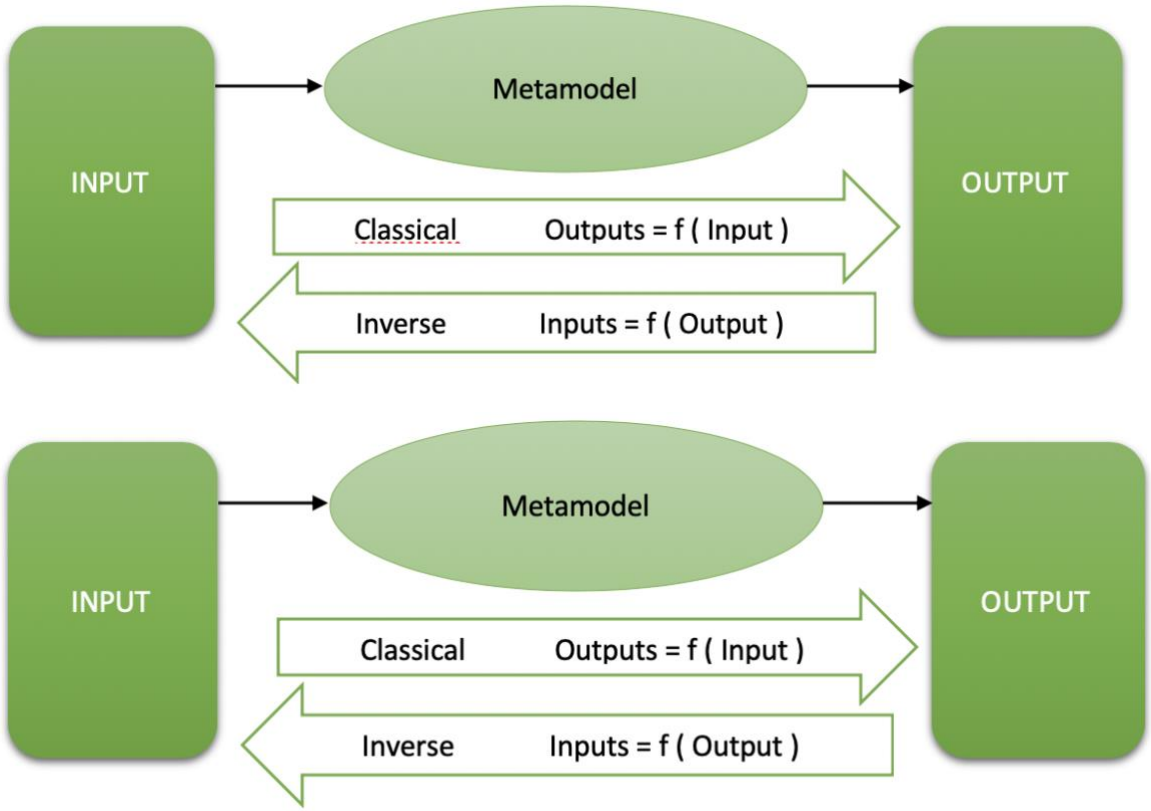


Figure 1 - Inverse and Classical Metamodelling illustration

Metamodelling of a deterministic dynamic model is a statistical approximation to the mapping of a number of parameters to a range of state variables. It can be exploited as a shortcut around heavy/expensive computations, and it is also (and maybe especially) relevant if one is more interested in summaries and descriptions of the resulting simulations rather than the exact simulation results in itself. For example: there might be of more interest to find state descriptive measures/tendencies (e.g. Regular Synchronized firing or Irregular Asynchrone firing behavior [3]) of the spiking behavior of a network, rather than the exact spike train for all neurons in the network (for all parameter combinations). To obtain a cheap way for accessing an estimate/indication of the behavior of the network, based on input parameters, without always computing simulation results can be of interest for many reasons. (e.g. in sensitivity analysis, further modelling in one particular parameter subspace etc.). It is important to emphasize that modelling often has a purpose/goal, and the field using the model results and or functionality will better assess and determine what can be described as acceptable results and predictive performance error.

Meta modelling is creating a model that predicts expensive (time, computational power, money) response variables from cheap (easily accessible) features. To run a simulation like

the one described in 6.1.1 for many hundred or even thousands of times, is a computationally expensive process, and if possible, it is very convenient to be able to predict how the output of such simulations would result in. The simulation might also create a lot of information that is not strictly necessary for a given research purpose, (i.e. the exact spiking times for every neuron for every simulations) when one might only interested in state describing measures (like correlations, covariances and other statistical measures, see section target features) of such spike trains. To use a lot of resources computing information that will go to waste should be avoided. Thus, to be able to predict the measures mentioned would be of great benefit.

4.1.2. HC-PLSR

HC-PLSR is an extension of the PLSR method and was proposed by Tøndel et. Al [2]. The HC-PLSR modelling pipeline consists of splitting the parameter space into regions where local PLSR models are created for each subspace. This method can reveal different behavior of the model for individual subspaces. The HC-PLSR approach has shown [2] useful for emulating models with complex non-linear characteristics, and the method can be adjusted to suite the complexity of the dynamic model behavior in a flexible way.

Existing metamodelling work does often [2] use OLS, that requires linearly independent input parameters (See section 4.1.5 PLSR regression). OLS regression-based modelling is primarily focused on modelling single output and cannot handle multicollinearity in the data parameters. The requirements for using OLS are not always met, and other techniques involving Deep Learning (ANN) exists. However, deep learning removes some of the interpretability, and this is when HC-PLSR can serve as a solution. HC-PLSR modelling uses (multivariate) PLSR regression for modelling local subspaces on the parameter space. In this way, it utilizes the intercorrelations between the output variables, and due to the clustering/separation of feature space into subspaces, it can also model highly non-linear and non-monotone input-output relations, which characterize many biological systems.

The implementation of metamodel exploration explained in section 6.6 Metamodelling Procedures, was created with the intention to explore the performance of HC-PLSR method [2]; a way of creating local regression models for each part of the nonlinear relationship between regressors and predictors (features and target). Implementations of the HC-PLSR

was based on an initial global (second order) PLSR using all observations from the training (or calibration) set, to identify a preliminary (global) model (source K). Next, a clustering of the output from the global model (X-/Y-scores of the PLSR model) was used to split the original observations into subsets where local polynomial regression models were hypothetically more likely to improve prediction results [2]. Finally, local PLSR models were created and calibrated individually for each of the clusters (with some exploration with regards to cluster restrictions). A 10-fold cross validation was also used here, to find the optimal number of principal components to include in the model. For an overview of the model pipeline, see Figure 2 - Illustration of the HC-PLSR approach from [2] [2] below.

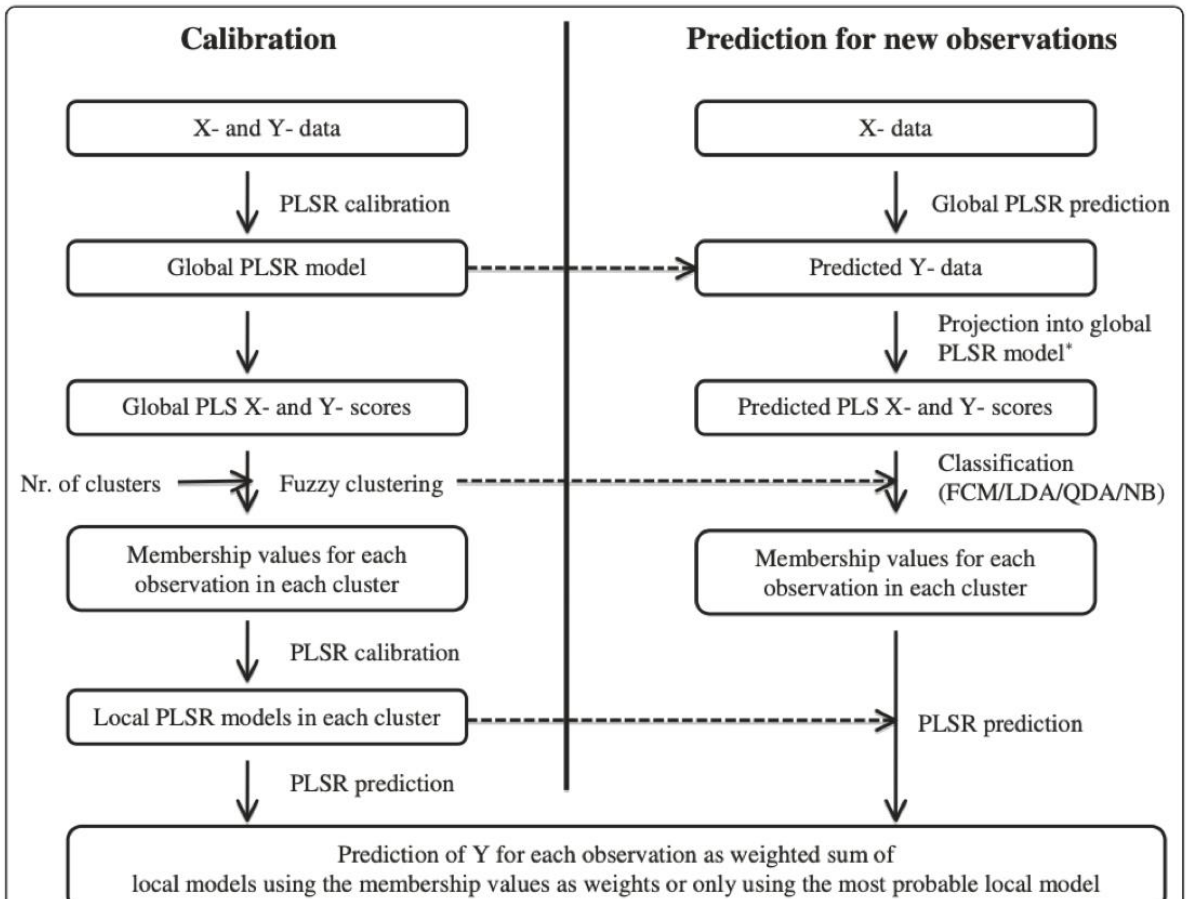


Figure 2 Illustration of the HC-PLSR approach. The HC-PLSR "pipeline" starts with calibration of an initial global polynomial PLSR using all observations in the calibration set. This global PLSR model provides PLS scores and loadings, which constitute the basis for separation of the calibration set observations into groups by fuzzy *C*-means (FCM) clustering [44,45]. Local PLSR models are then calibrated in each cluster. Predictions of response variables for new observations (or test set observations) are done by a) selecting the local model for the most probable cluster based on classification or by b) computing the regression coefficients as a weighted sum of the local models, where the weights are estimated cluster membership probabilities from the classification. *See Additional file 1: Appendix 1, Eq. A4 and A6.

Figure 2 - Illustration of the HC-PLSR approach from [2]

4.1.3. Artificial neurons

The cells that work as building blocks in the nervous system are called neurons. These specialized cells handle communication by electrical and chemical signaling between each other. There are two kinds of neurons; sensory and motor neurons. The sensory neurons carry information from the sensory receptors to the brain for interpretation and processing, and the motor neurons carry signaling from the brain to the muscle cells in the body. The neurons stop reproducing at birth, but the connections between them, the synapses, continue to develop and form throughout life.

The neuron is constructed as a cell body, dendrites and an axon (see Figure 3 - Components of the neuron from [7]). The cell body contains the cytoplasm and nucleus and is surrounded by a cell membrane to protect the cell. Dendrites are connected to (and surrounding) the cell body and act as receivers of signals that are transmitted from other cells and into the cell body. The axon extends from the cell body and ends up in axon terminals to form the end point of the neuron. This is where the Action Potential (AP, further described in [8]), initiated in the cell body, travels along before it reaches the contact point where the current cell is connected to other neurons. These connection points are called synapses and are the junction gaps between the axon terminal of one presynaptic neuron, and a dendrite of a postsynaptic neuron. There exists both electrical synapses, where ions flow across the gap, and chemical synapses where chemical signals, neurotransmitters, are released into the synapses to flow across. The receiver cell of the synaptic connection can either be less or more likely to fire an action potential after transmitting of the signal. This is the difference between excitatory and inhibitory synapses; the excitatory post synaptic potential is depolarizing, and the inhibitory post synaptic potential is repolarizing the cell body. In effect of this, signals from an excitatory synapse makes the receiving neuron more “likely” to fire an AP, and contrary makes is less likely to fire an AP if signals are from an inhibitory synapse. The post synaptic neuron summarizes the inputs from the excitatory and inhibitory synapses, and then “decides” (based on voltage change and the respective firing threshold) whether to fire an action potential or not.
It is important to remember that one axon can be connected to several neurons’ dendrites, and a neurons’ dendrites can receive signaling from several other neurons (and their respective axon terminals). This forms the basis of a neural network, further described in the section 4.1.4 Neural networks. The human brain and the nervous system are constructed by an

interconnected network of neurons. Thus, exploring its behavior (read: how signaling progresses and transmits throughout the network) and characteristics might be beneficial for the medical understanding of different diseases, conditions and behavioral patterns [9].

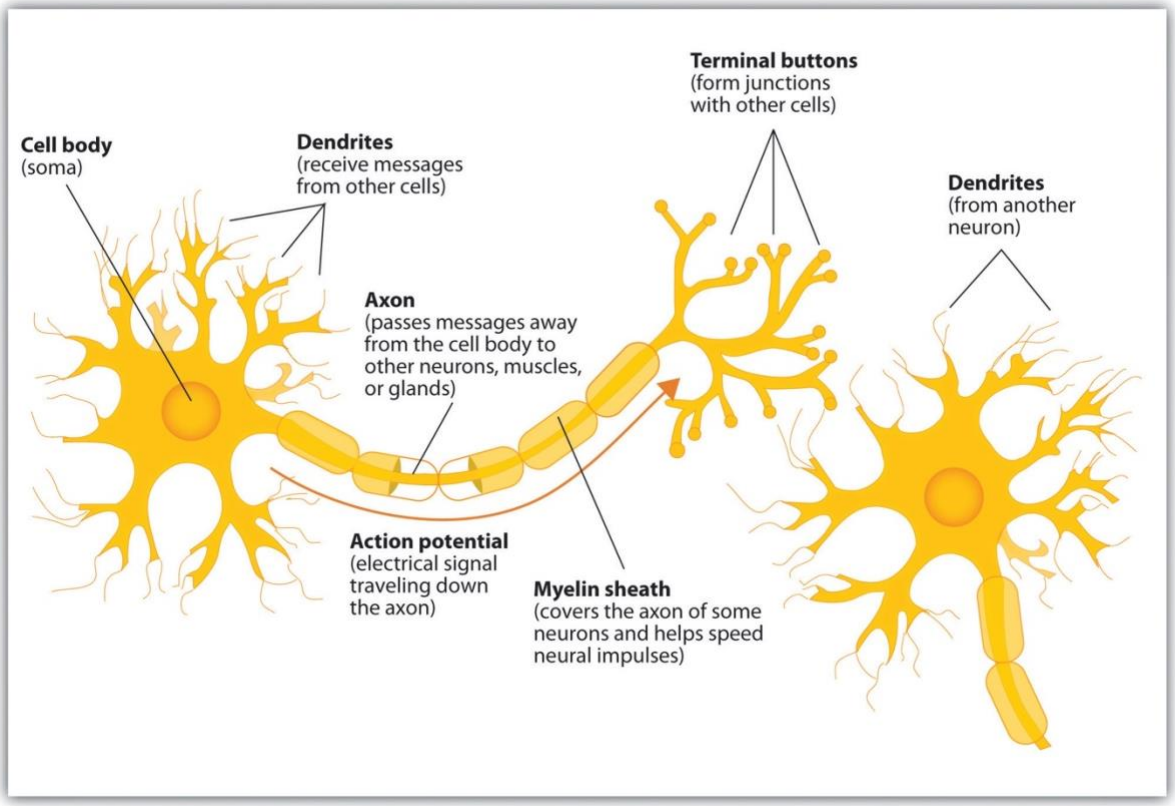


Figure 3 - Components of the neuron from [7]

Signaling in and between neurons can be of both chemical and electrical type. The signal is initiated in the cell body as a reaction to the received signals via dendrites. It then continues down along the axon to the axon terminals, in the form of an action potential. An action potential is essentially an electrical signal, which is initiated in the cell body by a rapid voltage change (polarization and depolarization) across the membrane. Figure 4 - The Action Potential from [10] illustrates the development of the (voltage) potential across the membrane.

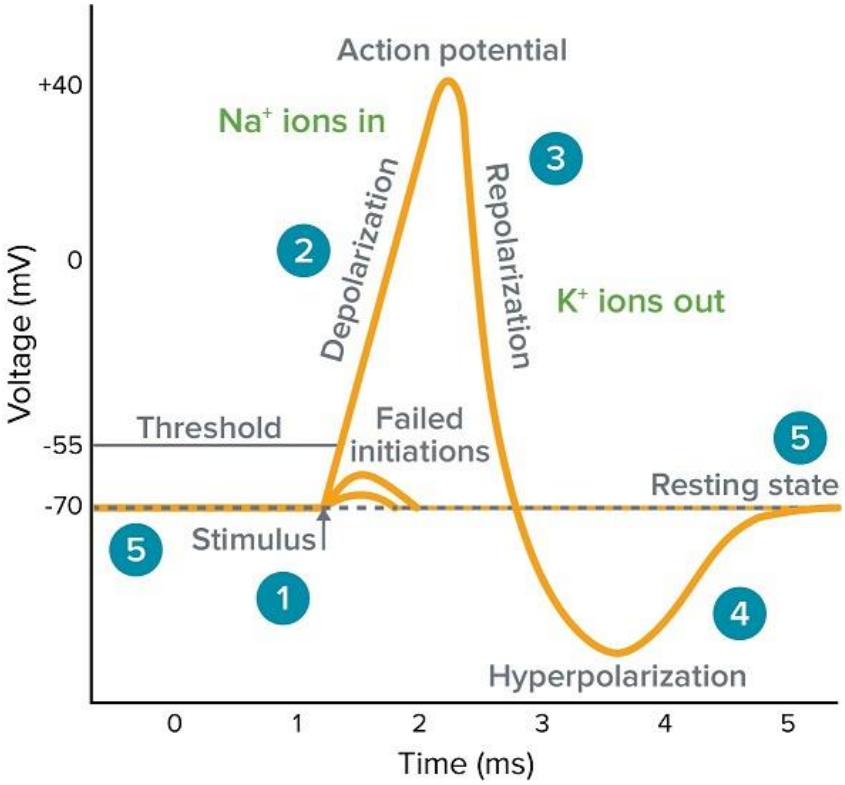


Figure 4 - The Action Potential from [10]

Before an action potential it is initiated, the cell body has a resting potential of around -65 mV. This means that the cell body is negatively charged, with reference to the outside of the cell membrane, and the charge is maintained (thus, resting) by controlling and positioning electrically charged ions. The resting membrane potential is maintained by pumps and channels in the membrane, and forces such as electrical drift and diffusion keeping a balance in the ion flow and net currents [1]. In the case of an action potential, the cell body signals for the membrane pumps and channels to allow for positively charged ions to enter the soma (cell body) and lets the negatively charged ions out. This causes a rise of charge in the soma, and when this reaches a certain threshold (approx. -55mV), an action potential is released. The firing of an action potential then propagates down the axon, which can be described as an electrical signal being sent down the axon. One important aspect here, is that the cell will either fire an action potential, or it will not. This means that there is no such thing as partially firing of an action potential, and the only thing deciding this is whether the soma reaches the threshold value. As a result of this, a cell will always fire with full strength, and the signal is also carried down the axon with full intensity. After an action potential has been fired off by the cell, there is a short (ms) refractory period inside the neuron. During this phase, a new action potential cannot form. The refractory period can be described as the process where channels and pumps in the membrane closes, and eventually causes the cell potential to reach

its resting potential again. When resting potential is reobtained, a new AP can potentially again be formed, sending new electrical signals down the length of the neuron's axon.
The construction and processes of a neuron for simulations can be quite complicated, and it can be desirable to make simplifications. Many reasons can be mentioned (see [8]) for why it can be appropriate to make simplifications;

- They are useful for incorporating into bigger networks (see section 4.1.3), because they are computationally cheaper, and can in turn be easier to analyze mathematically (depending on the simplified model used).
- The number of variables can be reduced while retaining many important and relevant properties of the neuron.
- When trying to understand the behavior of a network of neurons, simplified neuron models can be used to model the essential behavior of a neuron's mechanisms. This allows for faster simulations, less demanding computations and mathematical interpretability and analysis of the network. An example of this is if the "integrate-and-fire"-neuron simplification model (see next section) is used, consisting of two main components; a differential equation to describe the membrane potential, and a threshold for spike firing.

There are aspects to consider when deciding what type of neuron simplifications to use, and this is discussed further in [8]. A famously known and old neuron model is the "integrate and fire" neuron model (IF-model), where no variables of the neuron remains other than the membrane/cell potential. In order to produce spikes (APs), a spike-generating mechanism is also added to the model. Despite the biological complexity of generating an action potential, predicting the initiation of one is quite straight forward, and this is what the IF-model utilizes. As already mentioned, the membrane potential reaches the threshold potential, a spike is triggered and neurotransmitters are being released from the presynaptic terminal, to signal other neurons. The If model tries to capture the concepts where the membrane is being charged by flow of ion currents, then discharged again after the threshold potential is reached and an AP is being fired. This mechanism has an RC-circuit equivalent for conceptualizing, see [8]. The IF-model describes this using separated (not coupled) differential equations, i.e. it is faster to simulate and is thus very useful when handling larger network simulations. One of the most important applications of the IF neuron, is for understanding the variability of neuronal firing patterns. Patterns and "states" of neurons interconnected in a neural network

is explored and described by Brunel [3]. Important questions in computational neuroscience include the descriptions of input-output relations of connected neurons, and to better understand this relationship (and the variabilities of them), IF-models have played an important role. Especially in network simulations.

4.1.4. Neural networks

A neural network can be defined as a set of interconnected neurons, that communicates using action potentials (AP) via synapses. Biological neural networks hold many unanswered questions and is widely studied for exploration of behavioral and chemical analysis to mention some.
In order to explore and understand more of the human body and processes therein, simulations and modelling of different processes serve as an important tool. Complex events and behavior can be difficult to model, and it is therefore a current culture for "modelling what you can measure". This means that results from models are not yet useful if not comparable with real measurements that can confirm that the model behaves accurately. This means that there is a constant need for cooperation between the computational and experimental fields. One of the more famous studies on neural networks and computational neuroscience is Brunel's [3] work. In one of the published articles of this work [3], two different kinds of model designs were implemented to explore spiking tendencies within the network. The Brunel model [3] uses simple leaky IF-models in a network of randomly interconnected neurons, to explore the dynamics of the spiking patterns. Model A is used in this paper's implementation and is more elaborated in [8], but below is a summarized description.

The network neurons are connected using both excitatory and inhibitory synapses and are thus consisting of three neuron populations; Excitatory Neurons N_E, inhibitory neurons N_I, and a population of independent neurons representing external activity input N_{ext}. This population of identical neurons (N_{ext}) are represented by an independent Poisson process with firing frequency V_{ext} (from outside the network). The model is composed of N neurons, where the number relation between excitatory and inhibitory neurons is $N_E / N_I = 4$, thus there are four times as many excitatory neurons. The network is sparsely connected, meaning that each neuron is connected to (with a given probability) C randomly chosen other cells in the network.
Depolarization of the neurons (i in 0, 1,..., I, …, N) in the network follow the equation

$$\tau \dot{V}_i(t) = -V_i(t) + RI_i(t),$$

Equation 1

Where $I_i(t)$ is the synaptic currents arriving at the soma, i.e. the sum of all contributions of the spikes arriving at different synapses. These spike contributions are in Brunel (2000) [1] modelled as delta functions

$$RI_i(t) = \tau \sum_j J_{ij} \sum_k \delta\left(t - t_j^k - D\right),$$

Equation 2

Here, the first sum is a sum of different synapses, with postsynaptic potential amplitude J_{ij}. The second sum is the sum of all spikes arriving at the synapse J arriving at time t_{jk} + D where D is the transmission delay. Brunel's model has the same postsynaptic potential amplitude at each amplitude, J = J > 0 for excitatory synapses, and -g*J for the inhibitory. The external synapses are activated by a Poissonian spike train, (elaborated in [8]).
When the neuron reaches the threshold potential, an action potential is fired. After the repolarization phase, the neuron has a refractory period t_{rp}, during which the potential is insensitive to stimulation.

The Brunel model was developed for exploring global and local spiking in neural networks. The findings of Brunel states that networks of neurons can be found to collectively spike with different tendencies/behavior, referred to as states. The paper continues to describe these network states, affected by oscillations and frequencies of spike times, visualized in Figure 5 - Illustration of network states, (figure 8 from [3]). The paper continues to illustrate global and local states, that depend on the given input parameters for the simulation. The network design (model A) from Brunel's paper was implemented and confirmed to reproduce the results of Brunel's exploration of network characteristics in [3]. Simulation and network design were achieved using NEST Simulation tool (section 5.3.1) and resulting plots from these simulations can be explored/verified in [8]. NEST is a simulation framework that allows for network design and exploration of neuronal activity. A description of this tool is found in 5.3.1 NEST – Network simulation tool.

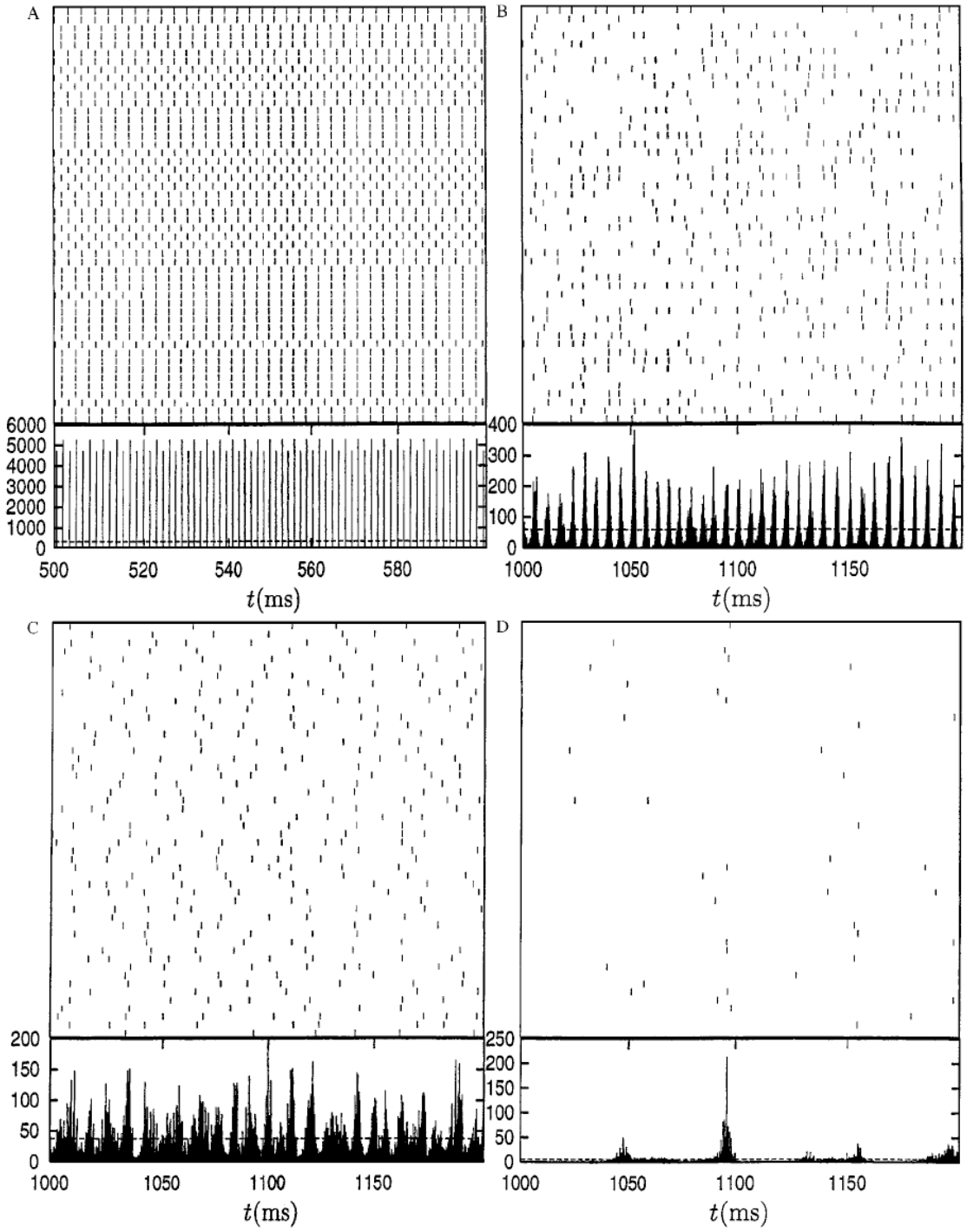


Figure 8. Simulation of a network of 10,000 pyramidal cells and 2,500 interneurons, with connection probability 0.1 and $J_E = 0.1$ mV. For each of the four examples are indicated the temporal evolution of the global activity of the system (instantaneous firing frequency computed in bins of 0.1 ms), together with the firing times (rasters) of 50 randomly chosen neurons. The instantaneous global activity is compared in each case with its temporal average (dashed line). **A:** Almost fully synchronized network, neurons firing regularly at high rates ($g = 3$, $\nu_{ext}/\nu_{thr} = 2$). **B:** Fast oscillation of the global activity, neurons firing irregularly at a rate that is lower than the global frequency ($g = 6$, $\nu_{ext}/\nu_{thr} = 4$). **C:** Stationary global activity (see text), irregularly firing neurons ($g = 5$, $\nu_{ext}/\nu_{thr} = 2$). **D:** Slow oscillation of the global activity, neurons firing irregularly at very low rates ($g = 4.5$, $\nu_{ext}/\nu_{thr} = 0.9$).

Figure 5 - Illustration of network states, (figure 8 from [3])

4.1.5. PLSR regression

In data Science and Machine Learning, supervised learning methods are divided into 2 main categories; classification and regression. Classification is where the Machine Learning algorithm predicts discrete outputs (samples are classified), and regression methods predicts continuous output, hence this can also in some cases be called Continuous Supervised Learning.

Regression can be univariate and multivariate; univariate regression modelling involves the analysis of a single variable. In Multivariate Regression, the target Y consists of several dimensions. Multivariate regression is a method for correlating one data matrix X, of predictor variables (features), to the information in a second data matrix, Y, of response variables (targets). Partial Least Square Regression (PLSR) is a way to do multivariate regression where intercorrelations between the Y-variables are utilized for model stabilization and is used in this model design (see section 6.1 Overall metamodelling pipeline).

The motivation for using PLS instead of OLS is that with bigger data it is harder to exclude the possibility of features being correlated. When using ordinary OLS, the calculation of the regression coefficients includes a matrix inversion, and this process becomes unstable with highly correlated features. This again will lead to regression coefficients that does not represent the individual variables "effect" on the response. The PLSR method handles this by decomposing the feature space to a subspace with orthogonal components, and with orthogonal components it is easier to separate the individual effect of features on the response (Y). Thus, the PLSR allows for the predictor variables to be highly correlated or even colinear. PLSR constructs new predictor variables (principal components) as linear combinations of the original predictor variables, similar to the method of Principal Component Regression (PCR). The difference from PCR, however, is the way the components are constructed. PCR creates components by assessing the variability in the predictor variables without considering the response variables. On the other hand, PLSR takes the response variables into account when creating the components, and thus are able to fit the response with fewer components. One might say that PLSR is sort of a PCR for X and Y. In PCA, what is optimized is the eigenvalues, but for PLSR what is maximized is the

covariance. It should also be mentioned that PLSR can be used to detect outliers in the relationship between X and y.

The core of PLSR is the idea that if U (the scores of Y) can be predicted, then Y can be calculated. And U can be predicted by using the scores of X. PLSR maximizes the covariance between the components through the scores of X and Y, such that the 1st component of X has the highest covariance with the first component of Y.

To predict using new data, the X scores are calculated by

$$X_{scores} = X_{new} * X_{loadings}$$

Then Y scores are calculated by

$$Y_{scores} = X_{scores} * R$$

where R is the relationship matrix (between X_{scores} and Y_{scores}) containing the inner relation regression coefficients on the diagonal. The prediction of Y is then calculated as

$$Y_{predicted} = Y_{scores} * Y_{loadings}$$

If one is just interested in the prediction of Y, one can collect the computations into one operation by using

$$Y_{predicted} = X_{new} * B$$

where B is the regression coefficients of the PLSR model.

$$B = X_{scores} * R * Y_{loadings}$$

For many models, the creation of new features (i.e. feature combinations such as cross terms of higher order terms) may improve performance. This can be the case for linear models especially, because by adding non-linear terms, new knowledge is added to the linear model (see section 4.2.2 Feature Engineering). One other advantage of PLSR is that many such non-linear combination of features may be added, and the resulting model will only use the first n components of the PLSR model. It can be difficult to know in advance what type of new features that will improve the model, so it saves time if one can use the PLSR as a way of performing dimensionality reduction/feature selection (see section 4.2.3 Feature selection).

