

**Miikka Mäkipörhölä**

# **Machine Learning Methods for Manufacturing**

**Faculty of Information Technology and Communication Sciences**

**M.Sc. thesis**

**April 2019**



# ABSTRACT

Miikka Mäkipörhölä: Machine Learning Methods for Manufacturing

**M.Sc. thesis**, 64 pages

Tampere University

Degree Programme in Computer Sciences

April 2019

---

Machine learning methods have become increasingly popular with the release of numerous open-source tools and libraries. Nevertheless the adoption of these techniques for use in manufacturing has been limited in practice. Manufacturing is still mostly dependent on traditional statistical methods and tools, even though machine learning methods could be applied to data that is already being collected from measurements done during manufacturing processes.

The purpose of this thesis is to introduce four different machine learning methods, that could prove to be useful in a manufacturing setting, and several different methods relating to the preprocessing of data and preliminary data analysis. The machine learning methods introduced are support vector machines, random forests, neural networks and NARX (non-linear autoregressive exogenous) neural networks. The algorithms and the history behind the methods introduced are explained, along with suggestions for some popular implementations of the algorithms, and the performance of each the methods is evaluated using a domain appropriate dataset.

Knowledge of the machine learning methods introduced in this thesis are an important addition to the toolkit of anyone doing predictive analytics.

Key words: machine learning, support vector regression, random forest, neural networks, NARX neural networks, manufacturing.

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.



# TABLE OF CONTENTS

1	Introduction . . . . .	1
2	Data and preprocessing . . . . .	3
2.1	Overview . . . . .	3
2.2	Input variables . . . . .	3
2.3	Outputs . . . . .	5
2.4	Output transformation . . . . .	5
2.5	Interquartile range and median filtering . . . . .	6
3	Clustering . . . . .	10
3.1	Overview of clustering and $k$ -means clustering algorithm . . . . .	10
3.2	Applying $k$ -means clustering to the output . . . . .	11
3.3	Visualization of the results . . . . .	12
4	Simple Linear Regression . . . . .	16
4.1	Overview of regression and simple linear regression . . . . .	16
4.2	Applying simple linear regression . . . . .	17
4.3	Results . . . . .	17
5	Support Vector Regression . . . . .	22
5.1	Overview of support vector machines and regression . . . . .	22
5.2	Applying support vector regression . . . . .	26
5.3	Results . . . . .	27
6	Random Forest Regression . . . . .	32
6.1	Overview of random forest regression . . . . .	32
6.2	Applying random forest regression . . . . .	35
6.3	Results . . . . .	35
7	Neural Networks and NARX neural networks . . . . .	40
7.1	Overview of neural networks and NARX neural networks . . . . .	40
7.2	Applying neural networks and NARX neural networks . . . . .	48
7.3	Results . . . . .	49
8	Conclusions . . . . .	58
	Bibliography . . . . .	60



# 1 INTRODUCTION

Machine learning methods have become more accessible during the last 5 years than ever before, because of the rapid development of easy to use open-source libraries and tools, which are readily available for everyone [Braiek *et al.*, 2018]. Machine learning methods have been popularized by companies that operate mostly in the digital space, because of the vast amounts of data collection possible inside the digital processes they control. This large-scale adaptation of machine learning processes in large technology companies has been ongoing for more than two decades. However, a large part of their work has been open sourced during the last 5 years, which has generated a lot of new interest in the field and has enabled companies and hobbyists with fewer resources to adopt machine learning methods [Braiek *et al.*, 2018].

Machine learning methods have also been adopted by large manufacturing companies, especially those producing high technology items, because of the large investment costs being easily offset by the small optimizations found in their production lines. Manufacturing is a prime field for machine learning, because of the finely tunable process parameters and the large amount of sensors utilized in production lines. Despite the possible advantages gained from machine learning methods, they are still not being applied on larger scales in manufacturing, with large amount of work which could be done by machine learning methods, is done instead with traditional methods [Sharp *et al.*, 2018].

Large amounts of time series data can be generated during a manufacturing process, and if properly logged and stored, this kind of data is ideal for machine learning applications. In a regular quality control method, when the manufactured goods fall outside the desired parameters, the generated data and the production output is analysed by a domain expert, who then implements or suggests changes to the manufacturing process. This analysis of data is predominantly done using traditional statistical methods and tools, and is reactive instead of pre-emptive. When problems in manufacturing arise a loss of production time and materials is guaranteed, due to the reactive nature of regular quality control. Time series data generated by manufacturing processes usually consist of a multitude of measurements collected from different types of sensors in the production line, along with output variables, which represent quality control measurements performed at the end of the manufacturing process. This kind of high dimensional time series data is difficult to analyse using traditional statistical methods [Johnstone & Titterington, 2009], with usually only a few key variables in the

data chosen for more careful observation. Rather than just using a few key variables, machine learning methods can enable the use of much larger input spaces, in order to create more accurate models than produced by traditional statistical methods.

This thesis introduces several different machine learning methods that might be suitable for predicting quality control measurements from manufacturing processes. The backgrounds and algorithms of the introduced machine learning methods are explained and their suitability is evaluated using domain appropriate high dimensional time series data.

Five different methods are introduced and used to build a predictive model in this thesis: linear regression, support vector regression, random forest regression, neural networks and NARX (Non-linear autoregressive with External Input) neural networks.

Linear regression is used to illustrate the difficulty of trying to build a simple linear regression model of the data with a more traditional statistical method. Support vector regression is used to create a more complex non-linear regression model of the data. Random forest regression is also used for building a non-linear regression model of the data. However, it is introduced as an example of how a large combination of weak predictors can be used as a strong predictor.

Neural networks and NARX neural networks are examined more in-depth, since they are the most sophisticated of the methods introduced, with two different models built using these methods and trained using the backpropagation algorithm, which is also explained. In addition to the previously mentioned methods,  $k$ -means clustering is introduced for the purposes of gaining a more comprehensive understanding of the distribution of values in the outputs.

Chapter 2 introduces the data used in this thesis and the preprocessing steps that were performed on it before its use. In Chapter 3  $k$ -means clustering algorithm is examined and is applied to the output for visualization purposes. In Chapter 4 linear regression is presented and simple linear regression models are created using the data. Chapter 5 introduces support vector regression, and several different models are trained using the algorithm and the performance of those models is evaluated. In Chapter 6 Random Forest regression is explained and the resulting models performance is examined in-depth. Chapter 7 covers neural networks, backpropagation and NARX neural networks. Neural networks and NARX neural networks are used to build two different types of predictive models trained using backpropagation. Chapter 8 summarizes the results from the previous chapters and suggests possible further research topics.

## 2 DATA AND PREPROCESSING

In this chapter the data used for this thesis is introduced, along with explanations and visualizations of some of the key variables in it. After the outline of data, the mandatory preprocessing steps for the use and analysis of the data are presented in the order they were performed. Moreover, the effects of these preprocessing steps are shown in visualizations, and the reasons for why the preprocessing steps were necessary to perform and what advantages they offer are explained.

### 2.1 Overview

The data used in this thesis consists of time series data with 75 input variables and 4 different outputs. The data can be split into four distinct datasets, with different outputs consisting of time spans in the range of 3-8 hours, with input variable measurements occurring every second and output measurements occurring every 12 seconds.

### 2.2 Input variables

The data contains 75 input variables, and these variables are similar to temperature, speed and pressure measurements typically encountered in a manufacturing settings. Precision of the variables is not fixed, with accuracies ranging from 1st to 4th decimal.

Of the 75 variables in the data, 26 were selected, with the help of a domain expert in manufacturing. The selected variables represent different types of measurements, with 20 representing temperatures, 5 representing speeds and one representing a size measurement. Missing values were present in the input data, but they covered less than 0.1% of the measurements and each missing value was replaced by the previous valid value during the preprocessing step.

Some of the variables are highly correlated, because changes in one variable might imply changes to other variables. These kinds of variable clusters, where the variables have a very high intercorrelation between themselves are common in a manufacturing setting. A good example of this kind of correlation is manufacturing speeds. If the speed of one part of the manufacturing process is changed, it usually requires adjustment of other speeds in the manufacturing process. A correlation coefficient matrix heat map of the selected input variables is shown in Figure 2.1.

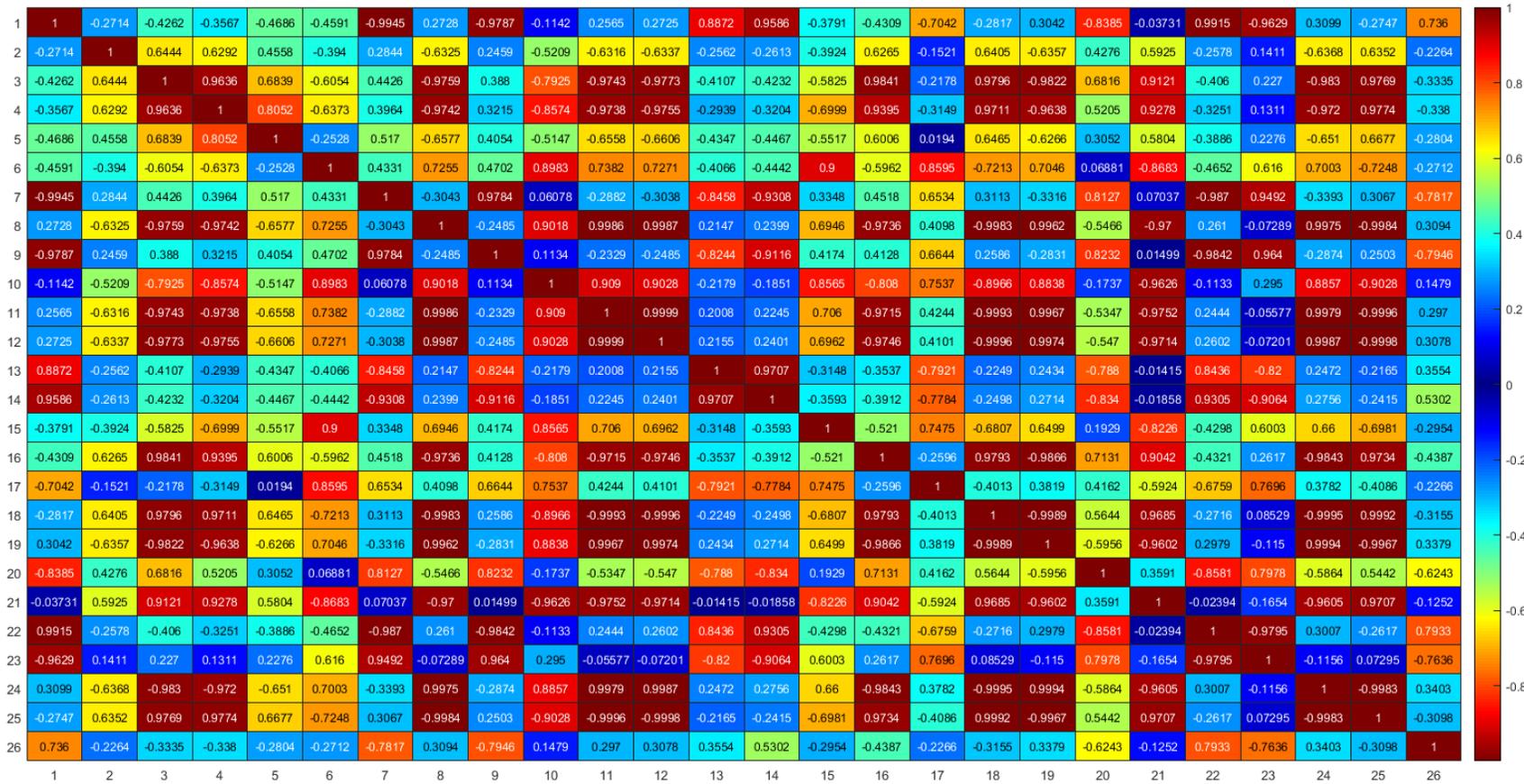


Figure 2.1: Correlation coefficients heat map of the 26 input variables used.

### 2.3 Outputs

Each distinct dataset contains 4 different outputs, and of these 4 outputs, 2 are used in this thesis. These two outputs are referred as outputs  $X$  and  $Y$ . The outputs consist of repeated vectors of 500 measurements in spans of 12 seconds. In addition to this, there is a break in the measurement process that occurs every 60 minutes.

Figure 2.2 contains a visualization of samples of the outputs  $X$  and  $Y$ , when missing measurements present in the output have been removed. In Fig. 2.3 a visualization of the sample of output  $X$  present in 2.2 is shown as a two dimensional image, the missing measurements are indicated in blue on the left and right side of the image. On average 30 measurements are missing from the start and end of the output vectors. The missing values are not errors, but rather indicate the start and end of the output measurement, which is why they are removed from the output for most use cases.

The visualization present in Fig. 2.3 was created in MatLab using the *imagesc*-function [Matlab, 2018c], which takes a matrix of data and scales the color range to the value range of the data. For this visualization the missing values were not removed, since one of the purposes was to see how the missing values were distributed on the sides of the output vectors. Instead of removing the missing values, they were replaced by zeroes, in order to make them distinct in the visualization, since all of the real values are positive.

The samples described and visualized are not representative of the whole output data used in the thesis, since the outputs of each dataset differ and all of the outputs do not contain the same extreme pattern visible in Fig. 2.2.

### 2.4 Output transformation

With the outputs  $X$  and  $Y$  consisting of vectors of 500 measurements per every 12 seconds and the input variables consisting of 26 inputs per second, it was necessary to match the output to the input data. For most methods the mean of the output vector was calculated and used as an output sample for each 12 second period, and a matching input variable window was selected for each of these output samples. Figure 2.4 shows samples of the mean values of output vectors  $X$  and  $Y$ . For support vector machine regression, random forest regression, neural networks and NARX neural networks the four input datasets and matching outputs were combined into a singular dataset, with training and testing split depending on

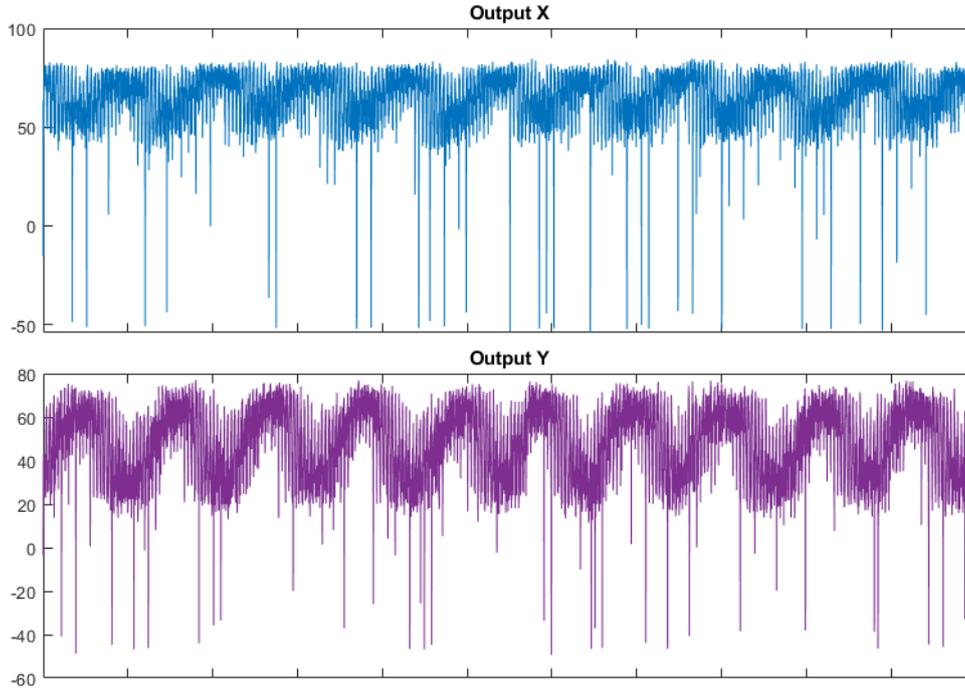


Figure 2.2: Samples of outputs  $X$  and  $Y$  from same dataset. Negative values in the samples are measuring errors.

the method.

The output vectors also contained missing values, in the form of padding in the start and end of the output vectors. The missing values were filtered out in the preprocessing step, except in the case of the datasets which were used to train the neural networks and NARX neural networks that were used for the prediction of the reduced output vectors. In addition to the missing values, every second vector of the output measurement was in reverse order, this was also corrected in the preprocessing step.

