Project report

Gaussian process regression machine

DRAFT

Daniël S. van der Westhuizen

17158680

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at the University of Stellenbosch

STUDY LEADER: Johan A. du Preez

DATE: 2016

I, the undersigned, hereby declare that the work contained in this report is my own original work unless indicated otherwise.

Signature Date

Summary

It is often the case that there are systems which has multiple variables. In certain cases the value of one of the variables are unknown when the values of all the other variables are known. In this study we aimed to investigate and implement the use of regression for predicting the unknown value using both linear and kernel (non-linear) methods. We started with direct linear regression. From there we progressed to Bayesian linear regression to determine predictive distributions. After that we investigated the relation between linear regression and kernel-based methods. Finally, we used the kernel-based method on its own and also applied them to Gaussian processes. We tested the predictive methods with both simulated as well as real-life data.

Dit is dikwels die geval dat daar stelsels is wat verskeie veranderlikes het. In sekere gevalle is die waarde van een van die veranderlikes is onbekend wanneer die waardes van al die ander veranderlikes bekend is. In hierdie studie was daar gepoog om te ondersoek en te implementeer die gebruik van regressie vir die voorspelling van die onbekende waarde deur die gebruik van beide lineêre en kern (nie-lineêre) metodes. Ons het begin met 'n direkte lineêre regressive. Van daar het ons gevorder na Bayesiese lineêre regressie om voorspellende verspreidings te bepaal. Daarna het ons die verhouding tussen lineêre regressive en kern gebaseerde metodes ondersoek. Laastens het ons die kern gebaseerde metode op sy eie gebruik en ook toegepas op Gaussiese prosesse. Ons het die voorspellende metodes getoets op beide gesimuleerde en werklike data.

List of Figures

List of Tables

List of Symbols

C	covariance matrix
D	dimensionality
M	basis function ticks for each dimension (i.e. M^D basis functions)
k	kernel function
k	column matrix of kernel function
K	Gram Matrix
t	the output value of a corresponding x vector
t	column vector of the output values of the training data
x	a vector of n values defining a location on the n-dimensional input space
x	a column vector of different x points
α	parameter
β	parameter
δ	Dirac delta function
θ	parameter
σ	variance
ϕ	basis function
Φ	design matrix
ω	weight

Contents

1. Introduction

The goal of this project was to investigate and implement the use of regression for predicting future values of a signal using both linear and kernel (non-linear) methods. We will progress from direct linear regression, through Bayesian linear regression and finally Gaussian Processes. Test with both simulated as well as real-life signals.

The problem statement is as follows. Often it is encountered that there is a scenario in which there is an output parameter and one or more continuous input parameters. An estimation needs to be made of the output value for a point on the input space (a test point) for which it is not possible to determine the output empirically. Thus, it is necessary to employ mathematical techniques to determine a probability distribution of the output at that input point. In order to do so, one needs to first have a data set of known output values and their corresponding points on the input space. There points are known as training data.

In this project, two such techniques were implemented and tested. The first of these is the predictive distribution for linear regression. It involves the definition of basis functions over the relevant part of the input space. The technique can then be used to determine to what degree each of the basis functions determine the output parameter. The weights that were assigned to the basis functions can then be used to determine a prediction of a test point.

The second technique that was used is called Gaussian processes for kernel regression. It involves the use of a carefully chosen kernel function that relates one training point to another. The kernel function has to be evaluated for each possible pairing of training points. This can then be used in an algorithm to make predictions for the test points.

Both of these regression methods were implemented in python. It was done using the Jupyter Notebook software in the WinPython package. It was set up in various Jupyter notebook files. Both of the regression methods were tested on simple synthetic one-dimensional data to evaluate the functionality of the written code and its performance on simple data set. It was then tested on more complex data and finally on real data sets. The results were evaluated, conclusions were drawn, and recommendations were made.

2. Literature Study

Much of the mathematics that used in this project is described in Chapters 2, 3, and 6 of Bishop's *Pattern Recognition and Machine Learning*. Chapter 2 describes probability distributions. It includes a lengthy section describing various concepts of Gaussian distributions, some of which is important to regression methods that we will implement.

Chapter 3 describes linear regression. It explains how to set up a model of basis functions. It goes on to explain parameter distribution and predictive distribution methods. The methods involve training data being fitted to the basis functions in order to make predictions.

Chapter 6 explains how kernels can be constructed and how it can be used in regression. It is possible to define kernels without basis functions and used effectively in Gaussian processes for kernel regression.

Chapter 45 in Mackay's *Information Theory, Inference, and Learning Algorithms* is also applicable. It includes the mathematics of Gaussian processes for kernel regression, except with different notation.

3. Linear Regression

3.1.Parameter Distribution

.

First we demonstrate how an output that is a sum of weighted functions can be solved. The weights are iteratively improved until they converge. This is called the parameter distribution method. The equations used are described by Bishop (2006).

A two-dimensional multivariate Gaussian distribution is used as the prior distribution for this method. A two-dimensional input array over weight space was defined in python. The points in the input array are a row-by-row grid of the pixels in figure 1. The probability of each point is defined as one-dimensional vector in which the value of each entry is the relative probability of the corresponding input point. The values of each point on the probability array are shown as the colour on the corresponding input value's pixel .Warmer colours correspond to areas of larger values on the output array.

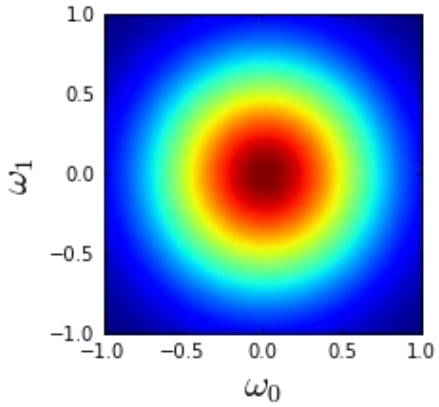

Figure 1: The prior distribution used for our demonstration of the parameter distribution method

A python function, called heatmap_random, was written to select randomly select a point on this space with a probability equal to its percentage of the total probability of all the points. For this demonstration, the output is be defined by equation (1).

$$y = \omega_0 + \omega_1 x \tag{1}$$

Below is an evaluation of equation (1) for six random samples.

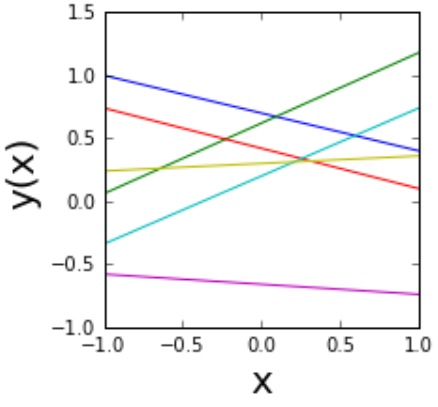

Figure 2: Equation (1) evaluated and plotted for six points on the weight space, selected with the heatmap_random function.

A point x on the x-axis is uniformly randomly selected on the interval [-1, 1]. Then for that point the equation

$$y = -0.3 + 0.5x + Normal(0, 0.2) \quad (2)$$

is evaluated. The equation shows that the actual values of ω_0 and ω_1 are -0.3 and 0.5 respectively. In this scenario, these two values are unknown and the parameter distribution method is used to recover it.

Using the particular random values of x and t that was selected, we determine and plot the probability of equation (1) over ω space, as shown on the left in figure 3. We then define a probability array for this image in the same way that the probability array for figure 1 was defined. This new probability array is then multiplied with the prior probability array to yield the distribution shown in the centre of figure 3. We then again generate samples from the weight space using the heatmap_random function, shown on the right.

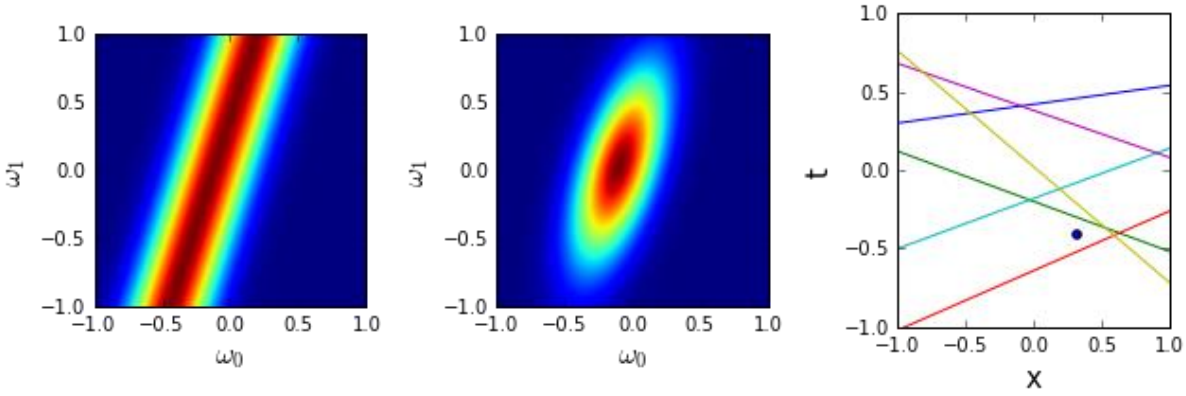

Figure 3: The steps in an iteration of the parameter distribution method. On the left is the pobability over weight space that equation (2) is true if x and t are evaluated to the point shown on the right. The plot in the centre, the new probability distribution over weight space, is the product of the plot on the left and the previous probability distribution over weight space.