For optimizing the PLSR model, the Mean Squared Error (MSE) of the validation data is calculated for different numbers of components to include in the final model. If all components are kept, this is the equivalent to OLS. PLSR is a linear regression model (see

section 4.1.9 Linear vs non-linear models), and adding non-linear feature combinations to the predictor variables (see section 5.3.2 Featuretools (Deep Feature Synthesis)) can make the PLSR modelling account for non-linearities in the dataset. The nonlinearity accounted for by the resulting polynomial PLSR model (including e.g. cross and higher order terms between features) is introduced by the new polynomial features.

4.1.6. Clustering methods

Clustering is an important technique for extracting useful information from various high dimensional datasets and is a useful data analysis tool [11].
Data analysis is commonly used in modern science research, for example in communication science, computer science and biology science [12] to mention some. Clustering plays a significant role in data analysis [12]. It can be used to discover hidden patterns in data, by grouping together similar objects and separating dissimilar objects. There are several defined measures of similarity, and a common example of this is the Euclidian distance measure (calculated in the feature space). The clustering of data is a way to organize data into categories, so that the similarity inside the cluster is maximized and dissimilarity from datapoints outside the cluster is maximized.
Clustering is an unsupervised learning method and is a common technique for statistical data analysis used in many fields [13]. But why is clustering relevant in metamodelling?
Clustering deals with data structure partitioning and can serve as a basis for further learning and understanding of the data.
Each clustering method make assumptions about the data points to constitute their similarities, and every strategy can result in different clusters. It is therefore important to not ignore the characteristics of each algorithm's strengths and weaknesses.

Measuring cluster algorithm performance is an important consideration regarding Machine Learning and modelling. Since clustering is an unsupervised learning technique, the evaluation of models is less straight forward than for supervised learning where the labels for every training sample is present. However, there are a few techniques for assessing and giving some insight to how the clusters change depending on what cluster method is used. It is still not possible to measure the validity of the model (because no labels exist), but the techniques can rather serve as a comparative analysis between different models. One method is assessing if the internal measures of the training data are similar to the measures in the test

data. Another technique is Silhouette Analysis, which can be used to explore the separation of the different clusters as illustrated in Figure 6 - Illustration of Silhouette Analysis concept, from [14] . This might serve as a tool to assess parameters of the cluster method visually [15].

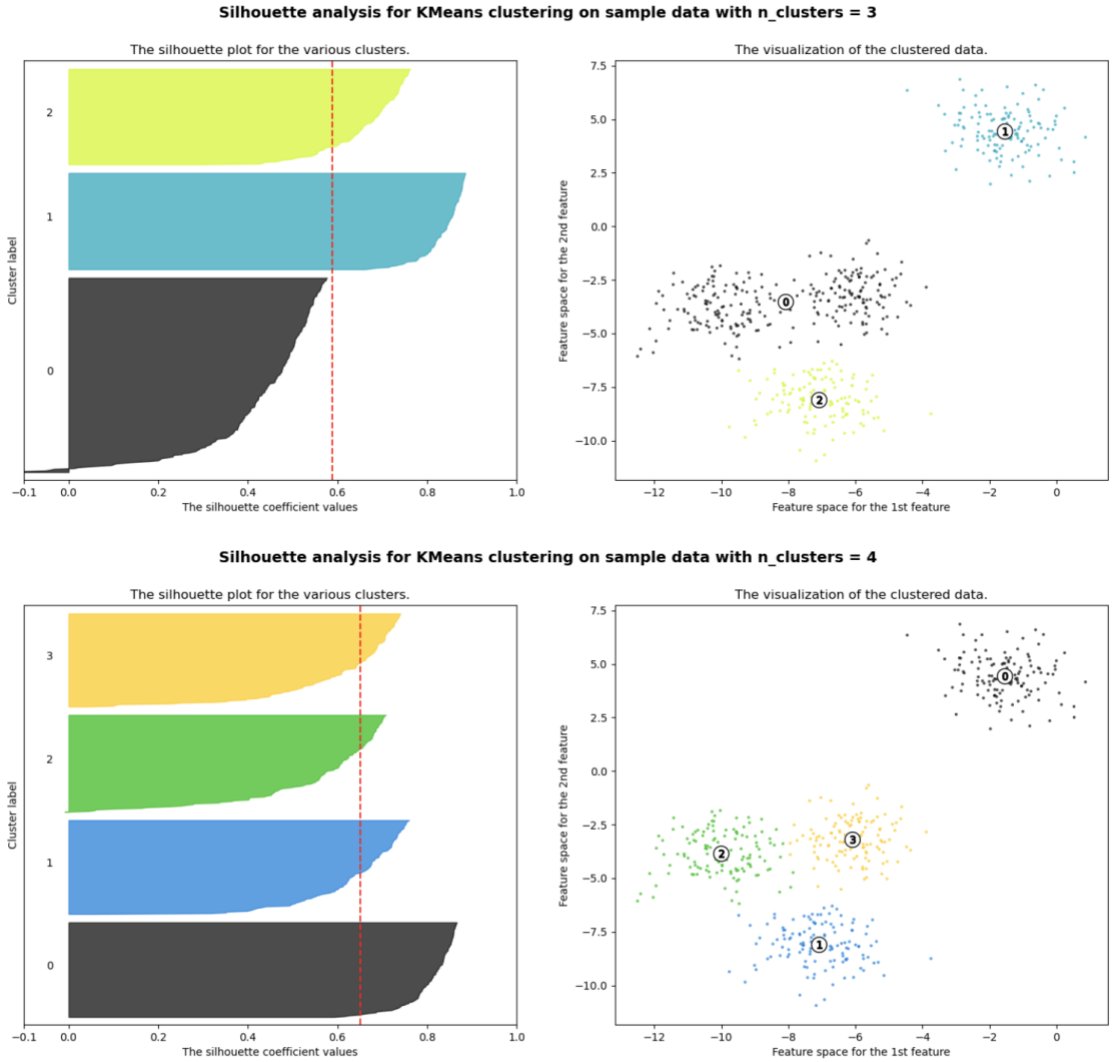


Figure 6 - Illustration of Silhouette Analysis concept, from [14]

4.1.7. K-Means Clustering

K-Means is probably the most well-known clustering algorithm. It has the advantage of being quite fast, as the computations involved mainly is calculating the distances between points and cluster centers. When using K-Means, the number of clusters needs to be predefined, which is not necessarily a trivial case. Since the clustering method can contribute to the discovery of hidden patterns in the data, the predefinition of clusters might limit that

exploration of hidden knowledge. Some of the insight the clustering method could have provided can be lost when the method cannot freely create as many clusters as is required. K-Means initializes the cluster centers randomly, and thus may yield different clustering results on different runs (if not a seed is set). One should note this lack of consistency and be aware that results might change [13], so it might be beneficial to test several seeds for a consistency check.

Pseudocode of K-Means Clustering:

- Select the number of classes and randomly initialize their respective center points.
- Compute the distance between datapoints and all cluster centers.
- Classify the datapoint to the closest cluster center
- Recompute the cluster center by calculating the mean of all points in the cluster.
- Repeat the steps above until 1) the cluster centers do not change much, or 2) has reached the maximum number of iterations.

4.1.8. Fuzzy C-Means Clustering

Clustering can be divided into two subgroups; hard and soft clustering. Hard clustering is about grouping data points in such a way that a data point can only belong to one cluster each (like K-Means Clustering). Soft clustering allows for data points to exist in multiple clusters. As a further development of the idea that a sample can exist in several clusters, Fuzzy C-Means returns the probabilities for a sample belonging to each of the clusters. The label for a sample belonging to a cluster is no longer a discrete value {0,1}, but is changed to a continuous variable interval [0,1].

4.1.9. Linear vs non-linear models

OLS regression is a Machine Learning technique that allows for associating one or more explanatory variables with a dependent variable (response). All Machine Learning models try to approximate the function, $f(x)$, that accurately describes the relationship between the dependent and the independent variable, and in Linear Regression it is assumed that this $f(x)$ is linear. The objective is then to approximate the coefficients, that is the intercept and the slope.

The term "linearity" refers to the linear relationship between two or more variables. If drawn in a two-dimensional space, the relationship would be a straight line, as illustrated in Figure 7 - Linear relationship illustration.

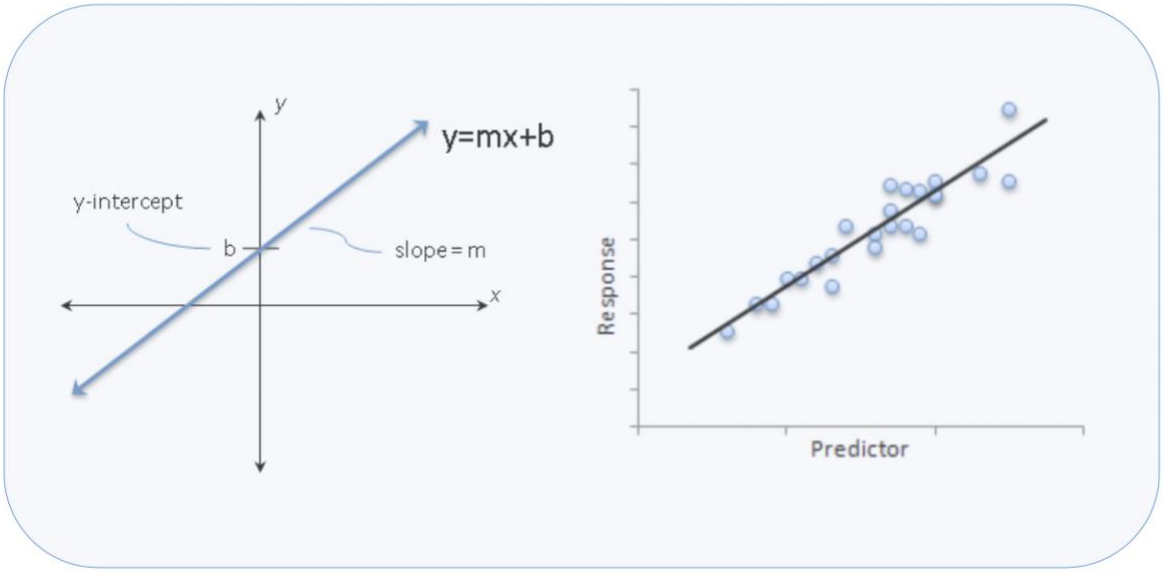


Figure 7 - Linear relationship illustration

The linear relationship (the regression line) is the line that fits the given datapoints the best (see 4.2.1 Measures of model performance). The job of the Regression Algorithm is to fit multiple lines to the datapoints, and then returning the line that results in the least error. This concept of regressing linear relationship between features, can be extended to cases where there are more than two variables, which is called *multiple* linear regression. A regression model involving multiple variables can now be represented can be explained as moving from

$$y = mx + b$$

to

$$y = b0 + m1b1 + m2b2 + m3b3 + ... mnbn$$

This is the equation of a hyperplane (since its more that 3 dimensions).

In the case of *multivariate* regression, we have more than one response variable, and the model finds the most optimal coefficients for all the attributes (intercepts and coefficients = intercepts and slope steepness). PLSR generates one model for all Y-variables at once through scores and loadings, instead of generating separate models (relationship explanations) for each response variable.

4.1.10. Collinearity vs Interaction terms

This section it means to provide an intuitive understanding on the differences between feature collinearity and interaction terms, since multivariate PLSR models allows for collinear features, and the model performance is increased when cross- and interaction terms is added to the feature space.

Collinearity between X_1 and X_2 means that X_1 is linearly correlated to X_2, that is,

$$X_1 \approx a + bX_2$$

It should be noted that the response, Y, is not considered when assessing collinearity.
Suppose the regression model is

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

When collinearity exists between X_1 and X_2, the model can be written:

$$Y = \beta_0 + \beta_1 (aX_2 + b) + \beta_2 X_2 + \varepsilon$$

which in turn reduces to

$$Y = \gamma_0 + \gamma_1 X_2 + \varepsilon.$$

Interaction terms are included in the model when the effects of X_1 and X_2 on the response are not additive (mark that the response, Y, is considered). To illustrate what is meant by "effects are not additive", consider the following:
If the "effect" of X_1 on Y is not "independent" of X_2, but is affected by the variable X_2, a typical solution is assuming that the coefficient of X_1 , β_1, is linearly correlated to X_2. In other words,

$$\beta_1 = a X_2 + b$$

Supposing that the regression model

$$Y = \beta_0 + \beta_1 X_1 + \beta_0 X_2 + \varepsilon$$

and that that β_1 is linearly dependent on X_2, the original model can be written as:

$$\begin{aligned} Y &= \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon \\ &= \beta_0 + (a X_2 + b) X_1 + \beta_2 X_2 + \varepsilon \\ &= \beta_0 + bX_1 + \beta_2 X_2 + a X_1 X_2 + \varepsilon \\ &= \gamma_0 + \gamma_1 X_1 + \gamma_2 X_2 + \gamma_3 X_1 X_2 + \varepsilon \end{aligned}$$

By this it is clear that collinearity and interaction terms affect the original model in different ways.

If X_3 is denoted as

$$X_3 = X_1 X_2$$

the above regression model with interaction terms is just a new model with one new variable, X_3, added. It is not always clear whether introducing $X_3 = X_1 X_2$ will cause the model to overfit. However, if there really exists an interaction between features, and this interaction is not included in the model, it will naturally underfit. On the other hand, the model might be overfitted if interaction terms are added where interaction between the features does not exist. However, there the risk of overfitting arises as long as new features are introduced to the model. It is therefore always important to include model validation techniques to ensure model performance on unseen data.

It is also interesting to note that the interpretation of the model is affected by the interaction terms. To exemplify this: if explaining

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

one may say that if X_2 is fixed, as X_1 increases by one unit, Y increases by β_1 units.
But if explaining

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \varepsilon$$

one could rather say that if X_2 is fixed, as X_1 increases by one unit, Y increases by $\beta_1 + \beta_3 X_2$ units. This shows that the unit contribution of X_1 to Y is a function of X_2, namely "interaction".

4.1.11. Latin Hypercube Sampling

The Latin Hypercube Sampling (LHS) is a semi random sampling procedure that is especially suitable for use in high-dimensional data. This is mainly because it separates into several hypercubes (more than 3 dimensions), and samples randomly within each hypercube. A detailed elaboration on the LHS can be found in the original paper [16].
The use of LHS experimental design is especially convenient when performing multiple automated simulations for parameter exploration, since variables are sampled from uniform distributions and ensures that the ensemble of parameters is representative of the natural variability of the systems input parameters.

The LHS is based on the Latin Square design, where a single sample exists in each row and column (in 2D space). A hypercube is a cube with more than 3 dimensions, and LHS is extended to allow for sampling of multi-dimensional feature spaces and hyperplanes. The key to LHS is stratification of the input probability distributions [17]. Stratification involves division of the cumulative curve (see Figure 8 - Sampling step in Latin hypercube sampling (LHS), from [18]) into even intervals on the cumulative probability scale.

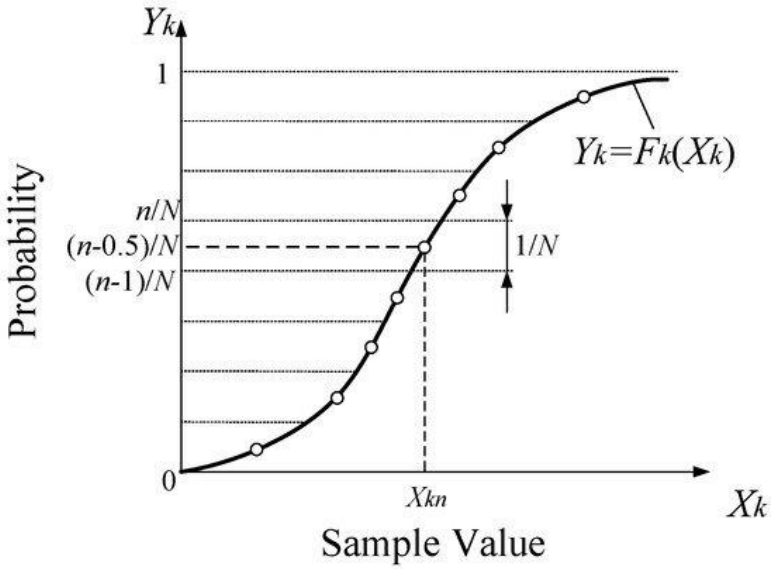


Figure 8 - Sampling step in Latin hypercube sampling (LHS), from [18]

A sample will then be taken randomly from every interval of the input distribution, as illustrated in Figure 9 and Figure 10 below. This method forces representation of values in all intervals (stratifications), and consequently forces recreation of the input probability distribution. In short; the space to be sampled from is divided into N equal partitions, and then choosing a random datapoint in each partition. The technique being used during LHS is "sampling without replacement".

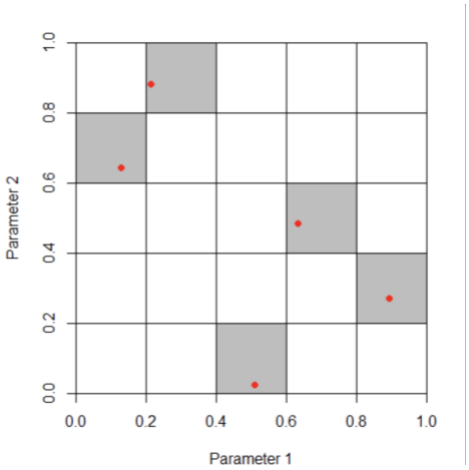


Figure 9 – Two-dimensional random sampling of a uniform Random LHS with 5 samples

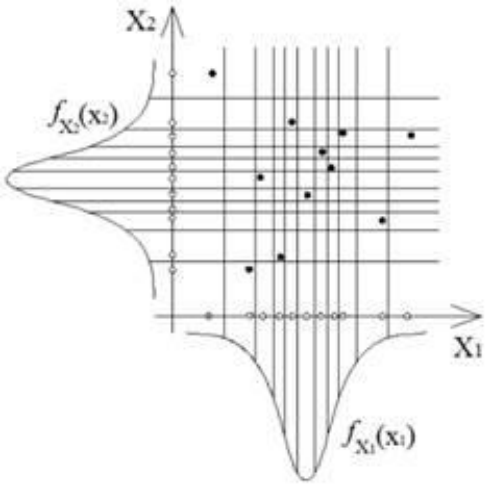


Figure 10 - Latin Hypercube Sampling Concept from [19]

4.2.Optimization

4.2.1. Measures of model performance

The coefficient of determination, R_2, is a regression score metric used as a measure of prediction accuracy. The best possible score is 1.0, and depending on how it is calculated, it can return negative values or 0 as the lowest score. In this project, the R_2 is calculated using:

$$R^2 = 1 - \mathrm{SSE}/\mathrm{SST} = 1 - \frac{\Sigma(y - \hat{y})^2}{\Sigma(y - \tilde{y})^2}$$

Where $\hat{y}$ and y are the predicted and true output values, respectively. $\tilde{y}$ is calculated as the mean of the true output values. When using this general form of calculating the coefficient of determination, it follows that it can have negative values. A constant model that always predicts the expected value of y, disregarding the input feature would get an R_2 score of 0.0.

R^2 can be considered a measure of explanatory power, (not necessarily model fit). High values indicate that the regression model has statistically significant explanatory power. The measure can also be viewed as the percentage of the response variable variation that is explained by a linear model, and 1) 0 % indicates that the model explains none of the variability of the response data arounds its mean and 2) 100 % indicates that the model explains all the variability of the response data around its mean.
However, while R-squared provides an estimate of the strength of the relationship between your model and the response variable, it does not provide a formal hypothesis test for this relationship. The F-test of overall significance determines whether this relationship is statistically significant. Low R^2 values are not always bad, and high R^2 values are not always good [20], but when comparing models and features included, one could use the R^2 score as an indicator of whether the models worsens or improves (increased value of R^2).

Mean Squared Error (MSE) is a common quality measure an estimator. It measures the average of the squares of the prediction errors (see equation XX). Here $\tilde{y}$ being the predicted values, and y being the vectors of true values.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

The best regression model in this project is the one that minimizes the function SSE while also optimizing the coefficient of determination, R^2.
In general, a model fits the data well if the differences between the observed values and the models predicted values are small and unbiased. Before one looks at the statistical measures for goodness-of-fit, the residual plots should be inspected [21]. Residual plots can reveal unwanted residual patterns that might indicate biased results more effectively than numbers.

4.2.2. Feature engineering

Feature engineering is a significant part in creation of intelligent systems. As illustrated in Figure 11, the engineering/creation of new features is an essential part of model development. Even though there exist good methodologies for (automated) machine learning, each problem is domain specific and will improve performance if given better features (suited to the

problem task). It is a chance for introducing human/domain knowledge into a model, to make the model more robust and thus perform better on unseen data.

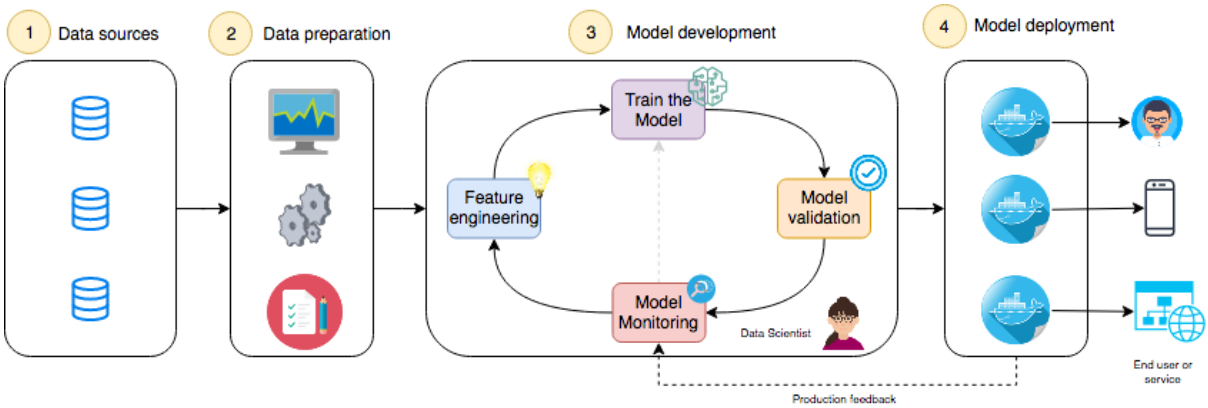


Figure 11 - Model development pipeline, the role of Feature Engineering from [22]

Feature engineering is improving the model by adding new features to the dataset, containing information that is expected to introduce new and relevant knowledge to the problem to be solved. For an extensive elaboration on practical feature engineering, see [23]

4.2.3. Feature selection

Feature selection is the process of reducing the number of input variables when developing a predictive model. It is desirable to reduce the number of features in a model, mainly for computational costs, but in some cases, it might also improve predictive performance. The goal is to reduce the number of features such that the remaining ones are the most relevant for prediction of the target variables. Some predictive modelling methods include a large number of variable inputs. This can slow down the development and training process, and also require larger system memory capacities. Feature selection can also be related to dimensionality reduction techniques since both methods result in fewer components for the model to be trained on, and thus increasing the computational efficiency. The difference here, however, is that dimensionality reduction is disregarding the features not considered relevant for the predictive mapping, whereas dimensionality reduction is creating a projection of the input data, resulting in entirely new input features.

5. Materials

This section contains information about the working environment setup and necessary packages for implementations of the models described in Chapter 6 Methods. It will also include a brief description of why these libraries were chosen. Full access to all files and implementation (including test scripts) can be found in the open GitHub repository [4].

5.1. Working environment

Some issues were encountered regarding packages and integration of NEST (see 5.3.1) and PyPet (see 5.3.5), that was discovered when attempting to hierarchically store simulation output/results in Pandas DataFrames handled by PyPet.
To resolve these issues in the environment setup, it was necessary to make some minor changes in the source code of the PyPet package. It issue was related to mishandling of data structures when using Pandas Dataframes from the NEST simulation output and followingly store it in the trajectory (hierarchical structure).
A change to one of the source files was necessary:

In the following file: “lib/python3.6/**site-packages/pypet/storageservice.py**
The following lines but be inactivated by commenting them out or deleting them:

Table 1 - Solving issue with package integration, Lines to inactivate

Line number	4186	4201	4202

After installing the packages/ libraries above and commenting out the lines mentioned, the environment is setup and ready.

PyPet requirements:
Pytables >= 3.1.1
Pandas >= 0.23.0
HDF5 >=1.8.9
Numpy >= 1.13.0

5.2. Modules in project

A complete list of project requirements can be found in the Appendix. An overview can be seen in table below:

Table 2 - Overview of important project modules

1	pandas==1.0.1
2	matplotlib==3.1.3
3	scikit_fuzzy==0.4.2
4	numpy==1.18.1
5	featuretools==0.16.0
6	scikit_learn==0.23.2
7	matplotlib==3.1.3
8	pypet==0.4.3
9	neo==0.8.0
10	smt==0.3.4
11	elephant==0.6.4
12	quantities==0.12.4
13	nest==1.3.0

5.3.Description of packages (external)

5.3.1. NEST – Network simulation tool

NEST [24] is a simulation tool that allows for exploration and understanding of biologically realistic neural networks. It is a high-performance neuronal network simulator that is used for diverse applications in computational neuroscience. Some key features include built-in methods for creating neurons, connecting them and assessing activities using measuring tools. NEST offers a selection of 50 different neuron models, 10 synapse models, and has minimal dependencies (only requires a C++ compiler). The implementation of NEST in this project is based on the work done in [8], where a part of the goal was to verify that the implementation could reproduce results described in Brunel's [3] paper on exploration of network states.

The network design specifications described in Brunel's model A are also used here, except the parameters tuned within the given parameter space (belonging to the AI-state) to create a dataset for further exploration of the metamodelling processes described in section 6.5

Metamodelling procedures. For an overview of parameters used in the NEST network simulation, see section 6.2.1 Simulations using NEST. Running a simulation using these parameters and specifications results in an overview of recorded spikes from a given number of neurons in the network. These spike-patterns serves as a basis for the creation of statistical measures used as targets/responses (Y) in the modelling procedure.

5.3.2. Featuretools (Deep Feature Synthesis)

Feature engineering can be crucial in many machine learning projects but can be difficult and time consuming if one is not deeply familiar with the data and domain. Featuretools [25] is a tool for automated feature engineering (see section 4.2.2 “Feature engineering”), by using Deep Feature Synthesis (DFS) on the given data. It can also handle temporal and relational datasets, and thus transforms it into feature matrices for machine learning. This allows for an automated process of creating new features, which is generalized and fits all datasets with equal structures (samples as rows and features as columns). It comes with a range of different options for aggregation of existing regressors, but one can easily create and add tailored functions, called primitives, (for example if adding domain knowledge-based criteria and filters) if necessary.

The features created by DFS are produced based on Feature Primitives, the building blocks of Featuretools. They define the individual computations that are applied to the raw data given as input to creates new features, and they are separated into two types; aggregations and transformations. The transformations (used in section 6.4.1 Transforming Features) are applied to columns in a table/dataset, whereas aggregations are applied across multiple tables with a defined relationship (parent/child relationship defined).
By breaking common feature engineering calculations into their primitive components, one is able to capture underlying structures of the features that human creates when doing feature engineering “manually”. Thus, by using this automated feature calculations/engineering one can include domain knowledge (see section 4.2.2 Feature engineering) into the model, without extensive work.
Featuretools provides a range of created primitives to choose from, but custom primitives are easily created and added if necessary. For an example on how to implement this, see Appendix F, where DFS is used for applying feature engineering to the dataset.

5.3.3. Scikit Learn – PLSR Regression

Scikit Learn [26] provides a class for PLSRegression; PLS2 for block regression and PLS1 in case of one-dimensional response. This class implements[1] the NIPALS[2] algorithm, and the method scales the input data (Model Parameter 'Scale' = True by default). In this model design (see section 6.6 Metamodelling Procedures), the global and local modelling is done using SkLearn PLSRegression.

5.3.4. SkFuzz – Fuzzy clustering and prediction

The SciKit-Fuzzy [27] library is a fuzzy logic toolbox for SciPy[3], written in python programming language. The fuzzy logic principles work by assigning (cluster) membership values to all samples in a multidimensional dataset. The membership value for a sample is calculated based on the similaritiy with each cluster center, and all samples has a sum of 100% cluster belonging in total. This is a soft clustering technique more elaborated in section 4.1.7 "Fuzzy C Clustering". The module creates cluster centers using training data in the skfuzzy.cmeans()-method, and followingly predicts cluster belongings to new and unseen data by skfuzzy.cmeans_predict(). The Fuzzy Partition Coefficient (FPC) is defined on the range [0,1], and describes the separability between the clusters, i.e. how cleanly the data is being described by a certain model. When FPC is maximized, the data is optimally described by the fuzzy clustering algorithm. To illustrate the FPC coefficient, see Figure 12 - Illustration of FPC from [27] below, where one clearly sees how the coefficient is at its highest when the clusters are separated as expected.

1 https://github.com/scikit-learn/scikit-learn/blob/0fb307bf3/sklearn/cross_decomposition/_pls.py#L266

2 https://cran.r-project.org/web/packages/nipals/vignettes/nipals_algorithm.html

3 https://www.scipy.org/

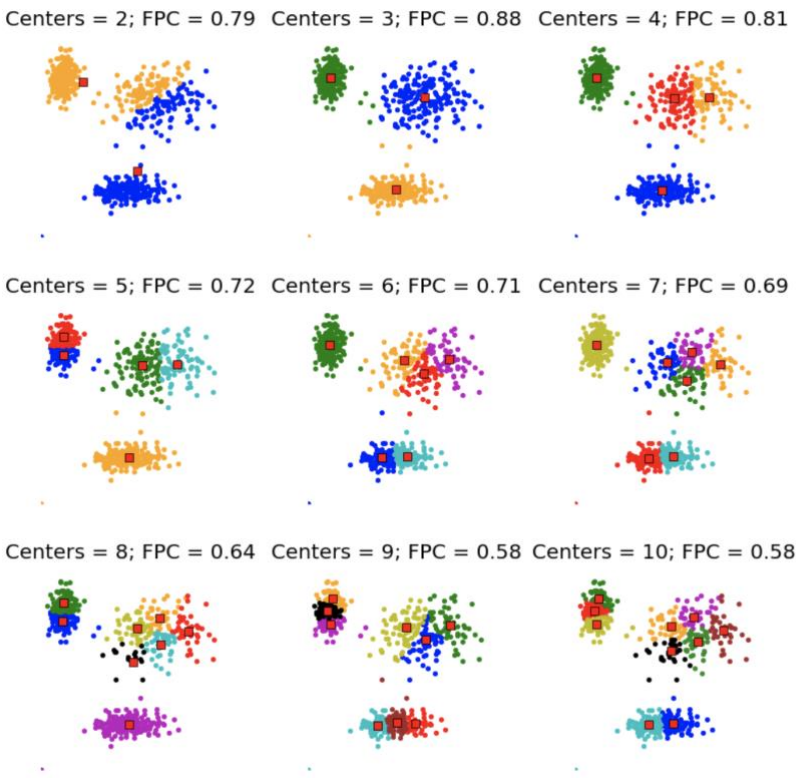


Figure 12 - Illustration of FPC from [27]

5.3.5. PyPet – Hierarchical organization of work

PyPet [28] is a python parameter exploration toolkit that provides a functional framework that manages exploration of a defined parameter space by running numerical simulations and storing the resulting data in a hierarchical manner. PyPet creates a trajectory containing all the data, and stores it as a hierarchical HDF5 file, which is easily accessible to read and navigate in using an HDF-file reader

It also organizes the resulting data (simulation outputs) together with the corresponding input-parameters, so that one easily can have full access to historical runs and parameters used. In addition to this, supplementary information can also easily be added to the hierarchy, such as extra summaries, calculations, statistical measures, textual descriptions and notes. It also allows for adding upcoming (new) simulations and parameter inputs, thus is a convenient tool if continuing work is to be done on the project.

Running an experiment in PyPet happens in an “Environment”, that works as the logic controller for filling parameters and results in the correct manner. The advantage here is that

the environment takes care of issues like logging (useful in cases of interrupted experiments), multiprocessing logistics, memory capacities, and other logistic matters. It also provides a direct integration with Git, offering a convenient way of storing backups and work progress. For explanation of the use of PyPet Trajectories in this project see Appendix H.

5.3.6. Latin Hypercube Sampling from SMT

The Latin Hypercube Sampling (LHS) [29] is a widely used method to generate controlled random samples [30], and is a way of generating random samples from a defined parameter space. The SMT (Surrogate Modelling Toolbox) has a library for sampling methods, and herein is the LHS method implementation. Five criteria for the construction of LHS are implemented in SMT[4] :

1. Center the points within the sampling intervals
2. Maximize the minimum distance between points and place the point in a randomized location within its interval
3. Maximize the minimum distance between points and center the point within its interval
4. Minimize the maximum correlation coefficient
5. Optimize the design using the Enhanced Stochastic Evolutionary algorithm (ESE)

5.3.7. Elephant – Adding Statistics

Elephant (Electrophysiology Analysis Toolkit) [31] is a library for analysis of electrophysiological data. It offers generic analysis functions for spike train data and time series recordings from electrodes (such as Local Field Potentials, LFP, of intracellular voltages). It is used in this project because it provides a consistent analysis framework that is built in a modular and manageable way. It is specialized in handling spike trains, the foundation of the data generated in section (data generation).