## 2.5 Interquartile range and median filtering

Interquartile range is commonly used to find outliers in data, and can be used with different ranges, with the most common use being the 25th and 75th percentile range, which is used for box plot visualizations of data. A higher range can be useful for filtering outliers present in a dataset. As can be seen in Fig. 2.2 the outputs contain many outliers, which are not representative of the data, and are most likely caused by errors in the measuring process. An interquartile range of

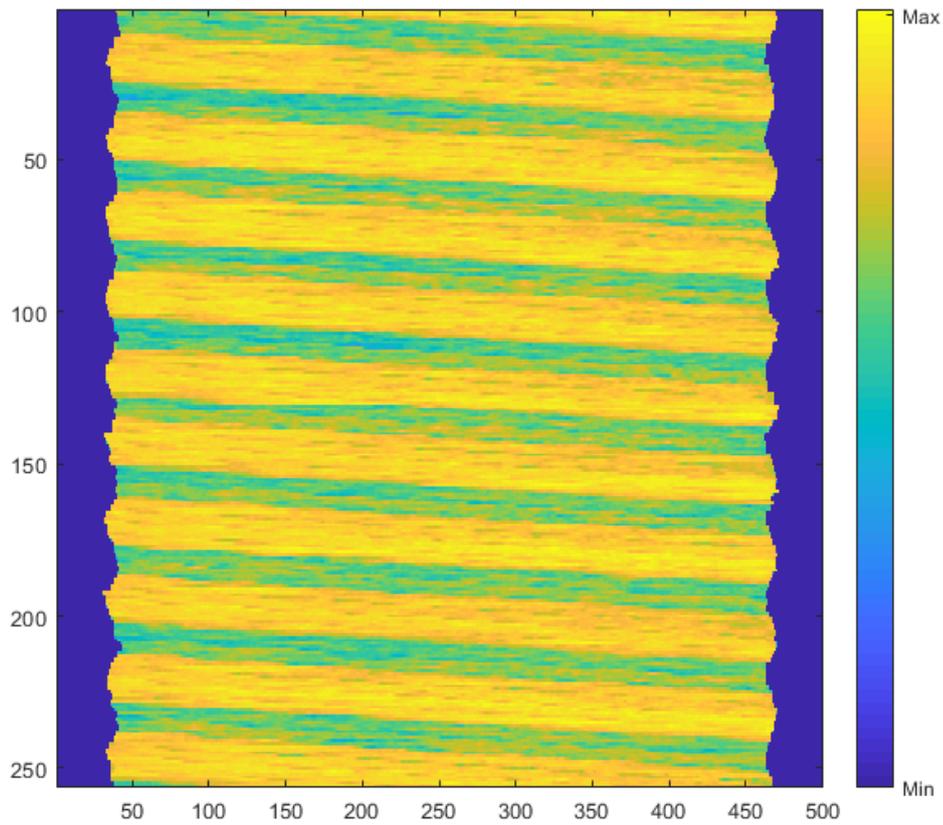


Figure 2.3: Visualization of sample vectors of output  $X$ , where y-axis represent sample  $n$  of the output and x-axis is the index of a sample vector, with missing values and measurement errors replaced by 0. The color range is scaled between the minimum and maximum value in the output vectors, with the minimum being 0.

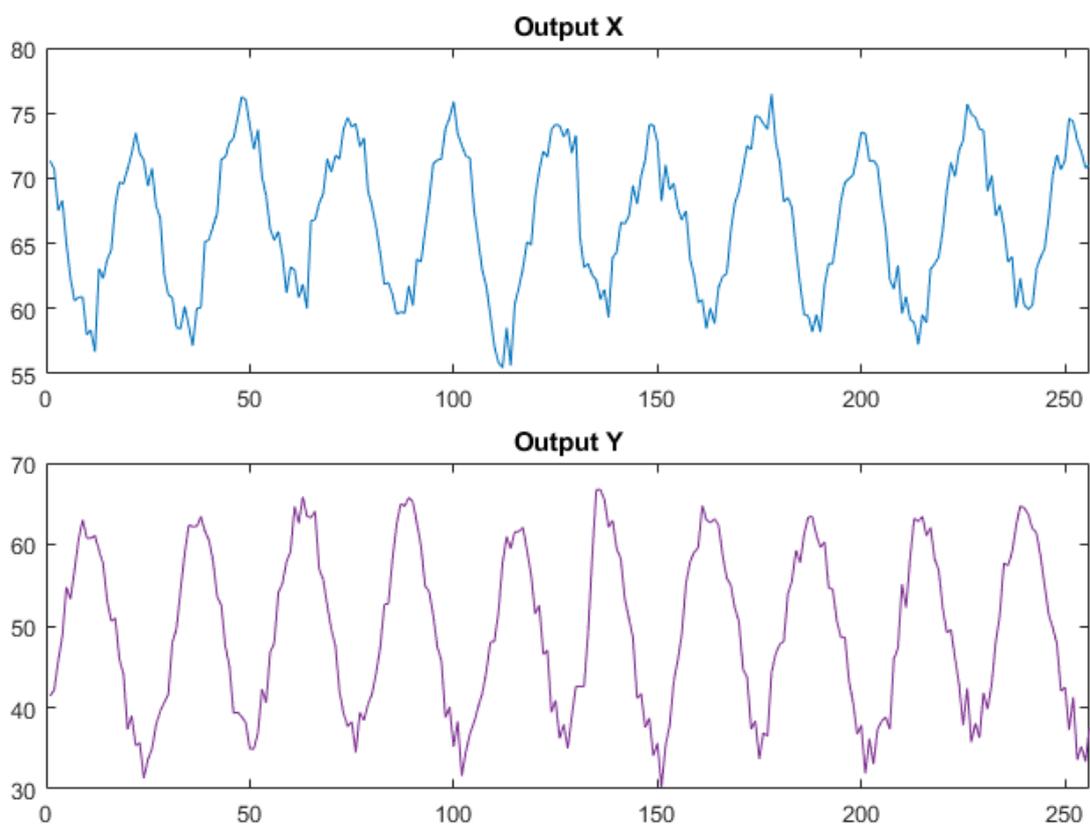


Figure 2.4: Samples of outputs  $X$  and  $Y$ , with each output vectors' mean used as a single data point.

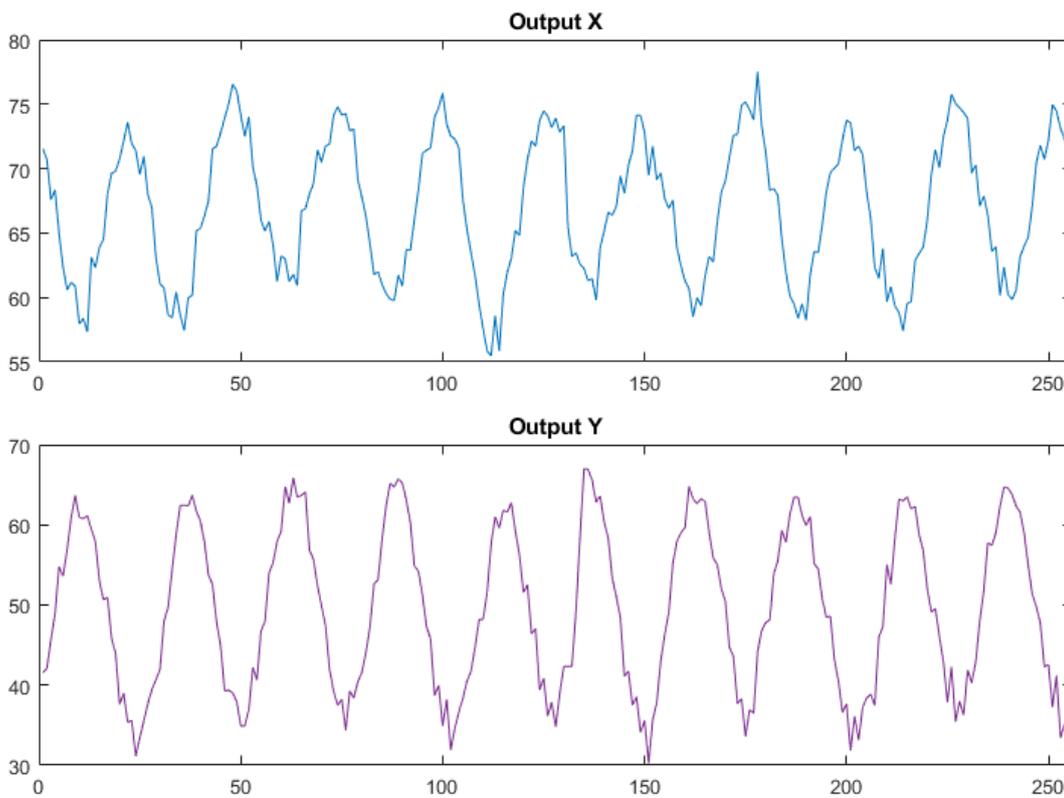


Figure 2.5: Samples of outputs  $X$  and  $Y$ , with each output vectors' mean used as a single data point, after removing outliers using interquartile range and filtering with a median filter with window size of 15.

5th to 95th percentile was set and all data points in the output vectors outside of this percentile were removed.

Further filtering of the outputs was necessary, because of noise present in the data. In this case the excess noise in the data was most likely caused by the analog nature of the measurement of the outputs. Noise should be filtered out because it could cause the machine learning methods to overestimate the effects of input variables on the output, which might in the worst case lead to overfitting of the models and render the models unusable for new data.

Median filtering was used to filter out the noise in the output data. A window size of 15 was used for the median filter, in order to not to lose too much precision, while still clearly smoothing out the noise present in the output. The effects of median filtering on outputs  $X$  and  $Y$  are shown in Fig. 2.5.

Both the interquartile range filtering and median filtering were done in Python 2.7 using the respective implementations in the SciPy library [Jones *et al.*, 2001].

### 3 CLUSTERING

Clustering was performed to get a better understanding of the value distribution of the output vectors and to better visualize potential patterns present in them.  $k$ -means clustering algorithm was chosen because of its simplicity and speed, since classification accuracy was not a priority.

In Section 3.1 the principles and functionality of  $k$ -means algorithm are examined and a pseudocode example of the algorithm is provided. In Section 3.2 the technical details on how to apply  $k$ -means clustering are shown, and the settings used for running the algorithm are provided. Finally, in Section 3.3 the results of applying  $k$ -means to the output are examined using visualizations created from the clustered data.

#### 3.1 Overview of clustering and $k$ -means clustering algorithm

Unsupervised machine learning methods, such as  $k$ -means clustering, are usually used when there is no knowledge of potential classifications in a dataset. Clustering can also be used to reduce data, for example, in color quantization of an image or dividing an image into separate regions are possible applications for clustering. In this thesis, clustering is used in order to gain a better understanding of the distribution of values in the output vectors, and for creating clear visualizations of the changes that occur in the output vectors over time.

$K$ -means clustering was chosen as the clustering algorithm, because it is one of the simplest clustering algorithms and one of the most widely used even today [Jain, 2010].  $k$ -means clustering also has a long history and has been studied extensively, with the first efficient implementations of the algorithm originating as early as the 1970s [Hartigan & Wong, 1979]. An implementation of  $k$ -means clustering can also be found in almost any machine learning framework or library, which makes it easy to use  $k$ -means in any programming environment.

$K$ -means clustering algorithm works by creating centroids in the data and associating each data point with the closest centroid, with Euclidean distance being usually used as the distance metric. Other distance or similarity measurements can also be used, with a Manhattan distance and cosine similarity being common choices. After assignment of data points to centroids, centroids are recentered to

```

Data: Data,  $k$ 
Result: Clustered Data
random starting centroids;
while no changes in cluster memberships do
    | assign each data point to the nearest centroid;
    | recalculate centroid locations to be the mean of all of the data points
    | assigned to them;
end

```

**Algorithm 3.1:**  $K$ -means clustering pseudocode.

the clusters center. This is repeated until there are no more changes in the centroid memberships and the system reaches equilibrium. A simplified pseudocode representation of the  $k$ -means clustering algorithm can be seen in Algorithm 3.1. A drawback of using  $k$ -means clustering is that the selection of  $k$  is not automatic. Choosing the right  $k$  value for  $k$ -means clustering is important, since  $k$  signifies the number of different classifications in the results. It might also be necessary to run the clustering algorithm several times to reach an adequate solution, since there is no guarantee of reaching a global optimum, because the result usually stabilizes in a local optimum.  $K$ -means++ initialization method for cluster centroids alleviates this somewhat, by providing better initialization of the centroids [Arthur & Vassilvitskii, 2007].

### 3.2 Applying $k$ -means clustering to the output

The output vectors were segmented to 20 segments of the same size before clustering, which equates to 25 measurements per segment. The segmentation was performed to eliminate possible effects of noise and to simplify the outputs feature space. Missing values were replaced by the segments median. Of the pre-processing steps discussed in Chapter 2, only the step of reversing every second measurement vector was performed before clustering.

The clustering was performed in Python using the SciKit-Learn library implementation of  $k$ -means clustering [Pedregosa *et al.*, 2011], and  $k$ -means++ method was used for the initialization of centroids [Arthur & Vassilvitskii, 2007]. SciKit-Learn's  $k$ -means implementation doesn't allow changing of distance metric, and uses Euclidean distance as the distance metric. SciKit-Learn's implementation of

using triangle inequality to calculate the distances between centroids and data points more efficiently was used [Elkan, 2003]. Other settings were left in their respective default values, since classification accuracy was not a priority. The default settings for SciKit-learns  $k$ -means clustering runs the algorithm 10 times, with different initial centroid locations. The default settings also set the maximum iteration count at 300 for a single run of  $k$ -means and a tolerance for declaring convergence.

MATLAB contains an implementation of the  $k$ -means clustering algorithm, but because of the different preprocessing steps for this method, the clustering was performed in python.

### 3.3 Visualization of the results

Lower values of  $k$  proved to be the best for visualization purposes, because they were the most effective in highlighting the changes in the value distribution of the output vectors. The best result for visualization purposes was achieved using a  $k$  value of 3, with the pattern present in the output measurement vectors being clearly visible. Using a  $k$  value of 3 caused the output measurements to have a binary classification, since the missing values at the start and end of the vectors were classified into a separate class. This clustering is shown in the Fig. 3.1. Clustering with higher values of  $k$  produced visualizations which were harder to interpret, but showed the same pattern visible in Fig. 3.1. An example visualization with  $k = 5$  is shown in Fig. 3.2.

The pattern visible in Fig. 3.1 matches the same pattern that can be seen in Fig. 2.2, and manifests as a waveform when the output measurement vectors are reduced to the mean values of the vectors.

Figure 3.3 shows samples of all of the outputs clustered with  $k = 3$ . The differences between datasets 2 and 4 to datasets 1 and 3 are clearly visible, with the pattern oscillating in the latter and being one directional in the former. Dataset 2 has the most uniform class distribution, although it shows some signs of the same pattern present in the dataset 4. Datasets 3 and 4 have more distinct patterns with clearly divisible regions of classes, than datasets 1 and 3.

The distribution of the classes in the outputs can help to understand why some of the outputs are harder to predict than others, and in the choosing of the right method for the purposes of modelling the dataset. For a dataset with repeating patterns, a method which takes into account the previous values might be the only way to model the changes accurately, if there exists a time related relationship

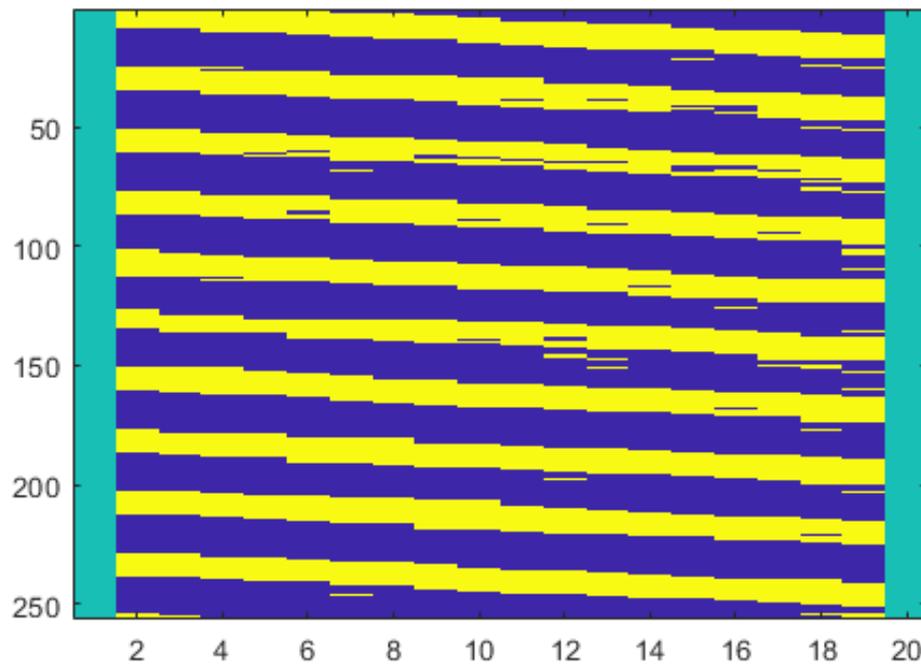


Figure 3.1: Sample of dataset 4 output vectors  $X$  clustered with a  $k$  value of 3.