This process can be repeated as many times as desired, each time using the new probability array as the prior. Below are the results after 20 iterations.

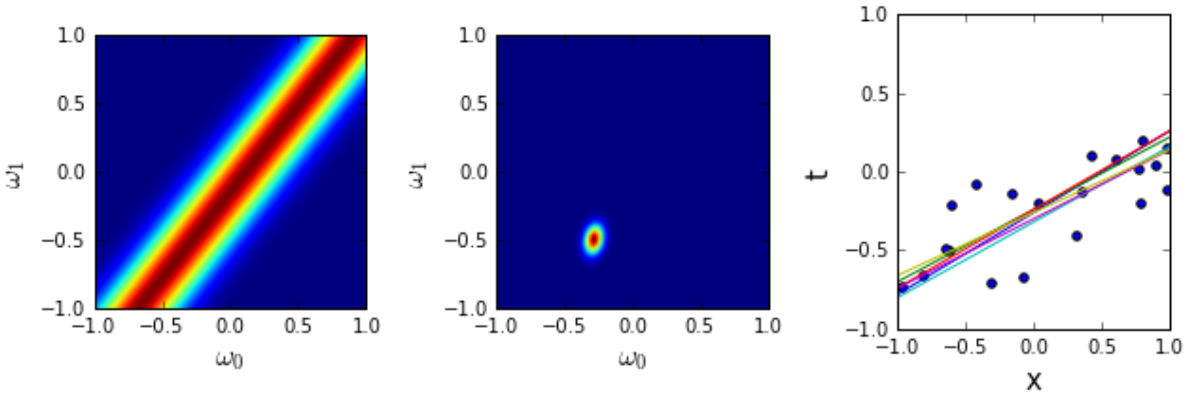

Figure 4: The same information shown in figure 3, except that this is the 20th iteration. The points that was randomly selected in each iteration is shown on the right.

This particular illustration is for the case of a one-dimensional input space and a single basis function. This method is not limited to these constraints, but a higher dimensional input space will require an exponentially larger number of computer calculations.

3.2.Predictive Distribution

3.2.1. One-Dimensional synthetic data

The predictive distribution method for the case of a one-dimensional synthetic input was implemented as described in here.

Here we aim to replicate the example in chapter 3.3 in Bishop's book. The code that is necessary to do so was implemented, but in such a way that it can easily be changed to accommodate multi-dimensional input, as will be necessary in a further demonstration.

The input x of the training data in this demonstration is 25 evenly spaced points on the interval [0, 1]. We assemble the vectors of these input points into a column vector denoted **x**, which is called the training input. The output is a sinusoid with Gaussian noise having a variance of 0.2, as shown in figure 5. The values of the output for each of these input points can be assembled into a corresponding column vector, denoted **t**.

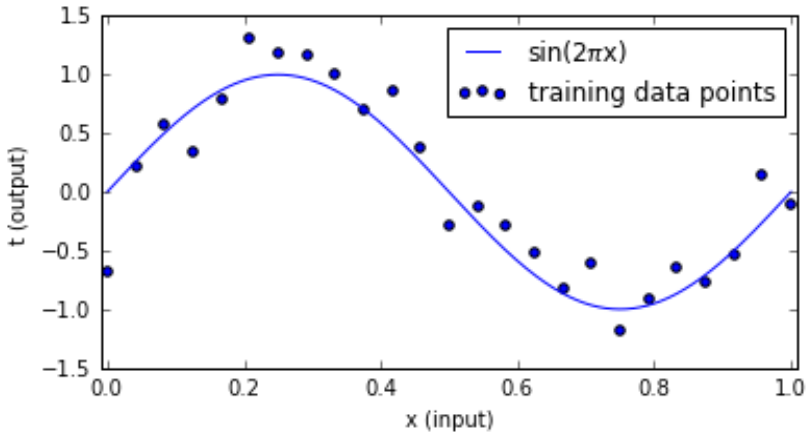

Figure 5: The training points used for the demonstrations in this chapter and the function they are derived from.

The basis functions that will be used for this demonstration consists of nine Gaussian distributions with means evenly spaced over the interval [0, 2] and each have variance of 0.2. We chose these basis functions because it is well spread over the area of interest, but not overly so. They are shown in figure 6

$$\phi_j(x) = exp\left\{-\frac{(x-\mu_j)^2)}{2s^2}\right\} \quad (3)$$

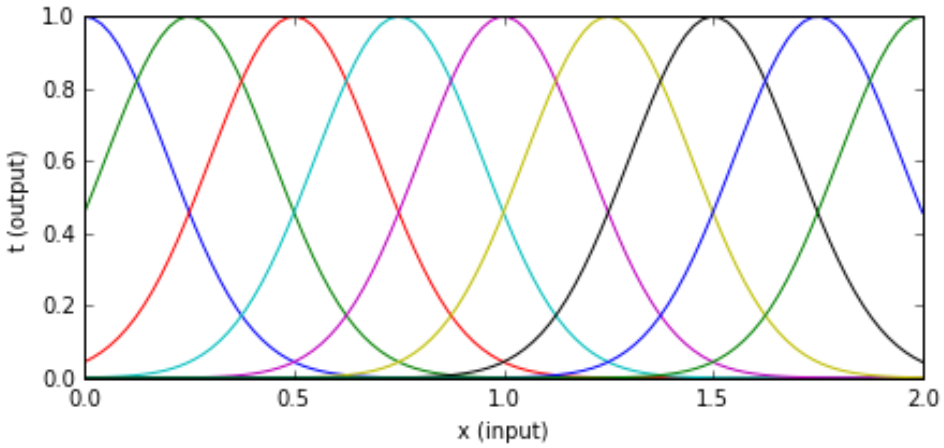

Figure 6: The basis functions for the predictive distribution method demonstration in this chapter.

For this demonstration the hyperparameters α and ß are arbitrarily chosen to be one and five. β was chosen based on randomly trying different values until one that works well was found. Specifically, it was α=1 and β = 5.

Each of the basis functions can be defined for each of the training data points. These definitions can be assembled into a matrix called the design matrix, denoted by Φ. It is shown in equation (4). We also define S_N, which will be necessary for further calculations in.

$$\mathbf{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix} \tag{4}$$

$$\mathbf{S}_N^{-1} = \alpha\mathbf{I} + \beta\Phi^{\mathsf{T}}\Phi \tag{5}$$

We now need a test data set. Unlike the training data set, only the input vectors of the test set have to be provided. There is no need to assemble all he test input vectors into a vector, because the prediction for each test point is made independently from the others. For this demonstration, we define the test vectors as a set of many equally spaced points over the interval [0, 2].

We now have everything we need to evaluate the means and variances of the prediction. These are defined in equations (6) and (7). The probability distribution for each test point, defined in equation (8), of the prediction is a Gaussian distribution defined over the output axis.

$$\mathbf{m}_N = \beta\mathbf{S}_N\Phi^{\mathsf{T}}\mathbf{t} \tag{6}$$

$$\sigma_N^2(\mathrm{x}) = \frac{1}{\beta} + \phi(\mathrm{x})^T \mathbf{S}_N \phi(\mathrm{x}) \tag{7}$$

$$p(t|\mathrm{x}, \mathbf{t}, \alpha, \beta) = \mathcal{N}(t|\mathbf{m}_N^T \phi(\mathrm{x}), \sigma_N^2(\mathrm{x})) \tag{8}$$

The result of the prediction is shown in figure 7. This method evidently works well with for the given hyperparameters. We attempted to implement a means of calculating the most optimal hyperparameters, but were unable to succeed. We decided to ensure that the hyperparameters we used here would always be adequate by always normalising the data sets we test this technique on.

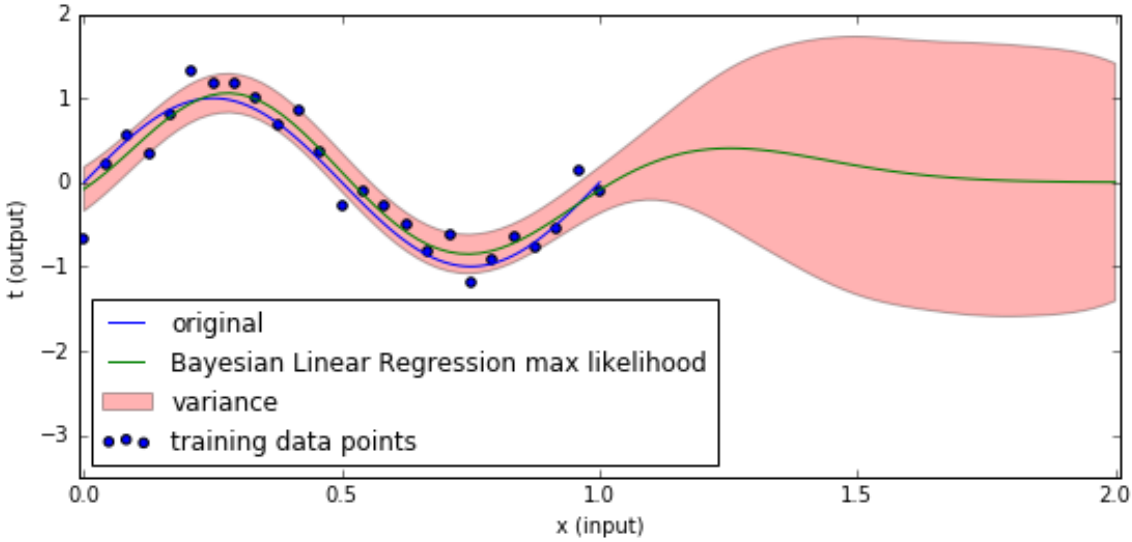

Figure 7: The predictive disribution for linear regression

3.2.2. Two-Dimensional Synthetic Data

Here we show how we used the predictive distribution method for linear regression and apply it to a synthetic data set with two inputs. We did this to see how well the method holds up when using a more complicated data set. This was also done so that we could have visual confirmation that the algorithm works when using a data set with multiple inputs

We arbitrarily defined a synthetic data that will be suitable for our demonstration. It is shown in figure 8. In the figure, warmer colours correspond to larger output values with the peaks having values of about one. Colder colours correspond to lower output values, with the lowest at zero.

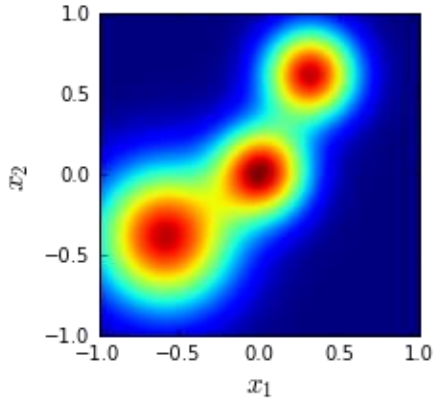

Figure 8:Arbitrarily defined synthetic data with a two-dimensional input, used for demonstrating the predictive distribution in this subsection. Warmer colours correspod to larger output values.

We generated random training points by uniformly randomly selecting values in the interval [-1, 1] for both the input dimensions. The points are shown in figure 9.

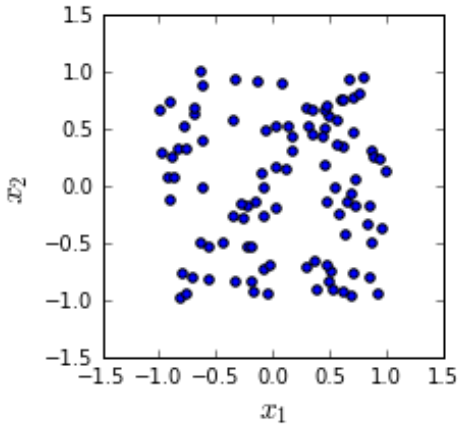

Figure 9: The training points used in for the demonstration in this subsection.

The means of the Gaussian basis function for each input dimension has to be chosen so that it corresponds to the mean and variance of the training points' location on that dimension. Alternatively, the training data can be normalised so that for each input dimension, it has a mean of zero and a variance of 0.5. In this study we opted to do the latter. The means and variances of the dimensions of the two-dimensional synthetic data set used in this subsection are already close to these specifications, so in this case it is not necessary to apply the normalization. For most real-life data sets, it will definitely be necessary.

Here we define means for the basis functions we will use for the demonstration in this subsection. If we would use, say, nine Gaussian basis functions for the one-dimensional case, then for an N-dimensional input space, there needs to 9^N basis functions. Their means must be evenly spaced throughout the normalised input space. Since the data set has two dimensions, it will have $9^2 = 81$ basis functions. The means of these basis functions are plotted in figure 10. Since all the input dimensions have the more or less the same variance they did in the demonstration of the technique for data with one-dimensional input, we continued using of 0.2, 1, and 5 as the values of σ, α, and β respectively.

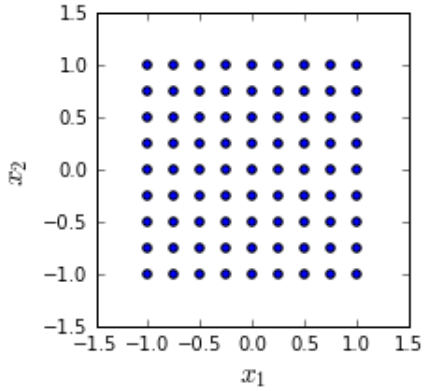

Figure 10: The means of the multivariate Gaussian basis functions used for the demonstration in thus subsection

We defined a test data set consisting of ten thousand points in a grid similar to the one shown in figure 10. We define it thus so that we can use the output values that will be yielded as pixels in a heat map image.

We continued to evaluate equations (4) - (8) in python just as we did for the data with a one-dimensional input. We fixed problems with the code that was not apparent when there was only one input dimension. The predicted means is shown as a heat map image in figure 11.

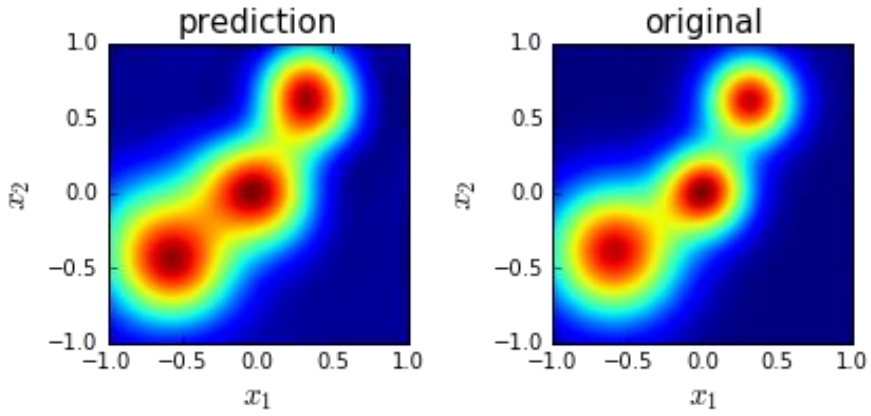

Figure 11: The predictive distribution for linear regression (left) of the test points shown in figure 10. The distribution that the training points were taken from is shown on the right for easy comparison.

As can be seen in the figure the prediction method does indeed hold up for this more complicated data set. The increased size of the data set and the large number of basis functions caused the computations to take a noticeable duration of time, namely a few seconds. This will exponentially increase as the number of dimensions and basis functions increase. We tested the effects of using fewer basis functions, specifically basis functions whose means are a 5x5 grid. The results were greatly inferior, showing us that it is indeed necessary to use as many basis functions as we did.

So, now that it is confirmed that the code works on this synthetic data set, we can proceed to test it on real world data sets. The data sets should not have many dimensions or the algorithm's computation time will be too long. For example, just setting up basis function

means (for M=9) for data with 9 or more dimensions requires more memory than the average computer has.

3.2.3. Real World Data Sets

In this subsection we evaluate the method's performance when tested on real-world data sets. The code used for each evaluation is provided in appendix C.

We decided to try the method with a wine quality data set. It has eleven input dimensions ad 1'599 entries. Its input dimensions are the properties of a wine product. Its output is a rating out of ten for that product.

It was here that we realised the problem of having too many input dimensions and decided to use only five of the attributes so that not too much computation is needed. In order to reduce computation time further, we limited the number of basis functions 5^5. We normalised the data to have a zero mean and a variance of 0.5. The necessity of this was explained in the previous section. We calculated the prediction just as we did before, using the same values of σ, α, and β.

Despite the limitations, the prediction took about a minute to be computed. In addition, the limitations had a price. The average of the predicted ratings was 4.61 but the actual ratings are 5.70. The mean predictions deviate from the actual value by an average of 11%. We decided to do further tests with this method only on data sets with a small number of dimensions.

Here follows another demonstration of predictive distribution for linear regression. Here we use a climate data set. The attributes of this data set are the date, minimum temperature, maximum temperature, and precipitation for every day of one year. We let the input space be the date, the minimum temperature, and the maximum temperature of a weather station. The precipitation was assigned to be the output.

We normalised the data and used every second entry as the training points. All the samples were used be the test data. The results of using various values of M are shown in table 1. The computation for M = 9 took a few seconds and for M = 10 it took several seconds. Any values of M lower than nine had negligible computation time. As can be seen in the table the accuracy of the prediction improves with more basis functions. Since the output data in this data set was only specific to the integer, we can consider this prediction satisfactory.

M	4	5	6	7	8	9	10
Prediction error	1.2	0.601	0.465	0.44	0.425	0.42	0.418

Table 1: The results of using various values of M when using the predictive distribution for linear regression to make predictions as described on the text.

4. Kernel Regression

When using kernel regression, usage is made of the kernel function k(**x**, **x**') that relates any 2 points on the input space. A Gram matrix **K**, whose elements are given k($\mathbf{x}_n$, $\mathbf{x}_m$), can be defined to relate each possible pairing of two input vectors. These kernel functions can be constructed from previously defined basis functions, like those used for linear regression. Since the basis functions has to be defined first in order to construct the Gram matrix in this way, any new prediction methods that uses the Gram Matrix will also be subjected to the same problematic restrictions that linear regression has. The restriction is that the number of basis functions needed is M^D. M is the number of basis functions that has to be defined for the variation of one input variable while all the other input variables are held constant.

A solution to this problem is to define kernel functions without involving basis functions. There are prerequisites which have to be satisfied when doing so. The Gram has to be a positive semi-definite matrix.

4.1.Nadaraya-Watson Model

Here we show how we attempted to replicate figure 6.3 in Bishop. In this example, the Nadaraya-Watson model is used to evaluate the output of the test input points using equation (9).

$$p(t|x) = \frac{\sum_n f(\mathbf{x} - \mathbf{x}_n, t - t_n)}{\sum_m \int f(\mathbf{x} - \mathbf{x}_m, t - t_m) dt} \tag{9}$$

f(x, t) is a zero-mean isotropic Gaussian with variance σ. m and n are any integers in the interval [0, N-1], where N is the number of training points. The input of the training points are evenly spaced points over the input space interval [0, 1] and the output has Gaussian noise. It is plotted in figure 12.

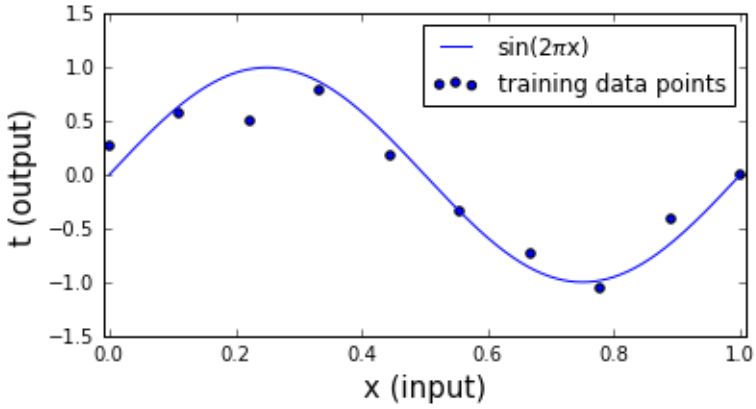

Figure 12: The training data points used for the demonstration of the Nadaraya-Watson model.

The equation for the best prediction is the same as equation (9), except that the *t* argument of f(x, t) is always zero and thus can be ignored. The integral becomes obsolete (equal to one) because its argument no longer contains the *t* variable. We are thus left with equation (10).

$$y(\mathbf{x}) = \frac{\sum_n g(\mathbf{x} - \mathbf{x}_n) t_n}{\sum_m g(\mathbf{x} - \mathbf{x}_m)} \tag{10}$$

The g(x) is the same as the f(x) of before except that the multivariate Gaussian with variance is now only over the input space.

We implemented the code necessary to calculate equation (10) in python. Below is the plot of the prediction after experimenting with different values of σ until one that works well (σ = 0.1) was found.

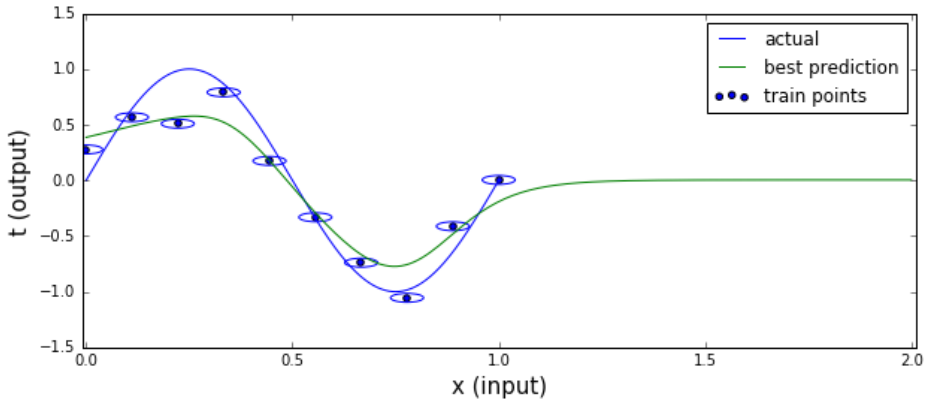

Figure 13: The prediction made by the Nadaraya-Watson model for the training points shown.

We also tried this technique on synthetic data with a two-dimensional input. We will use the same arbitrary data set we used for linear regression. It is shown in figure 14.

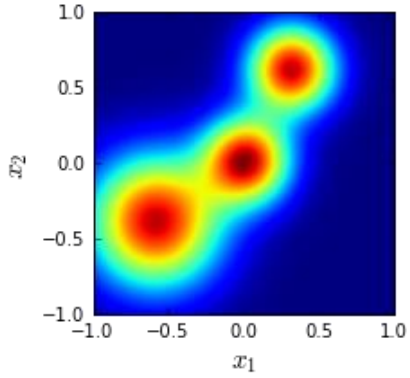

Figure 14: An arbitrary data set with a two-dimensional input, used for the illustration of the Nadaraya-Watson model.

Points on the input space were uniformly randomly selected over the intervals x_1 = [-1, 1] and x_2 = [-1, 1]. These points are plotted in figure 15.

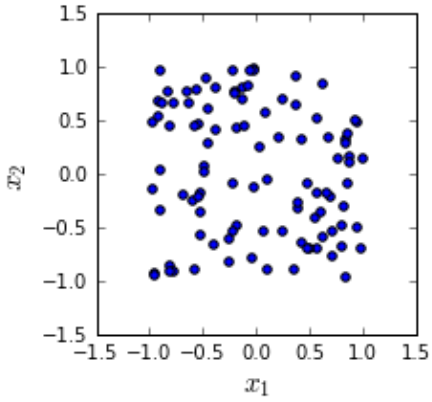

Figure 15: The training points used in for the demonstration in this subsection.

The most likely prediction made by the Nadaraya-Watson model using these training points is shown in the figure 16. The prediction is not very good and is very dependent on my choice of σ. Worse yet; it took several seconds to be computed.

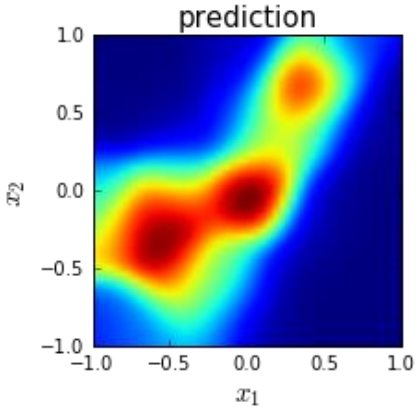

Figure 16: The prediction made using the Nadaraya-Watson model fitted with the training poins in figure

4.2.Gaussian Processes for Regression

Here follows an explanation of Gaussian process for kernel regression. All equations are from Bishop (2006). If input values x_1 ,..., x_N and corresponding target values $t_N = (t_1, ..., t_N)^T$ are provided, it can be used to predict the target value t_{N+1} separately for each of the given input values. The given input value for which a prediction is currently being made will be denoted x_{N+1}.

One widely used kernel function for Gaussian processes for kernel regression is given by

$$k(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 exp\{-\frac{\theta_1}{2}||\mathbf{x}_n - \mathbf{x}_m||^2\} + \theta_2 + \theta_3 \mathbf{x}_n^T \mathbf{x}_m \tag{11}$$

where θ_0,...., θ_3 are parameters.

C_{N+1} is an (N+1) × (N+1) covariance matrix with elements given by

$$C(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm} \tag{12}$$

δ_{nm} is the Dirac delta function, *n* is the row number, and *m* is the column number, which is zero when n and m are equal. The second term of the equation above is thus nonzero only for the main diagonal of C_{N+1}.

C_{N+1} can be partitioned into four different sections, each with their own notation as shown in equation (13) (equation 6.65 in Bishop [2006]). Equations (2.81) and (2.82) in Bishop's source can then be used to derive equations (14) and (15). These last two equations are the mean and variance of the prediction made by Gaussian processes for kernel regression.

$$C_{N+1} = \begin{pmatrix} C_N & \mathbf{k} \\ \mathbf{k}^T & c \end{pmatrix} \tag{13}$$

$$m(\mathbf{x}_{N+1}) = \mathbf{k}^T C_N^{-1} \mathbf{t} \tag{14}$$

$$\sigma^2(\mathbf{x}_{N+1}) = c - \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{k} \tag{15}$$

4.2.1. One-Dimensional Synthetic Data

Here follows a demonstration of Gaussian processes for kernel regression for a simple example. As usual, samples of sinusoid with Gaussian noise will be used as training points.

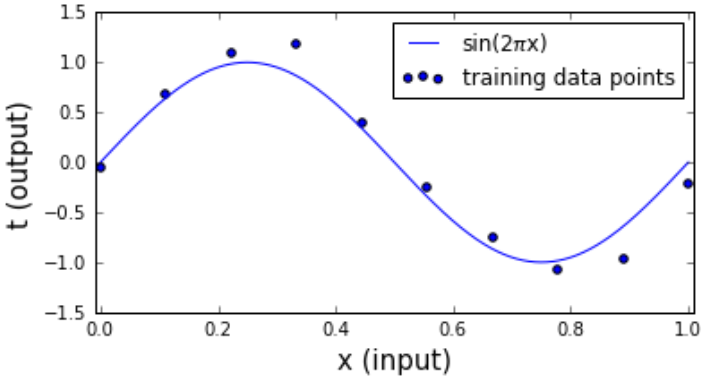

Figure 17: The training points used for the demonstration of a Gaussian processes for kernel regression.

After some experimentation and examining the Bishop's examples, we intuitively decided to initially use 1, 1, 5, and 1 as the respective initial values of the θ parameters. We found that the higher values of β has the effect of decreasing the variance of the prediction. These parameters values and the aforementioned training points were to evaluate $k(x_n, x_m)$. This was then used to calculate C_{N+1}, which was in turn used to predict the mean and variance of the output for inputs over the interval [0, -2]. The results are displayed in figure 18.

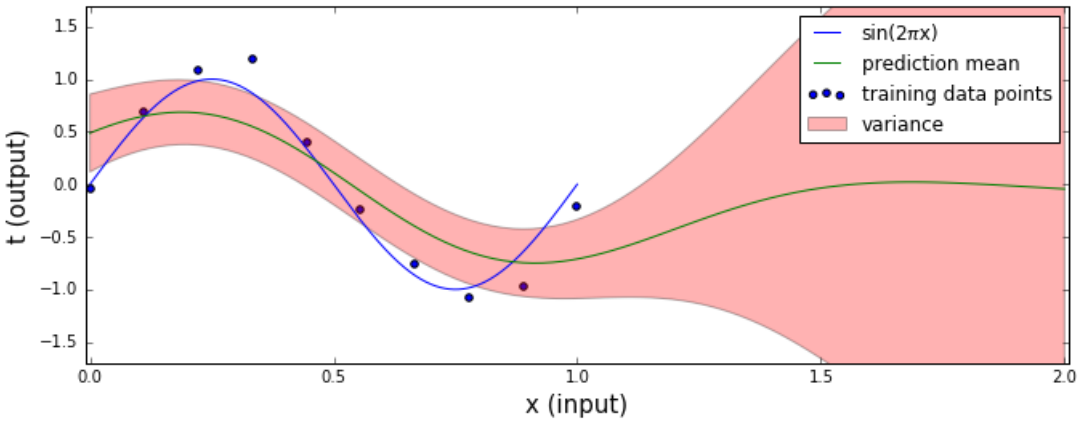

Figure 18: The training point and predictions made using Gaussian processes for kernel regression with selected parameter values.

The result is not satisfactory even after carefully selecting parameter values and thus this technique as described thus far has hardly any practical use. There exists a solution to this predicament. Bishop describes a way of calculating the probability gradient for each parameter value, which can then be used to find better values for the parameters. The equation for determining this is given below.

$$\frac{d}{d\theta_i}\ln p(\mathbf{t}|\theta) = -\frac{1}{2}Tr\left(C_N^{-1}\frac{dC_N}{d\theta_i}\right) + \frac{1}{2}\mathbf{t}^T C_N^{-1}\frac{dC_N}{d\theta_i}C_N^{-1}\mathbf{t} \tag{16}$$

The i variable can take on positive integer values up to the number parameters in the kernel function that is being used in this method. The one we used has four parameters. We wrote code to calculate the equation for each of the parameters. We used the same aforementioned values of the parameters as prior values for calculating the equation. After they were all calculated, we subtracted the value of the equation for each parameter from the respective parameter. We found that the new values of the parameters make the regression give better predictions. We repeated this process 20 times. The values of the parameters over the course of these iterations are shown in figure 19.

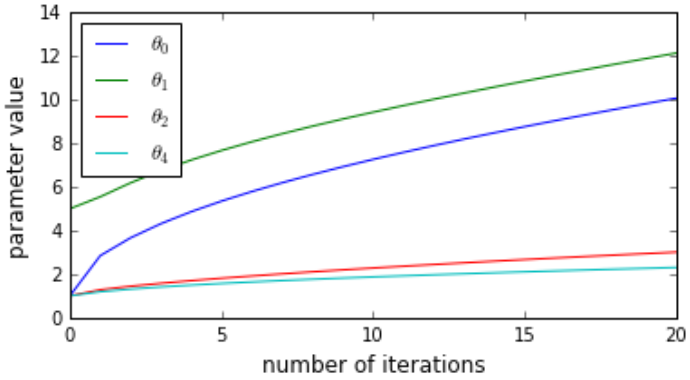

Figure 19: The values of the parameters of equation (11) after repeatedly subtracted the value of equation (16) for each parameter, and then subtracting it from that parameter. The training points in figure 17 was used for the evaluation of these equations. The values of the parameters at zero on the iteration axis are the arbitrarily prior values of the parameters chosen before iterations of the optimization.

Figure 20 shows the prediction using the parameters obtained after 20 iterations. This is good prediction.

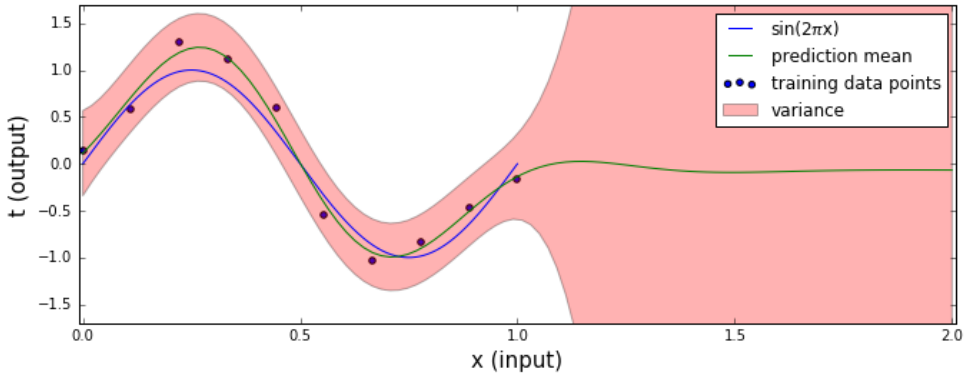

Figure 20: The results of using Gaussian processes for kernel regression after twenty iterations of optimization.

Using very many iterations or choosing very high prior parameters yields the problematic result in figure 21.

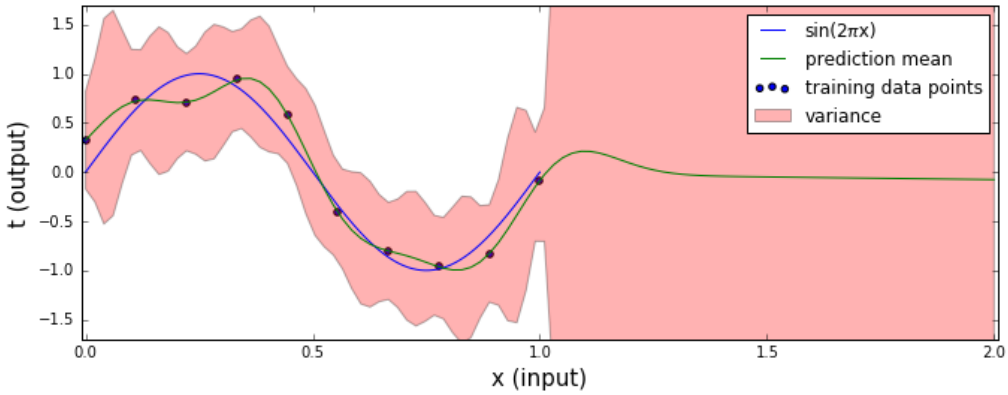

Figure 21: The results of using Gaussian processes for kernel regression after 500 iterations of optimization.

Although all the training points were predicted perfectly, the prediction no longer represents the sinusoidal wave that the training points were derived from. This is an example of what one would call “over fitting”. This technique would thus work very well if the training points are noiseless (unlike this example). Care needs to be taken when applying this parameter estimator.

4.2.2. Two-Dimensional Synthetic Data

Here we show we took Gaussian processes for kernel regression and applied it to synthetic data with a two-dimensional input. We will once again use the data set shown in figure 22.

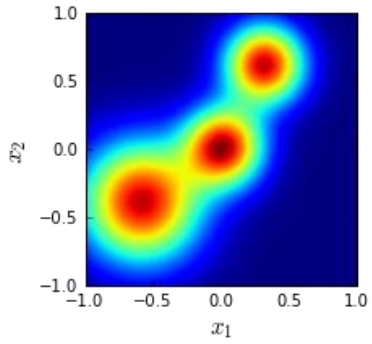

Figure 22: An arbitrarily defined synthetic data set with two inputs.

Again we use one hundred uniformly random points over the intervals x_1 = [-1, 1] and x_2 = [-1, 1], as shown in figure 23.

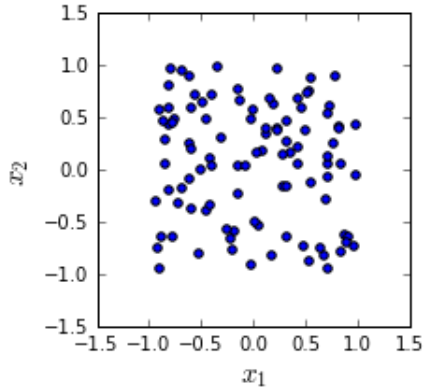

Figure 23: The training points used for the demonstration in this subsection.

In order to prevent the calculations from taking too long, we reduced the number of test points and by extension the resolution of the image render of the result, shown in figure 24.

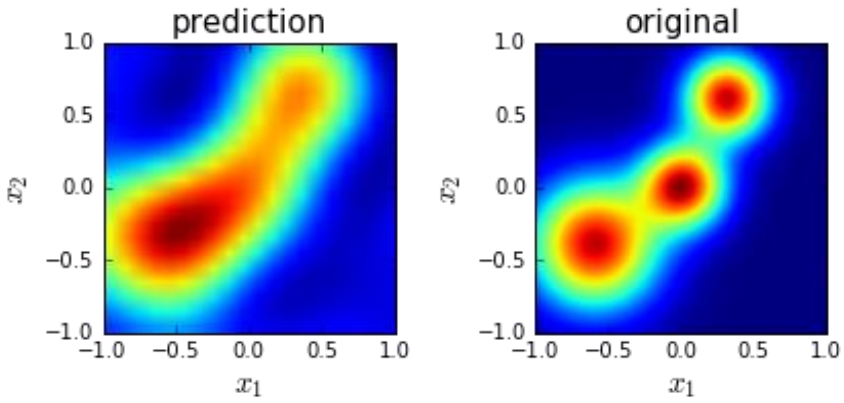

Figure 24: The prediction made by Gaussian processes for kernel regression (left). The distribution that the training points were taken from is shown on the right for easy comparison.

Bishop explains that number of calculations needed for this method is the function $O(N^3)$, where N is number of training points. This is contrary to linear regression, where the computational complexity function is $O(M^3)$ where M is the number of basis functions, which exponentially increases with the number of dimensions. It is thus unfeasible to use linear regression on data sets with large numbers (more than about five) of dimensions. Contrariwise, using Gaussian process for kernel regression is feasible with data sets with a large number of dimensions. Gaussian processes for kernel regression are instead practically limited by the number of training and test points.

We applied optimization just as we did for the one-dimensional data. It was computed very quickly. Figure 25 shows how the parameters changed over twenty iterations of optimization.

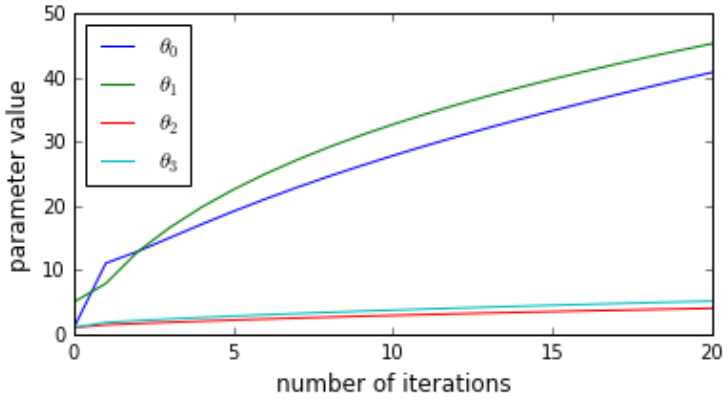

Figure 25: The values of the parameters during the course of twenty iterations of optimization.

Figure 26 shows the result of the predictions made using the parameters yielded after the optimization. Even though the resolution is low, we can still see that the predictions' distributions are accurate

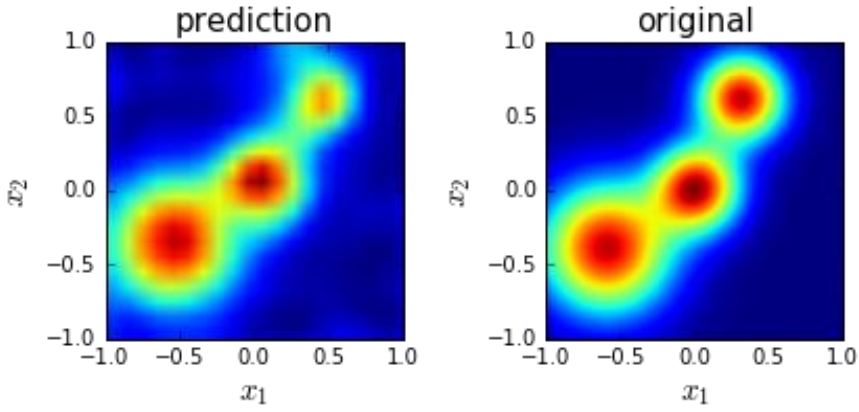

Figure 26: The prediction made by Gaussian processes for kernel regression (left) after twenty iterations of optimization, displayed as a heatmap. Note that the range of the ouput of the picture on the left is a bit smaller than the one on the right. The distribution that the training points were taken from is shown on the right for easy comparison.

We tested with different numbers of training points. The results are shown in table 386. It would appear that the calculation time increases quadratically with the number of training points. For some reason the average absolute value of the predictions' error doesn't improve with the number of training points. It rather converges to a constant.

Since figure 26 shows us that the shape of the predictions' distribution is close to the original, one might think that the prediction can be corrected simply by multiplying the predictions with a constant. This would not be right because, as illustrated in the next subsection, this method predicts the training points accurately. We believe the problem must therefore be that the predictions of the test points that are not close enough to the training points have an undesired decrease in their values.

# of training points	10	20	50	100
prediction time (seconds)	0.74	2.55	34	139
optimization time per iteration (seconds)	0.007	0.025	0.26	0.75
Average error	0.22	0.2	0.17	0.17

Table 2: The results of using Gaussian processes for kernel regression for various numbers of training points.

4.2.3. Real-World Data Sets

Here we show how we tested Gaussian processes for kernel regression. The code used for each evaluation is provided in appendix C. The first set we used was the same wine quality data set on which we tested linear regression. We used 50 of the entries as training points and used another 500 entries as test points. We made predictions of both the training points and the test points. The computation time of the prediction (both before and after optimization) was 44 seconds. The optimization took only a fraction of that time. The prediction of the test points had an average error of 0.672. The prediction of the training points had an average error of 0.0719.

We then tested it on a diabetes data set, which has eight inputs and one binary output. This data set has some missing data. We filled each missing data entry with the average of its respective attribute. We calculated the prediction with 500 test points as before, treating the output as a continuous variable in which the numbers zero and one are assigned to the two classes. We classified each test point to which binary class the prediction's value was closest to. Table 3 shows the results before and after optimization.

Training Points	50	100
Prediction computation time	49	200
number of erroneous classifications before optimization	150	135
number of erroneous classifications after optimization	139	138

Table 3: The results of using 50 or 100 training points when using Gaussian processes for kernel regression to make predictions based on the diabetes data set.

5. Conclusions

The methods that were tested in this project have limited usefulness as means of prediction. They work best when the input coordinates of the test data occurs in the same general area as that of the training data for each of the input dimensions. The method of using the predictive distribution for linear regression is feasible only when there are a small number of input dimensions. The method of using Gaussian processes for kernel regression, using the provided kernel function, can easily handle data with a large number of dimensions, but becomes unfeasible when the number of training points become large. It also has the problem reduced values for its predictions.

We would recommend that ways be found solves the unfeasibility problems stated previously. The flaws could be inherent in the methods or it could be a result of the way we implemented it. We also recommend finding methods that can make accurate predictions of test data that is not localized to the training data's general area on the input dimensions.

Appendices

Appendix A: Project Planning Schedule

The project leader and the student met shortly after the project allocations were released. It was agreed on that the student would take time during the June vacation to go through selected pieces of relevant literature. The student would familiarise himself with the content of the literature and identify parts of the literature that contains content that can be of use in this project.

The project leader and the student did not decide that a precise schedule has to be followed. The project progress during the second semester was more or less assigned to the schedule below. The student had to use his own judgment to decide if the sufficient progress has been made.

- First two weeks the first term: Study selected relevant sections in literature.
- Rest of the first term: Write the software necessary to implement linear regression. Test on all the necessary types of data sets and debug as necessary. Make appropriate observations
- Mid-semester break and first two weeks of second term: Write the software necessary to implement kernel-based regression. Test on all the necessary types of data sets and debug as necessary. Make appropriate observations.
- Rest of the second term: Finalise the report. Make changes and additions to previous work as appropriate.

Appendix B: Outcomes Compliance

Problem Solving

It was identified that the problem is there cases that a variable in a system is sometimes unknown while there are other known variables that remain known even when the former variable becomes is no longer known. It was also known that there is no mathematical model which relates the variables to each other. The solution was to make predictions using statistical models to determine the influence the known variables has on the unknown variables.

It was determined that the solution could be realised by consulting relevant literature to find and understand suitable models. Software would then be written to implement these models. These models would then be tested with simple synthetic data to determine how well they work and whether it was implemented correctly. The models would then be tested on real data and the results would be analysed. The implementation of the models and evaluation of the tests made on them is in the body of this report.

Application of Scientific and Engineering Knowledge

The problems was analysed and solutions were presented in a formal manner. The underlying concepts and theories of the solutions are explained in this report. In cases of uncertainty, informed judgments were made. The statistical models in this project have its basis in mathematical definitions that comply with the laws of physics.

The engineering problem that was explored in this project has already been the subject of research. Existing techniques of solving the problem were utilised and evaluated. An ultimate solution has yet to be determined for the problem and there is still room for more research on this topic.

Engineering Design

It was identified that the problem of predicting an unknown value in a system when other values in the system are known. The design process was explained in appendix A. Literature was consulted to acquire requisite knowledge which was then evaluated. The design task of writing the software to perform the desired algorithms was completed and optimised. We evaluated multiple solutions, judging them and comparing them. The report explains the logic behind the design and shows information that influenced it.

Investigations, experiments and data analysis

We planned our investigation by selecting relevant literature and studying to find solutions. We selected and use the Jupyter software in the WinPython package. We gave analyses and made interpretations of the results of our experiments. Conclusions were drawn in chapter 5.

Engineering Methods, Skills and Tools, Including Information Technology

In this project, appropriate regression methods were identified. They were implemented as custom made computer applications. They were tested on data sets in a logical manner. The results are presented with accompanying assessment and critique.

Professional and Technical Communication

The report was written according to the guidelines provided by the engineering faculty were studying in. The designs and experiments in this report are effectively illustrated with figures. The underlying mathematics are stated and a list of symbols are provided

Independent Learning Ability

As previously explained in this report, literature was studied and the acquired knowledge was reflected on. The sources are stated. Different regression methods was assessed and comprehended independently. The acquired knowledge was then used to implement computer programs without formal instruction.

Appendix C: Code

The code for doing linear regression is as follows. Note that some of the lines are split. After declaring the data set array as explained in the code's comments, the prediction can be calculated with the following code

```
import numpy as np
import pylab as pl
from numpy.linalg import inv

def basis(x,mu,s):
    #the function for evaluating a basis function given a point in N-dimensional input space
and also a mean and a variance.
    D = np.shape(mu)[0]
    magnitude = 0
    for a in range(0,D):
        magnitude += (x[a] - mu[a])**2
    magnitude = np.sqrt(magnitude)
    phi = np.exp(-(magnitude**2)/(2*(s**2)))
    return phi

#The following variables has to be declared first:
#       train_x: an (n, d) array, where d is the number of dimensions and n is the number of
training points
#       train_y: an (n, 1) array, where n is the number of training points
#       test_x: an (n, d) array, where d is the number of dimensions and n is the number of
test points
#       test_y: an (n, 1) array, where n is the number of test points

D = 5 #input dimensionality
M = 5 #basis function ticks for each dimension (i.e. M^D basis functions)
alpha = 1
beta = 5
s = 0.2

#Normalization
for a in range(0,D):
    train_x[:,a] = (train_x[:,a] - np.average(train_x[:,a])) /
(np.average(np.abs(train_x[:,a])))

means = np.zeros((M**D,D))
done = 0
mu_vector_amount = M**D
row_count = 0
column_count = 0
for a in range(0,mu_vector_amount*D):
    means[row_count,column_count] = ( (row_count//(M**column_count))%M*2/(M-1)) - 1
    row_count += 1
    if (row_count == mu_vector_amount):
        row_count = 0
        column_count += 1

design_matrix = np.zeros((1000,M**D))
for a in range(0,M**D):
    for b in range(0,1000):
        design_matrix[b,a] = basis(train_x[b,:],means[a,:],s)
S_N = inv( alpha*(np.identity(M**D)) + beta*np.dot(design_matrix.transpose(),design_matrix) )
m_N = beta * (   np.dot( np.dot(S_N,design_matrix.transpose()),
np.atleast_2d(train_y).transpose() ) )

pred_means = np.zeros(np.shape(test_x)[0])
for b in range(0,np.shape(test_x)[0]):
    phi_vector_pred = np.zeros(M**D)
    for a in range(0,M**D):
        phi_vector_pred[a] = basis(test_x[b,:],means[a],s)
    phi_vector_pred = np.atleast_2d(phi_vector_pred).transpose()
    pred_means[b] = np.dot( m_N.transpose(), phi_vector_pred )

pred_error = np.average(np.abs(pred_means - test_y))
```

After running the code the predictions of the output values is in the *pred_means* array. The average error of the prediction is the value of the *pred_error* variable.

The code for using Gaussian processes for kernel regression is as follows. Note that some of the lines are split. Explanations are included in the comments.

```
import numpy as np
import pylab as pl
from numpy.linalg import inv

def com_gauss_proc_kern(x_n, x_m, theta0, theta1, theta2, theta3):
    #the name is a shortened form of "common Gaussian process kernel".
    #This function calculates equation 6.63 for given parameters and 2 given x vectors.
    #Note: x_n and x_m has to be arrays. It has to be defined with functions like "np.array"
or
    #"np.atleast_2d". Declarations like "x = 1" or "x = [3,3]" will not work.
    #The theta arguments are single values
    x_n = x_n.ravel()
    x_m = x_m.ravel()
    term1 = theta0*np.exp( (-theta1/2)*((np.sqrt( ((np.subtract(x_m,x_n))**2).sum()  ))**2)  )
    result = term1 + theta2 + theta3*np.sum(x_n*x_m)
    return result

def gauss_proc_kern_reg_cov(x_train, x_test, theta0, theta1, theta2, theta3, beta):
    #the name is a shortened form of "Gaussian process kernel regression covariance"
    #x_train is a an (n, d) array, where d is the number of dimensions and n is the number of
training points.
    #if x train is one-dimensional, use np.atleast_2d on it before passing it ti this
definition
    #returns: cov N, k, k T, c, cov_full, inv_cov_N
    n, d = np.shape(x_train)
    x_full = np.append(x_train,np.atleast_2d(x_test)).reshape(n+1,d)
    cov = np.zeros([n+1,n+1])
    for a in range(0,n+1):
        for b in range(0,n+1):
            cov[a,b] = com_gauss_proc_kern(x_full[a,:], x_full[b,:], theta0, theta1, theta2,
theta3 )
            if (a==b):
                cov[a,b] += (1/beta)
    inv_cov_N = inv(cov[:n,:n])
    return cov[:n,:n], cov[:n,n], cov[n,:n], cov[n,n], cov, inv_cov_N

def gauss_proc_kern_reg_pred_mean(k_T,C_N,t, inv_cov_N):
    #the name is a shortened form of "Gaussian process kernel regression prediction mean"
    o = np.dot(k_T,inv_cov_N)
    i = np.dot(np.atleast_2d(o),np.atleast_2d(t.ravel()).transpose() )
    return i

def gauss_proc_kern_reg_pred_var(c,k_T,C_N,k, inv_cov_N):
    #the name is a shortened form of "Gaussian process kernel regression prediction variance"
    o = np.dot(k_T,inv_cov_N)
    i = c - np.dot(np.atleast_2d(o),k )
    return i

def theta0_deriv(x_n, x_m, theta0, theta1):
    x_n = x_n.ravel()
    x_m = x_m.ravel()
    result = np.exp( (-theta1/2)*((np.sqrt( ((np.subtract(x_m,x_n))**2).sum()  ))**2)  )
    return result
def theta1_deriv(x_n, x_m, theta0, theta1):
    x_n = x_n.ravel()
    x_m = x_m.ravel()
    result1 = theta0*np.exp( (-theta1/2)*((np.sqrt( ((np.subtract(x_m,x_n))**2).sum()  ))**2)
)
    result = result1*(-1/2)*((np.sqrt( ((np.subtract(x_m,x_n))**2).sum()  ))**2)
    return result
def theta3_deriv(x_n, x_m, theta3):
    x_n = x_n.ravel()
    x_m = x_m.ravel()
```

```
    result = np.sum(x_n*x_m)
    return result

def gauss_proc_kern_reg_deriv_values(x_train, theta0, theta1, theta2, theta3, beta):
    #the name is a shortened form of "Gaussian process kernel regression derivative values"
    #It returns all the values needed to solve equation (6.70)
    #x train is a an (n, d) array, where d is the number of dimensions and n is the number of
training points.
    #if x train is one-dimensional, use np.atleast_2d on it before passing it ti this
definition
    #returns: cov_N, k, k_T, c, cov_full
    n, d = np.shape(x_train)
    cov = np.zeros([n,n])
    cov_theta0_deriv = np.zeros([n,n])
    cov_theta1_deriv = np.zeros([n,n])
    cov_theta2_deriv = np.zeros([n,n])
    cov_theta3_deriv = np.zeros([n,n])
    for a in range(0,n):
        for b in range(0,n):
            cov[a,b] = com_gauss_proc_kern(x_train[a,:], x_train[b,:], theta0, theta1, theta2,
theta3 )
            cov_theta0_deriv[a,b] = theta0_deriv(x_train[a,:], x_train[b,:], theta0, theta1 )
            cov_theta1_deriv[a,b] = theta1_deriv(x_train[a,:], x_train[b,:], theta0, theta1 )
            cov_theta2_deriv[a,b] = 1
            cov_theta3_deriv[a,b] = theta3_deriv(x_train[a,:], x_train[b,:], theta3 )
            if (a==b):
                cov[a,b] += (1/beta)
    inv_cov = inv(cov) #inverse of the matrix covariance
    return cov, inv_cov, cov_theta0_deriv, cov_theta1_deriv, cov_theta2_deriv,
cov_theta3_deriv

def gauss_proc_kern_reg_deriv(x_train, t_train, inv_cov, cov_theta_deriv):
    term1 = (-1/2)*np.trace(np.dot(inv_cov,cov_theta_deriv))
    term2 = np.dot(t_train.transpose(),inv_cov)
    term2 = np.dot(term2,cov_theta_deriv)
    term2 = np.dot(term2,inv_cov)
    term2 = (-1/2)*np.dot(term2,t_train)
    result = term1 + term2
    return result

theta0 = 2
theta1 = 9
theta2 = 1
theta3 = 1
beta = 4

#prediction of the training points before optimization
train_pred_means = np.zeros(train_N)
train_pred_vars = np.zeros(train_N)
for a in range(0,train_N):
    C_N, k, k_T, c, C_N_full, inv_cov_N =
gauss_proc_kern_reg_cov(wine_train_x,wine_train_x[a,:],theta0,theta1,theta2,theta3,beta)
    train_pred_means[a] = gauss_proc_kern_reg_pred_mean(k_T,C_N,wine_train_y, inv_cov_N)
    train_pred_vars[a] = gauss_proc_kern_reg_pred_var(c,k_T,C_N,k, inv_cov_N)

#prediction of the test points before optimization
test_pred_means = np.zeros(test_N)
test_pred_vars = np.zeros(test_N)
for a in range(0,test_N):
    C_N, k, k_T, c, C_N_full, inv_cov_N =
gauss_proc_kern_reg_cov(wine_train_x,wine_test_x[a,:],theta0,theta1,theta2,theta3,beta)
    test_pred_means[a] = gauss_proc_kern_reg_pred_mean(k_T,C_N,wine_train_y, inv_cov_N)
    test_pred_vars[a] = gauss_proc_kern_reg_pred_var(c,k_T,C_N,k, inv_cov_N)

#The following pre-optimization variables has now been calculated:
#test_pred_means: predicted values of the test points
#train_pred_means: predicted values of the training points
#test_error_average: The average error of the test points

#Optimization
#arrays to hold the values of the parameters after each iteration
theta0_array = [theta0]
theta1_array = [theta1]
theta2_array = [theta2]
theta3_array = [theta3]
#variables to hold the value of the parameters after each iteration. They are initially given
the prior values.
```

```
theta0_new = theta0
theta1_new = theta1
theta2_new = theta2
theta3_new = theta3
N = 20 #Number of iterations of optimization
for a in range(0,N):
    q,w,e,r,t,y = gauss_proc_kern_reg_deriv_values(wine_train_x, theta0_new, theta1_new,
theta2_new, theta3_new, beta)
    cov = q
    inv_cov = w
    cov_theta0_deriv = e
    cov_theta1_deriv = r
    cov_theta2_deriv = t
    cov_theta3_deriv = y
    theta0_grad = gauss_proc_kern_reg_deriv(wine_train_x, wine_train_y, inv_cov,
cov_theta0_deriv)
    theta1_grad = gauss_proc_kern_reg_deriv(wine_train_x, wine_train_y, inv_cov,
cov_theta1_deriv)
    theta2_grad = gauss_proc_kern_reg_deriv(wine_train_x, wine_train_y, inv_cov,
cov_theta2_deriv)
    theta3_grad = gauss_proc_kern_reg_deriv(wine_train_x, wine_train_y, inv_cov,
cov_theta3_deriv)
    theta0_new -= theta0_grad
    theta1_new -= theta1_grad
    theta2_new -= theta2_grad
    theta3_new -= theta3_grad
    theta0_array = np.append(theta0_array,theta0_new)
    theta1_array = np.append(theta1_array,theta1_new)
    theta2_array = np.append(theta2_array,theta2_new)
    theta3_array = np.append(theta3_array,theta3_new)

#prediction of the training points after optimization
train_pred_means = np.zeros(train_N)
train_pred_vars = np.zeros(train_N)
for a in range(0,train_N):
    C_N, k, k_T, c, C_N_full, inv_cov_N =
gauss_proc_kern_reg_cov(wine_train_x,wine_train_x[a,:],theta0_new,theta1_new,theta2_new,theta3
_new,beta)
    train_pred_means[a] = gauss_proc_kern_reg_pred_mean(k_T,C_N,wine_train_y, inv_cov_N)
    train_pred_vars[a] = gauss_proc_kern_reg_pred_var(c,k_T,C_N,k, inv_cov_N)

#prediction of the test points after optimization
test_pred_means = np.zeros(test_N)
test_pred_vars = np.zeros(test_N)
for a in range(0,test_N):
    C_N, k, k_T, c, C_N_full, inv_cov_N =
gauss_proc_kern_reg_cov(wine_train_x,wine_test_x[a,:],theta0_new,theta1_new,theta2_new,theta3_
new,beta)
    test_pred_means[a] = gauss_proc_kern_reg_pred_mean(k_T,C_N,wine_train_y, inv_cov_N)
    test_pred_vars[a] = gauss_proc_kern_reg_pred_var(c,k_T,C_N,k, inv_cov_N)

#The following post-optimization variables has now been calculated:
#test pred means: predicted values of the test points
#train_pred_means: predicted values of the training points
#test_error_average: The average error of the test points
```

References

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. Print.

Cortez, P. UCI machine learning repository: Wine quality data set [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Wine+Quality [2016, October 9].

MacKay, David J. C. *Information Theory, Inference, and Learning Algorithms*. Cambridge, UK: Cambridge University Press, 2003. Print.

Sigillito, V. UCI machine learning repository: Pima Indians diabetes data set [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes [2016, October 9].

National climatic data center (NCDC) [Online]. Available: https://www.ncdc.noaa.gov/cdo-web/datasets. [2016, August 23].