4 https://smt.readthedocs.io/en/latest/_src_docs/sampling_methods/lhs.html

5.4. Implementation design choices

It is hard to know in advance exactly what constellation of modelling options that will result in the better performance results. Some sort of "trial and error"-way of working is often required, but to reduce the amount of tedious and cumbersome work is always to be preferred. It was therefore, as mentioned, a sub goal to structure the implementation in such a hierarchical way, so that a "general" model produces the benchmark for the dataset and the following options of architectures could be used strategically on the search for better performing model variations. Sometimes, the best of a few model options might pinpoint direction of which type of modelling strategy that works for the given dataset, and one can start tune and prune the better performing model to be further improved. The benchmark model (the main model) is the root of a tree with many branches representing options for model variations to test.

This can be a good way of creating modelling architectures, that encourages for different fields to also use and explore the possibilities of modelling without having to be a Data Scientist (or similar). To create standard methodologies that can be used by many scientists interested in modelling, without knowing exactly how to explore the infinite varieties of modelling strategies that can be applied, invites to cross-disciplinary projects and facilitates collaboration across fields of sciences.

6. Methods

6.1. Overall metamodelling pipeline

When data was formed and available by preprocessing and preparation, the strategy (of HC-PLSR [2]) was to model this data using a global model, then clustering the resulting model output (scores) into smaller segments, and followingly creating local regression models for each clustered batch of data.

The hypothesis is that this way of clustering the data will account for some of the non-linearity in the data, that one single regression model cannot capture alone. Several ways of combining different clustering methods and regression models was tested (further described in section 6.6). In general, the PLSR was used as a regression method, because it removes the issue of multicollinearity in the data, and also gives the option of modelling using only some of the principal components. Clustering is introduced as an attempt of handling non-

linearities in the data that one PLSR model cannot account for alone. Two different clustering methods were tested, the K-Means (see section 4.1.7) and the Fuzzy C Means (see section 4.1.8). Clustering was intended to pick up patterns from the scores created by a global PLSR model, and thus labelling and grouping together samples with more in common. Local PLSR models for each cluster was then created, based on the belonging samples.
Predictions were made by projecting new samples onto the new feature space defines by the PC's of the global model, and then a) predicting using the local PLSR model belonging to the cluster the sample was labeled with b) calculate a weighted sum of all prediction made by the local models, where the Fuzzy-C cluster probability serves as weights for the regression coefficients.

6.2.Data generation implementation. (network creation, simulation, storing)

The dataset is generated by using NEST (see section 5.3.1 NEST – Network simulation tool) for simulation of a neural network as described by Brunel [3] (mentioned in section 4.1.4 Neural networks). The simulation output results are one spike pattern/spike train for each recorded neuron, thus creating a matrix of all (recorded) neurons and their spikes at any given time (see Figure 5 for example). These spiking patterns (spike trains) form the basis for creation of response variables (see section 6.4.2 Adding statistics /spike train summaries using elephant), by calculating statistical measures described in section.
The characteristics of the spike trains change when varying the simulation input parameters g, j, eta and D [3].Thus, in order to simulate different outcomes and tendencies in the spike patterns, the parameter space was sampled randomly using LHS (see 4.1.11 Latin Hypercube Sampling from SMT). The four parameters were varied within the parameter ranges belonging to the AI-state of network activity. The AI-region of network behavior is the most common state to be measured in the cortex, and it was therefore decided to begin the model exploring on this parameter region. When a model created on this parameter range is sufficiently good, one might include wider parameter ranges for incorporating different network tendencies to the model.

6.2.1. Simulations using NEST

The network (Model A) described by Brunel was simulated and tested to verify that the input settings and the simulation outputs were consistent with the resulting states described in [3].

The tool used for creating and running the network simulation was, as mentioned previously, the NEST simulation toolbox. Description of how to implement and test the network design and simulation can be found in [8]. Parameter and variable settings are specified in the table below (see Table 3), and the script for creation of the network is included in Appendix K. Parameters marked with [*] are default values from Brunel (Model A) [3].

Table 3 – Table of parameters for simulation of Brunel Network Model

Symbol	**Value**	**Explanation**
dt	0.1	Simulation resolution in NEST, and bin size for histogram plots
simtime	1100 ms	Duration of the experiment simulation in [ms]
simlation Cutoff	100 ms	
D	Interval [1.0, 2.5]	Transmission delay, axonal propagation as a time delay. Represents Synapse delay between neurons, in [ms]
J	Interval [0.05, 0.4]	EPSP (Excitatory post synaptic potential) amplitude. Synapse weight between neurons.
eta	Interval [1.5, 3.0]	$\eta = \nu_{ext}/\nu_{thr}$
g	Interval [4.5, 6.0]	Relative strength of inhibitory synapses. Inhibitory synaptic strength, relative to excitatory synaptic strength.
V_ext		Frequency of external input
Epsilon [*]	0.1	Connection probability. Excitatory Neurons * epsilon = number of synapses per neuron
Order	2500	Defining relation between excitatory and inhibitory neurons. Relative number of neurons in network.
N_rec	50	Number of neuron to be recorded durng simulation
Num_threads	10	Number of cores used for processing. Simulation in threads for parallelizing.
print_report	True	Simulation variable, if set True; prints progress as output throughout simulation.

input_stop	False	If > 0, stops input signals after the given ms has passed
Ne [*]	10 000	Number of excitatory neurons
Ni [*]	2 500	Number of inhibitory neurons
Ce [*]	1 000	Number of excitatory connections per neuron
Ci [*]	250	Number of inhibitory connections per neuron

6.2.2. Sampled feature space

As previously mentioned, the varied simulation inputs were parameters g, j, eta and D (see section network desc), and the intervals sampled from is seen in Table 3 above. In order not to introduce bias in the model by introducing imbalanced data, the parameters were sampled evenly using the LHS sampling method (see 4.1.11 Latin Hypercube Sampling) with 500 sampling points (see script in appendix L). This also results in being able to ignore the possibility for outliers, since the data was created from deterministic modelling where features (the parameters) has been selected and created intentionally.

6.3. Data Handling

To attain the dataset, it was required to run several simulations. All simulation runs were using different combinations of parameters. It was a priority to handle this part of the modelling process in a functional way that transfers well across simulation tools, parameters to tune etc. This because new universal methodologies for creating models can widen the reach of use of such modelling methods and make it more accessible for other scientists to benefit from its advantages.

Accessing data via natural naming and grouping the data into meaningful categories and support for many different data formats are some of the most attractive features for this type of data generation and modelling. Therefore, the PyPet parameter exploration toolkit was a convenient tool for creating and storing the resulting data. Data formats that PyPet supports include python natives, lists, tuples, dictionaries, Numpy arrays and matrices, Scipy sparse matrices, Pandas Series, DataFrames and Panels (and BRIAN2 quantities and monitors). All of the mentioned datatypes happen to be used in this project, and PyPet allows/aims for it all

to be seamlessly combined into a convenient workflow. After simulation results were stored (together with the corresponding simulation parameters), the calculation of additional statistics (correlation of variation, covariance matrices, correlation matrices and fanofactor, see section 6.3.2) were inserted to a Pandas DataFrame and saved with the corresponding simulation run. The dataset used for further modelling where extracted from the trajectory as a simple csv-file. Using a dataset from a csv-file for modelling is the norm in Data Science, and thus was chosen as a natural choice for data flow for generalization purposes. The dataset (csv format) contains 500 simulations (rows), 4 simulation parameters (features, predictors) [g, j, eta, D] and 4 response variables (targets, Y), thus a 500*8-matrix. Both the trajectory file (HDF5 format), the resulting csv-file and the scripts for generating them can be found in the open project Github repository [4] and in the Appendix.

6.4.Data preprocessing

6.4.1. Transforming Features

As a part of comparing performance of different modelling strategies, an option to improve modelling performance for linear regression models is to include interaction terms and higher order terms to the regressors (X). See a discussion of collinearity and interaction terms section (4.1.10). This was done using the Python library Featuretools and the functionality Deep Feature Synthesis (see section5.3.2). To continue the work in a direction that generalizes well, a selection of common aggregations was used when creating the additional terms. Primitives included in this model were:
[add_numeric, multiply_numeric, logarithm, square_root, squared, cube]. The name Deep Feature Synthesis comes from the methods ability to stack these primitives to in turn generate more complex features (example multiply (log(a) + log(b))). Each time primitives are stacked, the “depth” of a feature is increased. This is controlled by the `max_depth` parameter, and it controls the maximum dept of the features returned by DFS. For some experimentation in the modelling process, some of the new features were excluded, but the widest set of new features was created using `max_depth=1`:

Original features:
['g', 'eta', 'J', 'D']

DFS features (new features, cross terms):

['g', 'eta', 'J', 'D', 'eta + J', 'D + g', 'D + eta', 'D + J', 'g + J',
'eta + g', 'eta * J', 'D * g', 'D * eta', 'D * J', 'g * J', 'eta * g',
'LOG(eta)', 'LOG(J)', 'LOG(g)', 'LOG(D)', 'SQUARE_ROOT(eta)',
'SQUARE_ROOT(J)', 'SQUARE_ROOT(g)', 'SQUARE_ROOT(D)', 'SQUARE(eta)',
'SQUARE(J)', 'SQUARE(g)', 'SQUARE(D)', 'CUBE(eta)', 'CUBE(J)',
'CUBE(g)', 'CUBE(D)']

The implementation of DFS and transformation of features is added in Appendix F and also available in the open project Github-repository [4].

6.4.2. Adding statistics /spike train summaries using elephant

When creating this metamodel, it has been an angle of approach to try to map the network spiking tendencies by (only) assessing the input parameters, and their resulting spiking state measures. Elephant (5.3.7), is an open source generic tool that provides analytic functions for spike train data and time series recordings, as well as statistics especially for spike trains.
A variety of different descriptive measurements/summaries of the spike patterns were computed, using this package (Appendix M + E).
Statistics used as targets for this modelling were summaries like:

Table 4 - Creating of Statistical measures overview

Type	**Shape**	**Comment**
Fanofactor	50x1 list	
Coefficient of Variation [CV]	50x1 list	
Coefficient of Correlation [CCorr]	50x50 matrix	Calculated the mean of the triangular matrix, thus a summary of the matrix becomes a 50x1-shaped list
Coefficient of Covariance [CCov]	50x50 matrix	Calculated the mean of the triangular matrix, thus a summary of the matrix becomes a 50x1-shaped list

6.4.3. Standardizing/scaling

The idea of scaling is to make the models more robust to analysis on feature spaces. Some algorithms, like PCA and KNN, are sensitive to the metric spaces, will be more weighted towards features with higher numbers (i.e. towards 5000g instead of 5 kg, even though they represent the same), and this is where scaling becomes handy. Scaling does not affect the significance of features; in contrast it improves analysis of data.
In this project, the standard scaling was done on the input data before creating PLSR-model (by using the internal scaler from Scikit Learn), and on the input data before creation of clusters (i.e. on the scores from the global PLSR model, see section 6.6 "Metamodelling Procedures")

6.5. Data inspection

The data is a result of deterministic modelling, and therefore; outliers and missing values are not present in the dataset. Below is the dataset visualized in two figures, firstly the original X and Y, and in the second figure, the first 15 scores are plotted against each other (X_{scores} vs. Y_{scores}). As seen, the cov_mean variable is fairly close to zero and might not contribute with much in the modelling. However, it was decided to be included, as it is expected to change characteristics when the parameter ranges are altered to contain values corresponding to the other network states (synchronous regular states [3]).

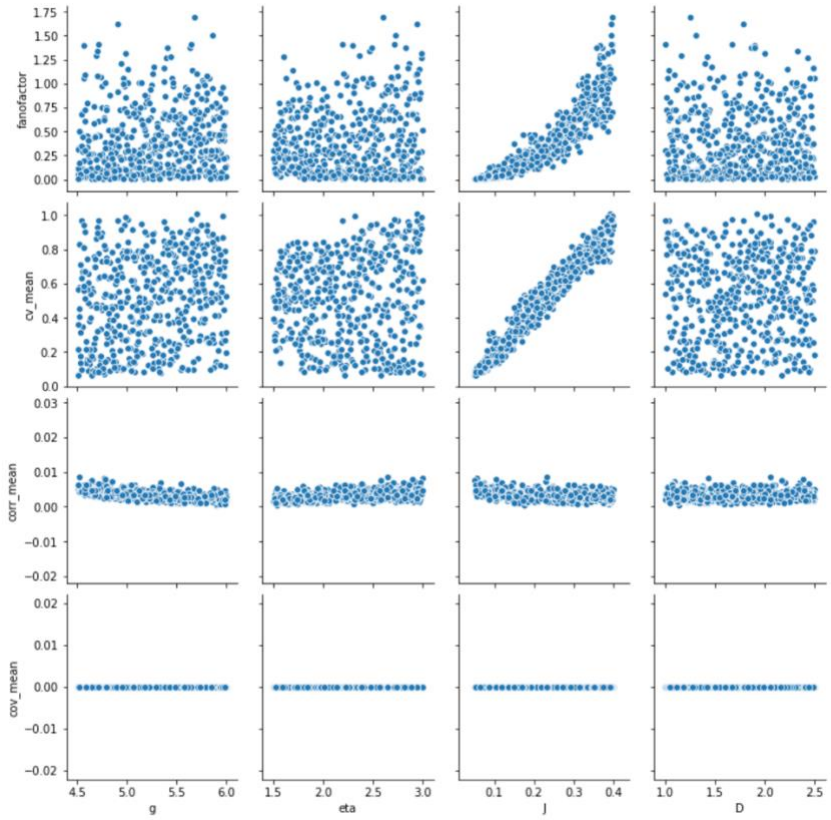


Figure 13 - Data inspection, X vs Y

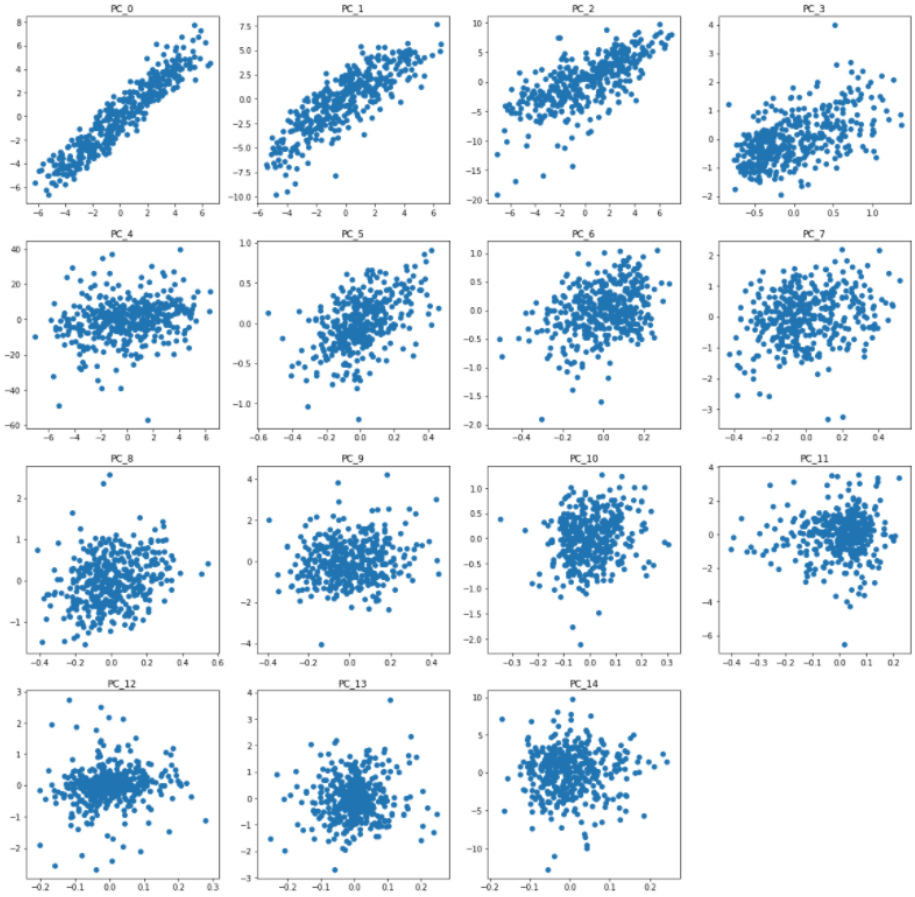


Figure 14 - Data inspection, X_scores vs Y_scores

6.6.Metamodelling Procedures

Two objectives were explicitly prioritized when designing the implementation and framework for this model. Firstly, it was a goal to implement a method that allows for different varieties of data, with equal structure (n samples * k features). In addition to this, the performance of models varies a lot when the dataset/type of data is altered. Therefore, as a second quality of this project was to prepare for exploration of different architecture-types /meta modelling design, to easily explore what type of structure design that optimized performance for the given dataset at hand.

Optimization of a metamodel is not a defined concept, since it is highly dependent on the type of data the model is used on. Thus, a selection of tested strategies and some further development on these ideas were selected. When that is stated, a list of recommendations for further exploration has been formed (section 0

Further work) that is especially suitable for low effort implementations (since already considered in creation of this project).

- Optimization techniques for the metamodel was in this project, the exploration of;
- different number of added features (cross and higher order terms)
- different clustering/classifier methods for clustering the scores
- Clustering on X scores vs clustering on y scores
- “Weighted Sum Prediction” vs. “Best Cluster Prediction”
- Parameter tuning for the clustering methods
- Outlier restrictions for cluster models and local models

Both the global and local PLSR models contains the optimal number of principal components, and this was determined based on the lowest MSE of the predictions of the response matrix Y, calculated by using cross-validation. A 10-fold cross-validation was used, where data was split randomly into ten segments. Then, 10 PLSR models were then created based on 9/10 segments and prediction was made by using the last segment.
The clustering method splits up the output space defined by the global model, i.e. clustering is then done on the scores from the fitted PLSR model. The original samples are then labeled according to the cluster belonging, and for each of the cluster labels, a local PLSR model is created (based on the samples for that cluster label). See figure below for illustration.

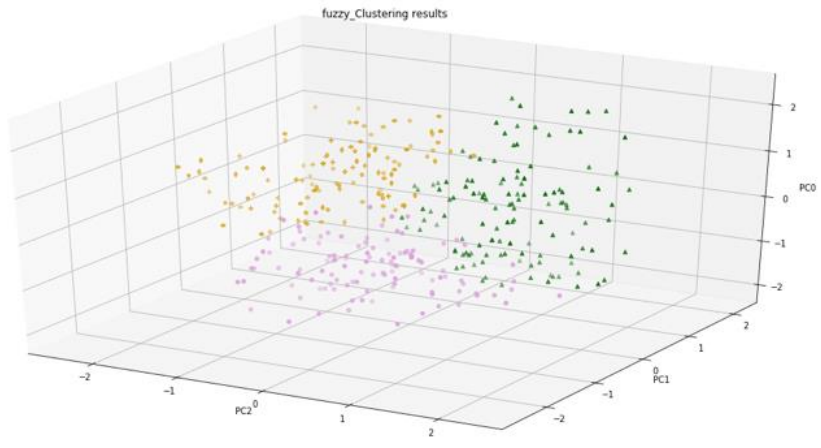


Figure 15 - Illustration of 3 PCs with cluster labels (colored green, yellow and pink)

Clustering of the scores were consequently always done on the first 3 components. The PLSR methods from Scikit Learn has a build in scaler that standardizes the data before fitting the model and turning this of worsens the resulting prediction. Thus, it was left “on”.

Standardization of input data for the PLSR models improved modelling performance, standard scaling of the PCs before clustering did also improve prediction accuracy.
This was implemented by 1) using the built in scaler (default=True) from SkLearn's PLSRegression method, and 2) using SkLearn's StandardScaler() for scaling the principal components before training the clustering algorithm. It should be noted that scaling for the PLS models happens after the DFS creates new transformed features, thus the new features are created on original regressors (X) and not on the scaled ones like done in [2].

In an attempt of improving performance, a plan to optimize the models was outlined. The approach resulted in a "modelling variations"-scheme, where the main steps of the HC-PLSR has tuning options defined in Table 5 - Model Variations for all steps in the meta model pipeline.
All these model variations were compared to the performance of a single PLSR, and the result is described in an overview table (section 7.3)

Table 5 - Model Variations for all steps in the meta model pipeline

Step 1	**Step 2**	**Step 3**	**Step 4**	
Global model	Clustering	Local models	Prediction	Comments
Original features	Algorithm		Weighted sum	
Added cross combinations and higher order terms	Outlier restrictions	Added cross combinations and higher order terms	Best cluster prediction	
	scores to cluster	Limit of minimum number of samples I a local model	Weighted sum	
		Limit (lower threshold) of cluster belonging (if soft clustering)	The N most relevant clusters form a weighted sum prediction	

Assessing and tuning of the model combinations described above were carried out in a standardized way. The hyperparameters tuned for each model variations are described in Table 6 - Hyperparameter tuning overview . The procedure of tuning was equal for all meta model variations, and the best performing hyperparameter combination were noted as the result in Table 11 - Model options and best performing model result.

Table 6 - Hyperparameter tuning overview

Global model	Clustering	Local models	Comments
CV	Fuzzy C Means: Fuzzifier	CV	
Number of principal components	Number of clusters	Number of Components	
Test size for dataset	HDBScan: Distance Minimum number of samples in cluster	Outliers excluded from local modelling	

7. Results

This section describes the resulting model architecture and performance of a selection of the tested model variants. The metamodelling descriptions are in turn split into 3 different varieties with interesting resulting performances. In general, clustering on X-scores of the global PLSR model yielded the better results for “weighted sum” prediction and for the “best cluster” prediction. However, it looks like clustering of the Y-scores were able to split/separate the datapoints into more “natural”-looking clusters This is illustrated below;

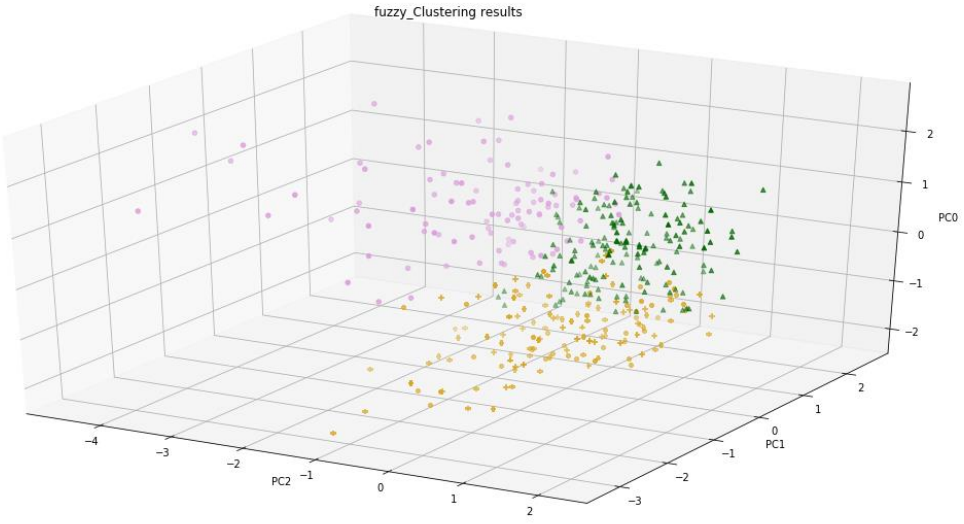


Figure 16 - Visualization of clustering on the Y_scores

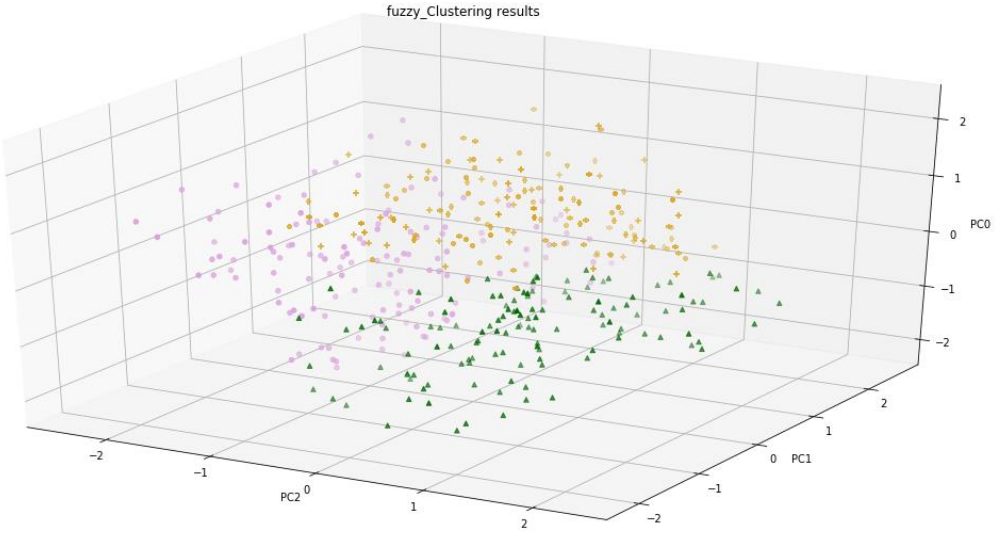


Figure 17 - Visualization of clustering of the X_scores

A variation of the metamodel architecture was tested (Called Version 2), where the global model is trained on the original features, and the local models are polynomial (added interaction terms). This variation gave the best improvement in prediction accuracy, compared to the signle PLSR model for comparison.

Since clustering on the Y_{scores} requires the global model to predict the target before it is transformed to the new feature space (spanned by the PCs), two global models were created; one polynomial for high prediction accuracy, and a second based on the original features that

is used for the projection of the datapoints before clustering. This gave a slight improvement in the prediction results.

7.1.Data preparation

Feature engineering as described in 6.4 Data preprocessing, resulted in the following feature transformations (cross and higher order terms):

Original features:

['g', 'eta', 'J', 'D']

DFS features (new features, cross terms):

['g', 'eta', 'J', 'D', 'eta + J', 'D + g', 'D + eta', 'D + J', 'g + J',
'eta + g', 'eta * J', 'D * g', 'D * eta', 'D * J', 'g * J', 'eta * g',
'LOG(eta)', 'LOG(J)', 'LOG(g)', 'LOG(D)', 'SQUARE_ROOT(eta)',
'SQUARE_ROOT(J)', 'SQUARE_ROOT(g)', 'SQUARE_ROOT(D)', 'SQUARE(eta)',
'SQUARE(J)', 'SQUARE(g)', 'SQUARE(D)', 'CUBE(eta)', 'CUBE(J)',
'CUBE(g)', 'CUBE(D)']

This is a result of `max_depth=1` in the DFS tool. The new dataset (with the transformed features) were then the input of the PLSR-optimizer function (see Appendix D), where the optimal number of PCs where included in the resulting PLSR models (used for creation of both global and local PLSR models).

In creation of response variables (Y variables), statistical calculations of the spike train matrices (NEST simulation results, section5.3.1) were created as described in (section 6.1.1 and 6.3.2). This resulted in a 4-dimensional response surface consisting of the measures: *fanofactor, coefficient of variation, coefficient of correlation, coefficient of covariance*. Since coefficient of variation, coefficient of correlation and coefficient of covariance are calculations for each neuron, a summary (mean, median and maximum value) were tested. The final dataset however, used *mean* as a statistical summary. When using median and maximum value, predictive performance for the methods dropped drastically.

7.2.Combined Model Variations

Here are some of the resulting model variations outlined and explained. For a complete overview, all model variations (and the resulting best performing version) that were tested are included in Table 11 - Model options and best performing model result. The "split" represents the train-test-split coefficient, indicating the relative (percent) amount of the total dataset used for testing. (e.g. split=0.2 indicated that 20 percent of the dataset was used as "unseen data" for model performance assessment).

7.2.1. PLSR original terms

Model created for comparison. No features added.

Split 0.3
R2 = 0.885
MSE = 0.07
MAE = 0.037

Split 0.4
R2 = 0.879
MSE = 0.009
MAE = 0.040

7.2.2. PLSR Polynomial

Model created for comparison. New features from DFS (see section) added.

Split 0.3
R2 = 0.914
MSE = 0.004
MAE = 0.028

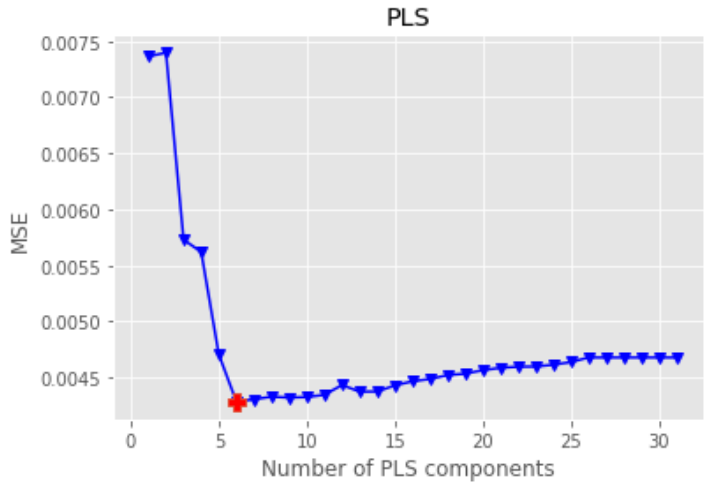


Figure 18 - PLSR optimizer function of optimal number of components

7.2.3. Variation a

HC-PLSR model variation *(a)* (Appendix A), where the clustering model is created on Y-scores from the global PLSR model. The PLSR models (both global and local) are including the transformed terms and components included in the models are chosen based on the optimize-function in script Appendix D.
Model specifications are stated in the table **Error! Reference source not found.**.

Table 7 - model specifications for HC-PLSR variation (a)

Split	0.3
Number of clusters	3
Fuzzifier coefficient	8
Scores	Y scores

In the following 3 figures, the clustering performance and the overall model performance is illustrated.