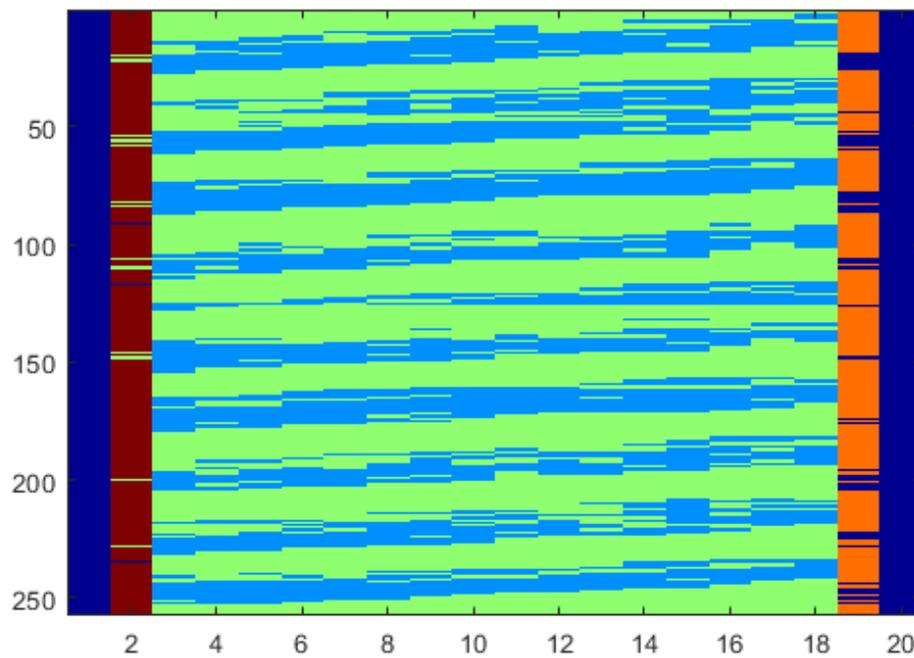


Figure 3.2: Sample of dataset 4 output vectors  $X$  clustered with a  $k$  value of 5.

between the outputs. However, a method that takes into account the previous values, might produce a worse model than traditional regression methods for a dataset where no such relationship exists, or the relationship is weak.

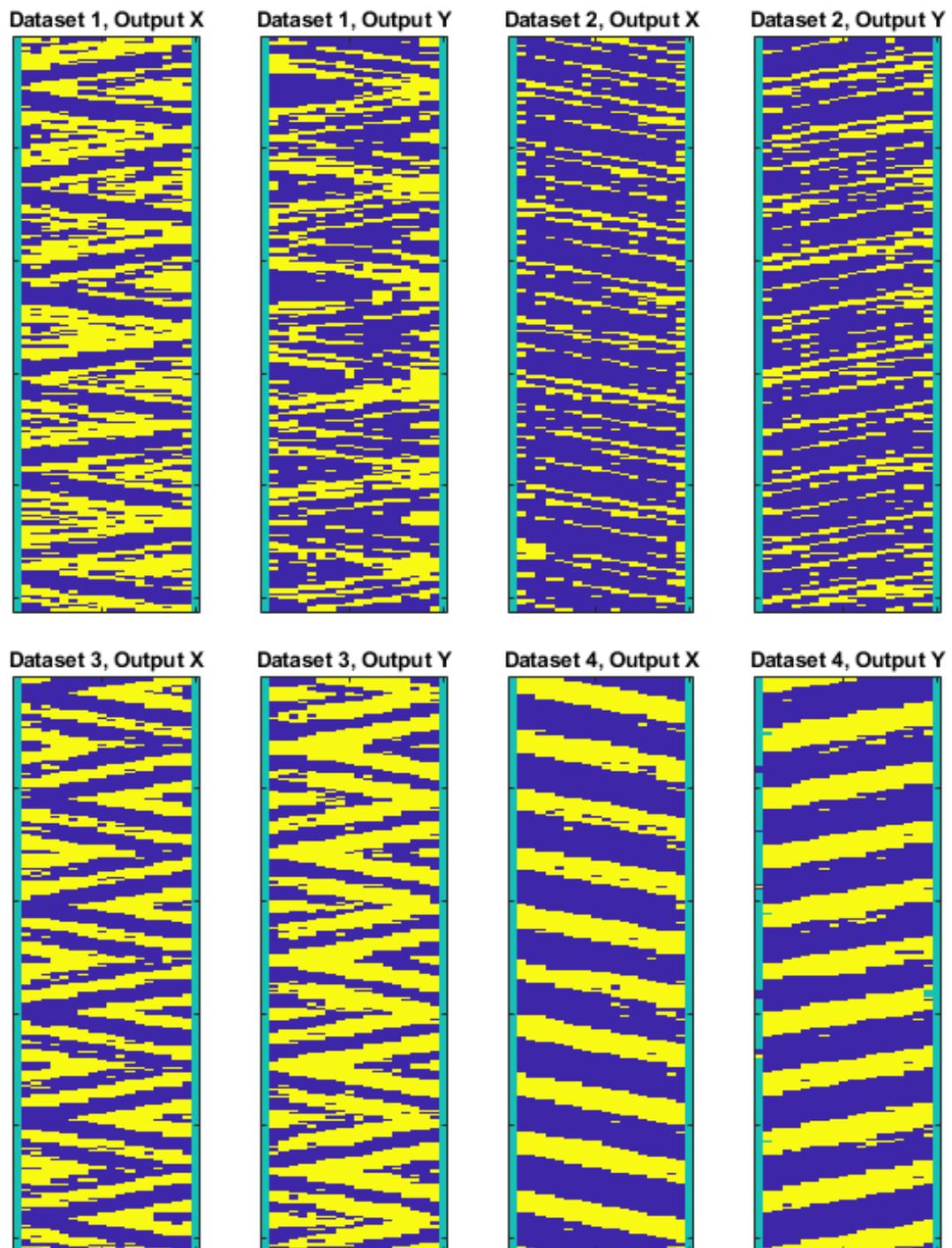


Figure 3.3: Clustered samples of the output vectors of all four datasets ( $k = 3$ ).

## 4 SIMPLE LINEAR REGRESSION

Linear regression is used for creating a simple linear regression model, this is done to make sure that there are no single variables that can be used to predict the output value, and to discover if there are some linear relationships present in the datasets. In Section 4.1 a simplified explanation of simple linear regression is provided. In Section 4.2 the technical details of how the simple linear regression models were created for the datasets are shown. Finally in Section 4.3, the created regression models' performance is examined and some of the results are visualized.

### 4.1 Overview of regression and simple linear regression

Regression analysis is one of the fundamental methods in statistical analysis, and it is still widely used in manufacturing. More complex regression methods and techniques have a lot in common with machine learning methods, but even with the more complex multivariate regression methods available, linear regression is still widely used in statistical analysis of manufacturing processes and quality control. Least squares estimation method is one of oldest forms of linear regression and it was originally published in the early 19th century [Yan & Su, 2009].

Linear Regression is a statistical tool and as is implied by the name, works only when a linear relationship exists between the independent variable and the response variable. The correlation coefficient between the independent variable and the response variable should be at least moderate for linear regression to be a viable choice for the task [Yan & Su, 2009]. According to Yan and Su [2009] a simple linear regression model can be stated in the form

$$y = \beta_0 + \beta_1 x + \epsilon$$

with  $y$  being the response variable,  $\beta_0$  being the  $y$  intercept,  $\beta_1$  as the slope of the simple linear regression line,  $x$  as the independent variable and  $\epsilon$  as the random error.

Least squares estimation works by finding the estimates for  $\beta_0$  and  $\beta_1$  so that the sum of the squared distances between the original response and the predicted response is minimal. This can be further simplified to finding the closest regression line to all of the data points [Yan & Su, 2009].

As linear regression is one of the fundamental methods in statistical analysis, it is still one of the most important statistical tools in a wide range of fields and is still used in numerous applications in a many fields of science, from social sciences

to finance. It is also still frequently used in machine learning and should be one of the first methods tried, when trying to build a regression model of a dataset, if there is a reasonable expectation that some linear relationships exist in the dataset.

## 4.2 Applying simple linear regression

A full sample of each dataset was used to create the simple linear regression models, and the model was tested against the same set it was created with. This had the risk of causing an overfit of the model, however since the resulting models performance was not acceptable and the purpose was to gain a better understanding of the relationships of single input variables on the output, instead of a producing a well performing regression model, a split into a training and testing datasets was not deemed necessary.

The correlation coefficients between the input variables and dependant output variable were examined by creating a heat map of the coefficient values, which can be seen in Fig. 4.1. Some of the correlation coefficients are relatively high, with variables 15-19 having notably high correlations with some of the outputs. However, instead of choosing those specific variables with high correlation values to create regression models with, each one of the variables were used to build simple linear regression models for each dataset and output combination.

The regression models were created in Matlab using the *polyfit* implementation [Matlab, 2018c]. The filtered output was given as the response variable and a single variable was given as an input. The process was repeated for all of the input variables and the resulting  $R^2$  and RMSE (root mean-squared error) values for the best three regression models of each dataset and output combination are shown in Table 4.1.

## 4.3 Results

The variables with high correlation coefficients produced the best fits, with variable 15, which had the highest correlation coefficient values of the input variables with the outputs, producing the best model for 6 of the 8 dataset and output combinations.

With an  $R^2$  value of 0.585, the regression model using variable 15 for dataset 4 and output value  $Y$  seems to exhibit a good fit. However, the high RMSE indicates that the model is not precise and is only good as indicator of the general direction

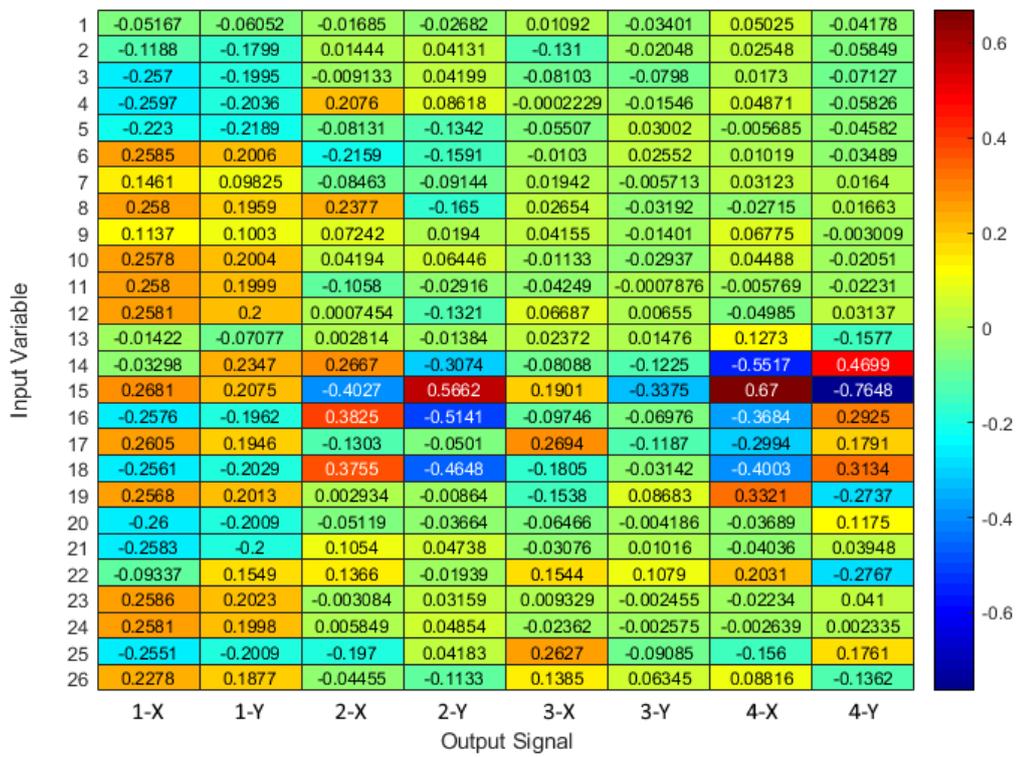


Figure 4.1: Correlation coefficients between the inputs and the outputs of each dataset.

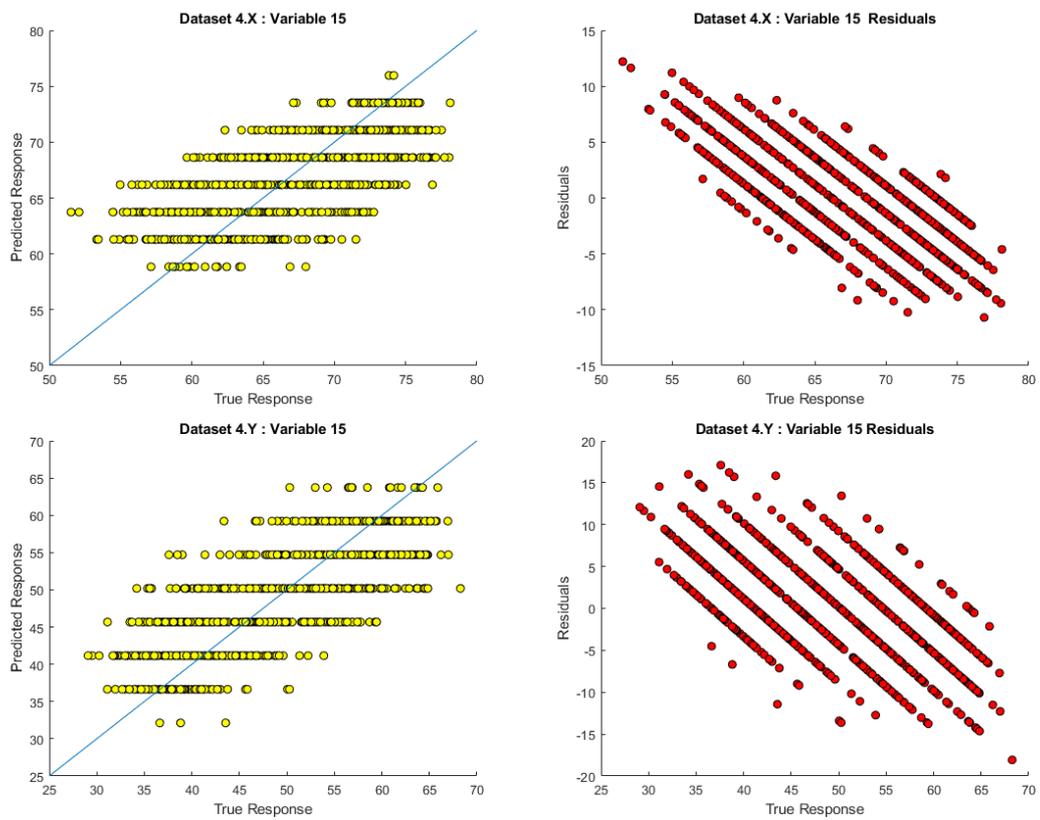


Figure 4.2: Response and residual plots for the two simple linear regression models with the best performance.

Dataset	Output	Variable	R <sup>2</sup>	RMSE
1	<i>X</i>	15	0.0719	1.8403
		17	0.0679	1.8442
		20	0.0676	1.8445
	<i>Y</i>	14	0.0551	1.2125
		5	0.0479	1.2171
		15	0.0431	1.2202
2	<i>X</i>	15	0.1622	2.3716
		16	0.1463	2.3940
		18	0.1410	2.4014
	<i>Y</i>	15	0.3206	3.4376
		16	0.2643	3.5770
		18	0.2161	3.6925
3	<i>X</i>	17	0.0726	0.9791
		25	0.0690	0.9810
		18	0.0361	0.9982
	<i>Y</i>	15	0.1139	0.7531
		16	0.0150	0.7940
		18	0.0141	0.7944
4	<i>X</i>	15	0.4489	4.3304
		14	0.3043	4.8655
		18	0.1602	5.3458
	<i>Y</i>	15	0.5850	6.0761
		14	0.2208	8.3255
		18	0.0982	8.9563

Table 4.1: Goodness of fit and error measurements of the best three created simple linear regression models for each dataset and output combination.

of the output. The same problem is present in the model created for dataset 4 and output value *X*, with an R<sup>2</sup> of 0.449. Response and residual plots for both of these models are shown in Fig. 4.2.

In the response plots, it can be seen that the response values are divided into several different groups on the *y*-axis, this is caused by the changes in variable 15 occurring in fixed steps, which is ultimately caused by the low precision of the variable.

As can be seen in the Table 4.1 the best models seem to concentrate on the variable range of 15 to 20. This group of variables has a relatively high intercorrelation and this explains the recurring presence of the variables as the best inputs for the regression models.

The only models with a decent performance are the models created for dataset 4, with only the regression models created for the combination of dataset 2 and output  $Y$  having somewhat comparable  $R^2$  values. Dataset 4 is different compared to the other datasets, in that its outputs have a relatively strong linear relationship with only a few of the input variables, while those same variables are not nearly as dominating on the outputs of the other datasets. This explains the low performance of the regression models created using the same input variable for the other datasets.

None of the created models accurately predict the small changes occurring in the outputs, even for dataset 4 which exhibits some anomalous features compared to the other datasets, which was also seen in the clustered outputs presented in the previous chapter. This makes the created regression models unusable when the goal is to predict occurring changes accurately, instead of a more general direction of change in the output. However, knowledge of the existence of linear relationships in the data is valuable information, when deciding on what other methods could possibly be used for predicting the output.

## 5 SUPPORT VECTOR REGRESSION

In this chapter support vector machine regression is used for creating a regression model with all of the selected variables as inputs. In Section 5.1 the principles and history of support vector machines and support vector regression are explained. In Section 5.2 the technical aspects of applying support vector regression are introduced and the parameters used for creating the support vector regression models are given. Finally in Section 5.3 the performance of the created models is analysed and some of the results are visualized.

### 5.1 Overview of support vector machines and regression

Support vector machines date back to the 1960s, with the first solution introduced in 1965 by Vladimir Vapnik, which solved the conceptual problem of finding a separating hyperplane which will generalize well for the case of optimal hyperplanes for separable classes. Vapnik defined the optimal hyperplane as a linear decision function with maximal margin between the vectors of the two classes [Cortes & Vapnik, 1995].