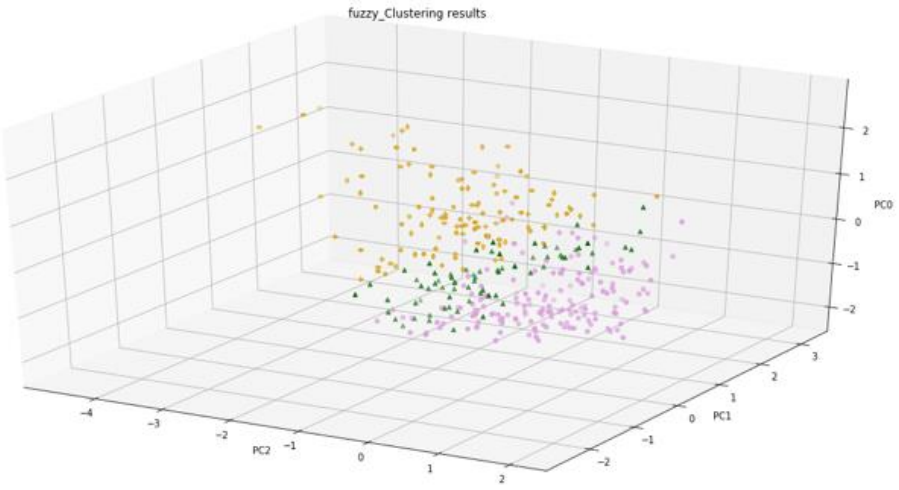


Figure 19 – Clustering result on Scores, for variation (a)

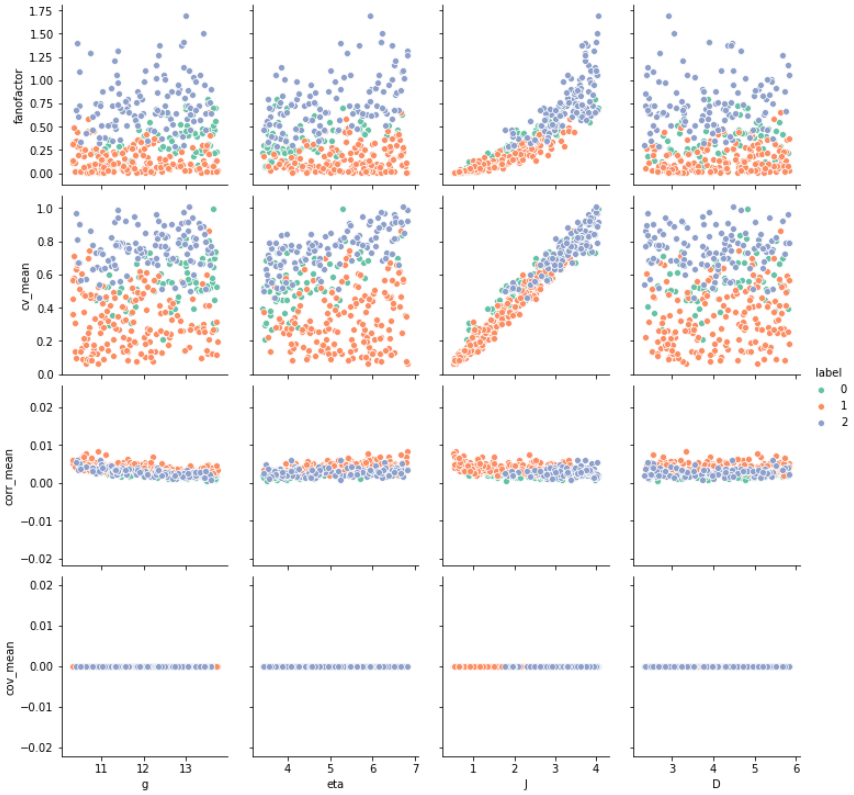


Figure 20 - Clustering results on original data, for variation (a)

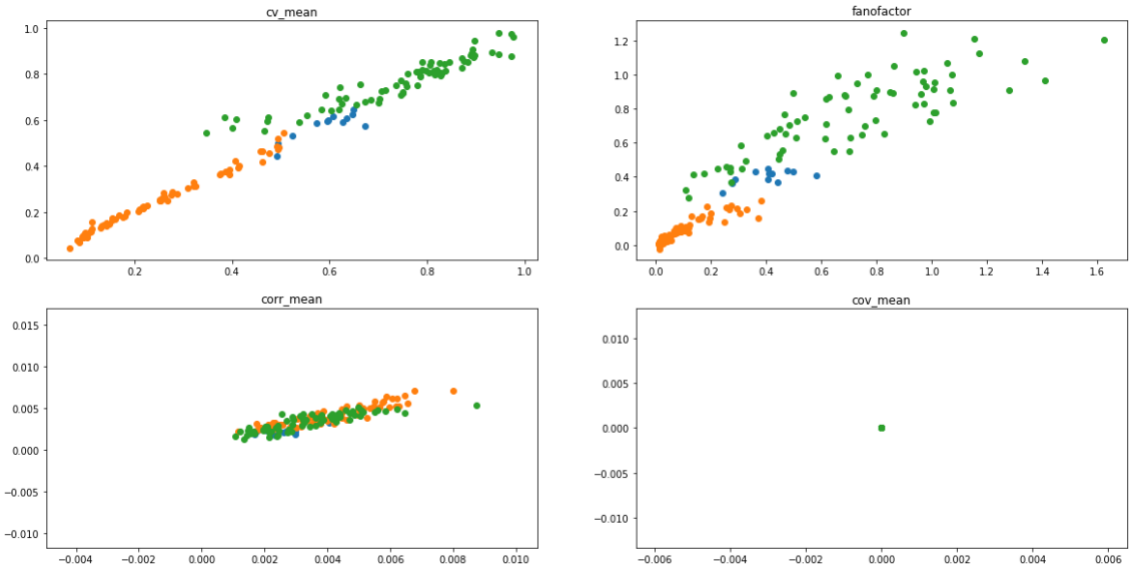


Figure 21 - Y_predicted vs Y_true plot, predictions from each cluster in different color. For variation (a)

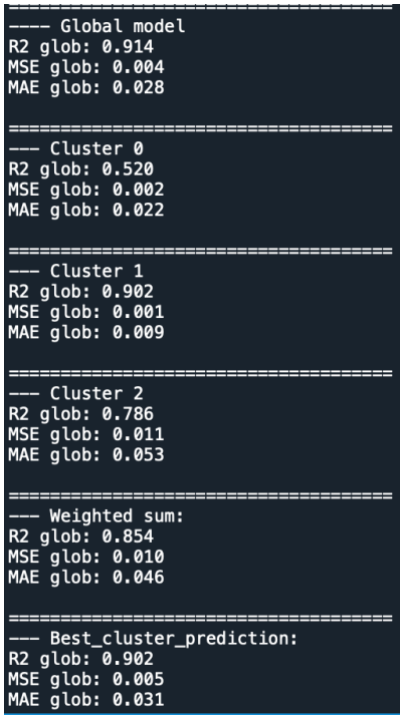


Figure 22 - Performance of model variation (a)

7.2.4. Variation b

HC-PLSR model variation *(b)* (Appendix A), where the clustering model is created on X-scores from the global PLSR model. The PLSR models (both global and local) are including the transformed terms and components included in the models are chosen based on the optimize-function in script Appendix D.

Model specifications are stated in the table **Error! Reference source not found.**.

Table 8 - model specifications for HC-PLSR variation (b)

Split	0.3
Number of clusters	3
Fuzzifier coefficient	1.1
Scores	X scores

In the following 3 figures, the clustering performance and the overall model performance is illustrated.

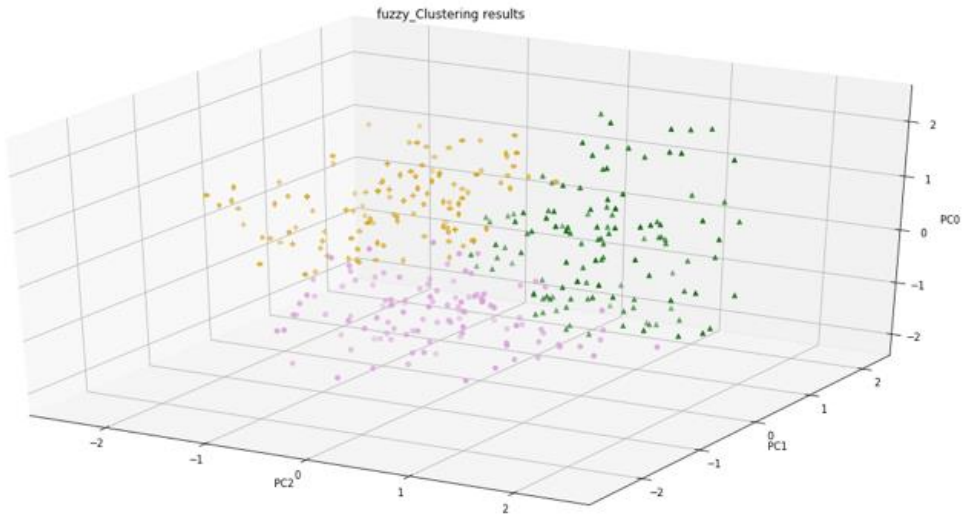


Figure 23 - Clustering result on Scores, for variation (b)

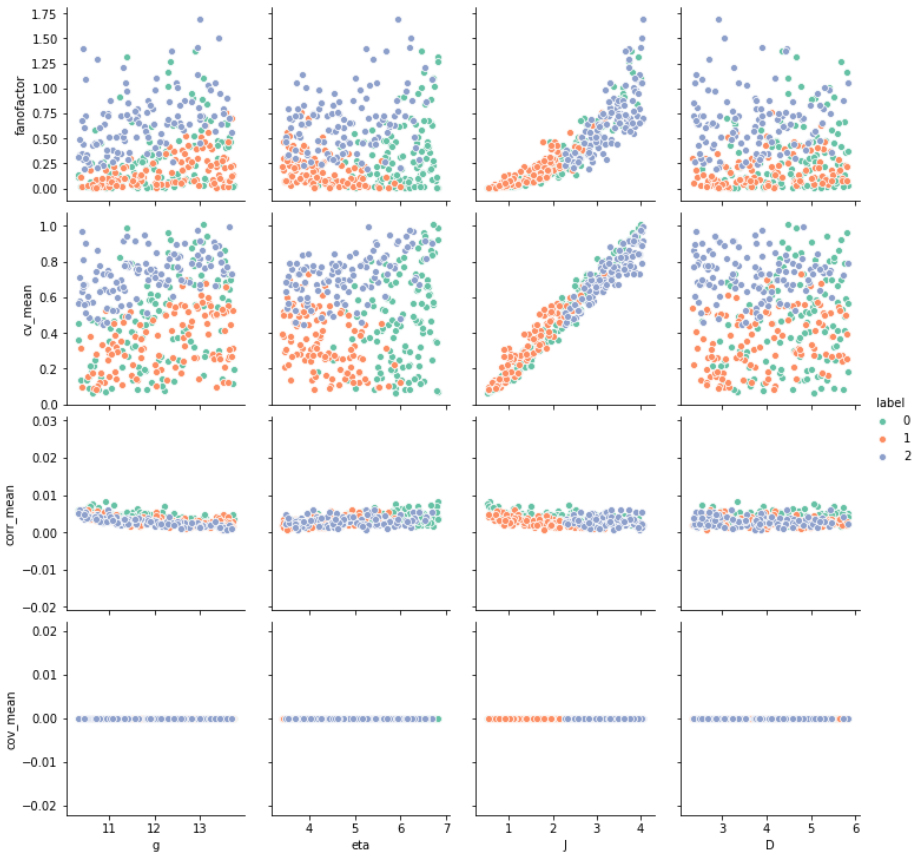


Figure 24- Clustering results on original data, for variation (b)

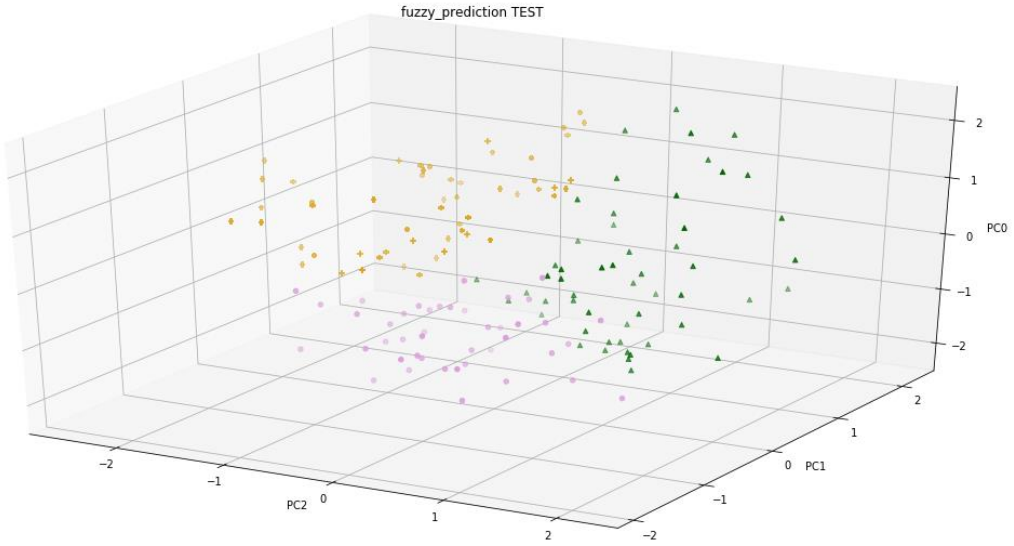


Figure 25 - Clustering Prediction method on scores, for variation (b)

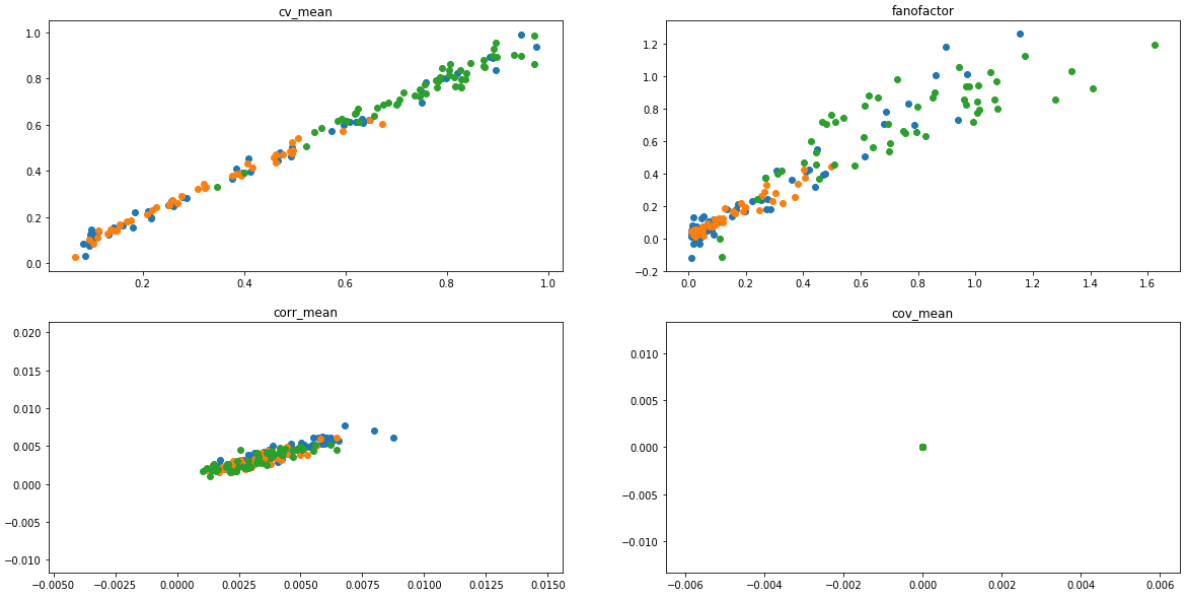


Figure 26 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (b)

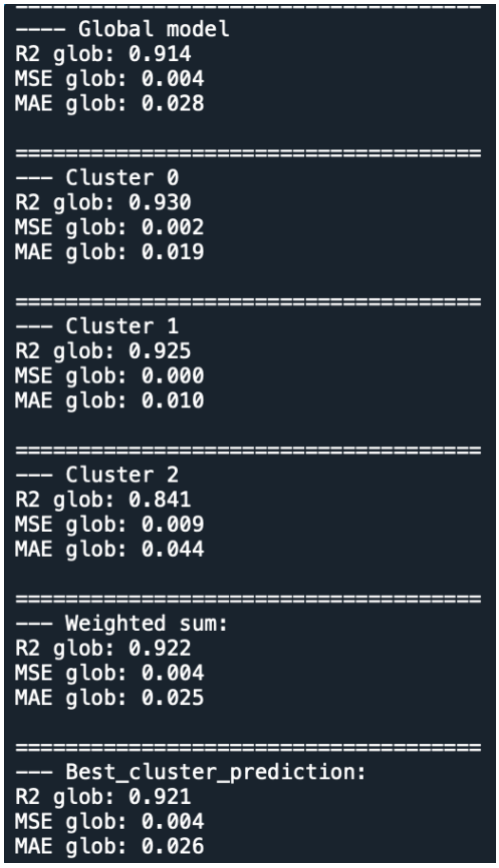


Figure 27 - Performance of model variation (b)

7.2.5. Variation c

HC-PLSR model variation *(c) as Version 2* (Appendix B), where the clustering model is created on X-scores from the global PLSR model. The global PLSR is created on original features, whereas the local PLSR models include the transformed terms. The number of components included in the PLSR models are chosen based on the optimize-function in script Appendix D.

Model specifications are stated in the table **Error! Reference source not found.**.

Table 9- model specifications for HC-PLSR variation (c)

Split	0.3
Number of clusters	3
Fuzzifier coefficient	5
Scores	X scores

In the following 3 figures, the clustering performance and the overall model performance is illustrated.

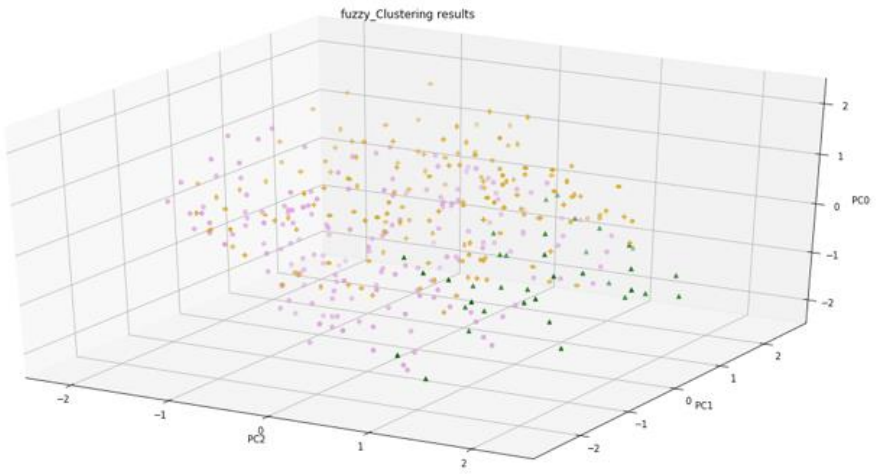


Figure 28 - Clustering result on Scores, for variation (c)

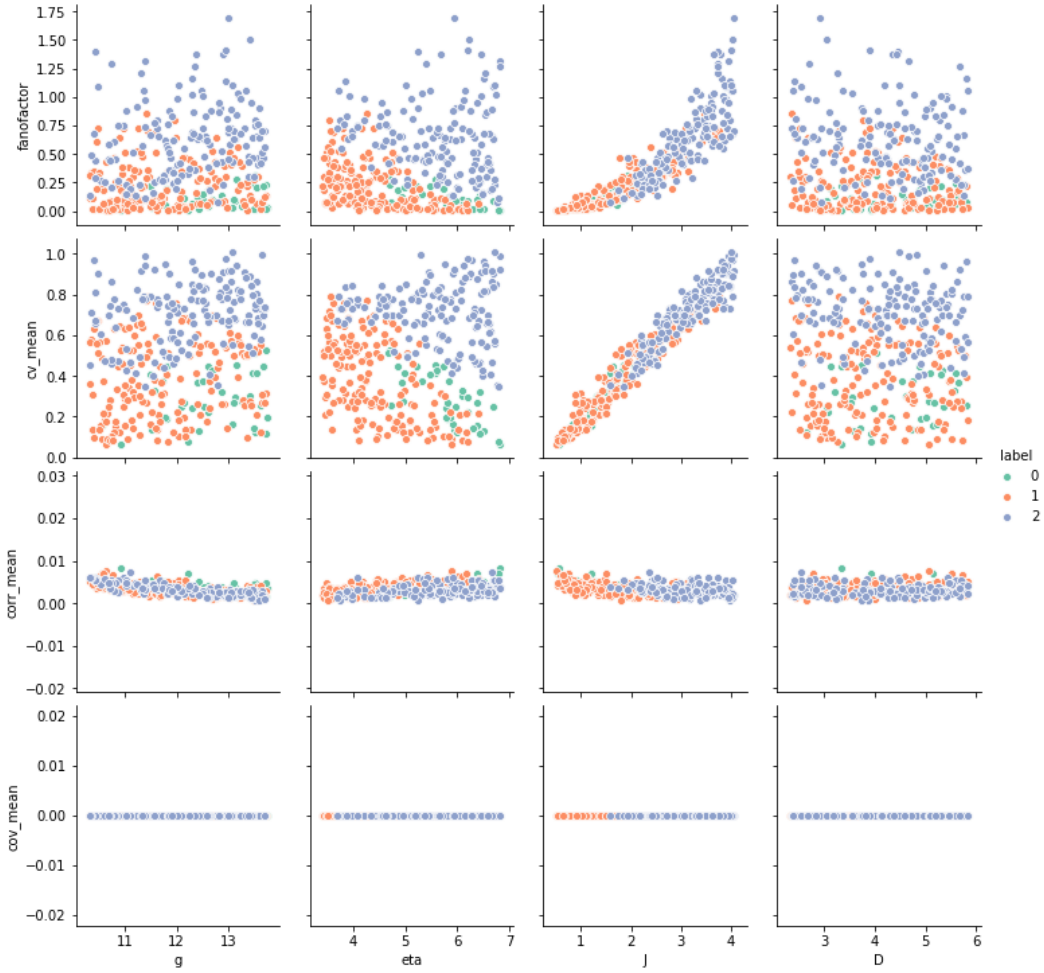


Figure 29 - Clustering results on original data, for variation (c)

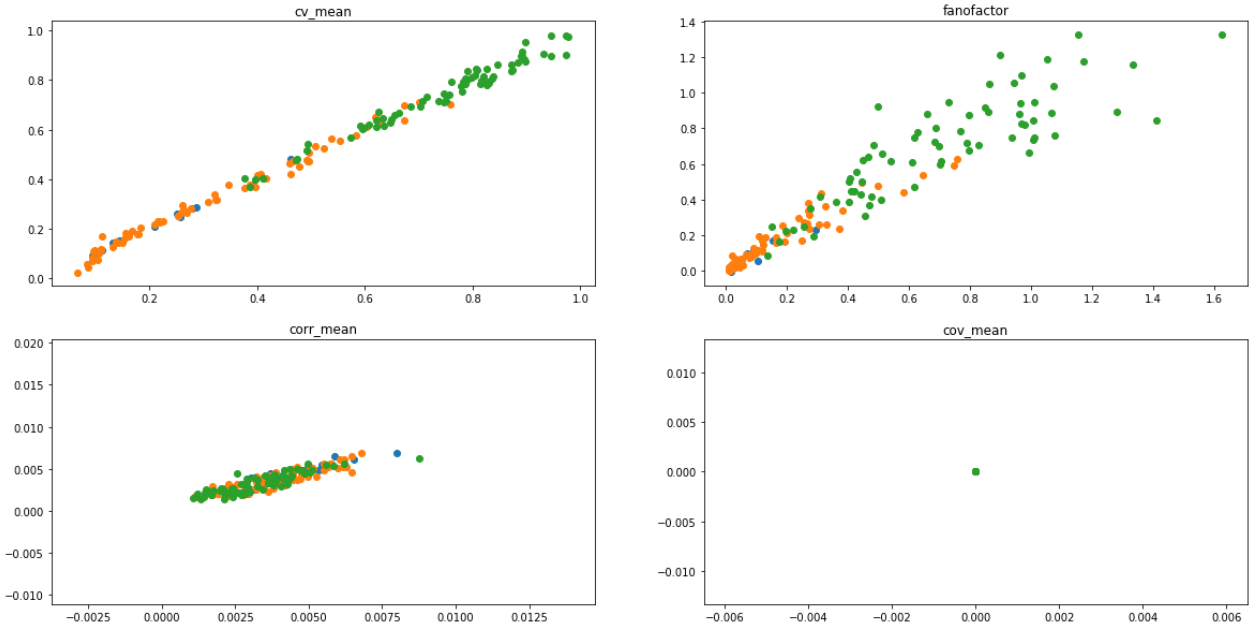


Figure 30 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (c)

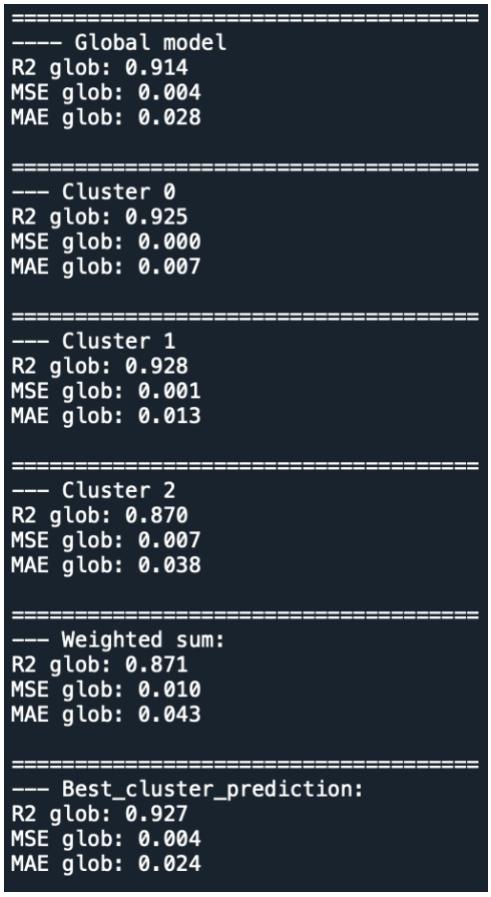


Figure 31 - Performance of model variation (c)

7.2.6. Variation d

HC-PLSR model variation *(d) as Version 2* (Appendix B), where the clustering model is created on Y-scores from the global PLSR model. The global PLSR is created on original features, whereas the local PLSR models include the transformed terms. The number of components included in the PLSR models are chosen based on the optimize-function in script Appendix D.

Model specifications are stated in the table **Error! Reference source not found.**.

Table 10 - model specifications for HC-PLSR variation (d)

Split	0.4
Number of clusters	3
Fuzzifier coefficient	2
Scores	Y scores

In the following 3 figures, the clustering performance and the overall model performance is illustrated.

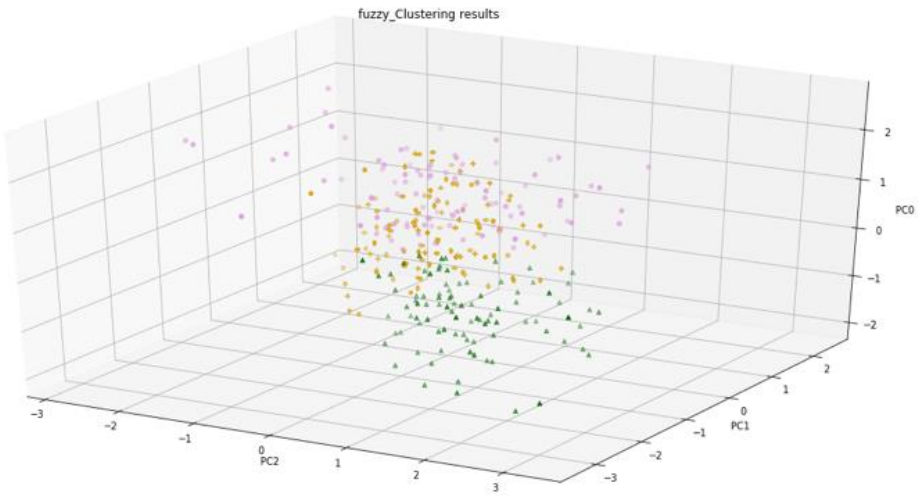


Figure 32 - Clustering result on Scores, for variation (d)

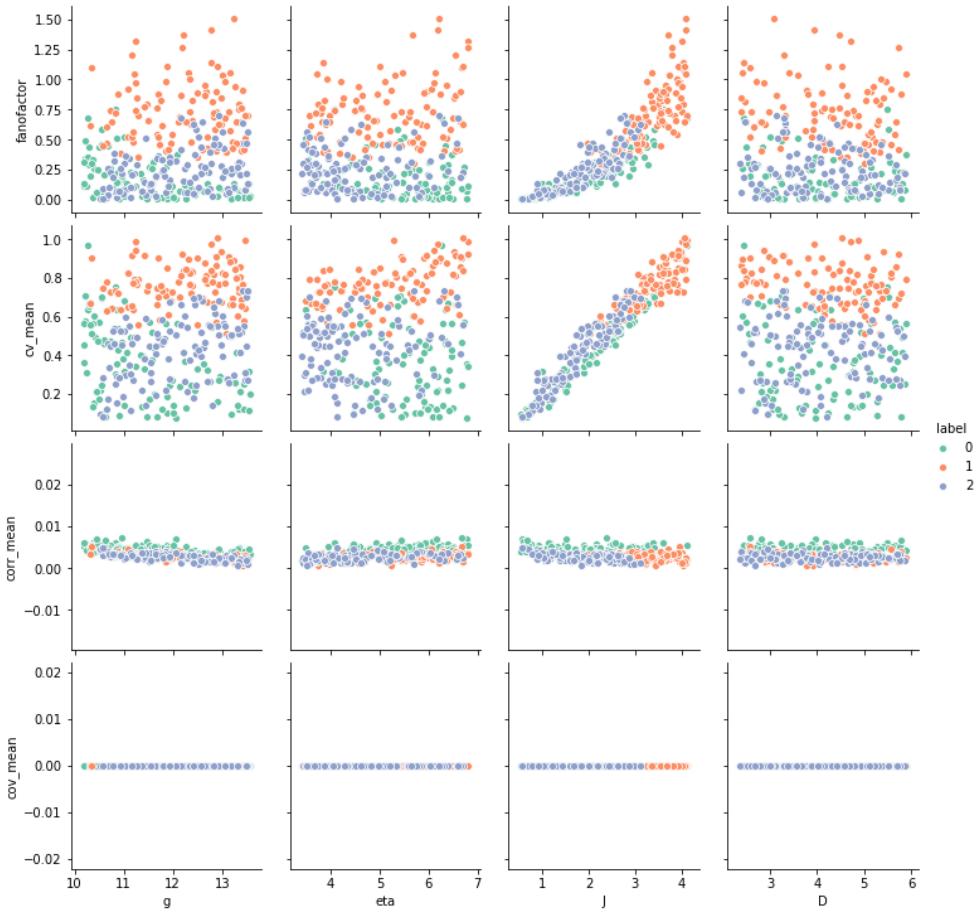


Figure 33 - Clustering results on original data, for variation (d)

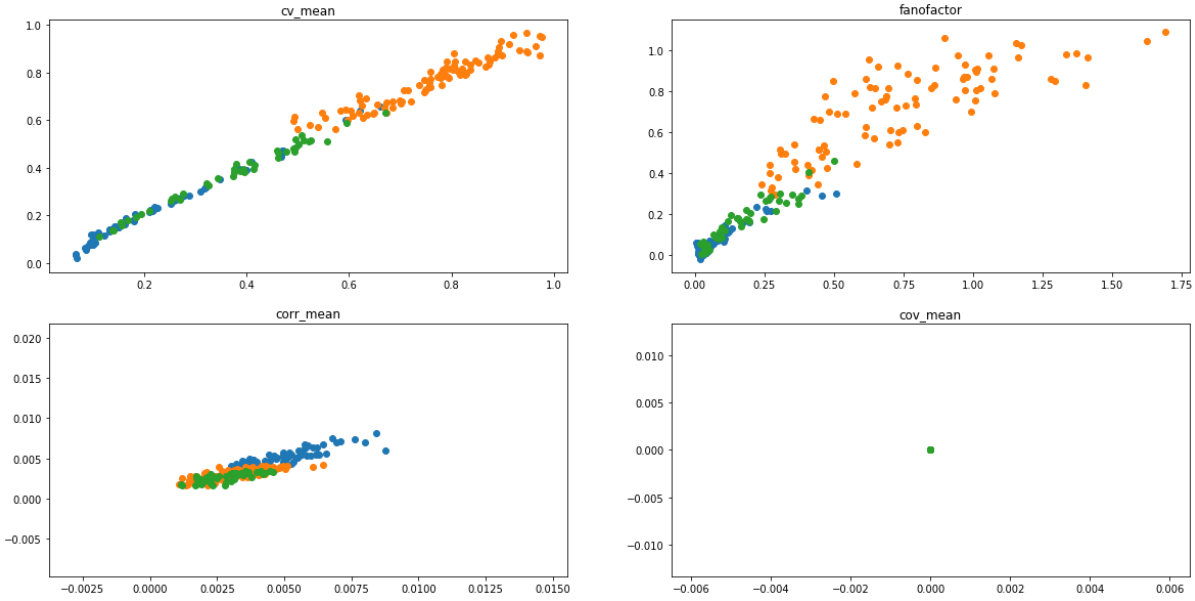


Figure 34 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (d)

```
====================================
---- Global model
R2 glob: 0.920
MSE glob: 0.006
MAE glob: 0.031

====================================
--- Cluster 0
R2 glob: 0.880
MSE glob: 0.000
MAE glob: 0.009

====================================
--- Cluster 1
R2 glob: 0.854
MSE glob: 0.001
MAE glob: 0.011

====================================
--- Cluster 2
R2 glob: 0.782
MSE glob: 0.011
MAE glob: 0.047

====================================
--- Weighted sum:
R2 glob: 0.913
MSE glob: 0.005
MAE glob: 0.030

====================================
--- Best_cluster_prediction:
R2 glob: 0.914
MSE glob: 0.005
MAE glob: 0.027
```

Figure 35 - Performance of model variation (d)

7.3.Overview of performance results

7.3.1. Total overview of model variations

Performance marked in green is exceeding the performance of the comparing model (PLSR). Version 1 models (V1) uses same variation of linear/non-linear on the global and local models, whereas Version 2 models (V2) uses linear PLSR model as a global model, and non-linear local modelling.