For a linearly separable dataset the optimal hyperplane can be constructed using only a small sample of the training data, these being the support vectors, this concept is visualized in the Fig. 5.1. The optimal hyperplane, which separates the classes in a dataset with maximal margin, can also be written in the form of

$$\mathbf{w}_0 \cdot \mathbf{x} + b_0 = 0$$

in which  $\mathbf{w}_0$  is a vector,  $b_0$  is a scalar and  $\mathbf{x}$  is a point. As a consequence the maximum-margin hyperplanes can be written in the form of

$$\mathbf{w} \cdot \mathbf{x}_i + b = 1$$

$$\mathbf{w} \cdot \mathbf{x}_i + b = -1$$

where  $\mathbf{w}$  is a vector,  $b$  is a scalar, and  $\mathbf{x}_i$  is point in the dataset. To limit the data points from being inside the margin the following constraints are given

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq 1 \quad \text{if } y_i = 1$$

$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1 \quad \text{if } y_i = -1$$

where  $y_i \in \{-1, 1\}$ , which indicates the class  $\mathbf{x}_i$  belongs to.

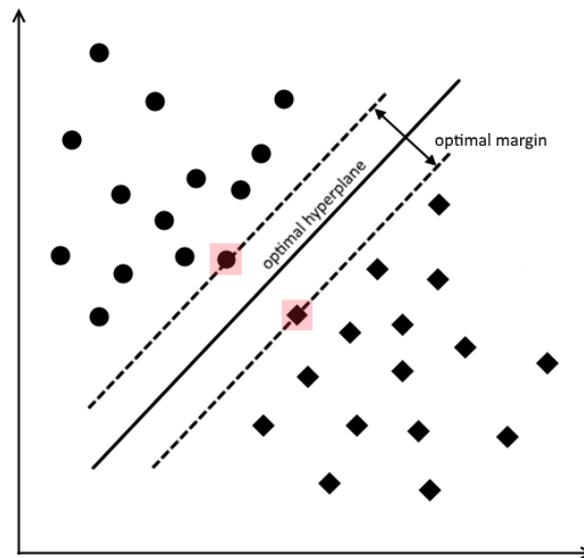


Figure 5.1: Visualization of the optimal hyperplane and maximum-margin hyperplanes for a linearly separable dataset, where the support vectors are indicated with red background.

These maximum-margin hyperplanes can then be used to classify the data, or similar data which is linearly separable. For a problem where the dataset is not linearly separable, a soft margin hyperplane, which separates the data with a minimum number of errors, can be used [Cortes & Vapnik, 1995].

Data that is linearly separable is rare in real situations and because of this a soft margin support vector machine is preferable in most cases, because it also works for linearly separable data, since it optimizes to the same solution as a hard margin support vector machine when applied to a linearly separable dataset. However, even though training a soft margin support vector machine on a non-linearly separable dataset does result in an optimal hyperplane that minimizes the errors, the resulting classification accuracy is usually very low, because the dataset is not linearly separable by nature and the optimal hyperplane cannot separate it in a reasonable way.

In the case of a non-linearly separable dataset a kernel trick can be applied to find the optimal hyperplane. Kernel tricks work by mapping the training data to a different feature space. The support vectors can then be searched for in the transformed feature space and the optimal hyperplane can also be constructed in the same feature space. The resulting optimal hyperplane is linear in the

transformed feature space, but might be non-linear in the original feature space. One of the most widely used kernel functions for support vector machines is the Gaussian kernel (Which is also called the Radial Basis Function kernel, or shortened to RBF kernel) which maps a dataset to an infinite dimensional space [Chang *et al.*, 2010]. Gaussian kernel can be written in the form of

$$K(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}}$$

where  $-\|\mathbf{x}-\mathbf{y}\|^2$  is the negative squared Euclidean distance between two vectors, and  $\sigma > 0$  is a free parameter. Gaussian kernel is in the shape of a bell curve, and the free parameter  $\sigma$  sets the width of the bell curve [Chang *et al.*, 2010].

Polynomial kernels are widely used also, with low-degree polynomial kernels used commonly for natural language processing tasks [Chang *et al.*, 2010]. When the original feature space is small, training a support vector machine with a polynomial kernel is usually faster than with a Gaussian kernel (when a low polynomial degree is used), with the drawback that the accuracy might be slightly worse. A polynomial kernel can be written in the form of

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + 1)^d$$

where  $\mathbf{x}$  and  $\mathbf{y}$  are vectors, and  $d$  is the degree of the polynomial,  $d = 1$  being a linear kernel.

Effects of applying a Gaussian kernel and quadratic polynomial kernel on a non-linearly separable dataset are shown in Fig. 5.2. As can be seen in the Fig. 5.2, finding the optimal hyperplane that separates the two classes is impossible in the original feature space, but in both of the transformed feature spaces it is a trivial task.

Support vector regression applies all of the aforementioned methods of finding the optimal hyperplane for the creation of a non-linear regression model. Support vector regression in its current form was proposed by Drucker *et al.* in 1997 [Drucker *et al.*, 1997], where as Vapnik introduced the first form of support vector regression in 1995 [Vapnik, 1995]. The main problem of finding an optimal hyperplane with support vector regression can be simplified to finding a solution to the optimization problem

$$\min \frac{1}{2} \|w\|^2$$

with the following constraints

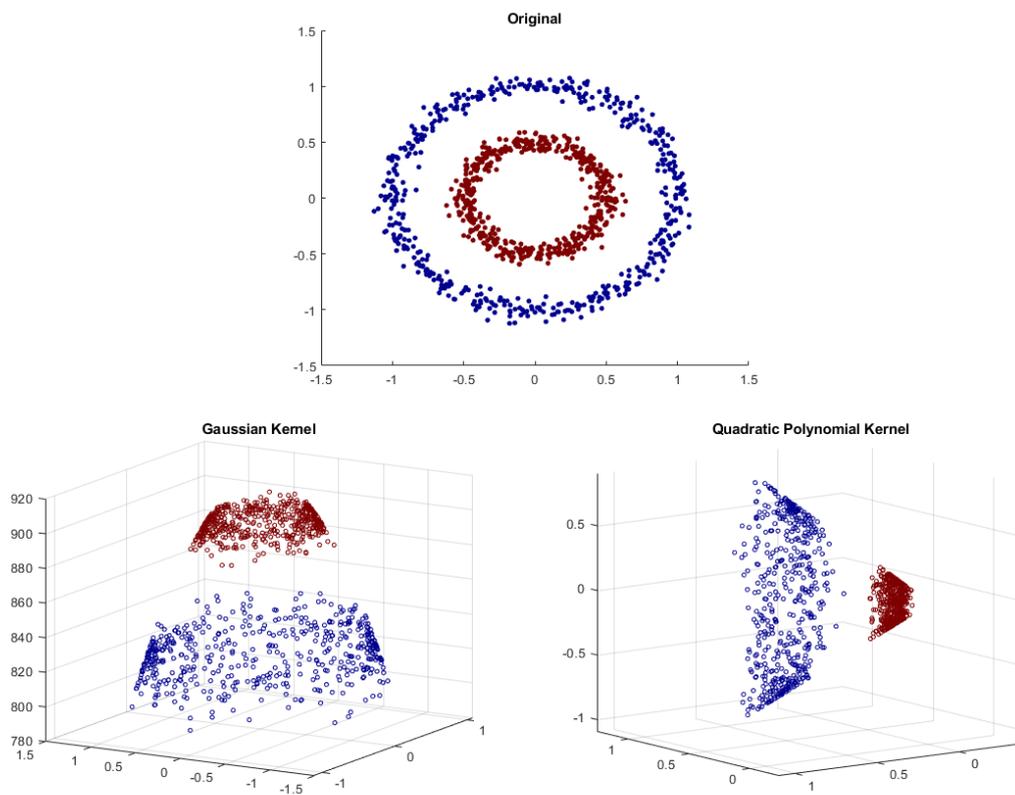


Figure 5.2: Effects of applying Gaussian kernel and quadratic polynomial kernel on a 2-dimensional non-linearly separable dataset.

$$y_i - \mathbf{w} \cdot \mathbf{x}_i - b \leq \epsilon$$

$$\mathbf{w} \cdot \mathbf{x}_i + b - y_i \leq \epsilon$$

where  $\epsilon$  is a free parameter which signifies the maximum deviation allowed (maximum absolute distance from the hyperplane),  $\mathbf{x}_i$  is a training sample, and  $y_i$  is the target value of  $\mathbf{x}_i$  [Smola & Schölkopf, 2004]. However, since a solution inside the given  $\epsilon$  might not exist, to cope with this slack variables are introduced in order to allow for errors [Vapnik, 1995], and with the slack variables the optimization problem is transformed into the form of

$$\min \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^l (\xi_i + \xi_i^*)$$

with constraints

$$y_i - \mathbf{w} \cdot \mathbf{x}_i - b \leq \epsilon + \xi_i$$

$$\mathbf{w} \cdot \mathbf{x}_i + b - y_i \leq \epsilon + \xi_i^*$$

$$\xi_i, \xi_i^* \geq 0$$

$$\epsilon \geq 0$$

where  $\xi_i$  and  $\xi_i^*$  are the slack variables, and  $C > 0$  determines the amount of deviation allowed from  $\epsilon$  [Smola & Schölkopf, 2004].

## 5.2 Applying support vector regression

The training and prediction using support vector regression was performed in Matlab, using the *ftrsvm* implementation in the Statistics and Machine Learning Toolbox [Matlab, 2018b]. Support vector regression is relatively popular and is implemented in various different machine learning libraries. Some notable libraries that implement support vector machine regression are e1071 package for R-language [Meyer *et al.*, 2019] and SciKit-Learn for Python [Pedregosa *et al.*, 2011].

The combination of the four distinct datasets was split into a training set and a testing set, with a split of 70/30. The dataset split was done with uniform random sampling of the data, using the *dividerand* function in Matlab, this was done to have a realistic sample of the data in the training and testing sets. Alternatively a hard split of each four datasets into a 70/30 division before combination of the

datasets could have been done, taking the first 70 percent of each dataset for training and the rest for testing.

Four different regression models were trained for both outputs  $X$  and  $Y$ , with the following kernel functions: linear kernel, quadratic polynomial kernel, cubic polynomial kernel and Gaussian kernel. The parameters used for *fitrsvm* in training each of the regression models are shown in Table 5.1.

Kernel Function	Kernel Scale	Standardization	Polynomial Order
linear	auto	true	N/A
polynomial	auto	true	2
polynomial	auto	true	3
Gaussian	auto	true	N/A

Table 5.1: Parameters used for training regression models using *fitrsvm* (parameters not present in the table were left at their default values).

The resulting models performance was tested using the test sets and the  $R^2$  value and RMSE (root mean-squared error) were calculated from these responses. All of the visualizations were also created from the responses to the testing sets.

### 5.3 Results

Support vector regression models trained using the Gaussian kernel method had the best performance. The performance of the models trained using regular linear kernel were only slightly worse than those trained with the Gaussian Kernel method. Models trained using the polynomial kernels performed worse than expected. Quadratic polynomial kernel produced decent results, however when compared to the models trained with the Gaussian and linear kernel methods, the performance was subpar. Models trained using the cubic polynomial kernel fared far worse than those with the quadratic polynomial kernel, with very high RMSE values and low  $R^2$  values. Table 5.2 contains  $R^2$  and RMSE values for each of the regression models created.

The training times for support vector regression machines using the polynomial kernel methods were much higher than for linear kernel or Gaussian kernel methods, which is to be expected when the input space is of a decent size. Table 5.3 lists the rounded average training times from 10 runs of training on the training datasets, which each consisted of 2610 samples. The training duration is

Target output	Kernel method	R <sup>2</sup>	RMSE
<i>X</i>	Gaussian	0.95	2.14
	Linear	0.89	3.03
	Quadratic polynomial	0.46	6.75
	Cubic polynomial	-25.10	47.57
<i>Y</i>	Gaussian	0.97	3.49
	Linear	0.94	4.53
	Quadratic polynomial	0.74	9.73
	Cubic polynomial	-9.55	62.07

Table 5.2: Goodness of fit and error measurements of all support vector regression models created, when predicting testing dataset response.

dependent on the hardware on which the training is done, but the magnitude of difference between the different kernel methods should be independent of the hardware the training is performed on. Lowering the size of the input feature space should be considered, if use of polynomial kernel methods is necessary.

Sample size $n$	Kernel method	Average training time
2610	Linear	2.5 seconds
	Quadratic polynomial	80 seconds
	Cubic polynomial	85 seconds
	Gaussian	1 second

Table 5.3: Average duration of 10 training runs of support vector regression models using different kernel method.

When evaluated using the testing set, the regression model trained for output  $X$  using the Gaussian kernel function produced a response with R<sup>2</sup> value of 0.95 and a RMSE of 2.14. The model trained with the linear kernel method had a similar performance, with an R<sup>2</sup> value of 0.89 and an RMSE of 3.03. In Fig. 5.3 the difference between the two models responses is visualized. The support vector regression model trained using the linear kernel method does not perform as well on the parts of the testing set that contains data from dataset 4.

Regression models created for output  $Y$  followed the same pattern as the ones for output  $X$ , with models trained using the Gaussian kernel having the best performance. The regression model trained using the Gaussian kernel for output

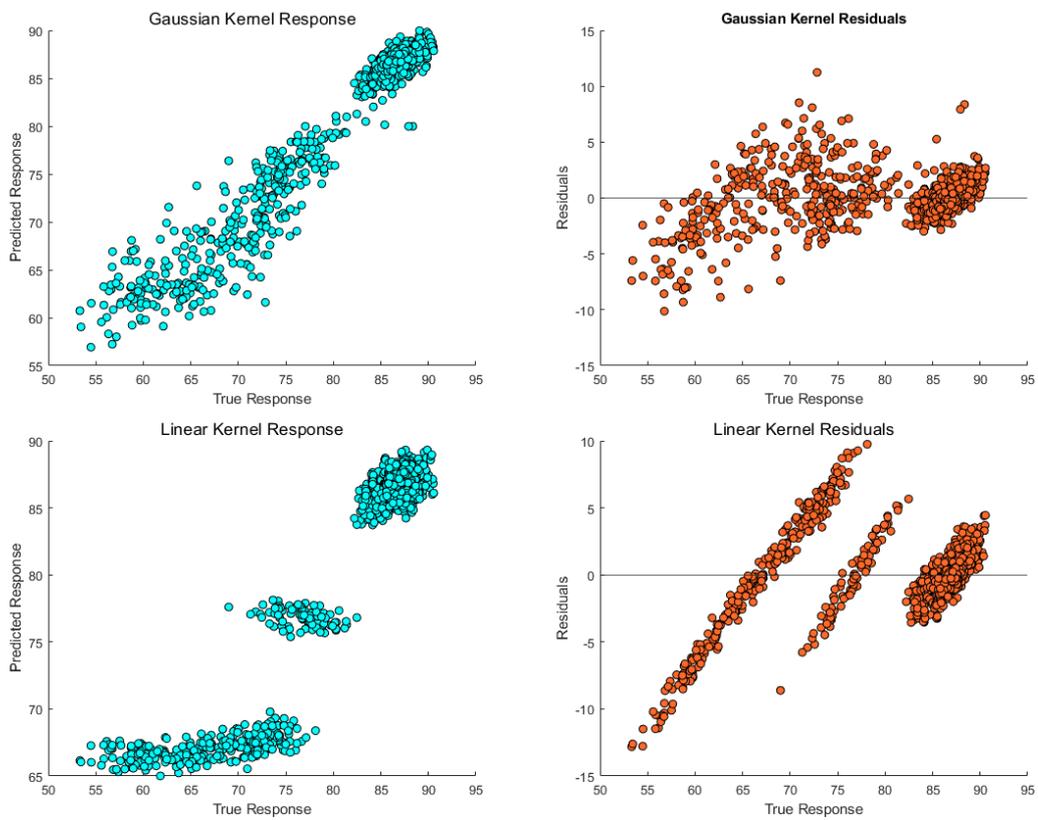


Figure 5.3: Response and residual plots for support vector regression models trained to predict output  $X$  with Gaussian and Linear kernel methods.

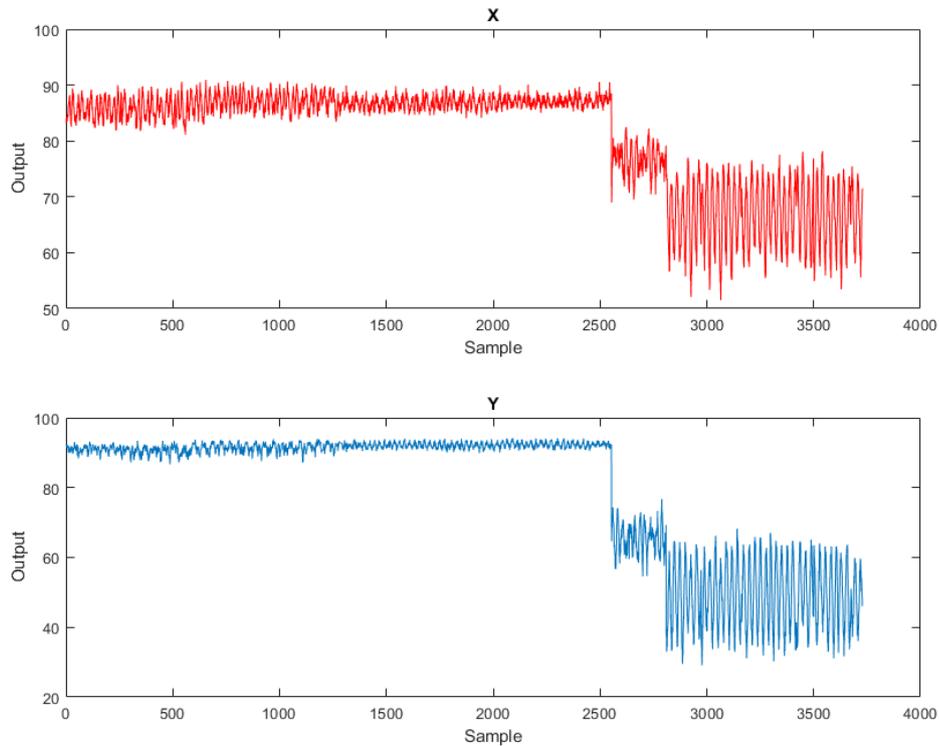


Figure 5.4: All samples of the mean vectors values of outputs  $X$  and  $Y$ .

$Y$  had an  $R^2$  value of 0.97, and an RMSE value of 3.49. Using the linear kernel for training resulted in a regression model with a  $R^2$  value of 0.9438 and an RMSE value of 4.53.

The performance of the models trained for output  $Y$  were slightly better than the ones trained for output  $X$ . This can be at least partly explained by there being more fluctuation in the output  $X$ , which can make training a regression model for the output more difficult. This difference can be seen in Fig. 5.4, which shows a side-by-side comparison of both of the outputs.

The overall performance of the trained regression models is acceptable, and would most likely suffice for many applications. However, the residual plots of the best regression models trained, shown in Fig. 5.5, indicate that the models under and over-predict the output response regularly. The trained regression models are not capable of predicting the large fluctuation present in the latter parts of the training set, which is mostly composed of data from dataset 4.

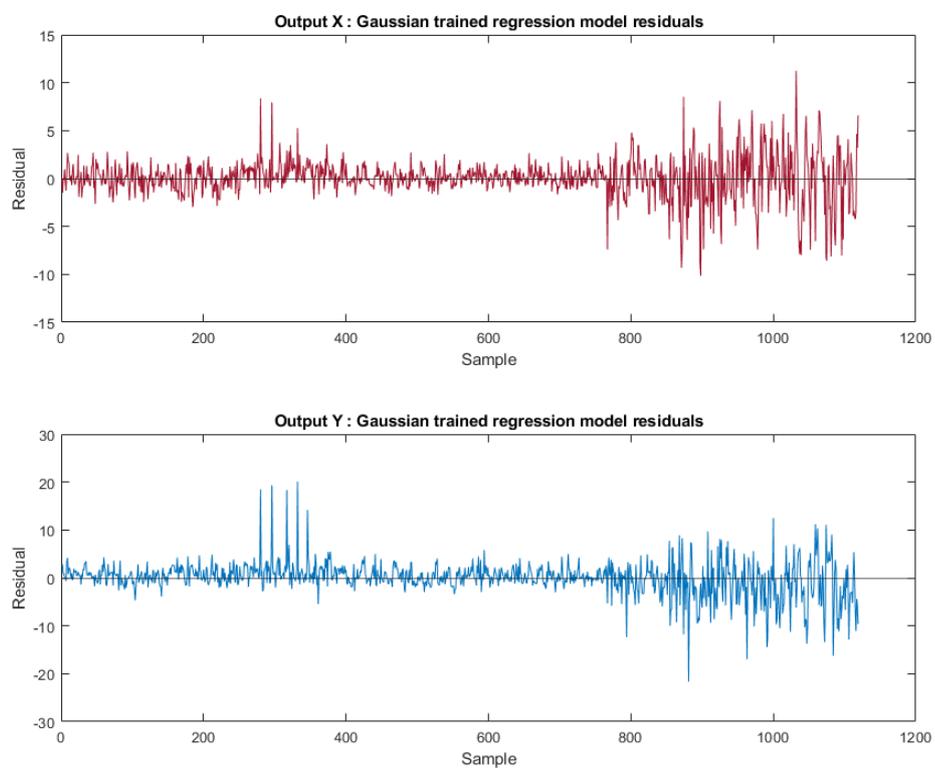


Figure 5.5: Residuals of the response outputs for the best support vector regression models trained for both outputs.

## 6 RANDOM FOREST REGRESSION

Random forest regression is introduced in order to give an example of a tree based method for training a regression model. The section 6.1 introduces random forest regression algorithm and provides several visualizations that demonstrate how the algorithm works with simplified examples. In Section 6.2 the parameters used for training are given and the implementation used in this thesis is introduced. In Section 6.3 the performance of the trained regression models is analysed and the results are visualized.

### 6.1 Overview of random forest regression

Random forest is a decision tree based method, which was developed by Leo Breiman in 2001 [Breiman, 2001]. Similar methods, in which weak random tree classifiers are combined to create a strong classifier existed before this, but Breiman combined several different methods used separately before, to form the basis for the random forest algorithm. The key methods used in random forest algorithm are bagging (also called bootstrap aggregating) used for combining random sampled weak classifiers, which was first proposed by Breiman in 1994 [Breiman, 1994] and random feature selection at decision tree splits (also called attribute bagging or feature bagging), which was proposed in 1995 by Tim Kam Ho [Ho, 1995].

Decision trees are an intuitive concept. Each node of the tree is a decision split, in which a certain feature is inspected, and based on the value of the feature a decision on which branch is traversed to the next node is made, and when a leaf node is reached, the algorithm terminates. Figure 6.1 contains an example of a simple decision tree with three nodes and two possible classification targets.

A popular recursive greedy algorithm for training a decision tree is that on each decision split, the feature to split on is chosen based on how well it divides the current dataset, based on a cost function, this process is repeated until a maximum depth or another predetermined condition is reached. Trees trained using this method are usually complex, which causes overfitting to the training data. The cost functions which are usually used are Gini gain and information gain. The depth and complexity of a trained tree can be somewhat alleviated by determining a maximum depth, or a minimum of inputs for a split to occur while training. Pruning branches that have a low effect on the accuracy of a grown tree is another possible remedy for overfitting.

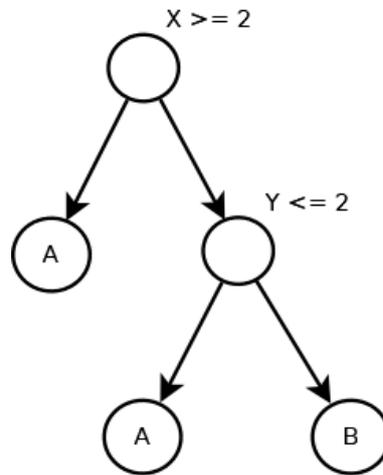


Figure 6.1: A simple decision tree, which takes values  $X$  and  $Y$  as input, and classifies the input to either class  $A$  or class  $B$ .

Regression trees can be trained in the same way. However, instead of a resulting classification at the leaf node, the mean of all of the data points used to grow the tree that belong to that leaf node is used as the response value. Figure 6.2 contains an example of a simple regression tree.

A single tree is almost useless in itself, since it is either very biased towards the training dataset and only performs well when applied to similar data, or is too general and only performs slightly better than random chance. However, multiple trees, which differ slightly from each other, can be combined to form a single strong learner. Combining a set of weak learners (in this case trees), to form a single strong learner is called boosting, the term was coined by Michael Kearns in 1988 [Kearns, 1988]. Using the multiple trees for predicting a single value is trivial, in that a single input is run through all trees and the results are combined to a single output in a manner fitting for the type of output, usually a majority vote is used for decision trees, and mean is used for regression trees.

When training multiple trees for use in boosting the algorithm, the problem of similar trees being trained repeatedly is encountered. Depending on the training algorithm for the trees, similar or even identical (if a deterministic algorithm is used) trees can be grown, this usually nullifies any advantages gained from boosting.

Bagging is used to circumvent the problem of creating correlated trees, by taking a random sample with replacement of the training data for use in training of the specific tree, this kind of sample is called a bootstrap sample. There is no clear

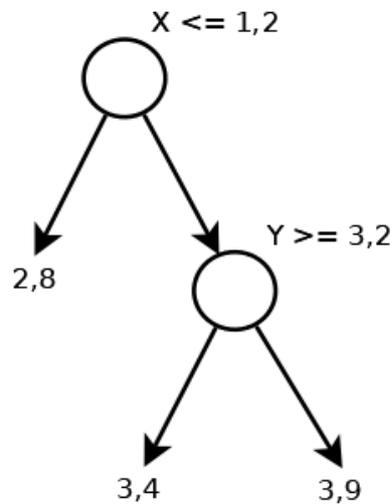


Figure 6.2: A simple regression tree, which takes values  $X$  and  $Y$  as input, and returns a response value.

definition on how many trees should be grown using bagging in order to get the best possible performance. Using bagging increases the accuracy of the complete model significantly [Breiman, 1994], however there is still a risk of overfitting to the training data.

While bagged trees by themselves already produce very good results in many cases, random forest regression incorporates random feature selection in to the training method. When using random feature selection, instead of finding the optimal split candidate from the whole feature space, a random segment of the feature space is used for determining each possible split [Breiman, 2001]. This injected randomness causes the grown trees to produce in more varied ways, which reduces variance of the model.

Random feature selection is extremely effective at reducing the trained models sensitivity to noise. An optimal ratio of features to randomly select and when to stop splitting depends on the data, but with a feature space of size  $p$ , suggested values for classification tasks are a random feature sample of size  $\lfloor \sqrt{p} \rfloor$  with a minimum node size of 1, and for regressions tasks a random feature sample of size  $\lfloor p/3 \rfloor$  and a minimum node size of 5 [Hastie *et al.*, 2009].

Random forests are also very fast to train, since the training of the trees is parallel by nature [Breiman, 1994]. The parallelization of the training process is possible, because each tree trained is a completely separate entity, with previously trained trees having no effect on the successive ones.

## 6.2 Applying random forest regression

The training of the random forest regression models was done in Matlab using the *TreeBagger* function, which is available in the Statistics and Machine Learning Toolbox [Matlab, 2018b], and can also be used for training other kinds of ensemble tree models.

Random forest algorithm is also widely implemented in several popular libraries, with SciKit-Learn for Python [Pedregosa *et al.*, 2011] containing implementations of random forest for both regression and classification, and in R a package named randomForest [Liaw & Wiener, 2002] is available, which can be used for classification and regression purposes.

The same training and testing split of the dataset specified in the support machine regression chapter was also used for training and testing the random forest regression tree models.

The training was done using the default settings of the *TreeBagger* function for performing regression. The default settings implemented are the same as those suggested by Breiman, with minimum node size being 5 and random feature sample of size  $\lfloor p/3 \rfloor$  for regression task. The number of trees grown was the only parameter changed, with the following values used for different models: 5, 10, 20, 30, 40, 50, 60, 100, 300, 600, 1000, 3000, and 5000. The models goodness of fit measurements, errors and residuals were calculated from the responses to the test sets. Training was not performed in parallel mode, which is reflected in the average training times shown in the Table 6.1.

## 6.3 Results

Three different models were trained initially, with tree counts of 100, 300 and 600. The performance of all of the three random forest regression models was practically identical, with only minimal differences. The goodness of fit and RMSE of the trained random forest regression models are visible in Table 6.1. The  $R^2$  value of each of the random forest regression models was around 0.97 for output  $X$  and close to 0.99 for output  $Y$ , with an RMSE of around 1.7 for output  $X$  and 2.3 for output  $Y$  respectively.

Random forest regression models with higher number of trees than shown in the Table 6.1 were also trained, with the highest number of trees grown being 5000. However, there was no measurable difference in the models performance when compared to a random forest regression model with 100 trees.

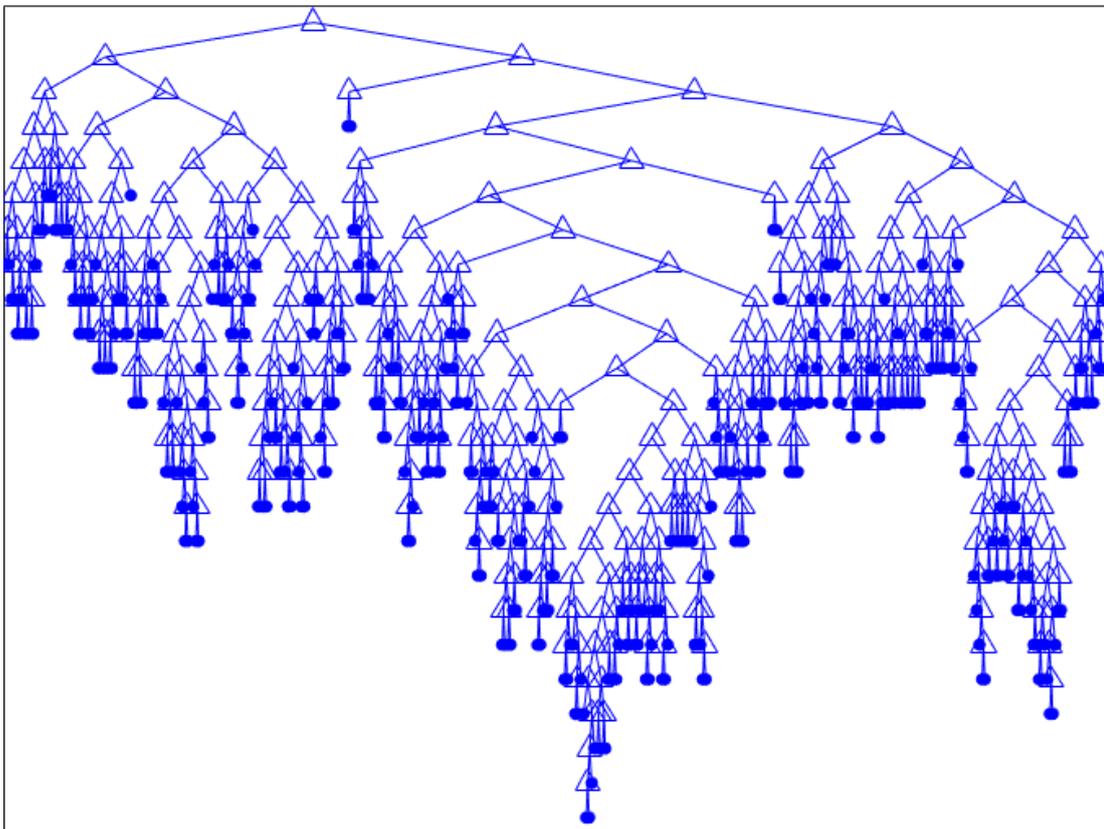


Figure 6.3: The structure of a tree in a random forest regression model trained for predicting output  $X$ , triangles represent splits and circles represent leaf nodes.

Number of trees	Output	Training time	R <sup>2</sup>	RMSE
100	X	1.28 seconds	0.97	1.67
	Y	1.17 seconds	0.99	2.32
300	X	5.75 seconds	0.97	1.65
	Y	3.60 seconds	0.99	2.29
600	X	7.45 seconds	0.97	1.67
	Y	6.99 seconds	0.99	2.30

Table 6.1: Training time, goodness of fit and RMSE of the initially trained random forest regression models with different number of trees.

Number of trees	Output	Average R <sup>2</sup>	Average RMSE
10	X	0.93	1.75
	Y	0.98	2.45
20	X	0.97	1.70
	Y	0.98	2.38
30	X	0.97	1.69
	Y	0.98	2.35
40	X	0.97	1.68
	Y	0.99	2.33

Table 6.2: Average goodness of fit and RMSE of trained random forest regression models with low counts of trees (100 random forest regression models were trained for each combination of output and the number of trees).

After noting that the number of trees above 100 had no significant effect on the performance of the created model, models with lower amounts of trees were trained to find the number of trees after which the models performance no longer increased. The average performance of random forest regression models trained using 10, 20, 30 and 40 trees are shown in Table 6.2. The performance plateau was reached at around 40 trees. There was a very slight increase in performance up to 40 trees at which point the performance was practically identical to a model with 100 trees. The largest leap of performance was between 10 and 20 trees. In Fig. 6.3 the structure of one of the trees in a random forest regression model of output  $X$  is visualized using the `view` function in Statistics and Machine Learning Toolbox [Matlab, 2018b], and it shows the complex structure of the

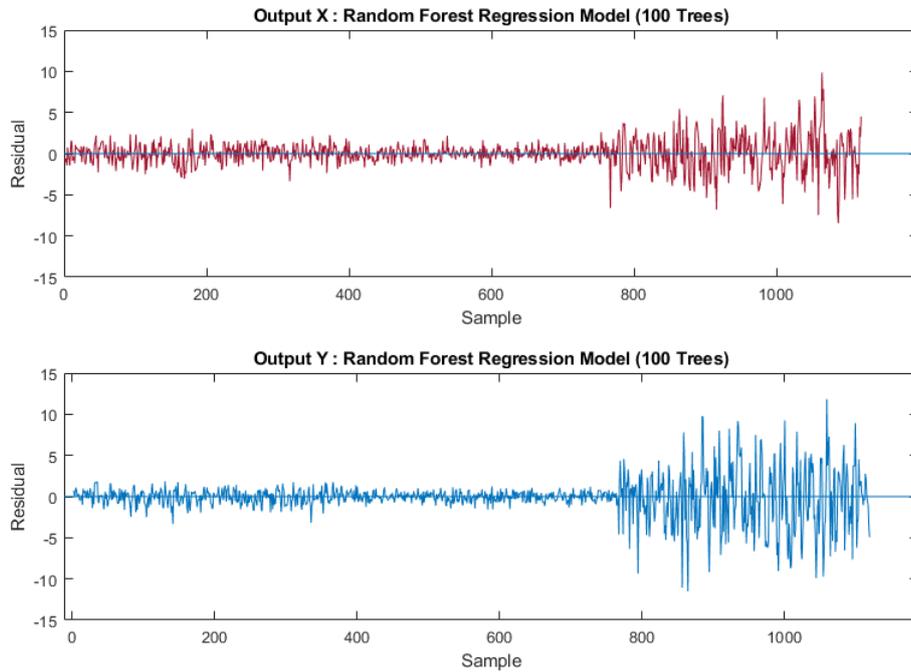


Figure 6.4: Residual plots of random forest regression models with 100 trees for outputs  $X$  and  $Y$ .

grown regression trees.

The random forest regression models trained cannot predict the fourth datasets outputs as well as the other datasets outputs, but the error rates are low enough that the overall performance of the trained models is good enough for practical uses. Residual plots of the random forest regression models with 100 trees for output  $X$  and output  $Y$  can be seen in the Fig. 6.4. The residuals are higher in the tail end of the response, which consists of the fourth dataset. The higher error rate in the predicted responses of the fourth datasets outputs were also present in the support vector regression models responses, although the errors are slightly smaller in the random forest regression models. Response plots for both output  $X$  and  $Y$  of the random forest regression models with 100 grown trees are shown in Fig. 6.5.

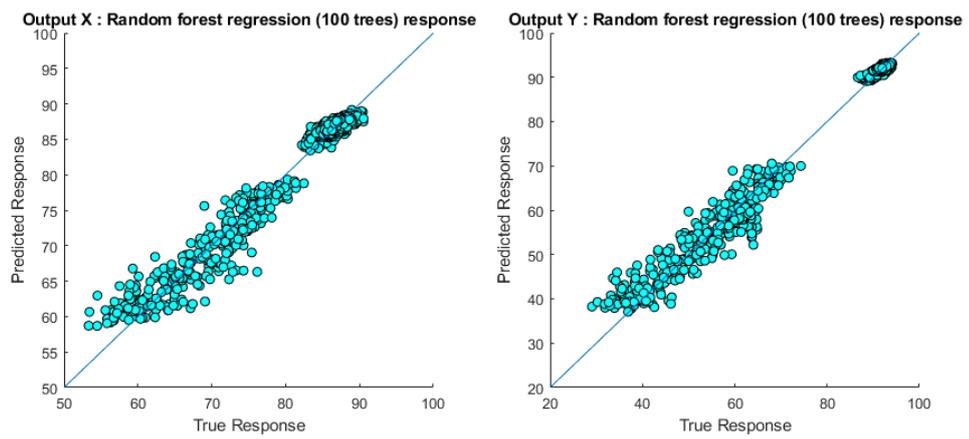


Figure 6.5: True response plots of random forest regression models with 100 trees for outputs  $X$  and  $Y$ .

## 7 NEURAL NETWORKS AND NARX NEURAL NETWORKS

In this chapter neural networks and an extended form of neural networks called NARX (non-linear autoregressive exogenous model) neural networks are introduced. In Section 7.1 the history of the development of neural networks from single layer perceptrons to modern multilayer neural networks is provided and the inner workings of neural networks and backpropagation are explained. In Section 7.2 some implementations for different programming environments are suggested, along with the parameters used for training and testing the models in Matlab. The Section 7.3 contains visualizations of the neural network models performance and the performance of the trained neural network and NARX neural network models are compared.

### 7.1 Overview of neural networks and NARX neural networks

Perceptrons are the earliest form of neural networks, and because of their simplicity are the best introduction to neural networks. Perceptrons were first created by Frank Rosenblatt in 1958 [Rosenblatt, 1958] as simplified models of biological neurons. A single layer perceptron can only be used as a linear classifier. A common example of a non-linear problem that cannot be solved with a single layer perceptron is the XOR function, which was proven to be impossible to create using a single layer perceptron by Minsky and Papert in 1969 [Minsky & Papert, 1969].

A single layer perceptron contains input nodes, weights for each of the input nodes and an output node, additionally a bias node can also be included. The steps taken when applying a single layer perceptron to an input are the following:

1. Multiply the input values with the corresponding weights
2. Sum the weighted input values and a possible bias
3. The sum of the weighted inputs and bias are given to the activation function
4. The output of the perceptron is the activation functions response

A commonly used activation function for single layer perceptrons is the binary

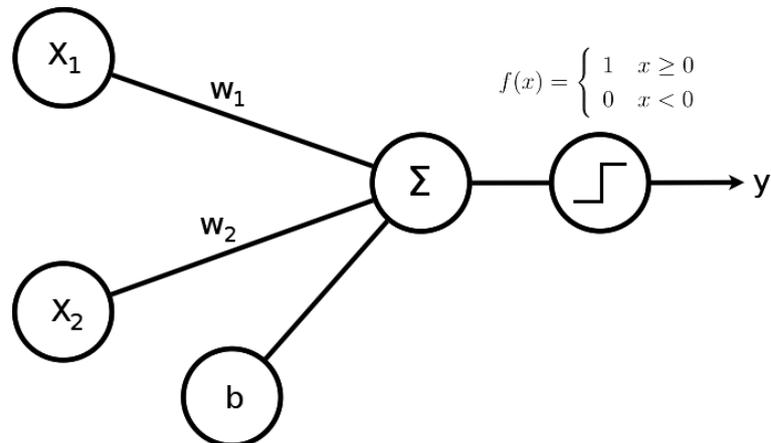


Figure 7.1: Perceptron structure for AND function,  $x_1$  and  $x_2$  are inputs, where  $x_i \in (0, 1)$ . The weights and bias for AND function are  $w_1 = 1$  and  $w_2 = 1$ ,  $b = -1, 5$ .

step function (also called Heaviside step function), which can be written as

$$H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

An example of a simple AND perceptron is shown in Fig. 7.1. Because of the simple structure of a single layer perceptron, it can also be written as the following function

$$f(x) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0, & \text{otherwise} \end{cases}$$

where  $\mathbf{x}$  is the input vector,  $\mathbf{w}$  is the weight vector,  $\mathbf{w} \cdot \mathbf{x}$  is the dot product, and  $b$  is the bias.

Training of a single layer perceptron can be done by adjusting the weights according to a learning rate to match the desired output classification. For a linearly separable dataset the training is repeated until an optimal hyperplane is found and all training data points are classified correctly. For a non-linearly separable dataset an error threshold or iteration limit must be set to arrive at a solution. Single layer perceptrons are too limited for many practical tasks. However McClelland and Rumelhart [1986] popularized the idea of using multiple processing layers between the input and output layers. Multilayer perceptrons contain at least one additional layer between the inputs and outputs, and these additional layers of nodes are called hidden layers. Hidden layers in combination with the

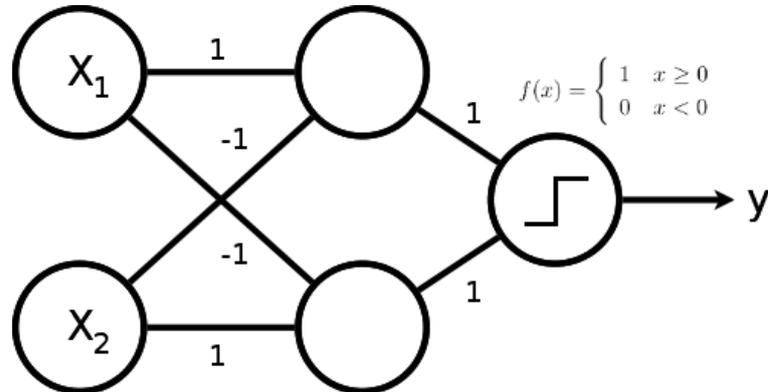


Figure 7.2: Multilayer perceptron for XOR-function,  $x_1$  and  $x_2$  are inputs, where  $x_i \in (0, 1)$ . The weights for the inputs and hidden layer outputs are marked on the connections.

use of a non-linear activation functions make it possible for multilayer networks to perform non-linear classification and regression tasks. An example implementation of a XOR-function using a multilayer perceptron with one hidden layer can be seen in Fig. 7.2.

Some common activation functions used with multilayer neural networks are sigmoid function, rectified linear unit (ReLU) function, softmax function and linear function. The sigmoid function

$$\phi(a) = \frac{1}{1 + e^{-a}}$$

produces an *s*-shaped curve, with the value scaled between 0 and 1. Rectified linear unit (ReLU) is currently the most commonly used activation function [Ramachandran *et al.*, 2017] and is defined as

$$f(x) = \max(x, 0)$$

Softmax function is usually only used in the output layer, since it depends on the other nodes outputs. It can be written as the equation

$$\phi_i = \frac{e^{\mathbf{z}_i}}{\sum_{j \in \text{output nodes}} e^{\mathbf{z}_j}}$$

where  $\mathbf{z}$  is the vector of outputs from all of the output nodes. Linear activation is also usually only used in the output layer, and is defined as the identity function  $f(x) = x$ .

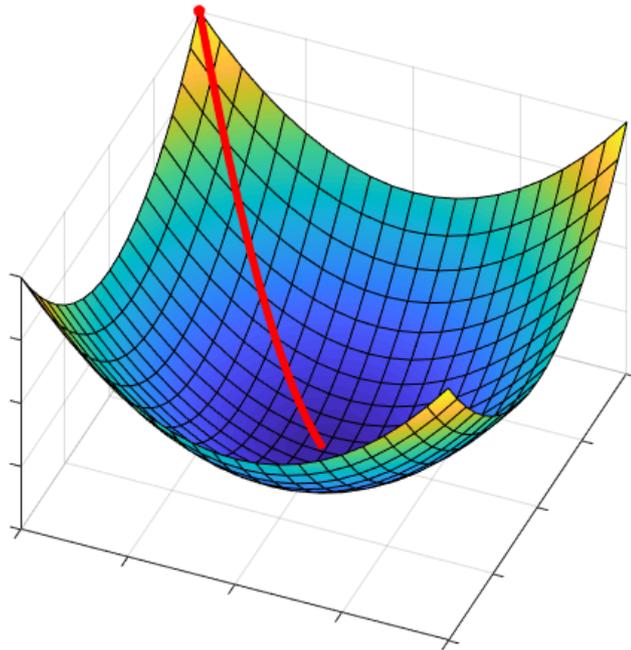


Figure 7.3: Illustration of gradient descent algorithm applied to the parabolic function  $x^2 + y^2$ .

Training of a multilayer neural network requires the use of backpropagation algorithm, which was first proposed by Paul Werbos in 1974 [Werbos, 1974]. Backpropagation is an algorithm that propagates the errors of the outputs backwards through the neural network in order to minimize the error criterion by adjusting the weights and biases. To limit overfitting a learning rate  $\eta$  is used as a multiplier for the weight and bias adjustments. A learning rate of 0.1 is usually used, but it is defined as  $0 < \eta < 1$ . The error criterion used usually for training a neural network is

$$E = \frac{1}{2} \sum_j (y_j - \hat{y}_j)^2$$

where  $j$  is an output node,  $y_j$  is the desired output and  $\hat{y}_j$  is the output nodes response.

A commonly used method for calculating the weight changes that minimize the error criterion is the gradient descent algorithm. Gradient descent tries to find the minimum of the function given to it by iteratively adjusting the parameter values. Figure 7.3 contains an illustration of gradient descent applied to the parabolic function  $x^2 + y^2$ . Gradient descent does not guarantee that a global optimum is found, since the algorithm can converge to a local optimum. Algorithm 7.1

contains a pseudocode implementation of the gradient descent algorithm.

```

Data:  $f, x, \gamma, \theta, i$ 
Result:  $y$ 
do
   $\beta = x;$ 
   $x = \beta - \gamma * f(\beta);$ 
   $\delta = x - \beta;$ 
  if  $|\delta| \leq \theta$  then
     $break;$ 
  end
   $i = i - 1;$ 
while  $i > 0;$ 
return  $x$ 

```

**Algorithm 7.1:** Gradient descent algorithm.  $f$  is the function to find the minimum for,  $x$  is the starting value used,  $\gamma$  is the multiplier used for change of  $x$  at each step,  $\theta$  is the threshold value for change between the steps,  $i$  is the maximum iteration limit and  $y$  is the local minimum of function  $f$  found.

Suggestions for additional methods to prevent the gradient descent algorithms tendency to converge on local minima have been made. However, in larger neural networks convergence on local minima is not a problem, and can actually be useful, since it can prevent overfitting the training data and finding the global minimum on a larger neural network is in some cases infeasible, because of the required training time [Choromanska *et al.*, 2015].

The following backpropagation algorithm is based on the article by Rumelhart, Hinton and Williams in 1986 [Rumelhart, 1986] and was formulated into this form by Martti Juhola [Juhola, 2019].

To understand the steps in the backpropagation algorithm, the following definitions have to be made. The activation level  $o_j$  of a hidden node or an output node is given by

$$o_j = \phi(\sum w_{ij}o_i - \theta_j)$$

where  $w_{ij}$  is the weight from input  $o_i$  to node  $j$ ,  $\theta_j$  is the node threshold and  $\phi$  is the activation function of the node. Weights are adjusted by

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

where  $w_{ij}(t)$  is the weight from node  $i$  to node  $j$  at iteration  $t$  and  $\Delta w_{ij}(t)$  is the weight adjustment. To speed up the convergence of the algorithm, a momentum term can be added

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) + \alpha(w_{ij}(t) - w_{ij}(t-1))$$

where  $0 < \alpha < 1$ . The weight change is computed by

$$\Delta w_{ij}(t) = \eta \delta_j o_i$$

where  $\eta$  is the learning rate,  $\delta_j$  is the error gradient at node  $j$ . The error gradient for the hidden nodes is given by

$$\delta_j = o_j(1 - o_j) \sum_k \delta_k w_{jk}$$

where  $o_j$  is actual output activation,  $\delta_k$  is the error gradient at node  $k$  to which hidden node  $j$  is connected. The error gradient for the output nodes is given by

$$\delta_j = \hat{y}_j(1 - \hat{y}_j)(y_j - \hat{y}_j)$$

where  $y_j$  is the desired output and  $\hat{y}_j$  is the output of output node  $j$ .

Using the aforementioned functions, the steps for using backpropagation algorithm for training a neural network are the following:

1. Set the neural networks initial weights and biases at small random values.
2. Input data to the neural network.
3. Adjust the weights starting from the output, progressing backwards to the hidden layers with the recursive equation.
4. Iterate the previous three steps until convergence with the selected error criterion is reached.

Adjustments of the weights when applying backpropagation can be done in several different ways. Batch gradient descent is the traditional method, in which

all of the gradients are calculated at once and adjustment of weights is only done once per iteration, this can lead to very slow convergence for larger datasets. A more common method is the stochastic gradient descent, where the training data is shuffled and weight adjustments are done after each samples gradient is calculated, this usually reduces the training time drastically for larger datasets, but can result in slightly reduced performance. Mini-batch gradient descent combines the batch gradient descent and stochastic gradient descent, the dataset is shuffled and divided into batches of desired size, and the weights are adjusted after the calculation of each batches gradients [Ruder, 2016].

Some threshold value must be set when checking the error criterion for reaching convergence, since in most cases the dataset is not linearly separable, and some number of errors must be tolerated. The threshold value can also prevent overfitting. Other possible error criteria are a sufficiently small error gradient, cross validation performance or iteration count.

A non-linear autoregressive exogenous (NARX) model takes into account the past value or values, which is an important feature when trying to model time series data that might contain relationships between the desired output and the previous outputs or inputs. A general NARX model can be written as

$$y(t) = f(u(t - n_u), \dots, u(t - 1), u(t), y(t - n_y), \dots, y(t - 1))$$

where  $u(t)$  is the input of the non-linear function  $f$  at time  $t$ ,  $y(t)$  is the output of the non-linear function  $f$  at time  $t$ ,  $n_u$  is the input order, and  $n_y$  is the output order. When  $f$  is a neural network the resulting system can be called a NARX neural network [Siegelmann *et al.*, 1997].

A NARX neural network takes as inputs the current input data and a number of previous inputs and outputs. The number of previous inputs and outputs used for computation of the output at time  $t$  is adjusted by a delay parameter  $d$ , with which the NARX neural network can be written in the following form

$$y(t) = N(u(t - d), \dots, u(t), y(t - d), \dots, y(t - 1))$$

where  $y(t)$  is the output at time  $t$ ,  $u(t)$  is the input at time  $t$  and  $N$  is the neural network.

NARX networks can be structured in an open-loop or closed-loop architecture. In an open-loop architecture the previous outputs for the input layer are taken from an external source. Whereas in a closed-loop architecture the computed outputs are fed directly into the input layer for the next computation. The open-loop architecture is used for training of a NARX neural network, because of the

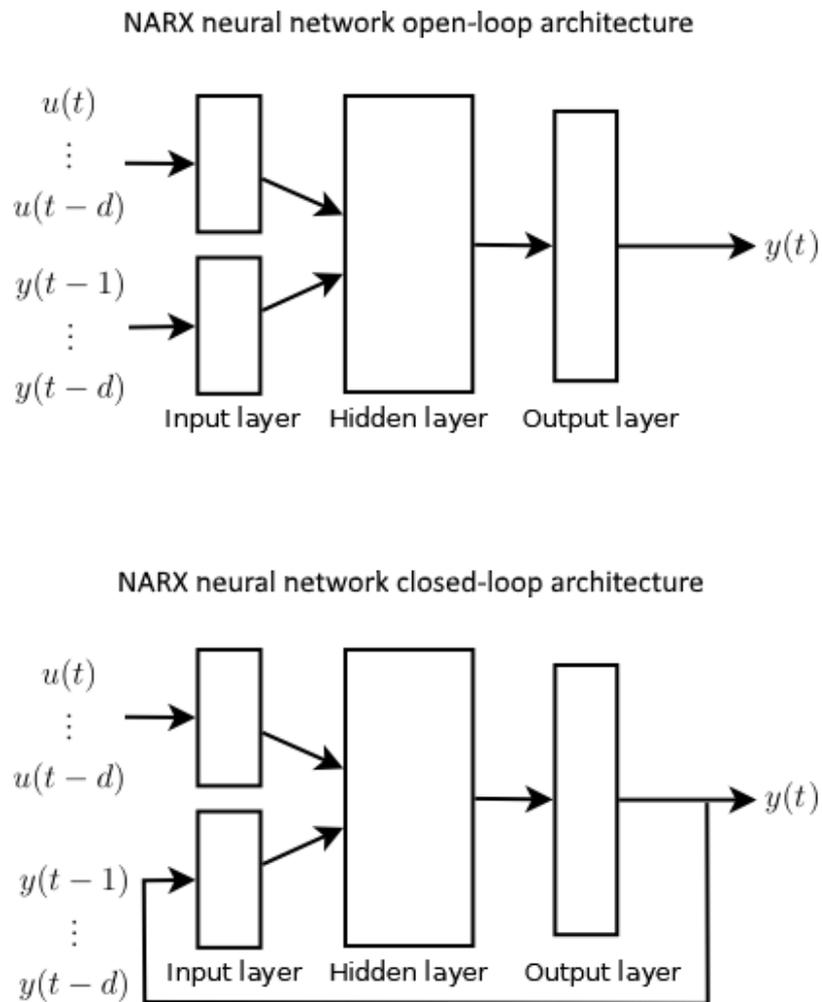


Figure 7.4: NARX neural network open-loop and closed-loop architectures.  $u(t)$  is the input at time  $t$ ,  $y(t)$  is the output at time  $t$  and  $d$  is the delay parameter.

possibility of using the correct outputs instead of the predicted outputs as the training inputs. Figure 7.4 contains a visualization of both architectures.

There have been some empirical tests that have shown that NARX neural networks outperform more traditional neural network when predicting complex time series data [Menezes Jr & Barreto, 2008]. However, NARX neural networks are not without problems, more complex NARX neural networks have a tendency to overfit the training data and might not be able to learn long time dependencies present in the data [Diaconescu, 2008].

## 7.2 Applying neural networks and NARX neural networks

Two different types of models were trained using neural networks and NARX neural networks, a model similar to the ones built using the other methods, which predicts a single value (the mean of the output vectors), and a model that takes advantage of the fact that the neural network models can have more outputs than one, which predicts a reduced form of the output vectors. Both types of models were trained for both outputs  $X$  and  $Y$ .

The training and prediction of the neural networks was done in Matlab, using the neural network tools found in the Deep Learning Toolbox [Matlab, 2018a]. The key functions used were *feedforwardnet* for neural networks, and *narxnet* for NARX neural networks. Neural network implementations are found for most programming languages, with some widely used libraries being Tensorflow [Abadi *et al.*, 2015], which has APIs for several different programming languages, and Keras [Chollet & others, 2015] for Python, which is not a neural network library by itself, but works as an interface for several machine learning libraries, including Tensorflow.

NARX neural networks were trained in open-loop architecture, and the test predictions were made in closed-loop architecture. The delays for NARX neural network input and feedback loops was set to 3. The training of the neural networks was done using the Levenberg-Marquardt Backpropagation algorithm, which is a backpropagation algorithm with an incorporated non-linear least squares optimization algorithm [Hagan & Menhaj, 1994].

For the models that predicted the output vectors, a new output dataset was created with a reduced output vector size. The reduced output vectors were created by segmenting each of the output vectors into 20 segments and then taking the median of each of those segments. Of the 20 segments the first three segments and the last three segments were removed. This was done to remove

all of the segments which contained missing values that were present at the start and end of the output vectors. The resulting output data contained 14 values, and thus reduced the size of the output layers from 500 nodes to a much more reasonable 14 nodes.

The dataset split used for neural networks and NARX neural networks was different than the split used for the previously introduced methods, since a validation set is needed for determining when to stop training of the neural networks. A split of 55 percent for training, 15 percent for validation and 30 percent for testing was used. The dataset division was handled by setting the created neural network *net* structures *divideParam* settings according to the desired split. The *divideFcn* was set to '*dividerand*' for neural networks and to '*divideblock*' for NARX neural networks, while *divideMode* was set to '*sample*' for neural networks and to '*time*' for NARX neural networks.

The structures of the neural networks and NARX neural networks trained are shown in Fig. 7.5. The structures used are simple feedforward networks with two hidden layers, with each containing 13 hidden nodes. The complexity of networks was not increased, because the shallow networks performance was sufficient for this use case and each of the models was trained 30 times to get an average performance and training time, which already resulted in multiple hours of training for the NARX neural networks with multiple outputs.

One of the common rules of thumb for the number of hidden nodes is that it should be less than the number of input nodes and greater than the number of output nodes [Heaton, 2008]. Each layer and node adds to the time complexity of the neural network, and thus increases the training time. Limiting the number of hidden layers and hidden nodes is especially important when considering the NARX neural networks, because the additional input nodes needed for the feedback loop and delayed inputs already increases the training time significantly compared to a regular neural network. The average training time of each trained neural network model is shown in Table 7.1.

### 7.3 Results

The neural network and NARX neural network models which predicted the mean value of the output vectors had a similar performance as the models created using random forest regression, with very high  $R^2$  values and low RMSE values. The average performance of each of the trained neural network types is shown in Table 7.1. Response plots of the models trained to predict the mean value for

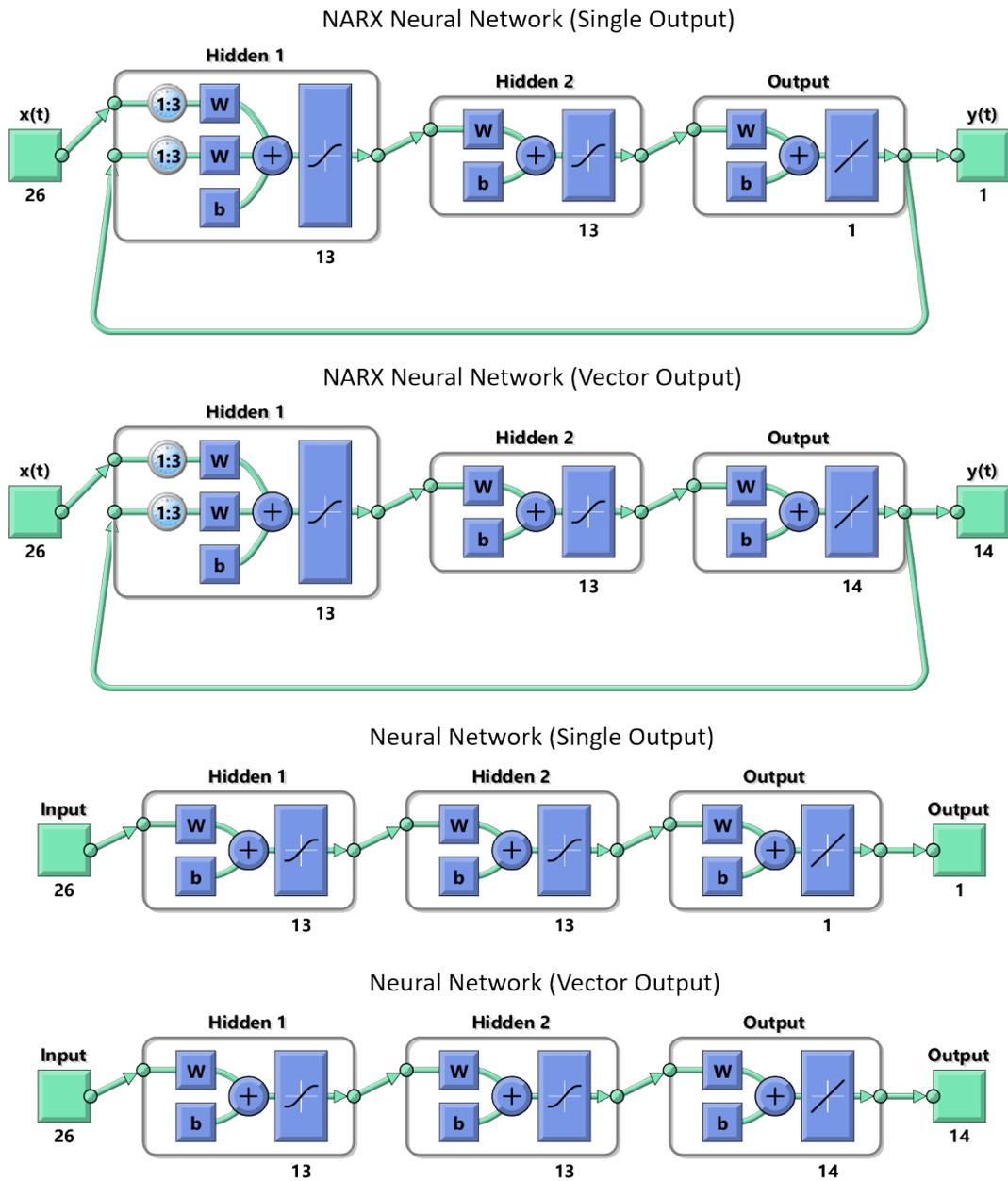


Figure 7.5: The network structures of the trained four different neural network models. The delay parameters is set to 3 for the NARX neural networks, which is indicated in the first nodes of the initial hidden layer. The figures were created using the *view* function found in Matlab Deep Learning Toolbox [Matlab, 2018a].

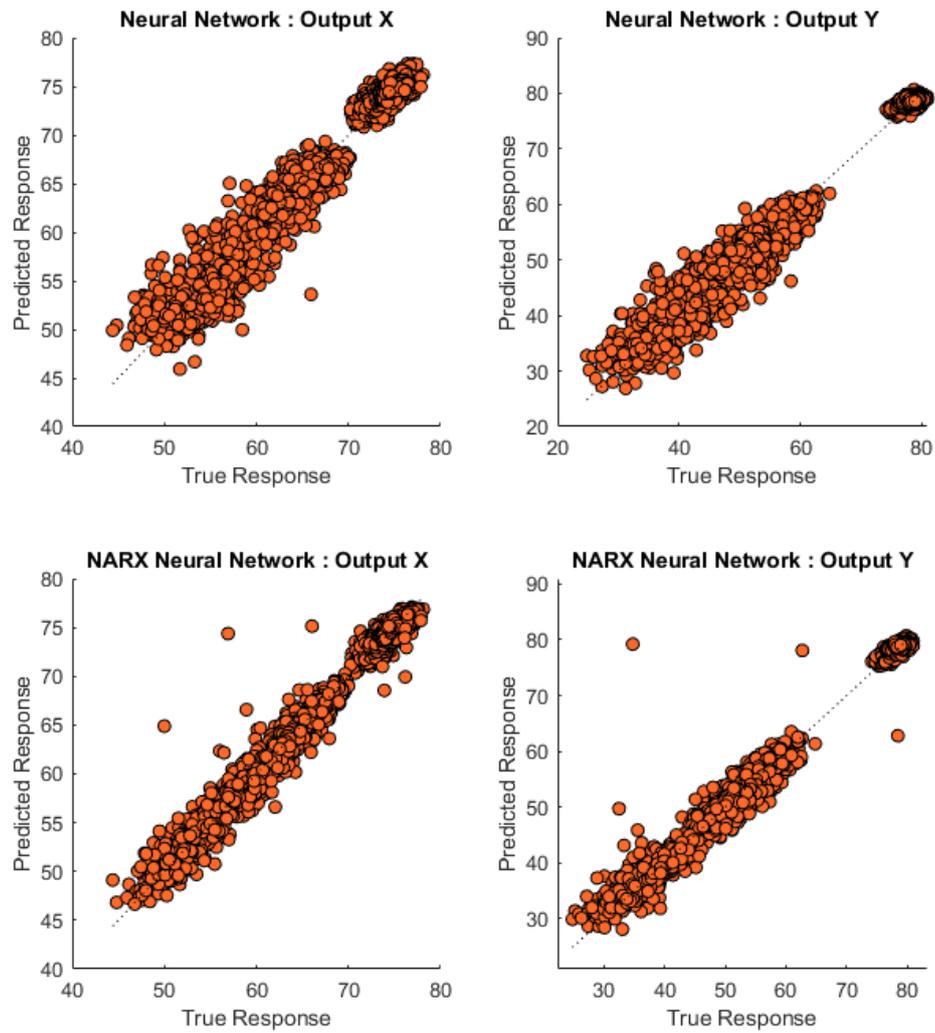


Figure 7.6: Response plots of the trained neural network and NARX neural network models for both outputs  $X$  and  $Y$ .

Output	Network Type	Output Type	Training Time	R <sup>2</sup>	RMSE
$X$	Neural Network	Mean	2s	0.989	1.618
	NARX	Mean	7s	0.993	1.315
	Neural Network	Vector	50s	0.942	4.845
	NARX	Vector	3m 10s	0.970	3.552
$Y$	Neural Network	Mean	2s	0.995	2.151
	NARX	Mean	7s	0.996	1.913
	Neural Network	Vector	35s	0.975	6.192
	NARX	Vector	3m 23s	0.984	4.994

Table 7.1: The average training times and performance measurements for all of the trained neural networks ( $n = 30$ ).

each output and network type combination with the lowest RMSE are shown in Fig. 7.6.

Residual plots of the neural network and NARX neural networks for output  $X$  which predict the mean value are shown in Fig. 7.7. The residual plots show that for both models the largest errors are mostly situated in the latter parts of the dataset which consists of dataset four, as was the case with the earlier methods also. In both outputs  $X$  and  $Y$  the NARX neural network model had a much higher error rate in the latter part compared to the regular neural network model, this is most likely caused by small errors in the output predictions being magnified over time by the feedback loop present in the NARX neural network structure. Even though the performance in the latter part is worse in the NARX neural network model, the overall errors before the portions corresponding with the fourth dataset are smaller, and this is reflected upon the RMSE of the models. The neural networks which were trained to predict the reduced output data had a similar distribution of errors as the mean value predicting neural networks for both outputs  $X$  and  $Y$ , with the NARX neural network performing worse in the latter parts of the dataset, and the regular neural network performing worse than the NARX neural network at the beginning portions of the dataset. Figure 7.8 contains samples from the beginning portion of the predicted reduced output  $Y$  in which the difference between the two types of neural network models performance is shown, with the NARX neural network model producing an output which resembles the original output much more closely than the output produced by the neural network model. In comparison, Fig. 7.10 shows the results of

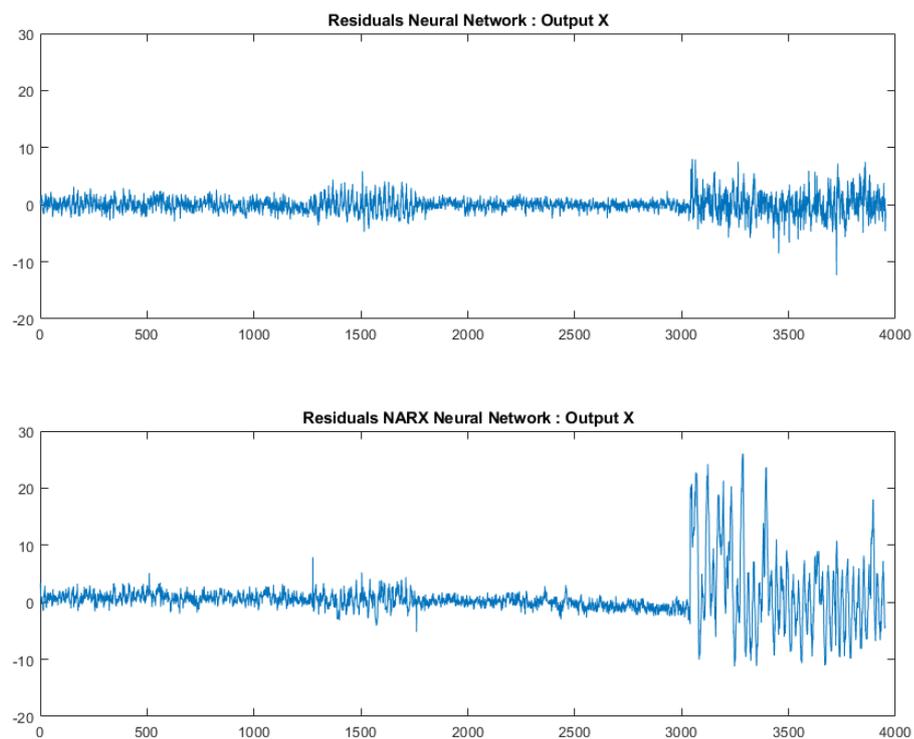


Figure 7.7: Residual plots of the predicted mean value of the output vectors  $X$  using a trained neural network and NARX neural network. The input data used for the response, from which the residuals are derived contained the training, validation and testing data.

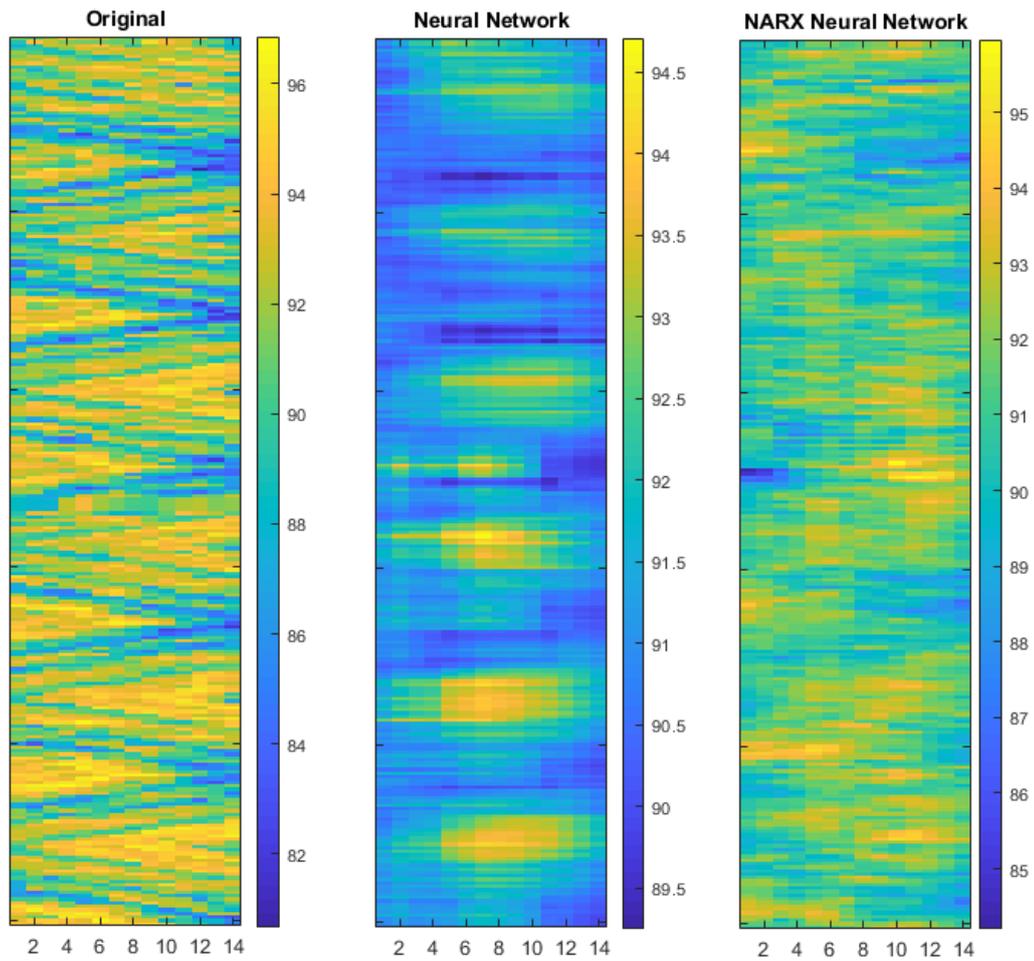


Figure 7.8: A sample from the beginning portion of the original reduced output  $Y$  and two corresponding predicted samples made using trained neural network and NARX neural network models.

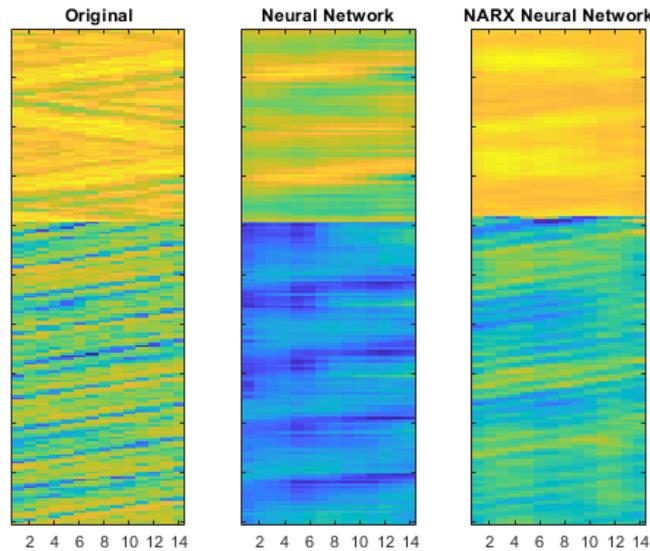


Figure 7.9: A sample from the beginning portion of the original reduced output  $X$  with a dataset change occurring in the middle and two corresponding predicted samples made using trained neural network and NARX neural network models.

predicting a sample of the latter portion of the reduced output, in which the regular neural network models prediction is much closer to the original output and manages to reproduce the pattern very well, while the predicted output using the NARX neural network manages to reproduce only some of the peaks present in the original output. Figure 7.9 contains a sample of the original and predicted reduced output  $X$ , with a dataset change occurring in the middle of the sample. The differences between the models performance in different portions of the dataset suggests that the underlying features that are responsible for the patterns in the outputs are different for each dataset, and training of a different model for each output and dataset combination might provide better results for this specific use case.

The average differences in the training times between neural networks and NARX neural networks are large, with the latter requiring on average three times the training time compared to the former. With a dataset of the size used and a shallow neural network structure the training time is not yet a problem, however the use of NARX neural networks should be carefully considered if a more complex neural network structure is necessary for the task or the dataset is much larger in size. In this case the training was not parallelized, but the magnitude of differ-

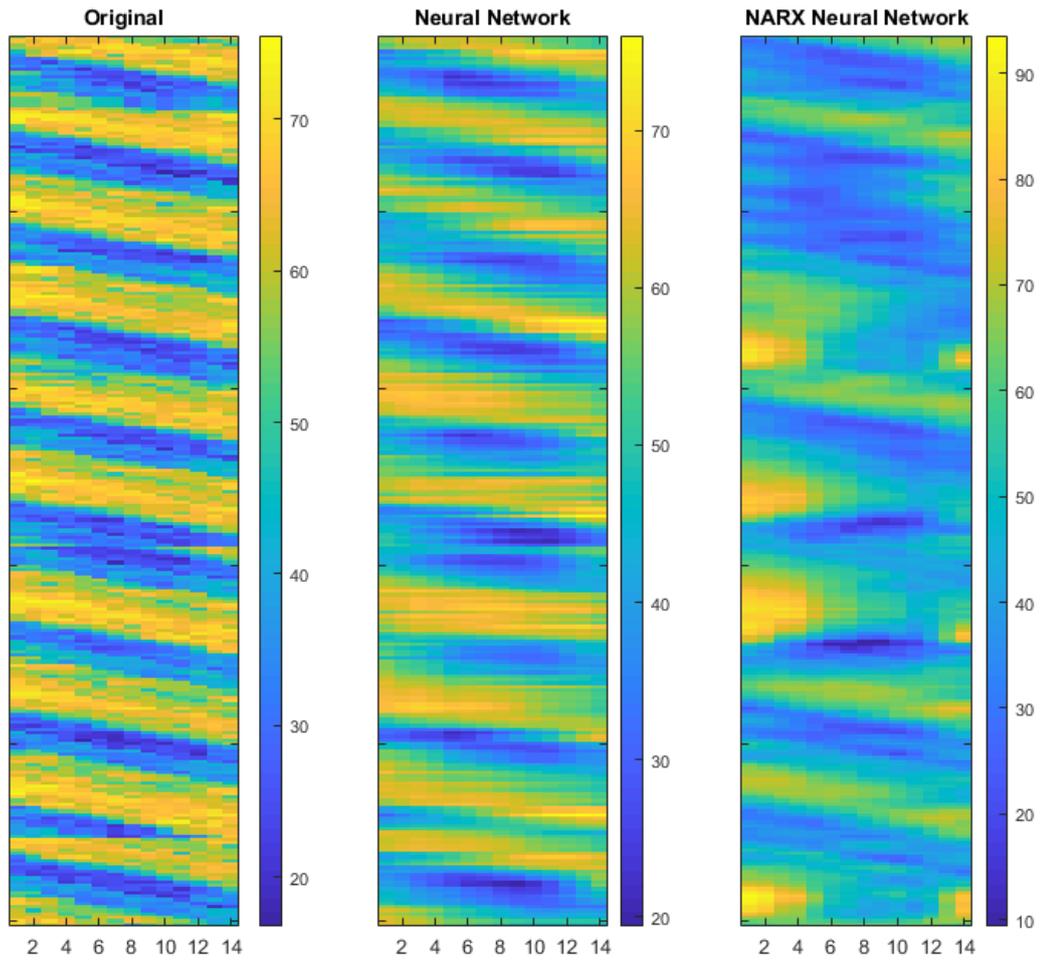


Figure 7.10: A sample from the end portion of the original reduced output  $Y$  and two corresponding predicted samples made using trained neural network and NARX neural network models.

ences should be applicable even if the training is done using a parallel algorithm, because of the increase in time complexity of the training algorithm resulting from the additional nodes in the input layer of the NARX neural network.

The differences between the trained NARX neural networks and neural networks was not insignificant, while both network structures produced models which had a similar performance when comparing the response plots,  $R^2$ , and RMSE values, the models had differences in which portion of the dataset the performance was optimal in. The neural network and NARX neural networks models which were trained to predict the mean value had a similar performance as the random forest regression models trained before, and with the random forest regression being relatively simpler method to use, careful consideration should be given into when a neural network is actually necessary for performing the required task.

Shallow neural networks are capable of producing quite complex outputs, as was seen with the capability of the trained neural network and NARX neural network models to reproduce the patterns in the reduced output vectors. However, the process of creating the reduced output data was quite similar to how the down-sampling and receptive field layers used in convolutional neural networks work, and implementing the process of down-sampling the output vectors in the neural network structures themselves should be considered if the increase in the training time of the resulting neural network is not a problem.

## 8 CONCLUSIONS

This thesis introduced four different machine learning methods which might be employed in a manufacturing setting for predicting the output of a quality control measurement. All of the introduced machine learning methods produced acceptable results for the dataset they were evaluated against.

Preprocessing is one of the most important tasks when using machine learning methods and a large portion of the performance can and should be attributed to extensive preprocessing of the data used in this thesis. Preprocessing should always be done carefully and should be the first step performed, before trying to build a model of the data. With good preprocessing of the data a simple machine learning method like support vector machine regression can produce results that can surpass traditional statistical methods with minimal training.

Differences between the best models built for predicting the mean value of the output vectors using the introduced machine learning methods were minimal, with the trained random forest, neural network and NARX neural network models having similar error rates and  $R^2$  values, while the support vector regression models trained using the Gaussian kernel had only slightly worse performance than the aforementioned models. All four methods were also capable of reproducing the fluctuation present in the mean value of the output vectors.

Training of most of the models created with the introduced machine learning methods and using the domain appropriate dataset was performed in less than 10 seconds, with only the neural network and NARX neural networks which predicted the reduced output having significantly higher training times. Training time with each method is of course dependent on the size, feature space and output type of the data used. If the training time of a machine learning model is an issue and the dataset or the feature space of the dataset is large, the decision on which method to use should be carefully weighed, because of the inherent differences in the methods training times. Incremental training of the models was not considered in this thesis, but should be possible for all of the models created with the introduced methods using appropriate training algorithms. Adjusting of the trained models periodically, while ensuring that the models previous performance is not altered, in a manufacturing settings is a topic that should be researched further.

Simple shallow neural networks can produce good results for many problems, as was seen in the neural network and NARX neural network models which were capable of reproducing the pattern in the reduced output vectors, but if the data

used is very complex, for example consisting of images or video, shallow neural networks will most likely fail to produce acceptable results. In this case more complex methods like convolutional neural networks should be considered. However, preprocessing of the data and training time are greater problems with convolutional neural networks than with the methods introduced in this thesis and the possibility of simplifying or reducing the data significantly should be considered first and foremost, before rushing to use more complex methods. Convolutional neural networks were outside of the scope of this thesis, but further research into the use of them for predicting the output of quality control measurements is also recommended.

The decision on which of the introduced methods, or any other machine learning methods, to use should always be guided by the principle of finding the simplest method which can still produce the desired results.

## BIBLIOGRAPHY

- [Abadi *et al.*, 2015] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, & Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from [tensorflow.org](http://tensorflow.org).
- [Arthur & Vassilvitskii, 2007] David Arthur & Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.
- [Braiek *et al.*, 2018] Housseem B. Braiek, Foutse Khomh, & Bram Adams. The open-closed principle of modern machine learning frameworks. pages 353–363, 2018.
- [Breiman, 1994] Leo Breiman. Bagging predictors. University of California, Technical Report no. 421. 1994.
- [Breiman, 2001] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [Chang *et al.*, 2010] Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, & Chih-Jen Lin. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11:1471–1490, 2010.
- [Chollet & others, 2015] François Chollet et al. Keras. <https://keras.io>, 2015.
- [Choromanska *et al.*, 2015] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, & Yann LeCun. The loss surfaces of multilayer networks. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 192–204, 2015.

- [Cortes & Vapnik, 1995] Corinna Cortes & Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
- [Diaconescu, 2008] Eugen Diaconescu. The use of narx neural networks to predict chaotic time series. *WSEAS Transactions on Computer Research*, 3(3):182–191, 2008.
- [Drucker *et al.*, 1997] Harris Drucker, Christopher J.C. Burges, Linda Kaufman, Alex J Smola, & Vladimir Vapnik. Support vector regression machines. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 155–161, 1997.
- [Elkan, 2003] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 147–153, 2003.
- [Hagan & Menhaj, 1994] Martin T Hagan & Mohammad B Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.
- [Hartigan & Wong, 1979] John A Hartigan & Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.
- [Hastie *et al.*, 2009] Trevor Hastie, Robert Tibshirani, & Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, 2nd edition, 2009.
- [Heaton, 2008] Jeff Heaton. *Introduction to Neural Networks with Java*. Heaton Research, Inc., 2008.
- [Ho, 1995] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282, 1995.
- [Jain, 2010] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

- [Johnstone & Titterington, 2009] Iain M. Johnstone & D. M. Titterington. Statistical challenges of high-dimensional data. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1906):4237–4253, 2009.
- [Jones *et al.*, 2001] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.
- [Juhola, 2019] Martti Juhola. Neurocomputing: Backpropagation training. <https://coursepages.uta.fi/tiets07/wp-content/uploads/sites/110/2019/02/Neurocomputing6.pdf>, 2019. Accessed: 2019-04-18.
- [Kearns, 1988] Michael Kearns. Thoughts on hypothesis boosting. *Unpublished manuscript*, 45:105, 1988.
- [Liaw & Wiener, 2002] Andy Liaw & Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.
- [Matlab, 2018a] Matlab. Matlab Deep Learning Toolbox, version 12.0 (r2018b), 2018.
- [Matlab, 2018b] Matlab. Matlab Statistics and Machine Learning Toolbox, version 11.4 (r2018b), 2018.
- [Matlab, 2018c] Matlab. version 9.5.0 (r2018b), 2018.
- [Menezes Jr & Barreto, 2008] José Maria P Menezes Jr & Guilherme A Barreto. Long-term time series prediction with the narx network: An empirical evaluation. *Neurocomputing*, 71(16-18):3335–3343, 2008.
- [Meyer *et al.*, 2019] David Meyer, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, & Friedrich Leisch. e1071: Misc functions of the department of statistics, probability theory group (formerly: E1071), tu wien, 2019. R package version 1.7-1.
- [Minsky & Papert, 1969] Marvin Minsky & Seymour Papert. *Perceptrons: An introduction to computational geometry*, pages 62–68. MIT Press, 1969.

- [Pedregosa *et al.*, 2011] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, & Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [Ramachandran *et al.*, 2017] Prajit Ramachandran, Barret Zoph, & Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [Rosenblatt, 1958] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [Ruder, 2016] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [Rumelhart & McClelland, 1986] David E. Rumelhart & James L. McClelland. Parallel distributed processing: 1, foundations / explorations in the microstructure of cognition. 1986.
- [Rumelhart, 1986] David E. Rumelhart. Learning representation by back-propagating errors. *Nature*, 323:533–536, 1986.
- [Sharp *et al.*, 2018] Michael Sharp, Ronay Ak, & Thomas Hedberg. A survey of the advancing use and development of machine learning in smart manufacturing. *Journal of Manufacturing Systems*, 48:170–179, 2018.
- [Siegelmann *et al.*, 1997] Hava T Siegelmann, Bill G Horne, & C Lee Giles. Computational capabilities of recurrent narx neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 27(2):208–215, 1997.
- [Smola & Schölkopf, 2004] Alex J Smola & Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.
- [Vapnik, 1995] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg, 1995.

[Werbos, 1974] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[Yan & Su, 2009] Xin Yan & Xiaogang Su. *Linear Regression Analysis: Theory and Computing*. World Scientific, 2009.