Table 11 - Model options and best performing model result

ID	Features in global PLSR	Scores used in Clustering	Clustering Algorithm	Features in Local PLSR	Result	Version	Notes
00	Original	-			Split 0.3 R2 = 0.885 MSE = 0.07 MAE = 0.037 Split 0.4 R2 = 0.879 MSE = 0.009 MAE = 0.040		Linear PLSR model for comparison, with optimal number of principal components included in the model Ch. 8.2.1
01	Polynomial	-			Split 0.3 R2 = 0.914 MSE = 0.004 MAE = 0.028		Non-linear PLSR model for comparison, with optimal number of principal components included in the model Ch. 8.2.2
02	Original	X	Fuzzy C	Original	Split 0.3 *Weighted sum:* R2 = 0.889 MSE = 0.006 MAE = 0.036 *Best cluster:* R2 = 0.897 MSE = 0.006 MAE = 0.036	V1	
03	Original	Y	Fuzzy C	Original	Split 0.4 *Weighted sum:* R2 = 0.859 MSE = 0.009 MAE = 0.039	V1	

					Best cluster: R2 = 0.896 MSE = 0.005 MAE = 0.029		
04	Polynomial	X	Fuzzy C	Polynomial	Split 0.3 *Weighted sum:* R2 = 0.922 MSE = 0.004 MAE = 0.025 *Best cluster:* R2 = 0.921 MSE = 0.004 MAE = 0.026	V1	
05	Polynomial	Y	Fuzzy C	Polynomial	Split 0.3 *Weighted sum:* R2 = 0.854 MSE = 0.010 MAE = 0.046 *Best cluster:* R2 = 0.902 MSE = 0.005 MAE = 0.031	V1	
06	Original	X	Fuzzy C	Polynomial	Split 0.3 *Weighted sum:* R2 = 0.871 MSE = 0.010 MAE = 0.043 *Best cluster:* R2 = 0.927 MSE = 0.004 MAE = 0.024	**V2**	
07	Original	Y	Fuzzy C	Polynomial	Split 0.4 *Weighted sum:* R2 = 0.913 MSE = 0.005 MAE = 0.030 *Best cluster:* R2 = 0.914 MSE = 0.005 MAE = 0.027	**V2**	

7.3.2. FCP Inspection

Fuzzy Partition Coefficient (FPC) for X scores using original:

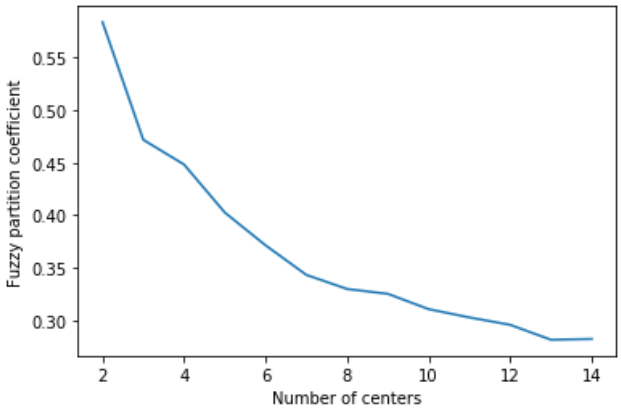


Figure 36 - Clustering on X_scores, Fuzzifier = 2

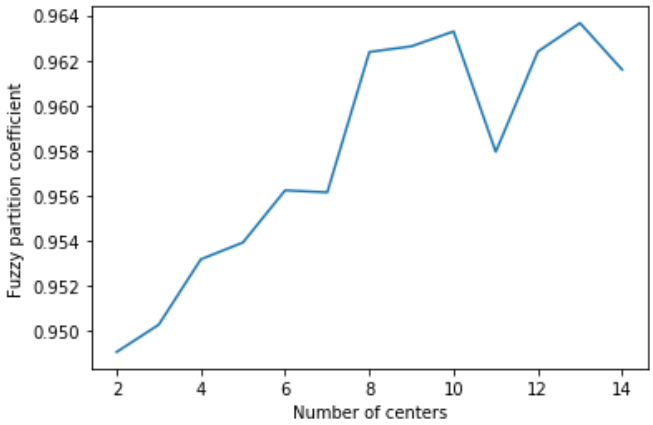


Figure 37 - Clustering on X_scores, Fuzzifier = 1.1

7.3.3. *Number of clusters and fuzzifier inspection*

Below are plots visualizing the model performance and how it is influenced by the fuzzifier-coefficient (clustering method parameter) and the number of clusters included in the model.

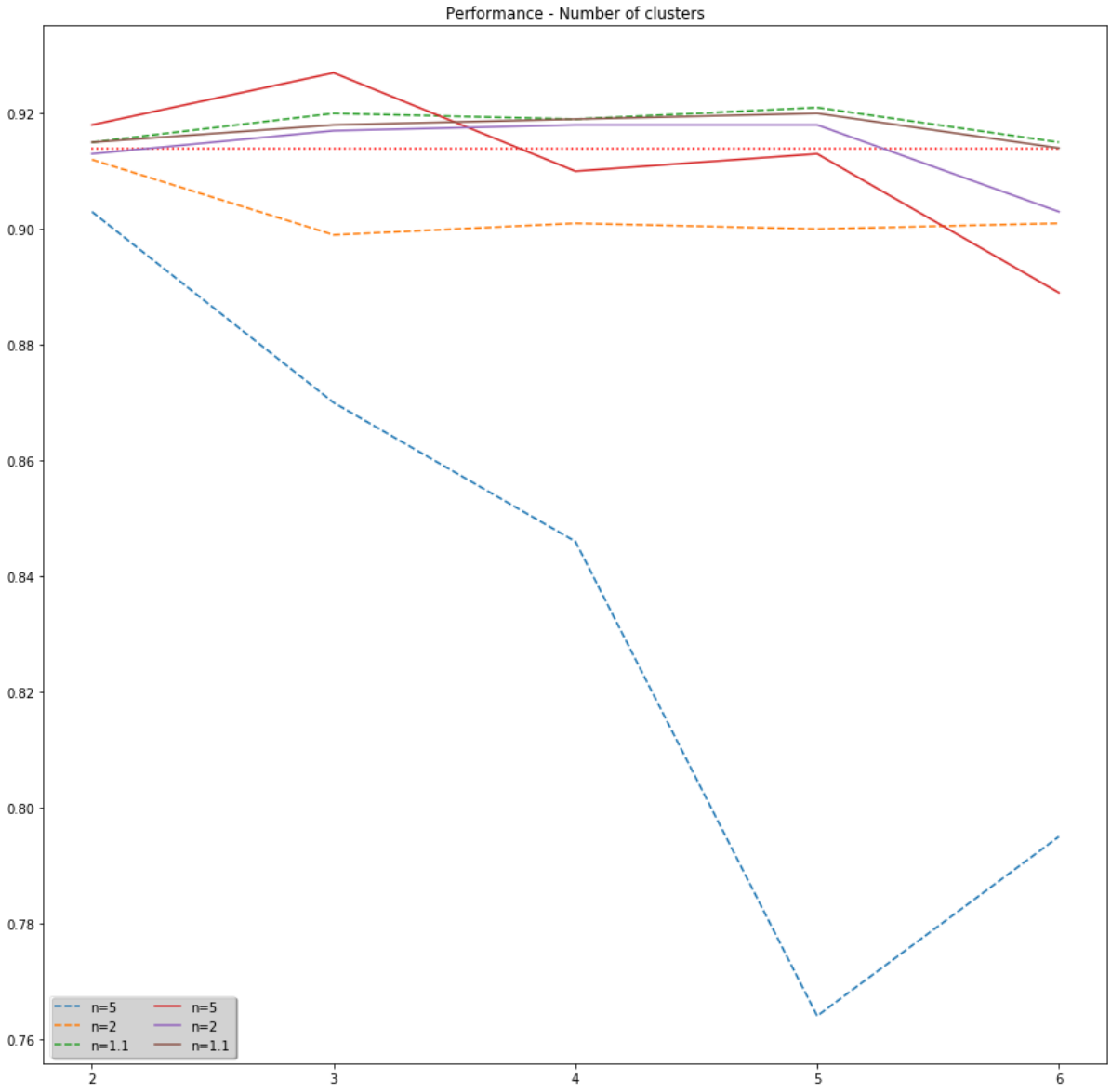


Figure 38 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 2"

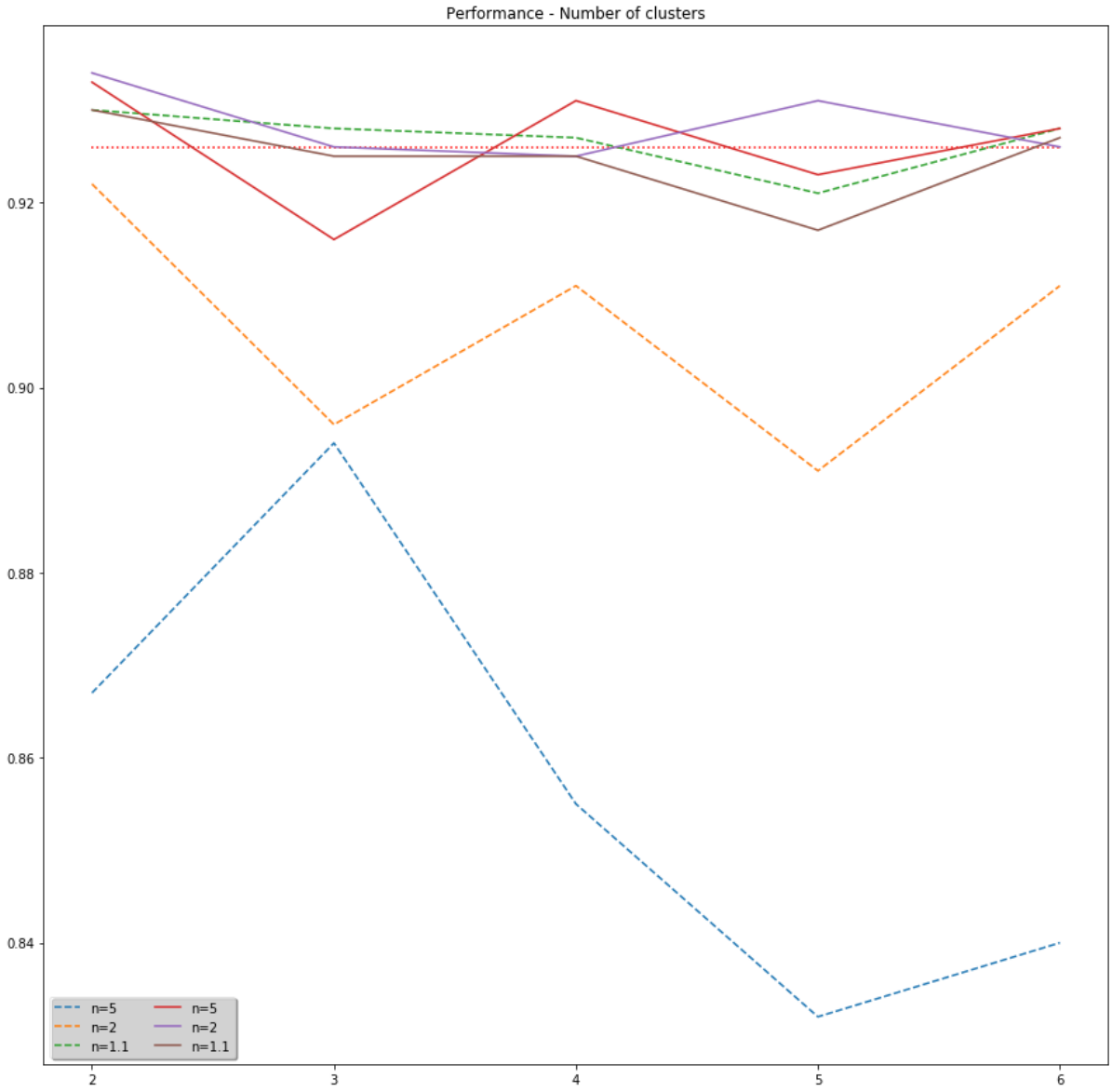


Figure 39 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 1" using polynomial PLSR

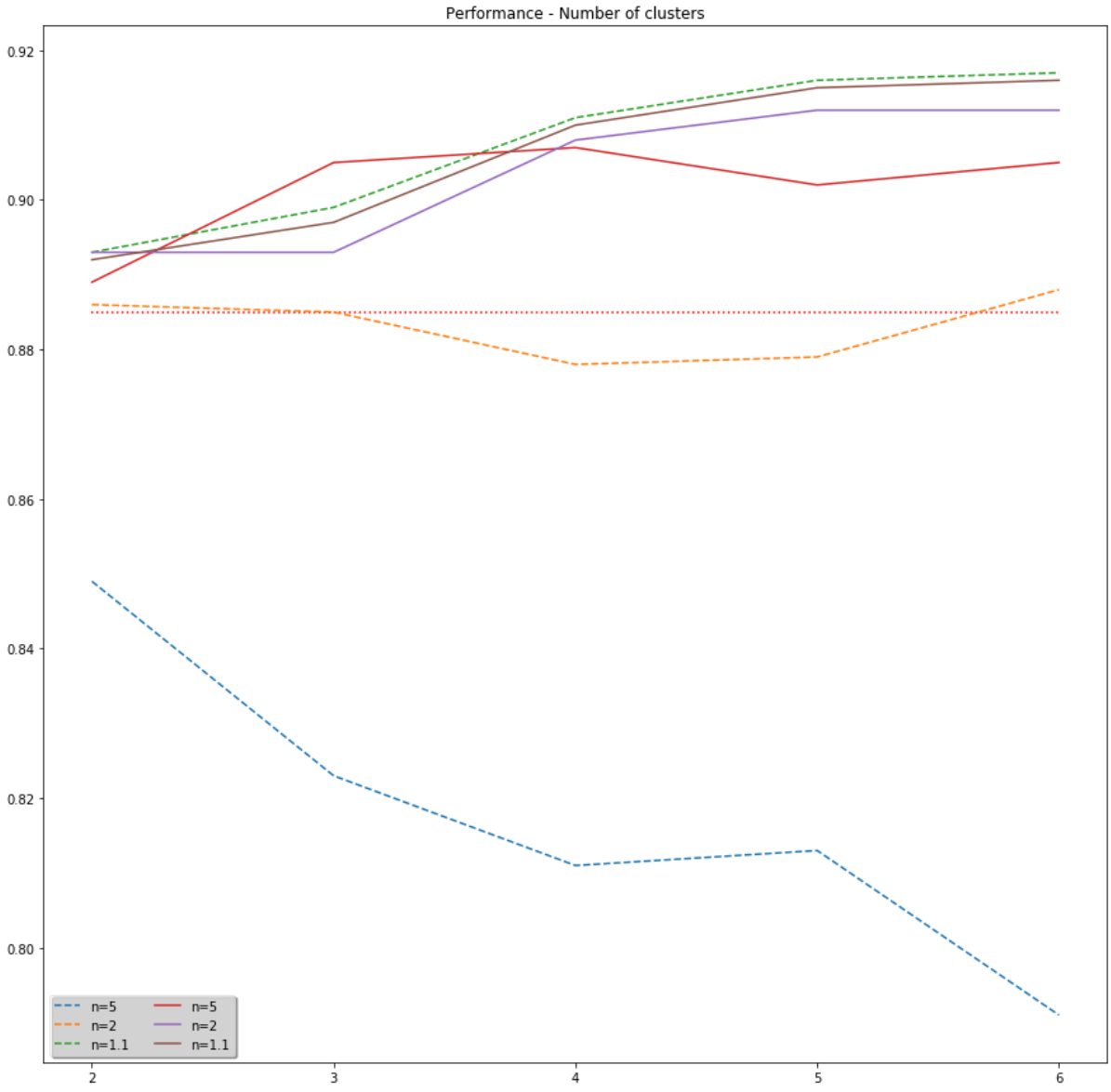


Figure 40 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 1", PLSR without cross/interaction terms

8. Discussion

The high prediction accuracy achieved by using a single polynomial PLSR, can reveal that only soft non-linearities exists in the behavior of the system. However, the increase in performance when clustering the subspace into regions, using HC-PLSR, might expose that abruptions of the subspace exists, that needs to be considered by a non-linear modelling technique. When comparing the HC-PLSR to a single PLSR model it is clear that the emulation capacities of the local linear modelling approach can account for a wider range of non-linearities, and thus result in greater prediction abilities.

Regarding the clusters influence prediction accuracy, two main aspects stand relevant for discussion. Firstly, the difference between clustering on X-scores and Y-scores. It is visually apparent that the clustering on Y-scores is managing the grouping of more “naturally” looking clusters. The space spanned from the first three X-components contains evenly distributed sample points, and the clustering method splits the subspace into smaller regions (rather than to extract new information about the data by exposing natural clustering patterns). If this is the case, there might exist more computationally effective ways of dividing the subspace into smaller regions. One could then debate the reasons for why the clustering on the Y-scores did not outperform the x-score clustering. One theory may be that the projections of new samples (for prediction using the HC-PLSR) when clustering on the y-scores are transformed by the PLSR model, after the target has been predicted from the same model. This means that prediction inaccuracies from the PLSR model is transferred when later projected onto the new feature space and labeled to the existing clusters, thus yield suboptimal results. In order to use the Y-scores as a basis for the clustering, the separation of the observations needs to be quite distinct [2], since the Y-scores contain some prediction error that may disrupt the classification when the clusters are not distinctly separated.

There were also some differences in the performance when considering the “Best cluster Prediction” versus the “Weighted sum Prediction”. The former used only the local model prediction from the most probable cluster for each sample, whereas the latter used the cluster belonging probabilities for all clusters as weights for the regression coefficient. Effectively the prediction was calculated as a weighted sum by utilizing the soft cluster belonging. These two different prediction methods gave slightly different performance results depending on what model combination was used. In [2] it is stated that the first approach performs better on

data with very distinct clusters, whereas the weighted sum of the local clusters probably will give better results on more continuous data. The performance of the prediction methods depend on how the cluster-algorithm separates the data into clusters, and this might explain the fluctuations in how the predictions options performed, even though this dataset is of a more continuous characteristic, where the clusters are not entirely separable. It is therefore hard to conclude on an optimal prediction strategy, whereas both should be included for comparison. However, if the "Best cluster" prediction method result in higher performance, it might be indicating that a fuzzy clustering method is unnecessary. This might also be explored by adjusting the fuzzifier-coefficient of the Fuzzy C Clustering algorithm. If a stronger separation of the fuzzy clusters improve performance, this may also indicate that an optional cluster method is favored. The Fuzzy C clustering is a fairly computationally demanding method compared to other "simpler" clustering techniques (e.g. K-nearest or K-Means clustering). One could also try to use the fuzzy-prediction method on clusters that were created as a result of hard clustering techniques, such as the two mentioned above. This will increase the computational efficiency, which is especially relevant if the dataset is relatively big and the fuzzy-prediction ("weighted sum"- prediction) accuracy is not outperforming the "best cluster" - prediction.

The HC-PLSR model structure can be altered in many ways while still applying the main concepts of locally linear modelling techniques. Both the clustering method (with all its parameter and optimization methods) and the local modelling present options for model architecture exploring. Examples of this might be outlier restrictions (for clustering and for creation of local models), or restrictions on amount of explained variance accounted for by the principal components. As a solution to the many modelling options for the HC-PLSR, a framework for testing several variations was the strategy when implementing the model design. The difference in performance depending on architectural designs, might be a usable and efficient tool for learning about the structures of the given dataset. As presented in [2], one of the strengths of HC-PLSR metamodelling lies in its ability to improve the analytical insights of the model being emulated. When little or no prior knowledge of the model behavior exists, is serves as a powerful tool for attaining knowledge about the system behavior that is not possible by a global model generalizing the entire dataset.

The predictive performance of a polynomial PLSR model is quite good and might indicate that the level on non-linearity in the given dataset is handled sufficiently well compared to an

HC-PLSR model. However, if no new cross- and higher order terms are added to the models, the HC-PLSR outperforms the PLSR by a greater margin. From this it is clear that the clustering is able to handle non-linearities in the data, and determining "how much better" it performs, is a matter of the degree of non-linearity in the data.
The new model strategy called "Version 2" in "Results", did result in the most improved prediction accuracy of all tested variants in this project. This might imply that the clustering of scores from a global model built on the original features (without interaction terms), is able to represent relevant non-linear structures in the data that diminishes when adding polynomial features.

The statistical summaries/measurements were used to represent the model output, were chosen based on general knowledge of relevant statistical aspects of spike-train characteristics. The coefficient of covariance was close to zero for almost all simulation runs, however it was included in the model because it is expected to change if the input parameter ranges is altered to include values from different states of the network. In addition to this, the statistical package "Elephant" does contain several other measurements, such as wave-to-noise-ratio or other descriptive values, that could uncover new knowledge about the neural network behavior that characterizes the defined states. This aspect highlights the need for interdisciplinary collaboration; by including more domain knowledge to the modelling, the prediction performance and interpretability can be optimized. Examples of this could be considering the choice of what simulation input parameters considered relevant, or defining the parameter regions to sample from, and by creating meaningful response parameters.

9. Further work

A focus for this implementation has been on generalizing the method to facilitate the exploring of what model is more efficient for the dataset at hand. It is clear that the clustering method and the resulting subspaces is of high importance for the HC-PLSR performance. The fuzzy clustering implemented herein, is computationally demanding for large sample sizes. A way to increase the computational efficiency of the HC-PLSR is to look for an alternative to Fuzzy-C Clustering method and it could also be interesting to see if there exists some relationship between the structure of the samples plotted on the principle components from the PLSR-model and the optimal cluster method for the given dataset.

Other suggestions for future work include:

- Use parameters regions that include different network states for the modelling
- Standardize a framework for easily altering as exploring model parameters such as: different clustering technique, different cluster prediction technique, number of components to cluster on, number of cluster centers, different distance measures, outlier restrictions, ways of including domain knowledge, prediction using the most probable clusters. This could look like the GridSearchCV from Sklearn, where the function would return the optimal parameter combination.
- Generalize for many types of data as a “plug and play” for inspection, would contribute largely for future work on the area.
- Use agglomerative Hierarchical clustering or HDBScan to inspect for cluster patterns before fuzzy (since K-Means and Fuzzy-C tends to create round clusters, and the density-based methods might pick up the anomalous patterns)
- Automize detection of best number of clusters, by measuring cluster performance. (using silhouette plots or "is the internal measures similar for the training and the test data?")
- Include criteria for explained variance of the latent variables included in the model
- Add framework with unit tests to verify code quality

10. Conclusion

This study is an exploration of the local multivariate modelling method called HC-PLSR. It also contains (model) implementations to examine the performance using a dataset generated from deterministic modelling of a Neural Network simulation (first described by Brunel [3]).

The main objective was to explore the possibilities of creating metamodels of Brunel's Neural Network model with good prediction accuracy. This may then be used to gain better insights and understanding of neural networks and their workings. In this context it was of interest to investigate how non-linear/non-monotone this system is by comparing PLSR (with and without higher order terms) and HC-PLSR, that accounts for different types of non-linearities.

One observes that when the global and local PLSR models are without polynomial terms (trained only using original features without added interaction terms), the HC-PLSR does indeed account for some non-linear relationships in the dataset that a single PLSR-model cannot. However, when added cross- and higher order terms to the features the HC-PLSR outperforms the polynomial PLSR model (for comparison) by a slighter margin. Another interesting result occurred from a new variant of the HC-PLSR modelling being tested. Here the global PLSR was without polynomial terms while the local PLSR-models were polynomial (added interaction and higher order terms). This variant of the model (called Version 2 in "Results") yielded the best predictive performance of all tested models and model combinations. These results indicate that the use of HC-PLSR modelling is an effective method for emulating non-linear mathematical models. Division of the parameter spaces into subspaces is the core idea of HC-PLSR and changing what the clustering method results in can impact the modelling performance a lot. Thus, continuing to explore the different clustering strategies seems like an effective approach for further optimization of the HC-PLSR.

A second aim of this study was to contribute to a methodology that invites for interdisciplinary understanding and collaboration. The framework for implementations of the models tested in this project is created on a modular basis and focuses on model exploration and development by facilitating testing of new modelling strategies. Important parts of model assembling of the HC-PLSR is safeguarded against implementation errors, by using tools and packages from external developers. In addition, the use of this model exploration technique

does not require prior knowledge of the system/model output or of the structure of the given dataset; a favored feature when different scientific fields cooperate on development of a model strategy.

11. References

[1] David Sterrat, Bruce Graham, Andrew Gillies, David Wilshaw, Principles of Computational Modelling in Neuroscience, Cambridge: Cambridge University Press, 2011.

[2] Kristin Tøndel1*, Ulf G Indahl1, Arne B Gjuvsland1, Jon Olav Vik1, Peter Hunter2, Stig W Omholt3 and Harald Martens1, "Hierarchical Cluster-based Partial Least Squares Regression (HC-PLSR) is an efficient tool for metamodelling of nonlinear dynamic models," BMC Systems Biology , Ås, Norway, 2011.

[3] N. Brunel, "Dynamics of Sparsely Connected Networks of Excitatory and Inhibitory Spiking Neurons," Kluver Academic Publishers, Paris, 2000.

[4] A. Stene, "open directory github.com," august 2020. [Online].

[5] Campbell K, McKay MD, Williams BJ, "Sensitivity analysis when model outputs are functions," Reliability Engineering & System Safety, 2006.

[6] Martens H, Veflingstad S, Plahte E, Martens M, Bertrand D, Omholt S, "The genotype-phenotype relationship in multicellular pattern-generating models - the neglected role of pattern descriptors," BMC Syst Biol, 2009.

[7] "Saylor Academy," 3.1 The Neuron Is the Building Block of the Nervous System, [Online]. Available: https://saylordotorg.github.io/text_introduction-to-psychology/s07-01-the-neuron-is-the-building-blo.html.

[8] A. Stene, "Reproducing Brunel's model and explaining fundamentals, Exploring Computational Neuroscience," Oslo, 2019.

[9] BrainFacts, "The Neuron, BrainFacts.org," april 2012. [Online]. Available: https://www.brainfacts.org/brain-anatomy-and-function/anatomy/2012/the-neuron. [Accessed august 2020].

[10] "Molecular Devices, What is an action potential?," [Online]. Available: https://www.moleculardevices.com/applications/patch-clamp-electrophysiology/what-action-potential#gref. [Accessed August 2020].

[11] S. K. W. Parvesh Kumar, "Comparative Analysis of k-mean Based Algorithms," IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.4, April 2010, 2010.

[12] Dongkuan Xu, Yingjie Tian, "A Comprehensive Survey of Clustering AlgorithmsAnn. Data. Sci. 2, 165–193 (2015). https://doi.org/10.1007/s40745-015-0040-1," 2015.

[13] G. Seif, "The 5 Clustering Algorithms Data Scientists Need to Know, https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68," 2015.

[14] SciKit-Learn, "Silhouette Analysis," [Online]. Available: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html. [Accessed August 2020].

[15] P. Jain, "Clustering Clearly Explained Intuitive and Informative guide about Clustering and Clustering Algorithms," 2019.

[16] Mckay, M.D. & Beckman, Richard & Conover, William, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code in wsc '05: proceedings of the 37th conference on winter simulation," Technometrics. 42. 202-208. , 2000.

[17] U. u. S. A, "Sampling Methods," University of Oslo, https://www.uio.no/studier/emner/matnat/math/STK4400/v05/undervisningsmateriale/Sampling%20methods.pdf, Oslo.

[18] Xia, S.; Song, L.; Wu, Y.; Ma, Z.; Jing, J.; Ding, Z.; Li, G, "An Integrated LHS–CD Approach for Power System Security Risk Assessment with Consideration of Source–Network and Load Uncertainties," Processes, 2019.

[19] K.R.M. dos Santosa , A.T. Becka , "A benchmark study on intelligent sampling techniques in Monte Carlo simulation," University of São Paulo. São Carlos School of Engineering Department of Structural Engineering, Sao Paolo, 2015.

[20] M. Editor, "Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit?," 2013.

[21] M. BlogEditor, "Why You Need to Check Your Residual Plots for Regression Analysis: Or, To Err is Human, To Err Randomly is Statistically Divine," 2012.

[22] M. Deeb, "Feature Engineering — Automation and Evaluation — Part 1," Medium, [Online]. Available: https://medium.com/ki-labs-engineering/feature-engineering-automation-and-evaluation-part-1-a34fb42e0bd4. [Accessed August 2020].

[23] Dipanjan Sarkar, Raghav Bali, Tushar Sharma, "Practical Machine Learning with Python," Apress, [Online]. Available: https://github.com/dipanjanS/practical-machine-learning-with-python. [Accessed August 2020].

[24] T. N. S. T. Initiative, "Nest-Simulator," 9th august 2020. [Online]. Available: https://www.nest-simulator.org/.

[25] "Featuretools," Copyright 2019, Feature Labs. BSD License Revision ff70beb6, [Online]. Available: https://docs.featuretools.com/en/stable/index.html#. [Accessed August 2020].

[26] s.-l. developers, "Scikit Learn, PLSR Regression," 2007 - 2020, scikit-learn developers (BSD License), [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cross_decomposition.PLSRegression.html. [Accessed 9th August 2020].

[27] S. F. D. Team, "Scikit Fuzzy - SkFuzz," [Online]. Available: https://pythonhosted.org/scikit-fuzzy/auto_examples/plot_cmeans.html. [Accessed 9 August 2020].

[28] P. Developers, "Pypet Documentation," [Online]. Available: https://pypet.readthedocs.io/en/latest/index.html. [Accessed 9 August 2020].

[29] J. Hwang, "Latin Hypercube Sampling Documentation," 2017. [Online]. Available: https://smt.readthedocs.io/en/latest/_src_docs/sampling_methods/lhs.html. [Accessed 9 August 2020].

[30] X. Li, "Numerical Methods for Engineering Design and Optimization, Latin Hypercube Sampling (LHS)," https://users.ece.cmu.edu/~xinli/classes/cmu_18660/Lec25.pdf, Pittsburg, 2014.

[31] E. A. a. Contributors, "Elephant Documentation," 2014-2020. [Online]. Available: https://elephant.readthedocs.io/en/latest/index.html#. [Accessed 9 August 2020].

[32] M. Editor, "Multiple Regression Analysis: Use Adjusted R-Squared and Predicted R-Squared to Include the Correct Number of Variables," 2013.

[33] R. Nau, "Statistical Regression," Fuqua School of Business, Duke University, 2019. [Online]. Available: http://people.duke.edu/~rnau/rsquared.htm.

12. Figures and tables

12.1. List of figures

Figure 1 - Inverse and Classical Metamodelling illustration 13
Figure 2 - Illustration of the HC-PLSR approach from [2] 15
Figure 3 - Components of the neuron from [7] 17
Figure 4 - The Action Potential from [10] 18
Figure 5 - Illustration of network states, (figure 8 from [3]) 22
Figure 6 - Illustration of Silhouette Analysis concept, from [14] 26
Figure 7 - Linear relationship illustration 28
Figure 8 - Sampling step in Latin hypercube sampling (LHS), from [18] 31
Figure 9 – Two-dimensional random sampling of a uniform Random LHS with 5 samples ..32
Figure 10 - Latin Hypercube Sampling Concept from [19] 32
Figure 11 - Model development pipeline, the role of Feature Engineering from [22] 34
Figure 12 - Illustration of FPC from [27] 39
Figure 13 - Data inspection, X vs Y 48
Figure 14 - Data inspection, X_scores vs Y_scores 48
Figure 15 - Illustration of 3 PCs with cluster labels (colored green, yellow and pink) 50
Figure 16 - Visualization of clustering on the Y_scores 52
Figure 17 - Visualization of clustering of the X_scores 52
Figure 18 - PLSR optimizer function of optimal number of components 55
Figure 19 – Clustering result on Scores, for variation (a) 56
Figure 20 - Clustering results on original data, for variation (a) 56
Figure 21 - Y_predicted vs Y_true plot, predictions from each cluster in different color. For variation (a) 57
Figure 22 - Performance of model variation (a) 57
Figure 23 - Clustering result on Scores, for variation (b) 58
Figure 24- Clustering results on original data, for variation (b) 59
Figure 25 - Clustering Prediction method on scores, for variation (b) 59
Figure 26 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (b) 60
Figure 27 - Performance of model variation (b) 60

Figure 28 - Clustering result on Scores, for variation (c) 61
Figure 29 - Clustering results on original data, for variation (c) 62
Figure 30 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (c) 62
Figure 31 - Performance of model variation (c) 63
Figure 32 - Clustering result on Scores, for variation (d) 64
Figure 33 - Clustering results on original data, for variation (d) 65
Figure 34 - Y_predicted vs Y_true plot, predictions from each cluster in different color, for variation (d) 65
Figure 35 - Performance of model variation (d) 66
Figure 36 - Clustering on X_scores, Fuzzifier = 2 69
Figure 37 - Clustering on X_scores, Fuzzifier = 1.1 69
Figure 38 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 2" 70
Figure 39 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 1" using polynomial PLSR 71
Figure 40 - R_2 Performance, Cluster and Fuzzifier inspection, Model "Version 1", PLSR without cross/interaction terms 72

12.2. List of tables

Table 1 - Solving issue with package integration, Lines to inactivate 35
Table 2 - Overview of important project modules 36
Table 3 – Table of parameters for simulation of Brunel Network Model 43
Table 4 - Creating of Statistical measures overview 46
Table 5 - Model Variations for all steps in the meta model pipeline 51
Table 6 - Hyperparameter tuning overview 51
Table 7 - model specifications for HC-PLSR variation (a) 55
Table 8 - model specifications for HC-PLSR variation (b) 58
Table 9- model specifications for HC-PLSR variation (c) 61
Table 10 - model specifications for HC-PLSR variation (d) 64
Table 11 - Model options and best performing model result 67

13. Appendix

Implementations included here are for adding information to the descriptive explanations in this paper. However, many more files of local modules tested were necessary, so ensure that all parts of the workflow worked individually before creating a model pipeline structured script. All of these tests and implementation blocks are available on the Open GitHub page, to simplify the continuing on this project or taking parts of it to use in new projects.

Appendix	**Filename**	**comment**
A	Meta_compressed_v1.py	Version 1 (main) implementation of HC-PLSR
B	Meta_compressed_v2.py	Version 2, optional variant of HC-PLSR
C	Fuzzy_cluster.py	Handles the creation of clusters and prediction of cluster labels (Fuzzy C Clustering implementation)
D	Optimize_PLSR.py	Finding optimal number of components in the PLSR model (minimizing MSE)
E	Split_data.py	Splitting data into training and test sets (randomly shuffled)
F	Dfs.py	Adding cross and higher order terms using Featuretools Deep Feature Synthesis
G	Performance.py	Calculating and printing performance of model
H	Run_exploration_simulation_NEST_pypet_LHS.py	Running a test exploration simulation
I	Create_csv_wSats_from_hdf5.py	Creating the CSV-file containing X and Y
J	Brunel_delta_ml.py	NEST [24] version of Brunel's model implemented

K	Brunel_delta.py	Altered implementation of Brunel's model A. Based on Brunel_delta_ml.py
L	Run_NEST_using_pypet_LHS.py	Running main exploration simulation
M	Add_statistics_summaries.py	Adding statistical summaries of the spike trains
N	Requirements.txt	Module/environment requirements

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 29 13:29:21 2020

@author: Anja Stene, anja.stene@nmbu.no
"""


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
from optimise_PLSR import optimise_pls_cv
from split_data import get_splitted_data
random.seed(100)
np.random.seed(100)
from sklearn.cross_decomposition import PLSRegression

df = pd.read_csv("data_500_params_stats_v5.csv")


def create_local_plsr_models(traindata_wLabels):

    unique_clusters = np.unique(traindata_wLabels['label'])
    print("uniq",unique_clusters)
    # collect datapoints belonging to each cluster in separate datasets
    data_split_by_clusters={}

    for i in unique_clusters:
        data_split_by_clusters["cluster{}".format(i)] = traindata_wLabels.loc[traindata_wLabels['label'] == i]

    #split to X and y for each cluster
    X_y_in_cl={}

    for i in unique_clusters:
        X_y_in_cl["X{}".format(i)] = \
        data_split_by_clusters["cluster{}".format(i)][features]
        X_y_in_cl["y{}".format(i)] = \
        data_split_by_clusters["cluster{}".format(i)][targets]


    pls_models_dct={}
    for i in unique_clusters:
        Xn = X_y_in_cl["X{}".format(i)]
        max_ncomps = len(Xn.columns)
        #Xn = scaler_pls.transform(Xn)
        yn = X_y_in_cl["y{}".format(i)]
        print(i,"",Xn.shape)
        pls_models_dct["plsmod{}".format(i)] = optimise_pls_cv(Xn, yn, max_ncomps, plot_components=True, title='PLSR for cluster {}'.format(i))
        #pls_models_dct["plsmod{}".format(i)] = PLSRegression(n_components=3, scale=True).fit(Xn, yn)
    return pls_models_dct


def create_best_localmodel_prediction_col(testdata_wLabels):

    ## Adding the prediction from the best results in as "best_cl_" to data_w_labels

    m = pd.DataFrame([])

    for i in uniquie_clusters:
        best_cl_data = testdata_wLabels.loc[testdata_wLabels['label']==i]

        best_cl_preds = best_cl_data[["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),

                                      "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i) ]].copy()

        best_cl_preds.columns = ['best_cl_fano', 'best_cl_cv', 'best_cl_corr', 'best_cl_cov']

        m=m.append(best_cl_preds)


    testdata_wLabels = testdata_wLabels.join(m)

    return testdata_wLabels
```

```
78
79
80  def calculate_weighted_sum(testdata_wLabels):
81
82      ## Create Weighted sums
83
84      weighted_sum_fano = 0
85      weighted_sum_cv = 0
86      weighted_sum_corr = 0
87      weighted_sum_cov = 0
88
89      for i in uniquie_clusters:
90          weighted_sum_fano += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_fano_{}".format(i)])
91          weighted_sum_cv += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_cv_mean_{}".format(i)])
92          weighted_sum_corr += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_corr_mean_{}".format(i)])
93          weighted_sum_cov += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_cov_mean_{}".format(i)])
94
95      testdata_wLabels['sum_pred_fano'] = weighted_sum_fano
96      testdata_wLabels['sum_pred_cv'] = weighted_sum_cv
97      testdata_wLabels['sum_pred_corr'] = weighted_sum_corr
98      testdata_wLabels['sum_pred_cov'] = weighted_sum_cov
99
100     return  testdata_wLabels
101 # ============================================================================================================
102
103 split = 0.2              #Train-test split to use
104 seed = 10                #random seed
105 ncenters = 3             #number of cluster centers to use in HC-PLSR
106 f = 2                   #Fuzzyfier component
107 components_to_cluster = 3   #Number of Principal Components to peform clustering on
108 score = 'x'              #Cluster on the X-Scores
109 #score = 'y'             #Cluster on the Y-Scores
110
111 # ============================================================================================================
112
113
114 X_train, y_train, X_test, y_test, data = get_splitted_data(df, split)
115
116
117 targets = ['cv_mean','fanofactor', 'corr_mean', 'cov_mean']
118 #y_train = data_train[targets]
119
120 # ============================================================================================================
121
122
123 from dfs import get_DFS
124
125 X_train_dfs, X_test_dfs = get_DFS(X_train, X_test)
126
127 #Assign DFS versions to X_train/test
128 X_train = X_train_dfs.copy()
129 X_test = X_test_dfs.copy()
130
131
132 features = X_train.columns  #column names to all features / regressors
133
134 # Global model
135 plsmod_full_dfs = optimise_pls_cv(X_train, y_train, len(X_train.columns), plot_components=True)
136
137 #Extracting the x_scores from the global plsrmodel to cluster
138 pls_x_scores = pd.DataFrame(plsmod_full_dfs.x_scores_)
139 pls_y_scores = pd.DataFrame(plsmod_full_dfs.y_scores_)
140
141
142 # ============================================================================================================
143
144
145 from fuzzy_cluster import fuzzy_cluster, fuzzy_prediction
146
147 if score == 'y':
148     pls_scores = pls_y_scores
149 else:
150     pls_scores = pls_x_scores
151
152 traindata_wLabels, scaler_cluster, cntr =  fuzzy_cluster(pls_scores, X_train, y_train, components_to_cluster, ncenters, f, seed)
153 uniquie_clusters = np.unique(traindata_wLabels['label'])
154 # ============================================================================================================
```

```
155
156
157
158 pls_models_dct = create_local_plsr_models(traindata_wLabels)
159
160 # ============================================================================================================
161 #Plotting original data for structure inspections after labeling
162
163 import seaborn as sns; #sns.set(style="ticks", color_codes=True)
164
165 p = sns.pairplot(traindata_wLabels, x_vars=['g', 'eta', 'J', 'D'],
166             y_vars = ['fanofactor','cv_mean','corr_mean', 'cov_mean'], hue="label", palette="Set2") # hue="cluster",
167
168 # ----------------------------------------------------------------------------
169 ## test data
170
171
172 """X_test_sc = scaler_pls.transform(X_test)
173 X_test_sc = pd.DataFrame(X_test_sc, columns = features )"""
174
175 #prediction  using global PLSR-model
176 y_pred_glob = plsmod_full_dfs.predict(X_test)
177
178 #Projecting/transforming X-data onto components from PLSR
179 transf_X, transf_y = plsmod_full_dfs.transform(X_test, y_pred_glob)
180
181 if score == 'y':
182     df_transf = pd.DataFrame(transf_y.copy())
183 else:
184     df_transf = pd.DataFrame(transf_X.copy())
185
186 testdata_wLabels = fuzzy_prediction(df_transf, X_test, y_test, scaler_cluster, components_to_cluster, cntr, f, seed, title="fuzzy_prediction TEST")
187
188
189 p = sns.pairplot(testdata_wLabels, x_vars=['g', 'eta', 'J', 'D'],
190             y_vars = ['fanofactor','cv_mean','corr_mean', 'cov_mean'], hue="label", palette="Set2") # hue="cluster",
191
192 # ============================================================================================================
193
194 ## predict using local plsr model and sum up using weights
195
196 #For all local models: Predict sample
197 # calculate weighted sum based on clusterbelonging and pred value from all local plsr models
198
199
200 for i in uniquie_clusters: #or for local_plsmodel in pls_model_dct:
201     #split data belonging to certain cluster into x and y
202     X = testdata_wLabels[features]
203     y_pred_loc = pls_models_dct["plsmod{}".format(i)].predict(X)
204
205
206     predicted_colnames = ["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),
207                           "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i)]
208
209     predicted_df = pd.DataFrame(y_pred_loc, columns = predicted_colnames)
210     testdata_wLabels = testdata_wLabels.join(predicted_df)
211
212 # ============================================================================================================
213
214 testdata_wLabels = calculate_weighted_sum(testdata_wLabels)
215
216 testdata_wLabels = create_best_localmodel_prediction_col(testdata_wLabels)
217
218 # ============================================================================================================
219 # PRINT PERFORMANCE
220
221 # -------- Global PLSR model performance on test
222
223 from performance import performance_data
224 performance_data( y_pred_glob, y_test, title="---- Global model ")
225
226 # -------- Local PLSR models performance
227
228
229 # split datapoint into groups based on clusterbelonging
230
231
232
```

```
# check performance from only that cluster
plt.figure(figsize=(20,10))

for i in uniquie_clusters:
    clusterdata = testdata_wLabels[testdata_wLabels['label']==i]
    y_pred = clusterdata[["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),
                          "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i)]]
    y_true = clusterdata[targets]

    #inspect local prediction accuracy
    for p in range(len(y_true.columns)):
        x=y_true[y_true.columns[p]]
        y=y_pred[y_pred.columns[p]]

        plt.subplot(2, 2, p+1)
        plt.scatter(x, y)
        plt.title(y_true.columns[p])


    if y_pred.empty: #If no samples was predicted with local model
        print("--- None in cluster {}".format(i))
        continue

    # Calculate r2 score & mean squared error for local models
    title=("--- Cluster {}").format(i)
    performance_data(y_pred, y_true, title)


# --------  Calculating MSE and R2 for weighted sum

y_pred = testdata_wLabels[['sum_pred_fano','sum_pred_cv',
                           'sum_pred_corr', 'sum_pred_cov']]
y_true = testdata_wLabels[targets]

performance_data(y_pred, y_true, title="--- Weighted sum: ")


# --------  Calculating MSE and R2 for Best_cluster_prediction

y_pred2 = testdata_wLabels[['best_cl_fano', 'best_cl_cv',
                            'best_cl_corr', 'best_cl_cov']]
y_true2 = testdata_wLabels[targets]

performance_data(y_pred2, y_true2, title="--- Best_cluster_prediction: ")

# ============================================================================================================
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 29 13:29:21 2020

@author: Anja Stene, anja.stene@nmbu.no
"""



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
from optimise_PLSR import optimise_pls_cv
from split_data import get_splitted_data
from performance import performance_data
random.seed(100)
np.random.seed(100)


df = pd.read_csv("data_500_params_stats_v5.csv")


def create_local_plsr_models(traindata_wLabels):

    uniquie_clusters = np.unique(traindata_wLabels['label'])

    # collect datapoints belonging to each cluster in separate datasets
    data_split_by_clusters={}

    for i in uniquie_clusters:
        data_split_by_clusters["cluster{}".format(i)] = traindata_wLabels.loc[traindata_wLabels['label'] == i]

    #split to X and y for each cluster
    X_y_in_cl={}

    for i in uniquie_clusters:
        X_y_in_cl["X{}".format(i)] = \
        data_split_by_clusters["cluster{}".format(i)][features]
        X_y_in_cl["y{}".format(i)] = \
        data_split_by_clusters["cluster{}".format(i)][targets]


    pls_models_dct={}
    for i in uniquie_clusters:
        Xn = X_y_in_cl["X{}".format(i)]
        max_ncomps = len(Xn.columns)
        #Xn = scaler_pls.transform(Xn)
        yn = X_y_in_cl["y{}".format(i)]

        pls_models_dct["plsmod{}".format(i)] = optimise_pls_cv(Xn, yn, max_ncomps, plot_components=True, title='PLSR for cluster {}'.format(i))

    return pls_models_dct


def create_best_localmodel_prediction_col(testdata_wLabels):

    ## Adding the prediction from the best results in as "best_cl_" to data_w_labels

    m = pd.DataFrame([])

    for i in uniquie_clusters:
        best_cl_data = testdata_wLabels.loc[testdata_wLabels['label']==i]

        best_cl_preds = best_cl_data[["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),
                                      "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i) ]].copy()

        best_cl_preds.columns = ['best_cl_fano', 'best_cl_cv', 'best_cl_corr', 'best_cl_cov']

        m=m.append(best_cl_preds)


    testdata_wLabels = testdata_wLabels.join(m)

    return testdata_wLabels
```

```
def calculate_weighted_sum(testdata_wLabels):

    ## Create Weighted sums

    weighted_sum_fano = 0
    weighted_sum_cv = 0
    weighted_sum_corr = 0
    weighted_sum_cov = 0

    for i in uniquie_clusters:
        weighted_sum_fano += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_fano_{}".format(i)])
        weighted_sum_cv += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_cv_mean_{}".format(i)])
        weighted_sum_corr += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_corr_mean_{}".format(i)])
        weighted_sum_cov += (testdata_wLabels[int("{}".format(i))] * testdata_wLabels["pred_cov_mean_{}".format(i)])

    testdata_wLabels['sum_pred_fano'] = weighted_sum_fano
    testdata_wLabels['sum_pred_cv'] = weighted_sum_cv
    testdata_wLabels['sum_pred_corr'] = weighted_sum_corr
    testdata_wLabels['sum_pred_cov'] = weighted_sum_cov

    return  testdata_wLabels
# ============================================================================================================


split = 0.2               #Train-test split to use
seed = 10                 #random seed
ncenters = 3              #number of cluster centers to use in HC-PLSR
f = 2                     #Fuzzyfier component
components_to_cluster = 3   #Number of Principal Components to peform clustering on
score = 'x'               #Cluster on the X-Scores
#score = 'y'              #Cluster on the Y-Scores

# ============================================================================================================


X_train, y_train, X_test, y_test, data = get_splitted_data(df, split)


targets = ['cv_mean','fanofactor', 'corr_mean', 'cov_mean']
#y_train = data_train[targets]

# ============================================================================================================


from dfs import get_DFS

X_train_dfs, X_test_dfs = get_DFS(X_train, X_test)

#Assign DFS versions to X_train/test
X_train = X_train_dfs.copy()
X_test = X_test_dfs.copy()

original_features = ['g', 'eta', 'J', 'D']
features = X_train.columns


# Global model
plsmod_full_dfs = optimise_pls_cv(X_train, y_train, len(X_train.columns), plot_components=True)

plsmod_original_features = optimise_pls_cv(X_train[original_features], y_train, len(original_features), plot_components=True)

#Extracting the x_scores from the global plsrmodel to cluster
#pls_x_scores = pd.DataFrame(plsmod_full_dfs.x_scores_)
#pls_y_scores = pd.DataFrame(plsmod_full_dfs.y_scores_)

pls_x_scores = pd.DataFrame(plsmod_original_features.x_scores_)
pls_y_scores = pd.DataFrame(plsmod_original_features.y_scores_)

# ============================================================================================================


from fuzzy_cluster import fuzzy_cluster, fuzzy_prediction

if score == 'y':
    pls_scores = pls_y_scores
else:
    pls_scores = pls_x_scores
```

```
traindata_wLabels, scaler_cluster, cntr =  fuzzy_cluster(pls_scores, X_train.copy(), y_train, components_to_cluster, ncenters, f, seed)
uniquie_clusters = np.unique(traindata_wLabels['label'])
# ================================================================================================================


pls_models_dct = create_local_plsr_models(traindata_wLabels)

# ================================================================================================================
#Plotting original data for structure inspections after labeling

import seaborn as sns; #sns.set(style="ticks", color_codes=True)

p = sns.pairplot(traindata_wLabels, x_vars=['g', 'eta', 'J', 'D'],
            y_vars = ['fanofactor','cv_mean','corr_mean', 'cov_mean'], hue="label", palette="Set2") # hue="cluster",


# ------------------------------------------------------------------------------
## test data


"""X_test_sc = scaler_pls.transform(X_test)
X_test_sc = pd.DataFrame(X_test_sc, columns = features )"""

#prediction  using global PLSR-model
y_pred_glob = plsmod_full_dfs.predict(X_test)
y_pred_small = plsmod_original_features.predict(X_test[original_features])
performance_data( y_pred_small, y_test, title="---- original data plsr model ")

#Projecting/transforming X-data onto components from PLSR
#transf_X, transf_y = plsmod_full_dfs.transform(X_test, y_pred_glob
#transf_X, transf_y = plsmod_original_features.transform(X_test[original_features], y_pred_glob)
transf_X, transf_y = plsmod_original_features.transform(X_test[original_features], y_pred_small)

if score == 'y':
    df_transf = pd.DataFrame(transf_y.copy())
else:
    df_transf = pd.DataFrame(transf_X.copy())

testdata_wLabels = fuzzy_prediction(df_transf, X_test, y_test, scaler_cluster, components_to_cluster, cntr, f, seed, title="fuzzy_prediction TEST")


# ================================================================================================================


## predict using local plsr model and sum up using weights

#For all local models: Predict sample
# calculate weighted sum based on clusterbelonging and pred value from all local plsr models


for i in uniquie_clusters: #or for local_plsmodel in pls_model_dct:
    #split data belonging to certain cluster into x and y
    X = testdata_wLabels[features]
    y_pred_loc = pls_models_dct["plsmod{}".format(i)].predict(X)


    predicted_colnames = ["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),
                          "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i)]

    predicted_df = pd.DataFrame(y_pred_loc, columns = predicted_colnames)
    testdata_wLabels = testdata_wLabels.join(predicted_df)

# ================================================================================================================

testdata_wLabels = calculate_weighted_sum(testdata_wLabels)

testdata_wLabels = create_best_localmodel_prediction_col(testdata_wLabels)

# ================================================================================================================
# PRINT PERFORMANCE

# -------- Global PLSR model performance on test

from performance import performance_data
performance_data( y_pred_glob, y_test, title="---- Global model ")

#Local PLSR models performance

```

```
233
234 # split datapoint into groups based on clusterbelonging
235 # check performance from only that cluster
236 plt.figure(figsize=(20,10))
237
238 for i in uniquie_clusters:
239     clusterdata = testdata_wLabels[testdata_wLabels['label']==i]
240     y_pred = clusterdata[["pred_fano_{}".format(i), "pred_cv_mean_{}".format(i),
241                           "pred_corr_mean_{}".format(i), "pred_cov_mean_{}".format(i)]]
242     y_true = clusterdata[targets]
243
244     #inspect local prediction accuracy
245     for p in range(len(y_true.columns)):
246         x=y_true[y_true.columns[p]]
247         y=y_pred[y_pred.columns[p]]
248
249         plt.subplot(2, 2, p+1)
250         plt.scatter(x, y)
251         plt.title(y_true.columns[p])
252
253
254     if y_pred.empty: #If no samples was predicted with local model
255         print("--- None in cluster {}".format(i))
256         continue
257
258     # Calculate r2 score & mean squared error for local models
259     title=("--- Cluster {}").format(i)
260     performance_data(y_pred, y_true, title)
261
262
263 # --------  Calculating MSE and R2 for weighted sum
264
265 y_pred = testdata_wLabels[['sum_pred_fano','sum_pred_cv',
266                            'sum_pred_corr', 'sum_pred_cov']]
267 y_true = testdata_wLabels[targets]
268
269 performance_data(y_pred, y_true, title="--- Weighted sum: ")
270
271
272 # --------  Calculating MSE and R2 for Best_cluster_prediction
273
274 y_pred2 = testdata_wLabels[['best_cl_fano', 'best_cl_cv',
275                             'best_cl_corr', 'best_cl_cov']]
276 y_true2 = testdata_wLabels[targets]
277
278 performance_data(y_pred2, y_true2, title="--- Best_cluster_prediction: ")
279
280 # ================================================================================================
281
282
283
284 import pkg_resources
285 installed_packages = pkg_resources.working_set
286 installed_packages_list = sorted(["%s==%s" % (i.key, i.version)
287    for i in installed_packages])
288 print(installed_packages_list)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 29 16:37:34 2020

@author: Anja Stene, anja.stene@nmbu.no
"""


import skfuzzy as fuzz
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

def fuzzy_cluster(pls_scores, data_wLabels, y_train, components_to_cluster,ncenters, f, seed,  title="fuzzy_Clustering results"):

    ### FUZZY C MEANS CLUSTERING


    # Cluster on scores
    pls_scores = pd.DataFrame(pls_scores.copy())
    length = len(pls_scores.columns)
    columns = ['sc{}'.format(i) for i in range(length)]
    pls_scores.columns = columns

    #components_to_cluster = 3# =length
    to_cluster = columns[:components_to_cluster]

    cluster_data_train = pls_scores[to_cluster] #pls_y_scores
    colnames=cluster_data_train.columns
    scaler_cluster = StandardScaler()
    cluster_data_train = scaler_cluster.fit_transform(cluster_data_train)
    cluster_data_train = pd.DataFrame(cluster_data_train, columns=colnames)

    #2. Inspect for optimal number of clusters? based on fcp

    error=0.005
    maxiter=1000

    fpcs=[]

    for i in range(2,15):
        _, _, _, _, _, _, fpc = fuzz.cluster.cmeans(
            cluster_data_train.T, i, f, error=error, maxiter=maxiter, init=None)
        fpcs.append(fpc)

        plt.title("FCP for #Clusters")
        #plt.plot(fcp)
    fig2, ax2 = plt.subplots()
    ax2.plot(np.r_[2:len(fpcs)+2], fpcs)
    ax2.set_xlabel("Number of centers")
    ax2.set_ylabel("Fuzzy partition coefficient")


    #3. create fuzzy C model with given number of clusters

    cntr, u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(
        cluster_data_train.T, ncenters, f, error=error, maxiter=maxiter, init=None, seed=seed)

    #Inspect cluster belongings
    cluster_membership = np.argmax(u_orig, axis=0)
    u_df=pd.DataFrame(u_orig)
    weights_df = u_df.T
    weights_df = weights_df.join(pd.DataFrame(cluster_membership, columns = ['label']))
    print("weights")
    print(weights_df)

    # Add weights to training data (scores matrix that were clustered)
```

```
    labeled_scores = cluster_data_train.copy()
    labeled_scores = labeled_scores.join(weights_df)

    ## -- plot clusters from training fuzzy C

    # ------- Plot
    plt.figure(figsize=(20,10))
    ax2 = plt.axes(projection='3d')

    markers = ['^', 'o', 'P', 'D', 's', 'd', 'X', '>']
    colors = ['darkgreen', 'plum', 'goldenrod','skyblue', 'mediumblue', 'limegreen', 'orangered', 'black']

    # Data for three-dimensional scattered points
    for label in (np.unique(labeled_scores['label'])):
        df = labeled_scores.loc[labeled_scores['label']==label].copy()

        xdata = df['sc2'] #0
        ydata = df['sc1'] #1
        zdata = df['sc0'] #2

        ax2.scatter3D(xdata, ydata, zdata, c=colors[label], marker = markers[label])

    ax2.set_xlabel('PC2')
    ax2.set_ylabel('PC1')
    ax2.set_zlabel('PC0')

    plt.title(title)
    plt.show()


    # for fuzzy C Means (Labeling)
    data_wLabels = data_wLabels.reset_index()
    data_wLabels = data_wLabels.join(weights_df)
    data_wLabels = pd.concat([data_wLabels,y_train.reset_index()], axis=1, sort=False)

    return data_wLabels.copy(), scaler_cluster, cntr


def fuzzy_prediction(pls_scores, data_wLabels, y_test, scaler_cluster, components_to_cluster, cntr, f, seed, title="fuzzy_prediction"):


    ### FUZZY C MEANS CLUSTERING

    #4. Predict new cluster membership with `cmeans_predict` as well as
    # `cntr` from the 4-cluster model

    length = len(pls_scores.columns)
    columns = ['sc{}'.format(i) for i in range(length)]
    pls_scores.columns = columns
    to_cluster = columns[:components_to_cluster]


    cluster_data_test = pls_scores[to_cluster]
    cluster_data_test = scaler_cluster.transform(cluster_data_test)
    cluster_data_test = pd.DataFrame(cluster_data_test, columns = to_cluster)


    # ------ use old model, label predicted data and plot clustered scores
    u, u0, d, jm, p, fpc = fuzz.cluster.cmeans_predict(
        cluster_data_test.T, cntr, f, error=0.005, maxiter=1000, seed=seed)


    #Inspect cluster belongings
    u_df=pd.DataFrame(u)
    cluster_membership = np.argmax(u, axis=0)


    weights_df = pd.DataFrame(u_df.T)
    weights_df = weights_df.join(pd.DataFrame(cluster_membership, columns = ['label']))

    labeled_scores = cluster_data_test.copy()
```

```
142     labeled_scores = labeled_scores.join(weights_df)
143
144     # ------- Plot
145     plt.figure(figsize=(20,10))
146     ax2 = plt.axes(projection='3d')
147
148     markers = ['^', 'o', 'P', 'D', 's', 'd', 'X', '>']
149     colors = ['darkgreen', 'plum', 'goldenrod','skyblue', 'mediumblue', 'limegreen', 'orangered', 'black']
150
151     # Data for three-dimensional scattered points
152     # df.loc[df['Color'] == 'Green']
153     for label in (np.unique(labeled_scores['label'])):
154         df = labeled_scores.loc[labeled_scores['label']==label].copy()
155         print("df.shape", df.shape)
156
157         xdata = df['sc2'] #0
158         ydata = df['sc1'] #1
159         zdata = df['sc0'] #2
160
161         ax2.scatter3D(xdata, ydata, zdata, c=colors[label], marker = markers[label], label="cluster{}".format(label))
162
163     ax2.set_xlabel('PC2')
164     ax2.set_ylabel('PC1')
165     ax2.set_zlabel('PC0')
166
167     plt.title(title)
168     plt.show()
169
170     #Adding labels and clusterweights to testdata (original, not projected)
171     data_wLabels = data_wLabels.reset_index()
172     data_wLabels = data_wLabels.join(weights_df)
173     data_wLabels = pd.concat([data_wLabels,y_test.reset_index()], axis=1, sort=False)
174
175     return data_wLabels.copy()
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 27 22:50:47 2020

@author: Anja Stene, anja.stene@nmbu.no
"""


from sys import stdout
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
import warnings

def optimise_pls_cv(X, y, n_comp, scale=True, plot_components=False, to_be_minimized='mse', title = "PLS"):
    '''Run PLS including a variable number of components, up to n_comp,
       and calculate MSE '''
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")

        mse = []
        component = np.arange(1, n_comp)  #if n_comp then OLS
        for i in component:
            pls = PLSRegression(n_components=i, scale=scale);
            # Cross-validation
            y_cv = cross_val_predict(pls, X, y, cv=10) #10
            mse.append(mean_squared_error(y, y_cv))

        # Calculate and print the position of minimum in MSE
        msemin = np.argmin(mse)
        print("Suggested number of components (MSE): ", msemin+1)

        if to_be_minimized == 'mse':
            minimise = mse
            arg_min=msemin

        stdout.write("\n")
        if plot_components is True:
            with plt.style.context(('ggplot')):
                plt.plot(component, np.array(minimise), '-v', color = 'blue', mfc='blue')
                plt.plot(component[arg_min], np.array(minimise)[arg_min], 'P', ms=10, mfc='red')
                plt.xlabel('Number of PLS components')
                plt.ylabel('MSE')
                plt.title(title)
                plt.xlim(left=-1)
            plt.show()

        # Define PLS object with optimal number of components
        pls_opt = PLSRegression(n_components=arg_min+1, scale=scale)

        # Fit to the entire dataset
        pls_opt.fit(X, y);
        y_c = pls_opt.predict(X)

        # Cross-validation
        y_cv = cross_val_predict(pls_opt, X, y, cv=10)

        # Calculate scores for calibration and cross-validation
        score_c = r2_score(y, y_c)
        score_cv = r2_score(y, y_cv)

```

```
65      # Calculate mean squared error for calibration and cross validation
66      mse_c = mean_squared_error(y, y_c)
67      mse_cv = mean_squared_error(y, y_cv)
68      print('R2 calib: %5.3f'  % score_c)
69      print('R2 CV: %5.3f'  % score_cv)
70      print('MSE calib: %5.3f' % mse_c)
71      print('MSE CV: %5.3f' % mse_cv)
72
73  return pls_opt
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 27 22:53:32 2020

@author: anja.stene
"""


import pandas as pd
import statistics
import re
import numpy as np
import random
random.seed(100)
np.random.seed(100)

from sklearn.model_selection import train_test_split




def get_splitted_data(df, split):



    ## -- Adding CV_mean to Y dataset
    cv_list = df['cv_list']
    cov_mean_list = df['cov_sparsematrix_mean_cols']
    corr_mean_list1 = df['corr_sparsematrix_mean_cols']
    corr_mean_list = corr_mean_list1
    print(corr_mean_list[1])

    #Extracting all floats in list. could not use split() because first and last element has a "[" or a "]" included
    cv_list = [re.findall(r"[-+]?\d*\.\d+|\d+", i) for i in cv_list]
    cov_mean_list = [re.findall(r"[-+]?\d*\.\d+|\d+", i) for i in cov_mean_list]
    corr_mean_list = [re.findall(r"[-+]?\d*\.\d+|\d+", i) for i in corr_mean_list]
    #re.findall(r'[\d\.\d]+', cv_list0) #Less robust?

    #Converting all elements in list to float, before calculating mean, by double list comprehension
    cv_list = [[float(i) for i in p] for p in cv_list]
    cov_mean_list = [[float(i) for i in p] for p in cov_mean_list]
    corr_mean_list = [[float(i) for i in p] for p in corr_mean_list]

    #Computing the mean for each spike train, across all neurons, for all 500 simulations
    cv_mean_list=[statistics.mean(i) for i in cv_list]
    #cov_meanofmean_list=[max(i) for i in cov_mean_list]
    #corr_meanofmean_list=[max(i) for i in corr_mean_list]
    cov_meanofmean_list=[statistics.mean(i) for i in cov_mean_list]
    corr_meanofmean_list=[statistics.mean(i) for i in corr_mean_list]

    #Adding the list of mean cv coefficients to the dataframe used for modelling
    df['cv_mean'] = cv_mean_list
    df['cov_mean'] = cov_meanofmean_list
    df['corr_mean'] = corr_meanofmean_list

    data=df[['g', 'eta', 'J', 'D', 'fanofactor', 'cv_mean','corr_mean', 'cov_mean']]


    targets = ['fanofactor','cv_mean','corr_mean', 'cov_mean']
    y = data[targets]
    X = data[['g', 'eta', 'J', 'D']]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=split, random_state=42, shuffle=True)

```

```
66      X_train = pd.DataFrame(X_train)
67      X_test = pd.DataFrame(X_test)
68      y_train = pd.DataFrame(y_train)
69      y_test = pd.DataFrame(y_test)
70
71      return X_train, y_train, X_test, y_test, data
72
73  #if __name__ == "__main__":
    #    print(get_splitted_data)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 27 22:58:45 2020

@author: Anja Stene, anja.stene@nmbu.no
"""



import featuretools as ft
from featuretools.primitives import make_trans_primitive
from featuretools.variable_types import Numeric
import numpy as np
from sklearn.preprocessing import StandardScaler
import pandas as pd


def get_DFS(X_train, X_test):


    features=X_train.columns

    scaler=StandardScaler(with_mean=False)
    X_train_sc = scaler.fit_transform(X_train)
    X_test_sc = scaler.transform(X_test)

    X_train = pd.DataFrame(X_train_sc, columns=features)
    X_test = pd.DataFrame(X_test_sc, columns=features)


    # Create two new functions for our two new primitives
    def Log(column):
        return np.log(column)
    def Square_Root(column):
        return np.sqrt(column)
    def Square(column):
        return np.square(column)
    def Cube(column):
        return np.power(column, 3)


    # Create the primitives
    log_prim = make_trans_primitive(
        function=Log, input_types=[Numeric], return_type=Numeric)
    square_root_prim = make_trans_primitive(
        function=Square_Root, input_types=[Numeric], return_type=Numeric)
    square = make_trans_primitive(
        function=Square, input_types=[Numeric], return_type=Numeric)
    cube = make_trans_primitive(
        function=Cube, input_types=[Numeric], return_type=Numeric)

    trans_primitives=['add_numeric', 'multiply_numeric']
    trans_primitives.append(log_prim)
    trans_primitives.append(square_root_prim)
    trans_primitives.append(square)
    trans_primitives.append(cube)


    # Make an entityset and add the entity
    es = ft.EntitySet(id = 'v1')
    es.entity_from_dataframe(entity_id = 'e1', dataframe = X_train,
                             make_index = True, index = 'index')


```

```
66  # Run deep feature synthesis with transformation primitives
67  X_train_dfs, feature_defs = ft.dfs(entityset = es, target_entity = 'e1',
68                                     trans_primitives = trans_primitives, max_depth=1)
69
70
71  ## --- DFS for Test data
72  # Make an entityset and add the entity
73  es2 = ft.EntitySet(id = 'v2')
74  es2.entity_from_dataframe(entity_id = 'e1', dataframe = X_test,
75                            make_index = True, index = 'index')
76
77  # Run deep feature synthesis with transformation primitives
78  #X_test_dfs, feature_defs = ft.dfs(entityset = es2, target_entity = 'e1',
79  #                                  trans_primitives = trans_primitives, max_depth=1)
80
81  X_test_dfs2 = ft.calculate_feature_matrix(features = feature_defs, entityset=es2)
82  X_test_dfs = X_test_dfs2
83
84  print("PRINTING", X_test['g'][0], X_test_dfs['g'][0])
85
    return X_train_dfs, X_test_dfs
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 30 12:07:44 2020

@author: anja.stene
"""

from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import pandas as pd

def performance_data(y_pred, y_true, title='performance'):

    y_pred = pd.DataFrame(y_pred)
    y_true = pd.DataFrame(y_true)

    score_glob = r2_score(y_true, y_pred)
    mse_glob = mean_squared_error(y_true, y_pred)
    mae_glob = mean_absolute_error(y_true, y_pred)


    print("==================================== ")
    print(title)
    print('R2 glob: %5.3f'  % score_glob)
    print('MSE glob: %5.3f' % mse_glob)
    print('MAE glob: %5.3f' % mae_glob)
    print(" ")


    return
```

run_exploration_simulation_NEST-Pypet-LHS

August 10, 2020

0.0.1 Running Network simulation on sampled feature space and stores hierarchical data using hdf5-format

Script for implementing LHS (sampling), NEST (network simulation) and Pypet (explolation and hdf5-storing)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from smt.sampling_methods import LHS
```

[2]:
```
def get_lhs_sampling_points(num_sampling_points):

    #AI parameterspace from Brunel
    g_space = [4.5, 6.0]
    eta_space = [1.5, 3.0]
    J_space = [0.05, 0.4]
    D_space = [1.0, 2.5]

    xlimits = np.array([g_space, eta_space, J_space, D_space])
    sampling = LHS(xlimits=xlimits)

    x = sampling(num_sampling_points)

    print(x.shape)
    print(x[:, 0])
    print(x[:, 1])
    print(x[:, 2])
    print(x[:, 3])

    return x[:, 0].tolist(), x[:, 1].tolist(), x[:, 2].tolist(), x[:, 3].
↪tolist()
```

[3]:
```
from elephant.statistics import isi, cv, fanofactor
from elephant.spike_train_correlation import corrcoef, covariance
from elephant.conversion import BinnedSpikeTrain
from neo.core import SpikeTrain
from quantities import Hz, s, ms
```

```
def get_statistics(spiketrains, t_stop):

    cv_list = [cv(isi(spiketrain)) for spiketrain in spiketrains]
    isi_list = [isi(spiketrain) for spiketrain in spiketrains]
    fano_factor = fanofactor(spiketrains)

    spiketrain_list = [SpikeTrain(spiketrain*s, t_stop=t_stop) for spiketrain␣
 ↪in spiketrains]
    binned_sts=BinnedSpikeTrain(spiketrain_list, binsize=10*ms)

    corr_coef = corrcoef(binned_sts, binary=False)
    cov_coef = covariance(binned_sts, binary=False)

    return cv_list, isi_list, fano_factor, corr_coef, cov_coef
```

[4]:
```
from pypet import Environment
import pandas as pd
import numpy as np
import os # To allow file paths working under Windows and Linux
from brunel_delta_ml import sim_brunel_delta
from neo.core import SpikeTrain
from quantities import Hz, s, ms


def run_simulation(g, eta, D, J, simtime, cutoff):

    df, spiketrains = sim_brunel_delta(g=g,
                                eta=eta,
                                J=J,
                                delay=D,
                                simtime=simtime,
                                cutoff=cutoff)
    return df, spiketrains


def my_pypet_wrapper(traj):

    df, spiketrains = run_simulation(traj.g, traj.eta, traj.D, traj.J, traj.
 ↪simtime, traj.cutoff)
    cv_list, isi_list, fanofactor, corr_coef, cov_coef =␣
 ↪get_statistics(spiketrains, traj.simtime)


    traj.f_add_result('$set.$.sim_res_df', df, comment='Result from simulation␣
 ↪i pandas dataframe`')
```

```
    traj.f_add_result('$set.$.cv_list', cv_list, comment='CV, Contains␣
↪coefficient of variation for every spiketrain')
    #traj.f_add_result('$set.$.isi_list', isi_list, comment='List of␣
↪interspikeintervals for all spiketrains')
    traj.f_add_result('$set.$.fanofactor', fanofactor, comment='fanofactor f =␣
↪var(v) / mean(v) where v is a list of the interspike interval variability')
    traj.f_add_result('$set.$.corr_coef', corr_coef, comment='CC, Coefficient␣
↪of correlation matrix, sparse')
    traj.f_add_result('$set.$.cov_coef', cov_coef, comment='CCov, Coefficient␣
↪of covariance')

def add_parameters(traj):
    """Adds all parameters to `traj`
    The parameters to be explored are also added here with
    default value that is equal to function defaults in brunel_delta.py.
    """
    print('Adding Parameters')

    traj.f_add_parameter('simulation.dt', 0.1, comment='Simulation Resolution␣
↪in NEST')
    traj.f_add_parameter('simulation.simtime', 1100.0, comment='Duration of the␣
↪experiment simulation in ms')
    traj.f_add_parameter('neuron.D', 1.5, comment='delay, synapse-delay between␣
↪neurons in ms')
    traj.f_add_parameter('neuron.g', 5.0, comment='Inhibitory synaptic strength␣
↪relative to excitatory')
    traj.f_add_parameter('neuron.eta', 2.0, comment='V ext / V thr')
    traj.f_add_parameter('neuron.epsilon', 0.1, comment='Excitatory Neurons *␣
↪epsilon = nr of synapses per neuron')
    traj.f_add_parameter('neuron.order', 2500, comment='Relative number of␣
↪neurons in network')
    traj.f_add_parameter('neuron.J', 0.1, comment='Synapse weight between␣
↪neurons')
    traj.f_add_parameter('neuron.N_rec', 50, comment='Number of neurons to␣
↪record during simulation')
    traj.f_add_parameter('simulation.num_threads', 10, comment='simulation in␣
↪threads for parallelizing')
    traj.f_add_parameter('simulation.print_report', True, comment='print output␣
↪during simulation')
    traj.f_add_parameter('simulation.stop_input', False, comment='Stop network␣
↪input in simulation after x ms')
    traj.f_add_parameter('simulation.num_sampling_points', 500, comment='Number␣
↪of sampling points in Latin Hypercube Sampling Method')
    traj.f_add_parameter('simulation.cutoff', 100, comment='Cutoff first x ms␣
↪to avoid transient effects, in ms')
```

```
def add_exploration(traj):
    """Explores different values of  g, eta, J and D ."""

    print('Adding exploration of g, eta, J and D')
    g_vals, eta_vals, J_vals, D_vals = get_lhs_sampling_points(traj.
 ↪num_sampling_points)
    explore_dict = {'neuron.g': g_vals,
                    'neuron.eta': eta_vals,
                    'neuron.J': J_vals,
                    'neuron.D': D_vals
                   }

    traj.f_explore(explore_dict)
```

[5]:
```
# Create an environment that handles running
filename = os.path.join('hdf5','biggest_set_updated.hdf5')

env = Environment(filename = filename,
                  overwrite_file = True)
traj = env.traj
```

```
MainProcess pypet.storageservice.HDF5StorageService INFO     I will use the hdf5
file `hdf5/biggest_set_updated.hdf5`.
MainProcess pypet.environment.Environment INFO     Environment initialized.
```

[6]:
```
# Add parameters
add_parameters(traj)


# Let's explore
add_exploration(traj)
```

```
Adding Parameters
Adding exploration of g, eta, J and D
(500, 4)
[5.5335 5.9535 4.6545 5.9955 5.4495 5.0175 5.6175 5.2305 4.8255 5.0145
 5.6715 5.9355 4.8795 4.9395 4.7205 5.4615 5.1435 4.7445 5.2005 5.7945
 5.7825 5.8275 5.7975 5.5695 4.8525 5.4975 4.6905 5.8995 5.9565 4.7655
 5.1165 5.6535 4.9335 5.7135 4.9665 5.4825 4.8075 5.6295 5.9295 5.4675
 4.6665 5.1705 4.8885 4.8765 4.6605 5.0805 5.3535 4.7115 5.4945 5.9625
 5.9895 5.3775 4.9755 5.2065 5.7735 4.8495 4.8915 4.6245 5.4555 4.6395
 4.9215 4.9965 5.8935 5.3205 5.9265 5.1285 5.2875 5.0295 5.2245 5.6835
 5.9805 5.8395 5.2635 4.8165 5.8545 5.1015 5.7315 5.5455 4.8615 5.1915
 5.1645 5.6955 5.4525 4.5705 4.8225 5.5245 4.9485 5.2755 5.5905 5.7345
 5.5365 5.9775 5.1315 5.1105 5.1765 5.9025 5.6235 5.7915 5.4585 5.9055
 5.8485 5.1465 5.8905 5.2905 5.3415 4.8435 5.2485 5.6595 5.4885 5.1405
```

```
 5.0835 5.8755 4.9605 5.8815 4.7295 4.8945 4.6185 4.6575 4.6095 5.9085
 5.5155 5.3085 5.4405 4.7235 5.3985 4.9515 4.6995 5.5305 4.9425 4.7415
 5.0475 5.7495 5.9685 5.8035 4.8735 4.9065 5.8965 4.6005 5.2815 5.6145
 5.8245 5.2965 5.9205 5.3385 5.0715 5.3745 4.5225 5.0505 5.1855 5.5425
 5.6865 4.8855 5.1975 4.6485 5.4225 5.3115 5.0445 5.6775 5.7525 5.5605
 4.5135 5.2155 4.5255 5.6025 4.8045 5.0745 5.3325 5.9145 5.0205 5.1075
 5.2125 5.2725 4.5465 5.8005 4.7715 4.5405 4.7745 5.8125 5.1885 5.0775
 4.5945 4.7535 4.8315 5.2425 4.7505 4.5555 5.5575 5.3955 5.4915 4.9725
 5.5035 5.5935 5.6055 5.0055 5.8425 5.3505 5.0325 5.5965 5.2455 5.4165
 4.8555 5.8725 5.4765 4.7565 4.9185 4.6845 4.7985 5.8305 5.4465 5.4435
 4.6305 4.7325 4.5315 5.3715 4.9545 4.8285 5.2335 4.6365 5.7645 5.2395
 5.3925 4.7175 5.4855 5.1045 5.7705 5.3835 4.9875 5.4075 5.1135 5.6655
 4.8105 5.0895 4.6785 4.5375 4.9035 5.6745 5.9985 5.2365 5.2665 4.6125
 5.7075 5.7795 5.5185 4.9695 5.7375 4.7265 5.1555 5.3055 5.3865 5.6445
 4.7625 5.8875 5.7015 4.5615 5.4735 5.7855 5.6505 5.3445 4.5495 5.2845
 5.1585 5.8095 5.5515 4.7385 5.2185 4.6335 4.7145 5.8155 5.6325 5.8605
 5.2035 5.5545 4.9245 5.5395 5.7555 5.3595 5.8635 4.8825 4.9935 5.5005
 5.3625 4.9575 4.5915 4.7775 4.5975 5.2545 4.9125 5.0655 5.7435 4.8405
 5.3355 4.6935 5.1795 4.9845 5.9925 5.2995 5.4105 5.5815 4.6035 5.4255
 4.5045 5.4345 4.5855 5.8575 5.6625 4.9305 4.6515 5.1495 4.9275 5.0265
 5.2605 5.9715 5.1345 5.8845 4.7475 4.8195 5.0385 4.7595 5.0565 5.9115
 4.6275 4.7355 4.5195 5.7465 5.5995 4.5525 5.6895 5.1675 5.6925 5.5065
 5.1375 5.8665 4.9905 4.9095 5.8695 4.5105 5.1945 5.2575 4.5795 5.6415
 5.6565 5.6985 5.5125 4.6215 4.5645 5.3145 5.6265 5.3295 5.9745 5.5845
 4.7085 5.4375 4.7025 4.7925 4.8585 4.7955 5.3235 5.3175 4.6875 4.8345
 5.9325 4.5825 5.2515 5.6685 4.7805 4.5345 5.2695 4.5585 4.5075 5.2275
 5.0955 5.0085 4.9365 5.4285 4.6965 4.6155 5.0535 4.5675 5.2785 5.7255
 4.9155 5.3895 5.9175 5.8785 5.7615 4.8465 4.8705 4.8135 5.0685 5.1735
 5.3655 5.8215 5.6385 5.7165 4.9635 5.0985 5.4015 5.8185 5.7405 5.4045
 4.6455 5.9445 5.0865 5.1615 4.9815 4.8375 5.6355 5.3565 4.7685 5.7675
 4.6635 5.9475 4.7895 4.6695 5.0415 5.6085 5.3685 5.0025 5.8065 5.3025
 5.9505 5.8335 5.9235 4.5285 5.1255 4.6725 5.5485 5.9385 5.3475 5.7585
 5.7225 5.1225 4.5435 4.5015 5.5275 5.0235 4.8675 5.0625 4.6065 5.2095
 5.3265 4.6815 5.4135 5.6805 5.0925 5.7045 5.9865 5.5095 5.9415 5.6205
 5.5665 4.7835 5.5755 5.4195 4.5765 5.5875 5.1195 5.7195 5.8365 4.8015
 5.7105 5.9655 5.5215 5.4645 4.8645 5.0355 5.5635 4.7865 5.9595 4.9995
 4.9005 5.0115 5.8515 5.3805 4.9785 5.2215 4.5735 4.8975 5.5725 5.7765
 4.6755 4.9455 5.6475 5.7885 4.6425 5.6115 5.0595 5.4705 5.4315 5.1525
 5.5785 5.7285 5.9835 5.8455 5.2935 4.5165 5.1825 4.5885 5.4795 4.7055]
[2.9355 1.5255 2.2725 2.0655 2.3985 1.9125 2.5335 1.6725 1.6335 2.9745
 2.4615 2.9565 1.5165 2.0835 2.2335 2.3055 2.5305 2.6385 2.9685 1.7595
 1.7745 2.6955 2.9805 2.9115 2.7345 1.9515 2.3625 2.0415 2.8005 2.9955
 2.4705 1.5945 2.0355 2.4405 1.6275 1.9845 1.5285 2.8305 2.8785 1.8765
 1.9785 2.7225 1.7835 2.1015 2.8125 2.5755 2.2875 2.1255 2.1165 1.9035
 2.8185 2.8725 2.6835 2.3535 2.8515 1.7565 2.0625 2.1075 2.3385 2.0715
 1.9275 2.0985 2.7675 2.3445 1.6785 2.6325 2.2755 1.9815 2.7105 2.3265
 2.9625 2.8905 2.7375 1.5435 2.6775 2.8755 1.8705 2.9175 2.1135 1.9965
 2.0325 2.6715 2.0535 1.5795 2.8605 2.5935 2.7405 2.4825 2.0385 2.7135
```

```
 2.8935 2.5035 2.4945 2.9025 2.3655 2.6265 2.0175 2.1735 1.7235 2.0505
 2.4885 2.1315 1.5375 1.5345 1.6065 2.9265 2.7585 2.2575 1.7055 2.2125
 2.4135 2.4195 1.6575 2.8395 2.4435 2.5875 1.5075 2.0685 2.5845 2.9205
 2.9145 2.5605 1.5915 2.7855 1.9605 1.7475 2.4675 2.1825 2.8695 1.6995
 2.6115 2.3325 2.3295 1.8615 1.6245 2.6595 1.6455 2.3685 2.6145 2.5395
 2.4225 2.0475 1.5045 2.9775 1.5735 2.8575 1.7385 1.5765 1.6485 1.5645
 2.8335 1.7325 2.9085 2.6055 1.8915 1.7985 2.8455 2.6025 2.3355 1.5015
 2.5995 2.6175 2.3805 2.6205 1.8105 1.8525 2.6475 2.1585 2.4795 1.7265
 2.7075 2.2485 2.0445 2.2305 1.6155 2.4165 1.9365 1.5615 2.2215 2.6685
 2.6895 2.2275 2.2455 2.5725 2.7795 2.3205 1.6395 2.9925 2.0265 1.7925
 2.1345 1.8975 2.1375 2.4495 2.6985 2.0295 2.6085 2.7975 1.8435 2.1615
 1.7865 2.2695 2.4645 2.8485 1.8675 1.9635 1.9425 1.7175 2.5215 1.9215
 1.8375 2.6865 1.8225 2.0895 2.0025 2.2035 2.6415 2.5485 2.8275 2.7765
 2.3175 2.1795 2.5095 2.2425 1.6035 1.5555 1.8645 2.3565 1.7685 1.9245
 2.4585 2.6235 1.6665 2.7555 1.6365 2.2635 2.5695 2.7285 2.0925 1.8735
 2.5905 2.2395 1.9335 2.9835 1.7085 2.2515 2.2545 1.9095 2.5065 2.5575
 1.6815 2.5005 2.9235 2.4765 2.9295 1.6905 1.8015 2.8365 2.3505 2.0775
 2.5185 2.9865 1.8135 1.6635 1.8195 1.6005 2.1975 1.9065 1.7355 2.7255
 1.6965 1.9695 2.6565 2.4555 1.8165 1.7805 1.9995 2.1555 2.0145 1.9005
 2.1105 2.4975 1.5495 1.6215 2.1645 2.2005 1.7295 1.6695 2.4465 1.8345
 2.3955 2.6295 2.7525 1.9875 1.6875 2.5455 2.0205 2.9715 2.4855 2.8545
 2.0115 1.5975 2.6445 1.9935 2.7885 2.2095 1.9545 2.5515 2.5275 1.5525
 2.5155 2.8845 1.6515 2.2815 1.9755 2.1765 2.0235 2.1945 2.8245 2.1885
 2.3745 2.6805 1.9155 2.0595 1.5135 2.8065 1.8795 2.2065 1.7205 2.8875
 2.5425 2.5125 2.2995 2.5965 2.7645 1.5315 1.9905 2.2365 2.9325 1.5885
 1.6935 2.2605 1.7655 2.7045 2.3115 2.4015 2.9475 2.1525 2.2845 1.6305
 2.0805 1.5225 2.7465 1.7955 1.8555 1.6605 2.7705 1.8885 2.7825 1.7715
 2.7735 2.1855 2.6925 2.2935 1.5465 2.1195 2.9055 2.2965 2.2785 1.5585
 2.9505 2.4345 1.6125 1.6755 1.9575 2.8035 2.6745 2.1225 1.7895 2.7315
 1.5705 2.0055 2.3475 2.4525 2.4285 1.6425 2.4105 2.8425 2.7495 2.7615
 2.9595 2.0865 1.9395 2.5665 2.6655 2.4375 1.8855 2.2905 2.8635 2.4915
 2.5785 2.0745 2.1675 2.1915 2.4315 1.8285 1.6845 2.5245 2.7015 2.7915
 1.8495 2.5545 2.1495 2.3595 2.3925 2.6355 2.5365 2.9655 2.3895 1.6545
 1.8465 2.4255 2.5815 2.1705 2.0955 2.1465 1.8945 2.3025 2.1285 1.5405
 1.5105 1.8315 1.9305 2.2185 1.9485 2.4075 1.8045 1.5825 2.8665 2.8095
 2.3085 2.3865 2.1435 1.6185 1.7505 2.9415 1.7445 2.7195 2.3145 2.8155
 2.9535 2.8965 2.0565 2.6625 1.8255 1.7535 2.5635 2.7945 1.8825 2.4045
 2.3715 2.8215 1.8585 2.4735 2.2155 1.7625 1.7025 1.7145 2.3235 1.5675
 2.9445 1.7775 1.9185 2.8995 2.9895 2.2665 1.9455 2.9985 1.7115 2.3415
 1.8075 1.8405 2.7165 1.6095 2.1045 2.6535 1.9665 2.9385 1.7415 1.9725
 2.2245 1.5195 1.5855 2.8815 2.0085 2.6505 2.7435 2.3775 2.1405 2.3835]
[0.31775 0.08535 0.13995 0.18615 0.39195 0.09585 0.30375 0.36395 0.13645
 0.09165 0.34015 0.19805 0.13085 0.29115 0.11685 0.36255 0.38005 0.38145
 0.36045 0.29955 0.20995 0.32335 0.38285 0.32895 0.33245 0.32475 0.36535
 0.30725 0.25125 0.05385 0.33315 0.18965 0.07345 0.33385 0.39405 0.34505
 0.30445 0.37095 0.25195 0.07835 0.14765 0.05595 0.05665 0.09095 0.14345
 0.32055 0.08955 0.29325 0.08885 0.09935 0.06365 0.31985 0.37935 0.25055
 0.29535 0.11265 0.27645 0.09795 0.16515 0.27855 0.31145 0.24495 0.07275
```

0.17075 0.30585 0.20505 0.12595 0.15885 0.21135 0.24355 0.31705 0.24425
0.38775 0.21275 0.12945 0.13715 0.14065 0.25475 0.09375 0.18685 0.32685
0.34575 0.27995 0.33455 0.20225 0.37305 0.07065 0.23305 0.32125 0.12805
0.05945 0.27925 0.15955 0.32965 0.15185 0.06505 0.11335 0.11475 0.37655
0.18895 0.26945 0.35135 0.17635 0.30305 0.25965 0.23025 0.23795 0.32405
0.20155 0.22675 0.12315 0.30515 0.22115 0.31355 0.10915 0.20715 0.19665
0.27015 0.07205 0.33805 0.17145 0.06645 0.20015 0.29185 0.24005 0.13435
0.39895 0.13925 0.36465 0.22465 0.37725 0.22885 0.24075 0.10145 0.25405
0.35625 0.38915 0.33945 0.12665 0.26035 0.23935 0.09865 0.17915 0.05245
0.06925 0.37375 0.28695 0.16375 0.37865 0.28765 0.27155 0.27505 0.26805
0.11755 0.11545 0.19875 0.19735 0.39825 0.35905 0.13155 0.24705 0.30655
0.32545 0.34435 0.09235 0.33735 0.21345 0.09305 0.33665 0.15325 0.08045
0.16935 0.36815 0.33595 0.23865 0.07975 0.38845 0.17425 0.26385 0.05735
0.28625 0.08465 0.14975 0.12105 0.23725 0.19945 0.18125 0.36605 0.30095
0.35835 0.18405 0.18825 0.11055 0.37165 0.35065 0.26735 0.08815 0.35345
0.38215 0.15815 0.10355 0.10705 0.28835 0.06995 0.06785 0.07135 0.15255
0.17495 0.08325 0.07485 0.11965 0.35975 0.18195 0.28275 0.36885 0.06435
0.21835 0.24215 0.35275 0.29815 0.26595 0.09025 0.10775 0.13855 0.20435
0.24565 0.33875 0.34855 0.13365 0.14555 0.25895 0.21905 0.15045 0.39615
0.29885 0.26665 0.08745 0.32615 0.07765 0.24635 0.08115 0.31285 0.29395
0.18265 0.28485 0.22535 0.18545 0.26105 0.25545 0.28135 0.29745 0.21625
0.34785 0.39965 0.12735 0.19525 0.19385 0.23165 0.31215 0.27435 0.20365
0.19455 0.38425 0.16445 0.19315 0.27085 0.39685 0.09725 0.33035 0.39475
0.35415 0.29605 0.13295 0.17845 0.14275 0.06575 0.38705 0.22185 0.14205
0.26245 0.09515 0.32195 0.27365 0.22745 0.11125 0.37795 0.24775 0.26175
0.18055 0.27715 0.13785 0.24145 0.22255 0.22955 0.09655 0.13225 0.24285
0.15605 0.28345 0.07905 0.28065 0.37515 0.10845 0.10985 0.05875 0.23235
0.06715 0.11195 0.16585 0.09445 0.06855 0.24845 0.31915 0.19035 0.34155
0.34295 0.06015 0.07695 0.22045 0.15395 0.21205 0.12245 0.38495 0.17985
0.08675 0.31565 0.25685 0.22395 0.25335 0.17215 0.11405 0.28975 0.31845
0.30795 0.06155 0.12875 0.14695 0.10215 0.38075 0.19105 0.38985 0.23585
0.10495 0.14625 0.27575 0.08395 0.21975 0.17705 0.16095 0.15535 0.35765
0.08185 0.37235 0.34645 0.18755 0.36115 0.05455 0.28415 0.23655 0.25265
0.27225 0.14135 0.36325 0.07555 0.23375 0.10075 0.25615 0.36675 0.20855
0.32265 0.24915 0.13015 0.20925 0.20575 0.17565 0.11895 0.16725 0.28555
0.12385 0.26455 0.15745 0.28905 0.31075 0.16025 0.23515 0.21415 0.30935
0.21485 0.26525 0.28205 0.33175 0.10285 0.10635 0.30165 0.07415 0.10565
0.16865 0.26875 0.38355 0.35695 0.05035 0.10005 0.34365 0.29675 0.14905
0.17005 0.20295 0.31635 0.35555 0.15675 0.34925 0.21555 0.16305 0.32825
0.06225 0.05805 0.12455 0.39265 0.38565 0.37585 0.20085 0.05175 0.26315
0.12525 0.24985 0.32755 0.05525 0.33105 0.21065 0.23445 0.17285 0.20785
0.30865 0.05105 0.22605 0.19595 0.16235 0.36185 0.30025 0.20645 0.22325
0.34085 0.12035 0.34225 0.18475 0.39125 0.31005 0.11615 0.06085 0.14485
0.13505 0.10425 0.29465 0.31425 0.30235 0.36745 0.18335 0.12175 0.16655
0.19175 0.21765 0.33525 0.14415 0.34715 0.07625 0.29045 0.19245 0.13575
0.39755 0.36955 0.39545 0.22815 0.17775 0.25755 0.38635 0.15115 0.25825
0.27785 0.27295 0.15465 0.05315 0.29255 0.39335 0.16165 0.11825 0.37025
0.08255 0.34995 0.39055 0.35485 0.31495 0.14835 0.08605 0.21695 0.17355

```
 0.23095 0.35205 0.16795 0.37445 0.06295]
[2.3155 2.0305 1.7455 1.5595 1.0225 2.3065 2.1895 1.1425 2.0455 2.0125
 2.0515 1.6975 1.9825 1.6435 1.2115 2.3095 1.5985 2.3875 2.3845 1.6915
 1.8235 1.4395 2.0695 1.2865 1.0465 2.0875 1.1575 1.8325 1.7155 1.4335
 1.7485 2.4355 1.1395 1.3615 2.4985 2.1355 1.5175 1.8625 2.1085 1.9255
 2.3695 1.3975 1.3555 1.9975 1.2685 1.4425 1.2805 1.7545 2.2495 2.3335
 1.6765 1.3285 1.2205 2.1955 1.8445 1.0975 1.0525 1.3915 1.9375 1.0585
 1.8595 1.9105 1.5925 1.6885 1.7575 2.3485 2.4805 2.3605 1.8685 2.1985
 1.2145 1.3645 1.4605 1.6945 1.9705 1.5835 1.5295 2.3755 1.1095 1.5685
 1.8925 1.0135 1.4275 1.9075 1.4845 1.7395 1.6255 2.1685 1.9675 1.5235
 1.1755 1.3885 1.0435 2.2915 1.1665 1.1365 1.2385 2.1625 2.2675 1.7245
 2.2765 1.1725 1.1305 2.1175 1.4965 1.0915 2.1835 1.0405 2.2195 2.0485
 1.7665 1.0555 1.4665 1.9945 1.8655 2.4685 2.3395 1.9045 2.3575 2.4025
 2.1205 1.5505 2.4055 1.8145 2.4235 2.4445 2.3275 1.4635 1.3945 1.4815
 1.3855 1.3825 1.7755 1.4575 1.9885 2.3635 1.6555 1.7185 2.2885 2.0365
 1.9915 1.1335 1.1275 1.6795 1.0825 2.4895 1.6825 2.3545 1.8505 1.7125
 1.6615 2.3365 2.2735 2.1055 1.5325 1.4725 2.0815 1.2445 1.5205 2.4835
 1.7695 2.0635 1.3735 2.1805 1.2265 2.2615 1.9405 1.9735 2.4625 1.1005
 1.2985 1.8565 1.3015 1.7515 2.4205 1.3075 1.2835 2.4775 2.1595 1.9615
 1.1635 1.7035 2.1115 1.0075 1.9135 1.2175 1.1455 2.4265 1.8715 2.0245
 2.0935 1.4455 1.8865 2.0545 2.4655 1.4695 2.0185 1.9225 2.3185 2.4145
 1.7275 1.2025 1.8055 1.0645 1.4785 1.7905 2.0005 1.3165 1.4875 2.2165
 2.4955 1.5805 2.1715 1.8175 2.1775 2.2315 1.9555 1.6405 2.3125 1.1965
 1.1185 1.5655 1.6345 1.1515 2.0095 2.0755 1.0105 2.1475 1.5895 1.1995
 2.4385 2.2585 2.2825 1.0375 1.4035 1.6375 2.1415 2.0065 1.4755 1.6225
 2.2375 1.1485 1.4065 1.8355 2.2345 1.1035 1.0315 1.3765 2.4715 1.2475
 1.9435 2.1235 2.1505 2.4505 1.3105 2.2555 1.0015 2.4925 1.2055 1.0705
 1.3435 2.1865 1.9345 2.1445 1.9195 1.6645 1.0045 1.3225 2.4745 1.3045
 2.2225 1.8985 1.5415 2.2855 1.5715 1.4905 2.2075 2.2405 1.3315 2.1325
 2.3455 1.9285 1.8415 1.5085 1.1695 1.7965 1.7215 1.7365 1.6045 2.0905
 2.1385 1.2535 1.4365 1.6465 2.0215 1.5115 1.2595 1.2625 1.0195 1.5355
 1.6585 1.4185 2.0785 1.7935 1.3705 2.4595 1.2505 2.0965 1.4515 1.5535
 1.2655 1.8025 2.2465 2.1265 1.7425 2.4565 2.4535 2.3005 1.8535 2.1745
 1.3495 1.2955 2.2795 2.3215 2.1295 1.1605 1.3375 1.6135 2.2015 1.5775
 1.8475 2.3725 2.3515 1.3405 1.8205 2.0665 1.8805 1.2415 1.0885 1.8835
 1.0345 1.9315 2.4415 1.1935 1.2295 2.0995 1.5445 1.5055 1.4305 1.5385
 1.1245 1.6015 1.9495 2.3245 1.3675 1.6735 1.2085 1.1215 1.4485 1.5145
 1.4125 2.0725 1.0945 1.7305 1.6165 2.2945 1.7605 1.9015 2.0155 1.6105
 2.3965 1.0165 2.2135 2.3305 1.3585 2.2045 2.3785 1.9525 2.4115 1.3255
 1.5265 1.1545 2.3665 2.3995 1.7845 1.7635 1.2715 1.6285 1.8775 1.0855
 2.0395 1.3195 1.4545 1.2235 1.0255 1.8385 1.8115 1.5865 2.2975 1.8955
 2.1655 2.4865 2.2435 2.3905 2.0275 1.1905 2.4175 1.1785 1.0285 1.8085
 1.2925 1.8265 1.1815 2.3425 1.8895 1.5745 1.7065 2.2255 1.2565 2.1565
 1.9765 1.3525 1.7335 1.6195 2.0845 2.1145 2.4295 1.5955 1.2355 1.4215
 2.2645 2.3035 1.6855 1.5565 1.9585 2.2705 1.6495 1.6315 1.0675 2.1025
 1.2325 1.9795 1.4155 1.9645 2.2525 1.9165 2.0425 1.3135 1.3345 1.3795
 1.0615 2.3815 2.4325 1.8745 1.0765 1.3465 2.0335 2.2105 1.2745 1.7095
 1.9465 1.4935 1.7815 1.1065 2.3935 1.7725 1.7005 1.4995 2.0605 1.4095
```

```
 1.7875 1.5625 1.5475 1.4005 2.0035 1.0795 1.0735 1.1125 1.0495 1.6525
 1.7785 2.2285 1.6675 1.7995 1.2895 1.1155 1.2775 1.9855 1.1875 1.8295
 1.4245 2.1925 1.1845 2.1535 2.4085 2.0575 1.6075 1.5025 1.6705 2.4475]
```

[7]:
```
# Run your wrapping function instead of your simulator
env.run(my_pypet_wrapper)
```

```
MainProcess pypet.environment.Environment INFO     I am preparing the Trajectory
for the experiment and initialise the store.
MainProcess pypet.environment.Environment INFO     Initialising the storage for
the trajectory.
MainProcess pypet.storageservice.HDF5StorageService INFO     Initialising
storage or updating meta data of Trajectory `trajectory`.
MainProcess pypet.storageservice.HDF5StorageService INFO     Finished init or
meta data update for `trajectory`.
MainProcess pypet.environment.Environment INFO
************************************************************
STARTING runs of trajectory
`trajectory`.
************************************************************

MainProcess pypet INFO     PROGRESS: Finished   0/500 runs [
]  0.0%
MainProcess pypet INFO
========================================
 Starting single run #0 of 500
========================================


Building network
Connecting devices
Connecting network
Excitatory connections
Inhibitory connections
Simulating
Brunel network simulation (Python)
Number of neurons : 12500
Number of synapses: 15637600
       Exitatory  : 12512500
       Inhibitory : 3125000
Excitatory rate   : 20.00 Hz
Inhibitory rate   : 20.80 Hz
Building time     : 3.04 s
Simulation time   : 36.14 s
Cutoff first ms   : 100.00

/Users/anjastene/anaconda3/envs/pypetNestFriend/lib/python3.6/site-
packages/pypet/storageservice.py:3110: FutureWarning:Conversion of the second
argument of issubdtype from `str` to `str` is deprecated. In future, it will be
```

create_csv_wStats_from_hdf5

August 10, 2020

0.0.1 Add statistical measures of the spike trains, create a csv of the resulting data of regressors and responses.

Create a csv file containing the data from simulations relevant for model training (i.e. exluding the spiketrains)

Workflow: * import modules * load trajectory for the correct file * run through dataset to pick up values * store everything in a pandas df * export df to csv

[13]:
```
import os
```

[14]:
```
# Create an environment that handles running

# using the same filename as for runninng the simulation
filename = os.path.join('hdf5','biggest_set_updated.hdf5')

# Reload the stored data from above.
# Do not need an environment for that, just a trajectory
from pypet.trajectory import Trajectory

# Create a new trajectory and pass it the path and name of the HDF5 file.

del traj
# Disable logging and close all log-files
env.disable_logging()
del env

traj = Trajectory(filename=filename)

# Load all stored data.
traj.f_load(index=-1, load_parameters=2, load_results=2)
```

[15]:
```
#ADJUSTING THE CORR_COEF AND COV_COEF TO DATAFRAMES
import pandas as pd

df = pd.DataFrame({'g': [], 'eta': [], 'J': [], 'D': []})

for run_name in traj.f_get_run_names():
```

```
    traj.f_set_crun(run_name)
    g=traj.g
    eta=traj.eta
    D=traj.D
    J=traj.J
    cv_list=traj.results.crun.cv_list
    fanofactor=traj.results.crun.fanofactor
    corr_coef=pd.DataFrame(traj.results.crun.corr_coef)
    cov_coef=pd.DataFrame(traj.results.crun.cov_coef)

    if run_name == "run_00000000":
        print(type(cv_list))


    typelist = [type(g), type(eta), type(J), type(D), type(cv_list),
                type(fanofactor), type(corr_coef), type(cov_coef)]

    df = df.append({'g': g, 'eta': eta, 'J': J, 'D': D, 'cv_list': cv_list,␣
↪'fanofactor': fanofactor,
                    'corr_coef': corr_coef, 'cov_coef': cov_coef},␣
↪ignore_index=True)

# Reset your trajectory to the default settings, to release its belief to
# be the last run:
traj.f_restore_default()
```

```
<class 'list'>
```

[16]:
```
df
```

[16]:

	g	eta	J	D	\
0	5.5335	2.9355	0.31775	2.3155	
1	5.9535	1.5255	0.08535	2.0305	
2	4.6545	2.2725	0.13995	1.7455	
3	5.9955	2.0655	0.18615	1.5595	
4	5.4495	2.3985	0.39195	1.0225	
..	…	…	…	…	
495	4.5165	2.6505	0.23095	2.0575	
496	5.1825	2.7435	0.35205	1.6075	
497	4.5885	2.3775	0.16795	1.5025	
498	5.4795	2.1405	0.37445	1.6705	
499	4.7055	2.3835	0.06295	2.4475	

				corr_coef	\
0	0	1	2	3	…
1	0	1	2	3	…
2	0	1	2	3	…

```
3              0         1         2         3     …
4              0         1         2         3     …
..                                                 …
495            0         1         2         3     …
496            0         1         2         3     …
497            0         1         2         3     …
498            0         1         2         3     …
499            0         1         2         3     …

                                                cov_coef  \
0                   0             1              2   …
1                   0             1              2   …
2                   0             1              2   …
3                   0             1              2   …
4                   0             1              2   …
..                                                 …
495                 0             1              2   …
496                 0             1              2   …
497                 0             1              2   …
498                 0             1              2   …
499                 0             1              2   …

                                                 cv_list  fanofactor
0    [0.8994818646428511, 0.8833214891550989, 0.728…    0.682474
1    [0.25165910206812425, 0.3926207436303479, 0.14…    0.080781
2    [0.20762749796857238, 0.21467588525256862, 0.2…    0.043621
3    [0.40877302892258993, 0.4463324448200583, 0.66…    0.234228
4    [0.6672862464314624, 0.9885214444992486, 0.956…    1.054967
..                                                 …         …
495  [0.3763781161989031, 0.45302429910672376, 0.38…    0.175074
496  [0.8478812817357193, 0.8386207340778052, 0.889…    0.873585
497  [0.2535824426631328, 0.24068543901411885, 0.24…    0.088251
498  [0.9498606846375839, 0.8073126445427456, 1.041…    0.655409
499  [0.0647609406282632, 0.07939149021435067, 0.08…    0.013249

[500 rows x 8 columns]
```

[17]:
```python
import numpy as np

corr_sparsematrix_mean_cols = []
cov_sparsematrix_mean_cols = []

for i in range(500):
    corr = df['corr_coef'][i]
    c2 = corr.copy()
    c2.values[np.tril_indices_from(c2)] = np.nan
    corr_sparse_mean = c2.mean().round(4)
```

```
    corr_sparsematrix_mean_cols.append(corr_sparse_mean.tolist())

    cov = df['cov_coef'][i]
    c2 = cov.copy()
    c2.values[np.tril_indices_from(c2)] = np.nan
    cov_sparse_mean = c2.mean().round(4)
    cov_sparsematrix_mean_cols.append(cov_sparse_mean.tolist())

df['cov_sparsematrix_mean_cols'] = cov_sparsematrix_mean_cols
df['corr_sparsematrix_mean_cols'] = corr_sparsematrix_mean_cols

df.to_csv('data_500_params_stats_v5.csv')
```

[32]:
```
#Print example of sparse matrix
listo = df['corr_sparsematrix_mean_cols'][40]
mean=pd.DataFrame(listo).mean()
print(listo)
```

```
[nan, -0.0041, 0.4979, 0.3306, 0.4979, 0.5983, 0.3306, 0.4262, 0.4979, 0.5537,
0.5983, 0.361, 0.4142, 0.5365, 0.5697, 0.3975, 0.4352, 0.4684, 0.4979, 0.4715,
0.4979, 0.5218, 0.4523, 0.4761, 0.4979, 0.518, 0.5365, 0.4421, 0.4621, 0.4806,
0.4979, 0.5141, 0.4665, 0.5131, 0.5266, 0.5408, 0.4423, 0.4573, 0.5242, 0.5364,
0.4478, 0.5345, 0.5456, 0.4396, 0.4524, 0.4646, 0.5196, 0.5299, 0.5397, 0.4468]
```

[12]:
```
df
```

[12]:

	g	eta	J	D	\
0	3.8815	2.5760	2.5760	2.0635	
1	3.9895	2.5056	2.5056	1.3255	
2	3.7505	2.2176	2.2176	1.6165	
3	3.8205	1.1936	1.1936	1.2655	
4	3.7455	2.8960	2.8960	1.1665	
..	…	…	…	…	
495	3.9605	1.8016	1.8016	1.4935	
496	3.9205	2.3776	2.3776	2.3635	
497	3.8105	1.7696	1.7696	1.5595	
498	3.5765	3.2864	3.2864	1.6315	
499	3.9035	3.1904	3.1904	2.2225	

	corr_coef \
0	[[1.0, 0.08769903861511744, 0.0784531101735543…
1	[[1.0, 0.01728911707080802, 0.0328569510763425…
2	[[1.0, 0.2480000894941341, 0.2386142184490019,…
3	[[1.0, 0.30912913506069245, 0.2901624570175456…
4	[[1.0, 0.6359804817245023, 0.3999531350309014,…
..	…
495	[[1.0, 0.03662724949678573, 0.1005310046419901…

```
496  [[1.0, 0.08716013215270216, 0.1196789682146686…
497  [[1.0, 0.14534771774327976, 0.0878020691827465…
498  [[1.0, -0.004277305918350799, -0.0042773059183…
499  [[1.0, 0.1988573568773955, 0.19913232796808136…

                                                cov_coef  \
0    [[0.0040922434832052195, 0.0003561073695711283…
1    [[0.0016155779928577862, 3.275980194861276e-05…
2    [[0.0040922434832052195, 0.0010193388534937094…
3    [[0.003253014614182443, 0.0009985970120389525,…
4    [[0.003920904322270698, 0.002493618619673402, …
..                                                 …
495  [[0.0019326317016105103, 5.24863449171852e-05,…
496  [[0.003550969141041778, 0.00031418731905827243…
497  [[0.0039389428333646, 0.0005751320879776593, 0…
498  [[0.004236482811000761, -1.8139999619831253e-0…
499  [[0.003929923660463029, 0.0007930652262128003,…

                                                 cv_list  fanofactor
0    [0.11955736909306632, 0.1421869977753263, 0.13…    0.019662
1    [4.150413981426069, 2.7431881025313944, 4.1881…   18.849440
2    [0.09357560331217467, 0.07995183704465272, 0.0…    0.008727
3    [0.1796574806985018, 0.18612593963940413, 0.17…    0.045734
4    [0.06651385216671868, 0.06797297812237542, 0.0…    0.002517
..                                                 …         …
495  [2.5284775395765395, 3.792132359368514, 3.0715…   11.909103
496  [0.6181448695096702, 0.593012056537501, 0.3060…    1.014082
497  [0.15317431647710103, 0.14271864634470188, 0.1…    0.019435
498  [0.02210814800649538, 0.022097086912079546, 0…    0.000464
499  [0.18577443915723346, 0.12260881688165026, 0.1…    0.040708

[500 rows x 8 columns]
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


def sim_brunel_delta(dt=0.1,
                     simtime=10.0,
                     delay=1.5,
                     g=5.0,
                     eta=2.0,
                     epsilon=0.1,
                     order=2500,
                     J=0.1,
                     N_rec=50,
                     num_threads=1,
                     print_report=True,
                     input_stop=False,
                     cutoff=0):

    # the following code is based on brunel-delta-nest.py
    # which is part of NEST.
    #
    # Copyright (C) 2004 The NEST Initiative
    # NEST is free software: you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation, either version 2 of the License, or
    # (at your option) any later version.
    #
    # NEST is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    # GNU General Public License for more details.
    #
    # You should have received a copy of the GNU General Public License
    # along with NEST.  If not, see <http://www.gnu.org/licenses/>.

    # This version uses NEST's Connect functions.

    # Link Quickref: https://www.nest-simulator.org/quickref/

    import nest
    import nest.raster_plot
    import time
    from numpy import exp

    nest.ResetKernel()
    startbuild = time.time()

    # Parameters for asynchronous irregular firing
    NE = 4 * order          #=10000 in simulation
    NI = 1 * order          #=2500 in simulation
    N_neurons = NE + NI

    CE = int(epsilon * NE)  # number of excitatory synapses per neuron
    CI = int(epsilon * NI)  # number of inhibitory synapses per neuron
    C_tot = int(CI + CE)  # total number of synapses per neuron
    cutoff = cutoff          # Cutoff to avoid transient effects, in ms

    # Initialize the parameters of the integrate and fire neuron
    tauMem = 20.0
    theta = 20.0

    J_ex = J
    J_in = -g * J_ex

    nu_th = theta / (J * CE * tauMem)
```

```
66  nu_ex = eta * nu_th
67  p_rate = 1000.0 * nu_ex * CE
68
69  if not print_report:
70      nest.set_verbosity('M_WARNING')
71
72  nest.SetKernelStatus({"resolution": dt, "print_time": True,
73                        "local_num_threads": num_threads,
74                        'overwrite_files': True})
75
76  print("Building network")
77
78  neuron_params = {"C_m": 1.0,
79                   "tau_m": tauMem,
80                   "t_ref": 2.0,
81                   "E_L": 0.0,
82                   "V_reset": 0.0,
83                   "V_m": 0.0,
84                   "V_th": theta}
85
86  nest.SetDefaults("iaf_psc_delta", neuron_params)
87
88  nodes_ex = nest.Create("iaf_psc_delta", NE)
89  nodes_in = nest.Create("iaf_psc_delta", NI)
90
91  # Stop input after x = input_stop ms if input_stop is not 0
92  # https://www.nest-simulator.org/helpindex/cc/poisson_generator.html
93  if input_stop:
94      params = {"rate": p_rate, "stop": input_stop}
95  else:
96      params = {"rate": p_rate}
97
98  nest.SetDefaults("poisson_generator", params)
99  noise = nest.Create("poisson_generator")
100
101 espikes = nest.Create("spike_detector")
102 ispikes = nest.Create("spike_detector")
103
104 nest.SetStatus(espikes, [{"label": "brunel-py-ex",
105                           "withtime": True,
106                           "withgid": True,
107                           "to_file": False}])
108
109 nest.SetStatus(ispikes, [{"label": "brunel-py-in",
110                           "withtime": True,
111                           "withgid": True,
112                           "to_file": False}])
113
114 print("Connecting devices")
115
116 nest.CopyModel("static_synapse", "excitatory", {"weight": J_ex, "delay": delay})
117 nest.CopyModel("static_synapse", "inhibitory", {"weight": J_in, "delay": delay})
118
119 nest.Connect(noise, nodes_ex, syn_spec="excitatory")
120 nest.Connect(noise, nodes_in, syn_spec="excitatory")
121
122 nest.Connect(nodes_ex[:N_rec], espikes, syn_spec="excitatory")
123 nest.Connect(nodes_in[:N_rec], ispikes, syn_spec="excitatory")
124
125 print("Connecting network")
126
127 # We now iterate over all neuron IDs, and connect the neuron to
128 # the sources from our array. The first loop connects the excitatory neurons
129 # and the second loop the inhibitory neurons.
130
131 print("Excitatory connections")
```

```
132
133 conn_params_ex = {'rule': 'fixed_indegree', 'indegree': CE}
134 nest.Connect(nodes_ex, nodes_ex + nodes_in, conn_params_ex, "excitatory")
135
136 print("Inhibitory connections")
137
138 conn_params_in = {'rule': 'fixed_indegree', 'indegree': CI}
139 nest.Connect(nodes_in, nodes_ex + nodes_in, conn_params_in, "inhibitory")
140
141 endbuild = time.time()
142
143 print("Simulating")
144
145 nest.Simulate(simtime)
146
147 endsimulate = time.time()
148
149 events_ex = nest.GetStatus(espikes, "n_events")[0]
150 rate_ex = events_ex / simtime * 1000.0 / N_rec
151 events_in = nest.GetStatus(ispikes, "n_events")[0]
152 rate_in = events_in / simtime * 1000.0 / N_rec
153
154 num_synapses = nest.GetDefaults("excitatory")["num_connections"] + \
155             nest.GetDefaults("inhibitory")["num_connections"]
156
157 build_time = endbuild - startbuild
158 sim_time = endsimulate - endbuild
159
160 print("Brunel network simulation (Python)")
161 print("Number of neurons : {0}".format(N_neurons))
162 print("Number of synapses: {0}".format(num_synapses))
163 print("       Exitatory  : {0}".format(int(CE * N_neurons) + N_neurons))
164 print("       Inhibitory : {0}".format(int(CI * N_neurons)))
165 print("Excitatory rate   : %.2f Hz" % rate_ex)
166 print("Inhibitory rate   : %.2f Hz" % rate_in)
167 print("Building time     : %.2f s" % build_time)
168 print("Simulation time   : %.2f s" % sim_time)
169 print("Cutoff first ms   : %.2f " % cutoff)
170
171 #nest.raster_plot.from_device(espikes, hist=True)
172
173 # The function shall return a tuple consisting of the excitatory and
174 # inhibitory spikes recorded, as Pandas data frames:
175 import pandas as pd
176
177 exc_spikes = nest.GetStatus(espikes, 'events')[0]
178 inh_spikes = nest.GetStatus(ispikes, 'events')[0]
179
180
181 # Excitatory spike trains
182 # Makes sure the spiketrain is added even if there are no results
183 # to get a regular result
184 #cutoff = 100
185 events_E = exc_spikes
186 nodes_E = nodes_ex
187 spiketrains = []
188
189 for sender in nodes_E[:N_rec]:
190     spiketrain = events_E["times"][events_E["senders"] == sender]
191     spiketrain = spiketrain[spiketrain > cutoff] - cutoff
192     spiketrains.append(spiketrain)
193
194 #Creating pandas df of spikeevents
195 df_e = pd.DataFrame(exc_spikes)
196 df_i = pd.DataFrame(inh_spikes)
197
```

```
198     #adding spike-type info to dataframe
199     df_e['excitatory'] = 1
200     df_i['inhibitory']= 1
201
202     #Appending dataframes to one big dataframe
203     df=df_e.append(df_i, sort=False, ignore_index=True)
204
205
206     return df, spiketrains
207
208     #return exc_spikes, inh_spikes
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Aug 10 16:58:59 2020

@author: Anja Stene, anja.stene@nmbu.no
"""


def sim_brunel_delta(dt=0.1,
                     simtime=10.0,
                     delay=1.5,
                     g=5.0,
                     eta=2.0,
                     epsilon=0.1,
                     order=2500,
                     J=0.1,
                     N_rec=50,
                     num_threads=1,
                     print_report=True,
                     input_stop=False,
                     cutoff=0):

    # the following code is based on brunel-delta-nest.py
    # which is part of NEST.
    #
    # Copyright (C) 2004 The NEST Initiative
    # NEST is free software: you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation, either version 2 of the License, or
    # (at your option) any later version.
    #
    # NEST is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    # GNU General Public License for more details.
    #
    # You should have received a copy of the GNU General Public License
    # along with NEST.  If not, see <http://www.gnu.org/licenses/>.

    # This version uses NEST's Connect functions.

    # Link Quickref: https://www.nest-simulator.org/quickref/

    import nest
    import nest.raster_plot
    import time
    from numpy import exp

    nest.ResetKernel()
    startbuild = time.time()

    # Parameters for asynchronous irregular firing
    NE = 4 * order          #=10000 in simulation
    NI = 1 * order          #=2500 in simulation
    N_neurons = NE + NI

    CE = int(epsilon * NE)  # number of excitatory synapses per neuron
    CI = int(epsilon * NI)  # number of inhibitory synapses per neuron
    C_tot = int(CI + CE)  # total number of synapses per neuron
    cutoff = cutoff          # Cutoff to avoid transient effects, in ms

    # Initialize the parameters of the integrate and fire neuron
    tauMem = 20.0
    theta = 20.0
```

```
65
66  J_ex = J
67  J_in = -g * J_ex
68
69  nu_th = theta / (J * CE * tauMem)
70  nu_ex = eta * nu_th
71  p_rate = 1000.0 * nu_ex * CE
72
73  if not print_report:
74      nest.set_verbosity('M_WARNING')
75
76  nest.SetKernelStatus({"resolution": dt, "print_time": True,
77                        "local_num_threads": num_threads,
78                        'overwrite_files': True})
79
80  print("Building network")
81
82  neuron_params = {"C_m": 1.0,
83                   "tau_m": tauMem,
84                   "t_ref": 2.0,
85                   "E_L": 0.0,
86                   "V_reset": 0.0,
87                   "V_m": 0.0,
88                   "V_th": theta}
89
90  nest.SetDefaults("iaf_psc_delta", neuron_params)
91
92  nodes_ex = nest.Create("iaf_psc_delta", NE)
93  nodes_in = nest.Create("iaf_psc_delta", NI)
94
95  # Stop input after x = input_stop ms if input_stop is not 0
96  # https://www.nest-simulator.org/helpindex/cc/poisson_generator.html
97  if input_stop:
98      params = {"rate": p_rate, "stop": input_stop}
99  else:
100     params = {"rate": p_rate}
101
102 nest.SetDefaults("poisson_generator", params)
103 noise = nest.Create("poisson_generator")
104
105 espikes = nest.Create("spike_detector")
106 ispikes = nest.Create("spike_detector")
107
108 nest.SetStatus(espikes, [{"label": "brunel-py-ex",
109                           "withtime": True,
110                           "withgid": True,
111                           "to_file": False}])
112
113 nest.SetStatus(ispikes, [{"label": "brunel-py-in",
114                           "withtime": True,
115                           "withgid": True,
116                           "to_file": False}])
117
118 print("Connecting devices")
119
120 nest.CopyModel("static_synapse", "excitatory", {"weight": J_ex, "delay": delay})
121 nest.CopyModel("static_synapse", "inhibitory", {"weight": J_in, "delay": delay})
122
123 nest.Connect(noise, nodes_ex, syn_spec="excitatory")
124 nest.Connect(noise, nodes_in, syn_spec="excitatory")
125
126 nest.Connect(nodes_ex[:N_rec], espikes, syn_spec="excitatory")
127 nest.Connect(nodes_in[:N_rec], ispikes, syn_spec="excitatory")
128
129 print("Connecting network")
130
```

```
131 # We now iterate over all neuron IDs, and connect the neuron to
132 # the sources from our array. The first loop connects the excitatory neurons
133 # and the second loop the inhibitory neurons.
134
135 print("Excitatory connections")
136
137 conn_params_ex = {'rule': 'fixed_indegree', 'indegree': CE}
138 nest.Connect(nodes_ex, nodes_ex + nodes_in, conn_params_ex, "excitatory")
139
140 print("Inhibitory connections")
141
142 conn_params_in = {'rule': 'fixed_indegree', 'indegree': CI}
143 nest.Connect(nodes_in, nodes_ex + nodes_in, conn_params_in, "inhibitory")
144
145 endbuild = time.time()
146
147 print("Simulating")
148
149 nest.Simulate(simtime)
150
151 endsimulate = time.time()
152
153 events_ex = nest.GetStatus(espikes, "n_events")[0]
154 rate_ex = events_ex / simtime * 1000.0 / N_rec
155 events_in = nest.GetStatus(ispikes, "n_events")[0]
156 rate_in = events_in / simtime * 1000.0 / N_rec
157
158 num_synapses = nest.GetDefaults("excitatory")["num_connections"] + \
159             nest.GetDefaults("inhibitory")["num_connections"]
160
161 build_time = endbuild - startbuild
162 sim_time = endsimulate - endbuild
163
164 print("Brunel network simulation (Python)")
165 print("Number of neurons : {0}".format(N_neurons))
166 print("Number of synapses: {0}".format(num_synapses))
167 print("       Exitatory  : {0}".format(int(CE * N_neurons) + N_neurons))
168 print("       Inhibitory : {0}".format(int(CI * N_neurons)))
169 print("Excitatory rate   : %.2f Hz" % rate_ex)
170 print("Inhibitory rate   : %.2f Hz" % rate_in)
171 print("Building time     : %.2f s" % build_time)
172 print("Simulation time   : %.2f s" % sim_time)
173 print("Cutoff first ms   : %.2f " % cutoff)
174
175 #nest.raster_plot.from_device(espikes, hist=True)
176
177 # The function shall return a tuple consisting of the excitatory and
178 # inhibitory spikes recorded, as Pandas data frames:
179 import pandas as pd
180
181 exc_spikes = nest.GetStatus(espikes, 'events')[0]
182 inh_spikes = nest.GetStatus(ispikes, 'events')[0]
183
184
185 # Excitatory spike trains
186 # Makes sure the spiketrain is added even if there are no results
187 # to get a regular result
188 #cutoff = 100
189 events_E = exc_spikes
190 nodes_E = nodes_ex
191 spiketrains = []
192
193 for sender in nodes_E[:N_rec]:
194     spiketrain = events_E["times"][events_E["senders"] == sender]
195     spiketrain = spiketrain[spiketrain > cutoff] - cutoff
196
```

```
197         spiketrains.append(spiketrain)
198
199
200
201
202     return pd.DataFrame(exc_spikes), pd.DataFrame(inh_spikes), spiketrains
203
204     #return exc_spikes, inh_spikes
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Aug 10 16:58:59 2020

@author: Anja Stene, anja.stene@nmbu.no
"""

import numpy as np
from smt.sampling_methods import LHS
from pypet import Environment
import pandas as pd
import os # To allow file paths working under Windows and Linux
from brunel_delta_ml import sim_brunel_delta
from neo.core import SpikeTrain
from quantities import Hz, s, ms
from elephant.statistics import isi, cv, fanofactor
from elephant.spike_train_correlation import corrcoef, covariance
from elephant.conversion import BinnedSpikeTrain


def get_lhs_sampling_points(num_sampling_points):

    g_space = [4.5, 6.0] # up to 5
    eta_space = [1.5, 3.0]
    J_space = [0.05, 0.4]
    D_space = [1.0, 2.5]

    xlimits = np.array([g_space, eta_space, J_space, D_space])
    sampling = LHS(xlimits=xlimits)

    x = sampling(num_sampling_points)

    print(x.shape)
    print(x[:, 0])
    print(x[:, 1])
    print(x[:, 2])
    print(x[:, 3])

    return x[:, 0].tolist(), x[:, 1].tolist(), x[:, 2].tolist(), x[:, 3].tolist()


def get_statistics(spiketrains, t_stop):

    cv_list = [cv(isi(spiketrain)) for spiketrain in spiketrains]
    isi_list = [isi(spiketrain) for spiketrain in spiketrains]
    fano_factor = fanofactor(spiketrains)

    spiketrain_list = [SpikeTrain(spiketrain*s, t_stop=t_stop) for spiketrain in spiketrains]
    binned_sts=BinnedSpikeTrain(spiketrain_list, binsize=10*ms) # binsize = simulation resolution?

    corr_coef = corrcoef(binned_sts, binary=False)
    cov_coef = covariance(binned_sts, binary=False)

    return cv_list, isi_list, fano_factor, corr_coef, cov_coef



def run_simulation(g, eta, D, J, simtime, cutoff):

    df, spiketrains = sim_brunel_delta(g=g,
                           eta=eta,
                           J=J,
                           delay=D,
                           simtime=simtime,
                           cutoff=cutoff)
    return df, spiketrains


def my_pypet_wrapper(traj):

    df, spiketrains = run_simulation(traj.g, traj.eta, traj.D, traj.J, traj.simtime, traj.cutoff)
    cv_list, isi_list, fanofactor, corr_coef, cov_coef = get_statistics(spiketrains, traj.simtime)

```

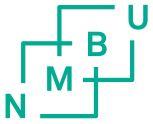
```
    traj.f_add_result('$set.$.sim_res_df', df, comment='Result from simulation i pandas dataframe`')
    traj.f_add_result('$set.$.cv_list', cv_list, comment='CV, Contains coefficient of variation for every spiketrain')
    #traj.f_add_result('$set.$.isi_list', isi_list, comment='List of interspikeintervals for all spiketrains')
    traj.f_add_result('$set.$.fanofactor', fanofactor, comment='fanofactor f = var(v) / mean(v) where v is a list of the interspike interval variability')
    traj.f_add_result('$set.$.corr_coef', corr_coef, comment='CC, Coefficient of correlation matrix, sparse')
    traj.f_add_result('$set.$.cov_coef', cov_coef, comment='CCov, Coefficient of covariance')

def add_parameters(traj):
    """Adds all parameters to `traj`
    The parameters to be explored are also added here with
    default value that is equal to function defaults in brunel_delta.py.
    """
    print('Adding Parameters')

    traj.f_add_parameter('simulation.dt', 0.1, comment='Simulation Resolution in NEST')
    traj.f_add_parameter('simulation.simtime', 1100.0, comment='Duration of the experiment simulation in ms')
    traj.f_add_parameter('neuron.D', 1.5, comment='delay, synapse-delay between neurons in ms')
    traj.f_add_parameter('neuron.g', 5.0, comment='Inhibitory synaptic strength relative to excitatory')
    traj.f_add_parameter('neuron.eta', 2.0, comment='V ext / V thr')
    traj.f_add_parameter('neuron.epsilon', 0.1, comment='Excitatory Neurons * epsilon = nr of synapses per neuron')
    traj.f_add_parameter('neuron.order', 2500, comment='Relative number of neurons in network')
    traj.f_add_parameter('neuron.J', 0.1, comment='Synapse weight between neurons')
    traj.f_add_parameter('neuron.N_rec', 50, comment='Number of neurons to record during simulation')
    traj.f_add_parameter('simulation.num_threads', 10, comment='simulation in threads for parallelizing')
    traj.f_add_parameter('simulation.print_report', True, comment='print output during simulation')
    traj.f_add_parameter('simulation.stop_input', False, comment='Stop network input in simulation after x ms')
    traj.f_add_parameter('simulation.num_sampling_points', 500, comment='Number of sampling points in Latin Hypercube Sampling Method')
    traj.f_add_parameter('simulation.cutoff', 100, comment='Cutoff first x ms to avoid transient effects, in ms')


def add_exploration(traj):
    """Explores different values of  g, eta, J and D ."""

    print('Adding exploration of g, eta, J and D')
    g_vals, eta_vals, J_vals, D_vals = get_lhs_sampling_points(traj.num_sampling_points)
    explore_dict = {'neuron.g': g_vals,
                    'neuron.eta': eta_vals,
                    'neuron.J': J_vals,
                    'neuron.D': D_vals
                   }

    traj.f_explore(explore_dict)



# Create an environment that handles running
filename = os.path.join('hdf5','biggest_set_updated.hdf5') #'thisfile.hdf5'

env = Environment(filename = filename,
                  overwrite_file = True)
traj = env.traj


# Add parameters
add_parameters(traj)


# Let's explore
add_exploration(traj)


# Run your wrapping function instead of your simulator
env.run(my_pypet_wrapper)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Aug 10 16:58:59 2020

@author: Anja Stene, anja.stene@nmbu.no
"""

import os

# Create an environment that handles running

# using the same filename as for runinng the simulation


filename = os.path.join('hdf5','biggest_set_updated.hdf5') #'thisfile.hdf5'

# reload the stored data from above.
# need an environment for that, just a trajectory.
from pypet.trajectory import Trajectory

# So, first let's create a new trajectory and pass it the path and name of the HDF5 file.
# Yet, to be very clear let's delete all the old stuff.
#del traj
# Before deleting the environment let's disable logging and close all log-files
#env.disable_logging()
#del env

traj = Trajectory(filename=filename)

# Now we want to load all stored data.
traj.f_load(index=-1, load_parameters=2, load_results=2)

# ADJUSTING THE CORR_COEF AND COV_COEF TO DATAFRAMES
import pandas as pd

df = pd.DataFrame({'g': [], 'eta': [], 'J': [], 'D': []})

for run_name in traj.f_get_run_names():
    traj.f_set_crun(run_name)
    g=traj.g
    eta=traj.eta
    D=traj.D
    J=traj.J
    cv_list=traj.results.crun.cv_list
    fanofactor=traj.results.crun.fanofactor
    corr_coef=pd.DataFrame(traj.results.crun.corr_coef)
    cov_coef=pd.DataFrame(traj.results.crun.cov_coef)

    if run_name == "run_00000000":
        print(type(cv_list))


    typelist = [type(g), type(eta), type(J), type(D), type(cv_list),
                type(fanofactor), type(corr_coef), type(cov_coef)]
    #df = df.append({'g': g, 'eta': eta, 'J': J, 'D': D}, ignore_index=True)
    df = df.append({'g': g, 'eta': eta, 'J': J, 'D': D, 'cv_list': cv_list, 'fanofactor': fanofactor,
                    'corr_coef': corr_coef, 'cov_coef': cov_coef}, ignore_index=True)

# Don't forget to reset your trajectory to the default settings, to release its belief to
# be the last run:
traj.f_restore_default()
#df.to_csv('data_500_params_stats_v5.csv')
```

```
#df['corr_coef'][0].mean()

import numpy as np

corr_sparsematrix_mean_cols = []
cov_sparsematrix_mean_cols = []

for i in range(500):
    corr = df['corr_coef'][i]
    c2 = corr.copy()
    c2.values[np.tril_indices_from(c2)] = np.nan
    corr_sparse_mean = c2.mean().round(4)
    corr_sparsematrix_mean_cols.append(corr_sparse_mean.tolist())

    cov = df['cov_coef'][i]
    c2 = cov.copy()
    c2.values[np.tril_indices_from(c2)] = np.nan
    cov_sparse_mean = c2.mean().round(4)
    cov_sparsematrix_mean_cols.append(cov_sparse_mean.tolist())

df['cov_sparsematrix_mean_cols'] = cov_sparsematrix_mean_cols
df['corr_sparsematrix_mean_cols'] = corr_sparsematrix_mean_cols

df.to_csv('data_500_params_stats_v5.csv')


import pandas as pd

df = pd.DataFrame({'g': [], 'eta': [], 'J': [], 'D': []})

for run_name in traj.f_get_run_names():
    traj.f_set_crun(run_name)
    g=traj.g
    eta=traj.eta
    D=traj.D
    J=traj.J
    cv_list=traj.results.crun.cv_list
    fanofactor=traj.results.crun.fanofactor
    corr_coef=traj.results.crun.corr_coef
    cov_coef=traj.results.crun.cov_coef

    typelist = [type(g), type(eta), type(J), type(D), type(cv_list),
                type(fanofactor), type(corr_coef), type(cov_coef)]
    #df = df.append({'g': g, 'eta': eta, 'J': J, 'D': D}, ignore_index=True)
    df = df.append({'g': g, 'eta': eta, 'J': J, 'D': D, 'cv_list': cv_list, 'fanofactor': fanofactor,
                    'corr_coef': corr_coef, 'cov_coef': cov_coef}, ignore_index=True)

# Don't forget to reset your trajectory to the default settings, to release its belief to
# be the last run:
traj.f_restore_default()
```

Appendix N requirements.txt

```
matplotlib==3.1.3
numpy==1.18.1
pypet==0.4.3
neo==0.8.0
smt==0.3.4
elephant==0.6.4
quantities==0.12.4
pandas==1.0.1
nest==1.3.0
```



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway