UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

FACULTY OF ENGINEERING, BUILT ENVIRONMENT AND INFORMATION TECHNOLOGY

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING
http://www.up.ac.za/eece

EAI732: Intelligent Systems

Assignment 3: Gaussian Processes, Support Vector Machines and Graphical Models

Compiled By
Sholto Armstrong
14036071

16th May, 2018

DECLARATION OF ORIGINALITY

UNIVERSITY OF PRETORIA

The University of Pretoria places great emphasis upon integrity and ethical conduct in the preparation of all written work submitted for academic evaluation.

While academic staff teach you about referencing techniques and how to avoid plagiarism, you too have a responsibility in this regard. If you are at any stage uncertain as to what is required, you should speak to your lecturer before any written work is submitted.

You are guilty of plagiarism if you copy something from another author's work (e.g. a book, an article or a website) without acknowledging the source and pass it off as your own. In effect you are stealing something that belongs to someone else. This is not only the case when you copy work word-for-word (verbatim), but also when you submit someone else's work in a slightly altered form (paraphrase) or use a line of argument without acknowledging it. You are not allowed to use work previously produced by another student. You are also not allowed to let anybody copy your work with the intention of passing if off as his/her work.

Students who commit plagiarism will not be given any credit for plagiarised work. The matter may also be referred to the Disciplinary Committee (Students) for a ruling. Plagiarism is regarded as a serious contravention of the University's rules and can lead to expulsion from the University.

The declaration which follows must accompany all written work submitted while you are a student of the University of Pretoria. No written work will be accepted unless the declaration has been completed and attached.

Full names of student:	Sholto Armstrong
Student number:	14036071
Topic of work:	Assignment 3

Declaration

1. I understand what plagiarism is and am aware of the University's policy in this regard.
2. I declare that this assignment report is my own original work. Where other people's work has been used (either from a printed source, Internet or any other source), this has been properly acknowledged and referenced in accordance with departmental requirements.
3. I have not used work previously produced by another student or any other person to hand in as my own.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

SIGNATURE: _____ DATE: 15 May 2018

TABLE OF CONTENTS

1 Introduction

Machine learning can be split into two groups of algorithms, one being parametric algorithms and the other kernel based approaches. Parametric algorithms consider the training data and attempt to create a model by tweaking parameters. The training data can then be discarded and the model alone can be used to create predictions on new data. On the other hand, kernel based approaches require the training data in order to make prediction on new data. This can be advantageous as they often require less training data and need little to no time to train. In this assignment we begin by exploring the use of the Gaussian process for regression. Thereafter, a sparse kernel algorithm, known as the Support Vector Machine(SVM), is used for classification. Finally, graphical models are explored and used to create an image de-noising algorithm.

2 Abstract

In this assignment, we begin by exploring the effects of various parameters on a Gaussian process. Thereafter, the Gaussian process is applied to a non-linear mapping problem and the effects of ARD are discussed. SVM's are then explored using the Iris dataset [7]. Finally, graphical models are used to derive an algorithm which can be used to de-noise images.

3 Part A: Gaussian Process Regression

The Gaussian process is a stochastic process whereby a collection of random variables are normally distributed. It is possible to create models for this process which depend on input features. If the covariance matrix of this model correlates features which are similar, it is possible to use these models for regression. This can be done by conditioning the model on known features. In doing so, features which are similar to the training feature will be strongly correlated to it. To derive this, as shown in [1], we start with linear regression, whereby an output $\mathbf{y}$ is dependant on weight parameters $\mathbf{w}$ and the design matrix Φ, as shown in Eq. (3.1). Here the design matrix is used to map the input-space to feature-space.

$$\mathbf{y} = \Phi \cdot \mathbf{w} \tag{3.1}$$

It is possible to set a prior distribution over the weight parameters $\mathbf{w}$ as shown in Eq. (3.2).

$$p(\mathbf{w}) = \mathcal{N}\left(\mathbf{w}|0, \alpha^{-1}\mathbf{I}\right) \tag{3.2}$$

Since Φ only depends on input parameters and the mapping function $\phi(\mathbf{x})$, it is possible to describe the prior over $\mathbf{y}$ with a normal distribution derived from $p(\mathbf{w})$. This is given in Eq. (3.3).

$$p(\mathbf{y}) = \mathcal{N}\left(\mathbf{y}|0, \alpha^{-1}\Phi\Phi^T\right) \tag{3.3}$$

Using the kernel trick, it is possible to rewrite $\Phi\Phi^T$ as $\mathbf{K}$, where $K_{i,j} = k(x_i, x_j)$. Here the function $k(x_i, x_j)$ is known as the kernel function. This function allows one to evaluate $\Phi\Phi^T$, independent of the number of features in the feature-space $\phi(\mathbf{x})$. It is however not enough to define a prior for the underlying value y_n as there is often an element of noise added to y_n to produce t_n. This can again be modelled with the use of a normal distribution as shown in Eq. (3.4).

$$p(\mathbf{t}|\mathbf{y}) = \mathcal{N}\left(\mathbf{t}|\mathbf{y}, \beta^{-1}\mathbf{I}\right) \tag{3.4}$$

Once can now use Eq. (3.4) and Eq. (3.3) to determine $p(\mathbf{t})$ by marginalising over $\mathbf{y}$. This is done in Eq. (3.5). Here $\mathbf{C}$ is given as $\mathbf{K} + \beta^{-1}\mathbf{I}$.

$$p(\mathbf{t}) = \mathcal{N}(\mathbf{t}|0, \mathbf{C}) \tag{3.5}$$

It is now possible to perform regression by evaluating $p(t_{N+1}|\mathbf{t})$, assuming that one has chosen a relevant kernel function. Equation (3.6) gives an example of a commonly used kernel function for regression.

$$k(x_i, x_j) = \theta_0 \exp\left(-\frac{\theta_1}{2}||\mathbf{x}_i - \mathbf{x}_j||^2\right) + \theta_2 + \theta_3 x_i^T x_j \tag{3.6}$$

Automatic Relevance Determination(ARD) is a method used to weight the correlation of each input dimension on the output dimension. This will minimise the effect of parameters which have less influence on the overall system, which leads to a sparser representation of the data. The kernel function with ARD is shown in Eq. (3.7).

$$k(x, x') = \theta_0 \exp\left(-\frac{1}{2}\sum_{i=1}^{D}\eta_i(x_i - x_i)^2\right) + \theta_2 + \theta_3 x_i^T x_j \tag{3.7}$$

3.1 Sampling Functions From A Prior

Using Eq. (3.3) and Eq. (3.6), it is possible to draw random prior functions with different parameters for $(\theta_0, \theta_1, \theta_2, \theta_3)$. This helps develop an intuition on how these parameters affect the functions produced by the Gaussian process. This is done in Figs. 1 to 5. When examining Fig. 1, it is possible to see that θ_0 is responsible for setting the magnitude of the functions around the mean. This is intuitive as the variance of $p(y)$ increases as this parameter increases. One can also assume that the variance of y, when moving away from a known point, would increase more rapidly with a larger θ_0. Figure 2 varies θ_1 and keeps the remaining hyper-parameters constant. From this, one can see that θ_1 is responsible for setting the correlation between a point and its neighbouring points. When θ_1 is low, points which are closer will have a higher correlation, as the relative covariance is low. This leads to graphs which are smother. Therefore, it is possible to generate higher frequency graphs by increasing θ_1. Figure 3 explored the effect of θ_2, we can see from these graphs that increasing θ_2, increases the magnitude of the graphs produced. This id due to the fact that it increases the covariance. However, it is interesting to see that the graphs in Fig. 3 are all horizontal lines. This is due to the fact that θ_2 sets a bias to each element in the covariance matrix. Therefore, θ_2 controls how much the mean of the training samples should contribute to the output. Increasing this parameter also produces more smooth outputs as each point is more dependant on the mean of the outputs. This can be seen in Fig. 5. Finally, we can explore how the hyper-parameter θ_3 affects possible distributions, from the probability distribution p(y), by examining Fig. 4. These all consist of diagonal lines, which suggests that this term adjusts how much variations in x should affect the output. Increasing this parameter increases the likelihood of functions which increase or decrease linearly as x increases.

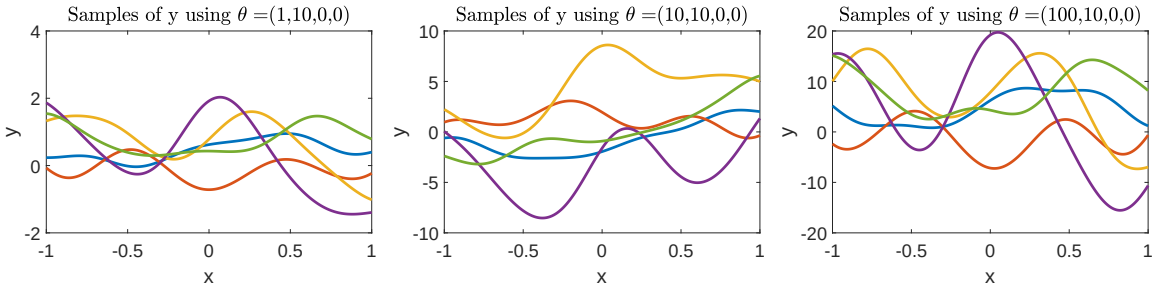


Figure 1. Random samples from the probability distribution $\mathcal{N}(\mathbf{y}|0,\mathbf{K})$, varying the hyper-parameter θ_0

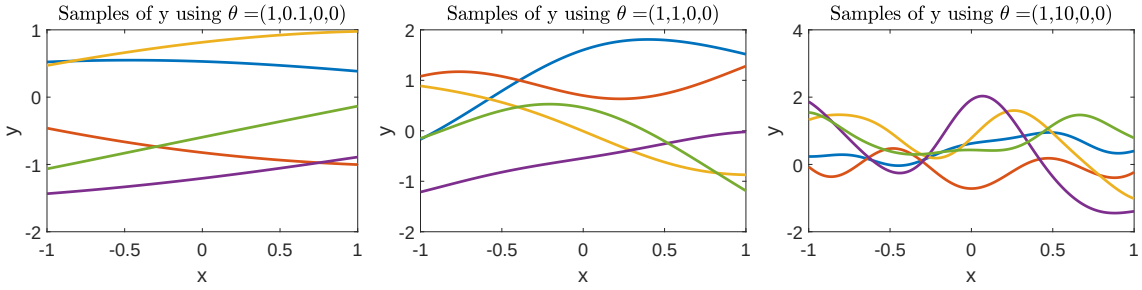


Figure 2. Random samples from the probability distribution $\mathcal{N}(\mathbf{y}|0,\mathbf{K})$, varying the hyper-parameter θ_1

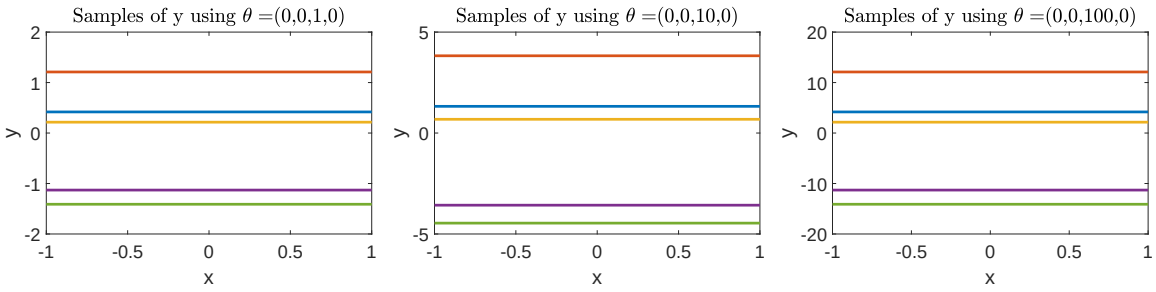


Figure 3. Random samples from the probability distribution $\mathcal{N}(\mathbf{y}|0,\mathbf{K})$, varying the hyper-parameter θ_2

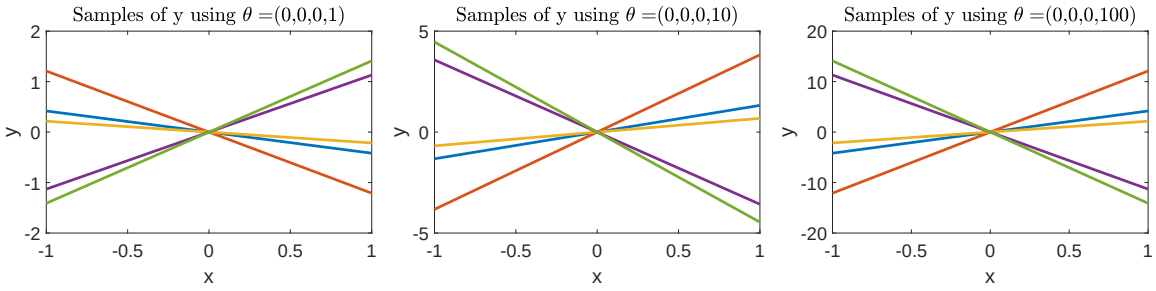


Figure 4. Random samples from the probability distribution $\mathcal{N}(\mathbf{y}|0,\mathbf{K})$, varying the hyper-parameter θ_3

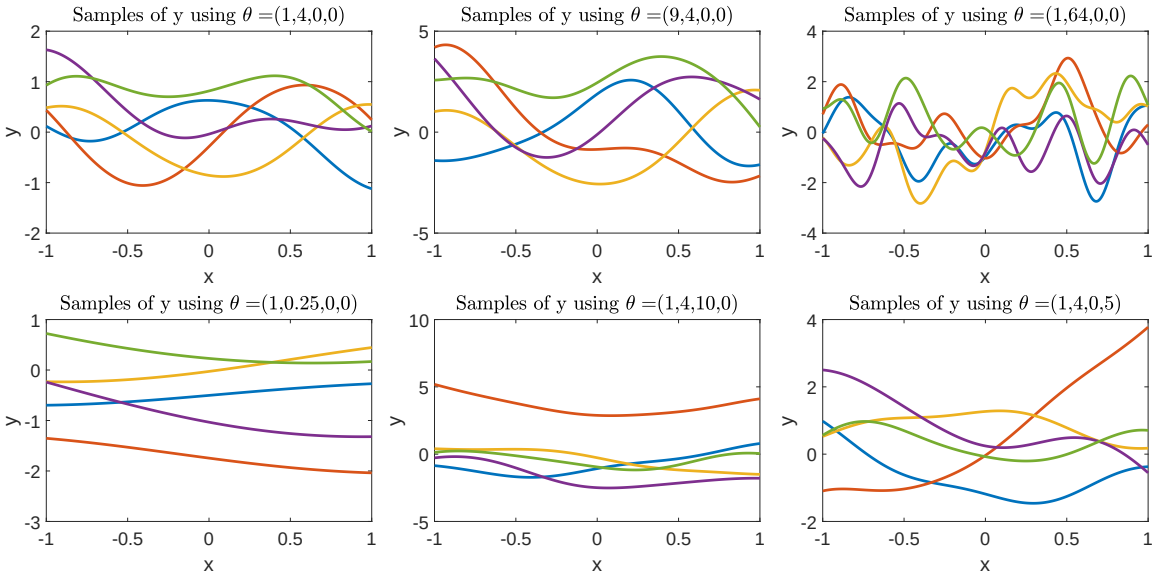


Figure 5. Random samples from the probability distribution $\mathcal{N}(\mathbf{y}|0,\mathbf{K})$, using various hyper-parameters

3.2 Regression of Robotic Arm Data: Part 1

We will now apply the Gaussian Process to a regression problem introduced by MacKay[2]. This involves the mapping of a 2D non-linear robotic arm. The function used to generate this mapping is given by Eq. (3.8), where $\mathbf{x} = (x_1, x_2, x_3, x_4)^T$.

$$f(\mathbf{x}) = 2\cos(x_1) + 1.3\cos(x_1 + x_2) \tag{3.8}$$

A dataset containing 200 training points and 200 test points was used. These points where generated by randomly choosing x_1 from the intervals $(-1.932, -0.45)$ and $(0.45, 1.932)$ and x_2 from the range $(0.534, 3.142)$. The variables x_3 and x_4, were chosen from a Gaussian distribution with a zero mean and a unit variance. The outputs were obtained by adding Gaussian noise to $f(\mathbf{x})$ with a variance of 0.0025. The 3D plots of $f(\mathbf{x})$ given with the training and test data is shown in Fig. 6.

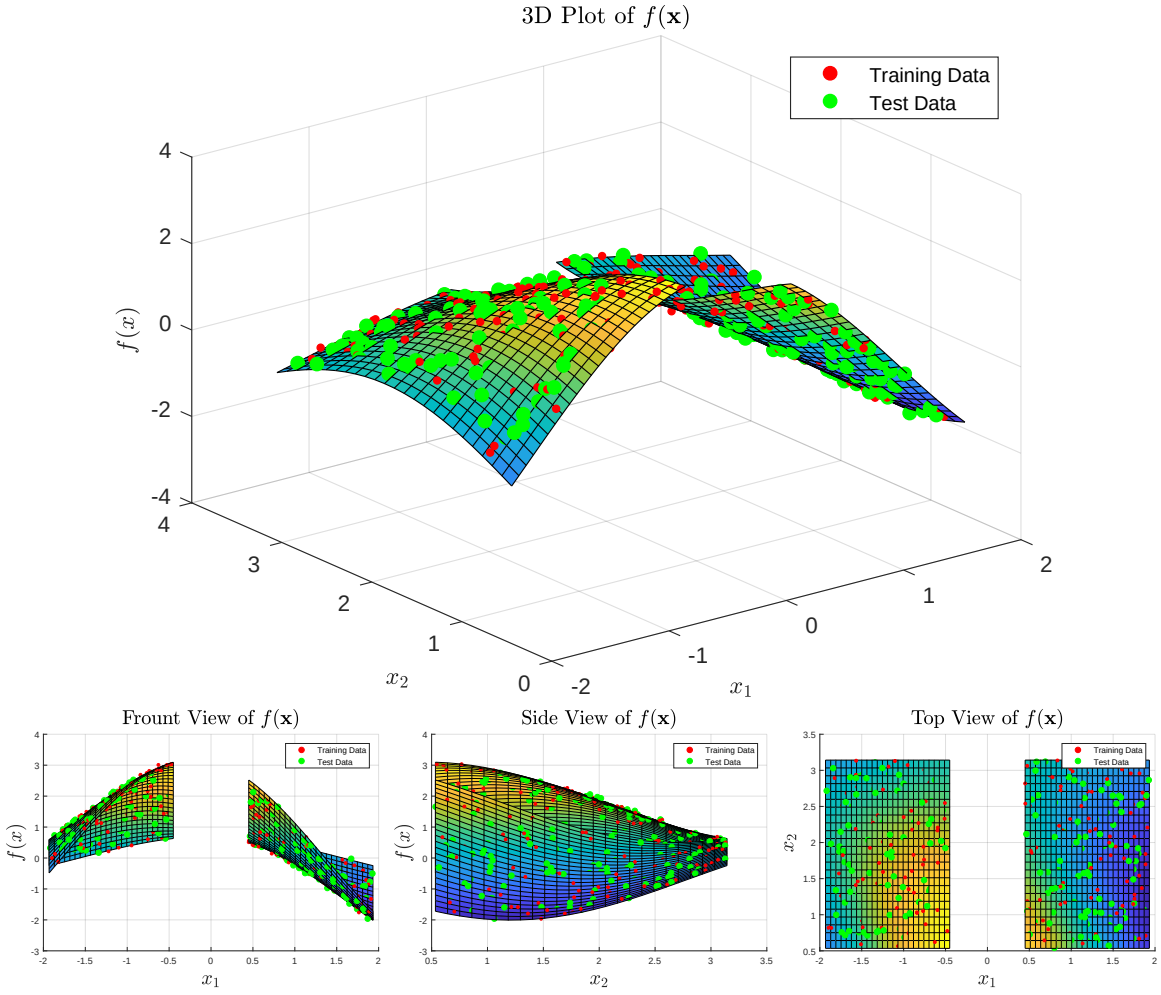


Figure 6. Plots of $f(\mathbf{x})$ from various angles.

3.2.1 Design

It is assumed that the full range of the input variables and output variables is not known. Therefore, The input variables are normalised from the training data using Eq. (3.9), where $\max(\mathbf{X})$ and $\min(\mathbf{X})$ returns a vector containing the minimum and maximum element for each dimension of X. This is done in order to ensure that each variable has an equal prior weighting.

$$\mathbf{X}_{normalised} = (\mathbf{X} - \min(\mathbf{X}))/(\max(\mathbf{X}) - \min(\mathbf{X})) \tag{3.9}$$

The training output variables are also shifted to have a zero mean. This is done using Eq. (3.10).

$$\mathbf{Y}_{shifted} = \mathbf{Y} - \text{mean}(\mathbf{Y}) \tag{3.10}$$

The GPML Toolbox[3] was used to implement the Gaussian Process, using a composite covariance function comprising of a squared-exponential with ARD plus noise. ARD should eliminate x_3 and x_4 as they have no correlation with the output data. The parameters of this kernel function are then optimised multiple times using random seeds between 0 and 1. This helps prevent finding a local minimum, due to the gradient decent optimisation used. The source code for this is given in Fig. 7.

```
meanfunc = [];
covfunc = {'covSum', {'covSEard','covNoise'}};
likfunc = @likGauss;

besthyp = 0;
besthypScore = inf;
for c = 1:5
  hyp = struct('mean', [], 'cov', log(rand(1,6)), 'lik', -1);
  [hyp2, fX, iterations] = minimize(hyp, @gp, -2000, @infGaussLik,
     meanfunc, covfunc, likfunc, normX, yShifted);
  if fX(end) < besthypScore
    besthyp = hyp2;
    besthypScore = fX(end);
  end
end
```

Figure 7. Optimisation of the hyper-parameters

The test data can then be normalised using the same normalisation constants found for the training dataset. This is then fed into the Gaussian Process in order to make predictions of this data. The mean squared error(MSE), root mean squared error(RMSE), and normalised root mean squared error(NRMSE) can be calculated using Eqs. (3.11) to (3.13).

$$\text{MSE}(y,\hat{y}) = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{3.11}$$

$$\text{RMSE}(y,\hat{y}) = \sqrt{\text{MSE}(y,\hat{y})} \tag{3.12}$$

$$\text{NRMSE}(y,\hat{y}) = \frac{\text{RMSE}(y,\hat{y})}{y_{max} - y_{min}} \tag{3.13}$$

This was then implemented in Matlab as shown in Fig. 8.

```
[m, s2] = gp(besthyp, @infGaussLik, meanfunc, covfunc, likfunc,
   normX, yShifted, normXstar);

MSETest = sum(power(m+meanY-ystar,2))/length(ystar);
RMSETest = sqrt(MSETest);
NRMSETest = RMSETest/range(ystar);

[mTr, sTr2] = gp(besthyp, @infGaussLik, meanfunc, covfunc, likfunc,
    normX, yShifted, normX);

MSETraining = sum(power(mTr-yShifted,2))/length(ystar);
RMSETraining = sqrt(MSETraining);
NRMSETraining = RMSETraining/range(yShifted);
```

Figure 8. Code used to make predictions and calculate the errors

3.2.2 Results

The results for the optimisation are shown in Table 1, where the parameter sigma is the likelihood noise hyper-parameter, and the remaining hyper-parameters are from Eq. (3.7).

Initial Parameters $\eta_1, \eta_2, \eta_3, \eta_4, \theta_0, \beta^{-0.5}, \sigma$	**Final Parameters** $\eta_1, \eta_2, \eta_3, \eta_4, \theta_0, \beta^{-0.5}, \sigma$	**Cost**
0.28, 0.55, 0.96, 0.96, 0.16, 0.97, 0.37	4.71e-01, 7.68e-01, 1.52e+04, 75.58, 2.50, 6.92e-03, 5.08e-02	-2.513e+02
0.96, 0.49, 0.80, 0.14, 0.42, 0.92, 0.37	4.71e-01, 7.68e-01, 1.79e+04, 75.59, 2.50, 9.91e-03, 5.03e-02	-2.513e+02
0.79, 0.96, 0.66, 0.04, 0.85, 0.93, 0.37	4.71e-01, 7.68e-01, 1.15e+04, 75.59, 2.50, 5.47e-04, 5.12e-02	-2.513e+02
0.68, 0.76, 0.74, 0.39, 0.66, 0.17, 0.37	4.71e-01, 7.68e-01, 9.09e+03, 75.58, 2.50, 5.06e-02, 8.31e-03	-2.513e+02
0.71, 0.03, 0.28, 0.05, 0.10, 0.82, 0.37	4.71e-01, 7.68e-01, 3.78e+04, 75.59, 2.50, 3.25e-05, 5.12e-02	-2.513e+02
0.69, 0.32, 0.95, 0.03, 0.44, 0.38, 0.37	4.71e-01, 7.68e-01, 1.18e+04, 75.59, 2.50, 3.64e-02, 3.61e-02	-2.513e+02
0.77, 0.80, 0.19, 0.49, 0.45, 0.65, 0.37	4.71e-01, 7.68e-01, 1.50e+04, 75.58, 2.50, 2.43e-02, 4.51e-02	-2.513e+02
0.71, 0.75, 0.28, 0.68, 0.66, 0.16, 0.37	4.71e-01, 7.68e-01, 1.01e+04, 75.58, 2.50, 5.09e-02, 5.50e-03	-2.513e+02
0.12, 0.50, 0.96, 0.34, 0.59, 0.22, 0.37	4.71e-01, 7.68e-01, 1.38e+04, 75.59, 2.50, 5.12e-02, 2.92e-03	-2.513e+02
0.75, 0.26, 0.51, 0.70, 0.89, 0.96, 0.37	4.71e-01, 7.68e-01, 1.63e+04, 75.58, 2.50, 5.49e-03, 5.09e-02	-2.513e+02

Table 1. Results of parameter optimisation

The errors produced by the Gaussian Process are shown in Table 2.

	Training Data	**Testing Data**
MSE(%)	0.23	0.25
RMSE(%)	4.82	4.98
NRMSE(%)	0.97	1.06

Table 2. Results of the Gaussian Process

Figure 9, shows the predictive function over x_1 and x_2, where $x_3 = x_4 = 0$.

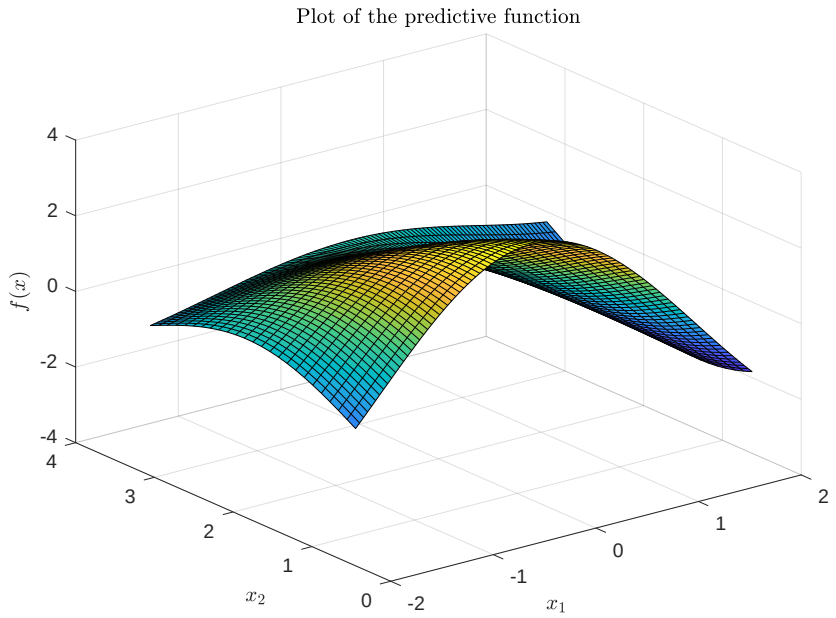


Figure 9. Absolute error of the predicted function

Figure 10, shows the absolute error of the predicted function over x_1 and x_2, where $x_3 = x_4 = 0$.

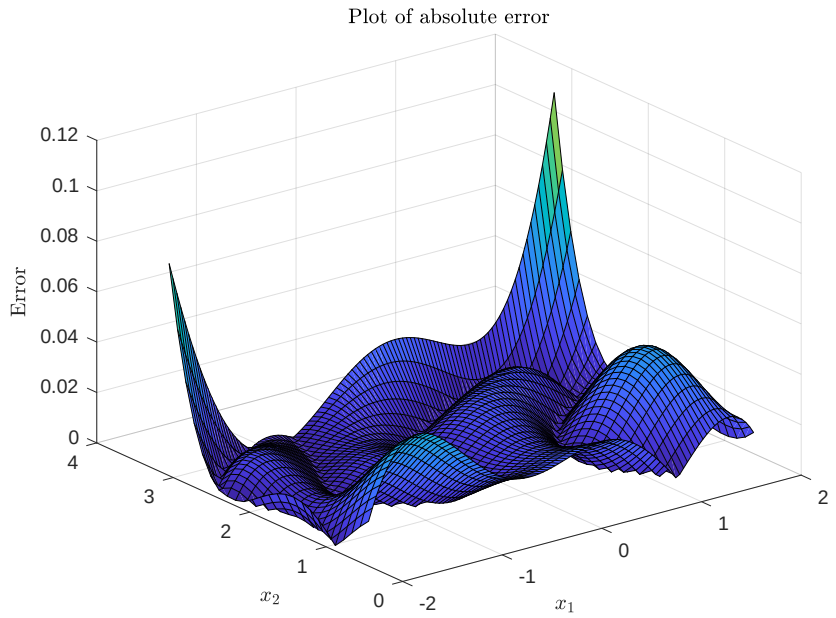


Figure 10. Plot of predictive distribution

The confidence interval of the predictive function is plotted in Figure 11, where $x_3 = x_4 = 0$.

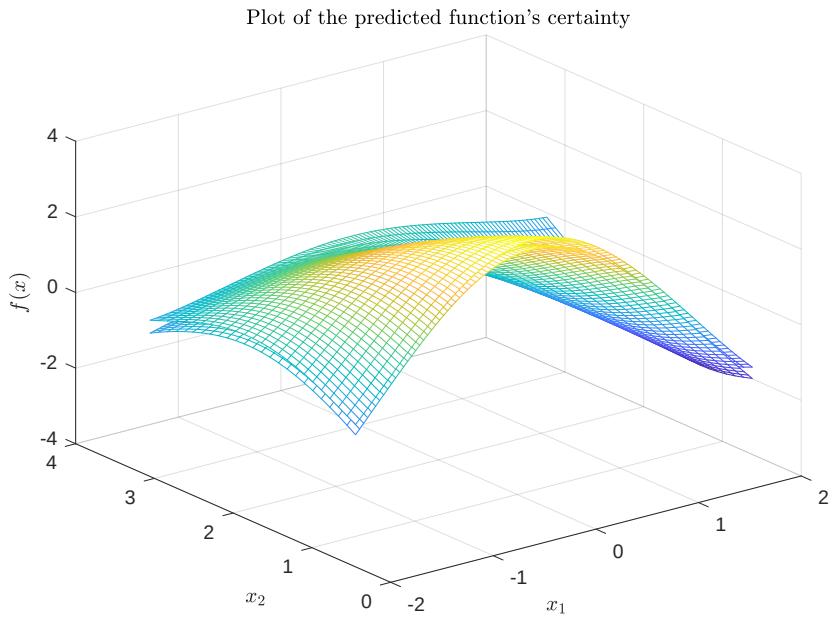


Figure 11. Plot of the 2 sigma confidence interval

3.2.3 Discussion

From Table 1, we can see that η_3 approximately converges to $1e4$ and η_4 approximately converges to 75.5. Both of these values are large when compared to η_1 and η_2. This causes the correlation to decrease rapidly for small changes in $(x_i - x_i')^2$, which means that x_3 and x_4 will have a negligible affect on y. One can see that out of the 10 runs performed, most of the hyper-parameters converged to the same value. This means that the optimum found is most likely a global solution, and that the search space is not a complex function with many local minima. Another interesting property is that σ usually converges to values close to $5e-2$, therefore $\sigma^2 \approx 0.0025$. These are good estimates to the actual noise in the training data. When looking at Table 2, we can see that the MSE of the training data is 0.23% and the MSE of the test data is 0.25%. The MSE of the training data is as large as the initial error of the training inputs. This means that the Gaussian Process was able to predict the model with an extremely high accuracy, given the amount of input-noise. This is reflected in Fig. 10, where there is a low error for most of the points near the centre. As the points reach the end of the input-space, we can see that the error quickly increases as there are less training points near the edge. However, all of the individual errors are below 0.12, which is an extremely good approximation.

3.3 Regression of Robotic Arm Data: Part 2

We now consider a different mapping function for the robotic arm. This is given in Eq. (3.14).

$$f(\mathbf{x}) = 2\cos(10x_1) + 1.3\cos(x_1 + x_2) \tag{3.14}$$

In order to get an appropriate value for the amount of training samples required to approximate this function, we can turn to the Nyquist Sampling Theorem[4], which states that the minimum frequency required to properly sample a function must be at least double the highest frequency component of the function. The highest frequency component of x_1 is $\frac{5}{\pi}\text{rad}^-1$ and x_2 is $\frac{1}{2\pi}\text{rad}^-1$. The minimum number

of samples required can then be calculated as shown in Eq. (3.15).

$$\begin{aligned} N &= 2 \times f_{\max x_1} \text{Range}(x_1) \times 2 \times f_{\max x_2} \text{Range}(x_2) \\ &= 4 \times \left(2 \times (1.932 - 0.45) \times \frac{5}{\pi} \right) \times \left((3.142 - 0.534) \times \frac{1}{2\pi} \right) \\ &= 7.83 \end{aligned} \tag{3.15}$$

In order to ensure that enough points are generated for regression, N is multiplied by a factor of 8 leading to a total of approximately 65 samples. The 65 training points and a further 400 test points are randomly selected from the ranges provided in Section 3.2. When calculating the outputs, no noise is added to the target variables, furthermore x_3 and x_4 are ignored for this part. The new mapping function with the training and test points are shown in Fig. 12.

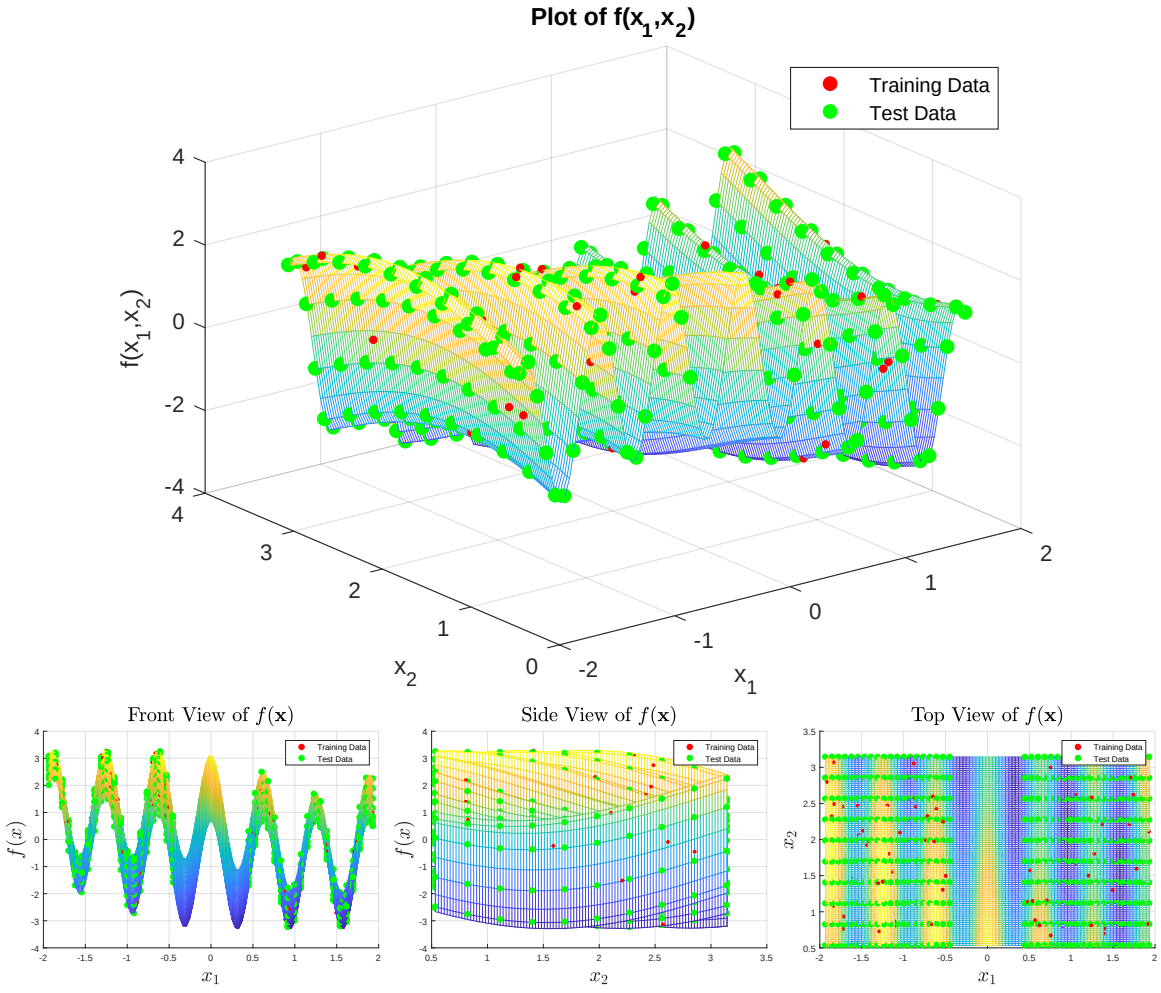


Figure 12. Plots of $f(\mathbf{x})$ from various angles.

The optimisation and regression is then preformed in a similar fashion to Section 3.2.1, apart from the hyper-parameter σ which is initialised to $1e-8$.

3.3.1 Results

The results for the optimisation are shown in Table 3, where the parameter sigma is the likelihood noise hyper-parameter, and the remaining hyper-parameters are from Eq. (3.7).

Initial Parameters $\eta_1, \eta_2, \eta_3, \eta_4, \theta_0, \beta^{-0.5}, \sigma$	**Final Parameters** $\eta_1, \eta_2, \eta_3, \eta_4, \theta_0, \beta^{-0.5}, \sigma$	**Cost**
0.69, 0.76, 0.43, 0.66, 1.00e-09	1.61e-01, 5.11e-01, 1.04, 1.35e+00, 1.00e-09	1.188e+02
0.11, 0.93, 0.19, 0.27, 1.00e-09	1.61e-01, 5.11e-01, 1.04, 1.35e+00, 1.00e-09	1.188e+02
0.80, 0.49, 0.77, 0.40, 1.00e-09	7.36e-02, 1.33e+00, 4.73, 6.69e-04, 1.00e-09	-1.210e+01
0.27, 0.04, 0.67, 0.43, 1.00e-09	2.95e-01, 1.13e-02, 1.09, 1.25e+00, 1.00e-09	1.233e+02
0.45, 0.61, 0.06, 0.32, 1.00e-09	7.36e-02, 1.33e+00, 4.72, 6.68e-04, 1.00e-09	-1.210e+01
0.77, 0.70, 0.13, 0.13, 1.00e-09	7.36e-02, 1.33e+00, 4.72, 6.68e-04, 1.00e-09	-1.210e+01
0.09, 0.01, 0.42, 0.66, 1.00e-09	1.61e-01, 5.11e-01, 1.04, 1.35e+00, 1.00e-09	1.188e+02
0.72, 0.53, 0.11, 0.63, 1.00e-09	7.36e-02, 1.33e+00, 4.73, 6.69e-04, 1.00e-09	-1.210e+01
0.13, 0.13, 0.10, 0.14, 1.00e-09	7.36e-02, 1.33e+00, 4.72, 6.67e-04, 1.00e-09	-1.210e+01
0.17, 0.20, 0.32, 0.32, 1.00e-09	1.61e-01, 5.11e-01, 1.04, 1.35e+00, 1.00e-09	1.188e+02

Table 3. Results of parameter optimisation

The errors produced by the Gaussian Process are shown in Table 4.

	Training Data	**Testing Data**
MSE(%)	4.5354e-09	0.4205
RMSE(%)	6.7345e-04	6.4844
NRMSE(%)	1.0489e-04	0.9998

Table 4. Results of the Gaussian Process

Figure 13, shows the predictive function over x_1 and x_2, where $x_3 = x_4 = 0$.

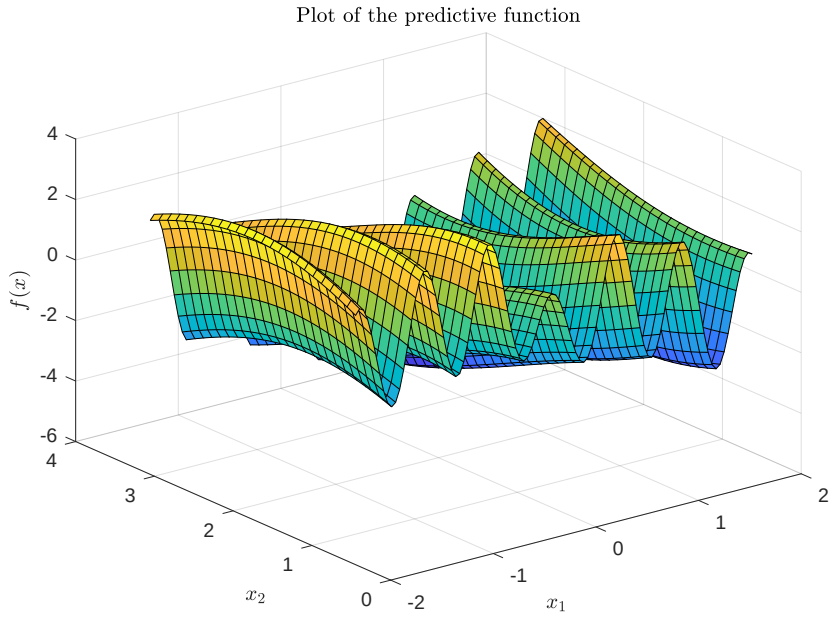


Figure 13. Plot of predictive distribution

Figure 14, shows the absolute error of the predicted function over x_1 and x_2.

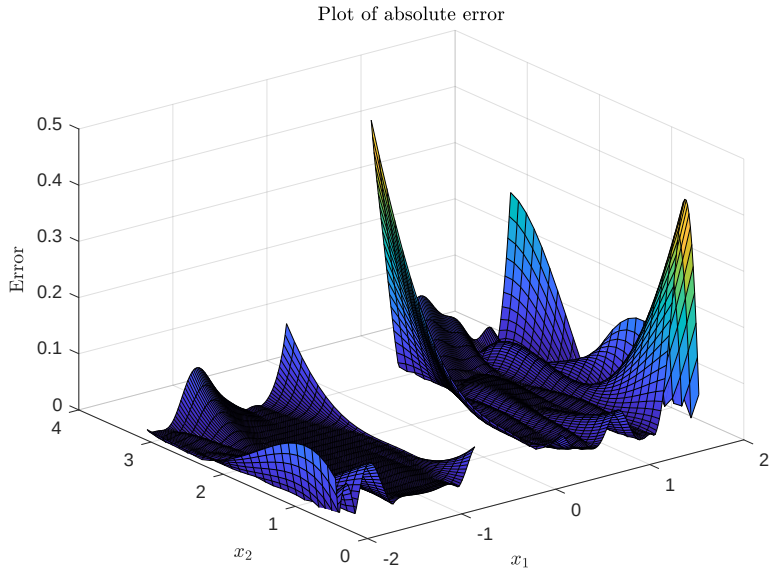


Figure 14. Absolute error of the predicted function

The confidence interval of the predictive function is plotted in Figure 15.

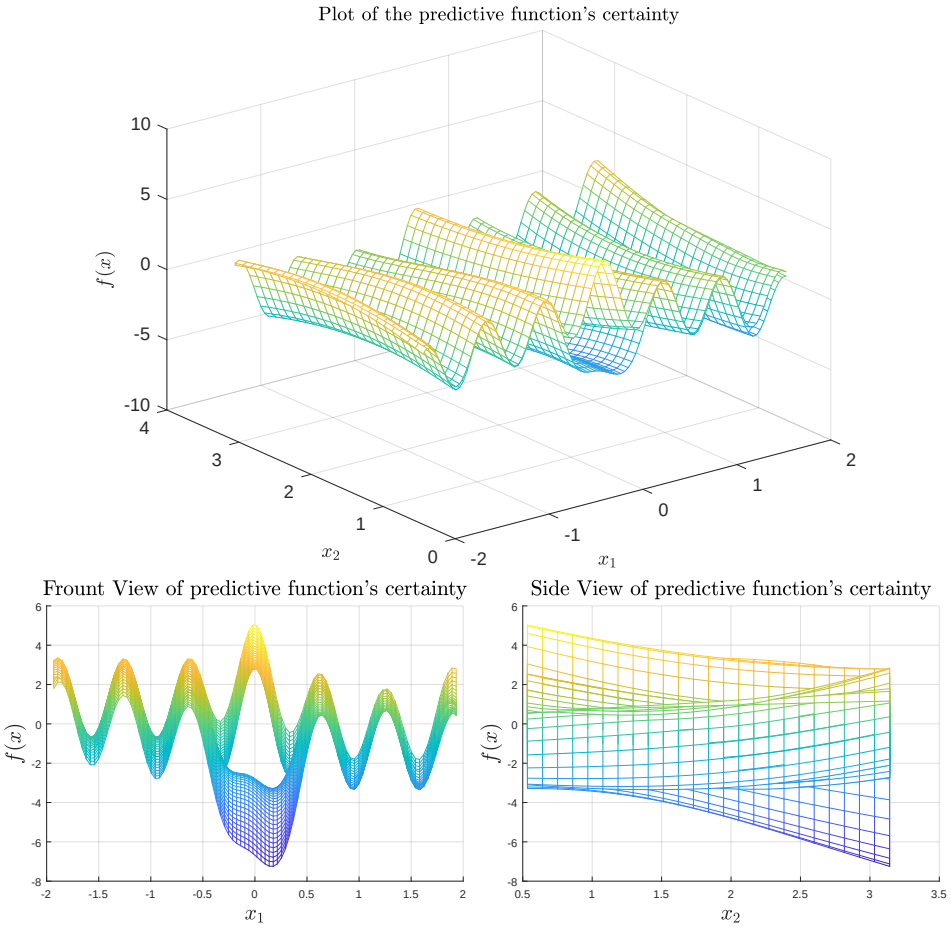


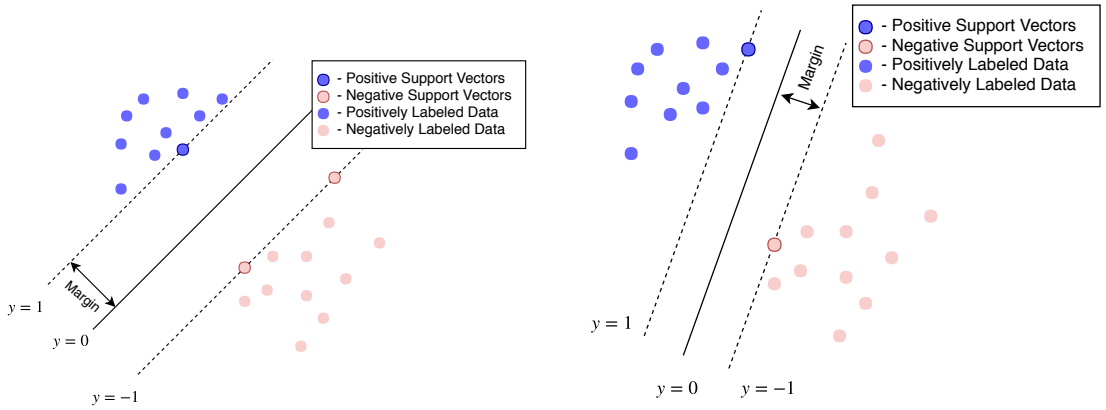
Figure 15. Plot of the 2 sigma confidence interval

3.3.2 Discussion

From Table 3, we can see that the search space of this function is more complex as it takes more iterations to yield low cost values. This is partly due to the reduced number of training points, but also because of the more complex function. The more complex the function is, the harder it is to distinguish between the noise and actual values. When looking at the hyper-parameters with the smallest cost values, we can see that η_1 is far smaller that η_2. This is due to the fact that the outputs are far more correlated with x_1 than they are with η_2. It is also interesting to see that $B^{-0.5}$ converges to a really small value, which corresponds to the fact that no noise was added to the outputs. This is also reflected in the training NRMSE, shown in Table 4, which is negligibly small. The testing error is also quite small, having an NRMSE smaller than the error achieved in Section 3.2. This is can also be attributed to the noiselessness of the training data. When examining Fig. 15, unlike in Fig. 11, we can see that the confidence interval of the predictive function rapidly increases toward the centre of the graph. This is due to the increased frequency of y in relation to x_1. The Gaussian Process has to "react" faster to changes in x_1 than before, leading to a graph which is more uncertain the further it is from known points. This is reflected in Figs. 13 and 14, where errors caused by a change in x_1 vary dramatically but not so much by a change in x_2.

4 Part B: Support Vector Machine

Gaussian processes can also be used to classify data, however it it highly inefficient in doing so. Therefore, we introduce a new sparse kernel method, namely the Support Vector Machine(SVM), which is an effective classifier. The premise behind SVM's is to find the hyperplane which maximises the perpendicular distance between the nearest opposing classes. This is termed the margin. Figure 16a provides an example of an optimal boundary line, where the margin between the classes in a maximum. An alternative boundary line is given in Fig. 16b. This boundary line has a smaller margin, therefore it is more likely to incorrectly label data. The equation of the decision boundary is given in Eq. (4.1).



(a) A optimal boundary line

(b) A suboptimal boundary line

Figure 16. Examples of possible boundary lines between classes

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{4.1}$$

As shown in Bishop[1], the perpendicular distance from a point to the decision boundary is given in Eq. (4.2).

$$\frac{t\left(\mathbf{w}^T \phi(\mathbf{x}) + b\right)}{||\mathbf{w}||} \tag{4.2}$$

In order to maximise the size of the margin, it is convenient to define $y(\mathbf{x})$ at the closest correctly classified $\mathbf{x}$ as 1. This is possible as scaling the values of $\mathbf{w}$ and b leaves the perpendicular distance to $\mathbf{x}$ unchanged. Doing so creates the constraint given in Eq. (4.3).

$$t\left(\mathbf{w}^T \phi(\mathbf{x}) + b\right) \geq 1 \tag{4.3}$$

We now merely seek to maximise $||\mathbf{w}||^{-1}$. This can be done by minimise the value

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

. Minimising this with the constraints given in Eq. (4.3), can be achieved with Lagrange multipliers. This is equivalent to maximising the value given in Eq. (4.4) , subject to Eqs. (4.5) and (4.6).

$$\sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \tag{4.4}$$

$$a_n \geq 0 \tag{4.5}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{4.6}$$

Predictions can now be made with Eq. (4.7).

$$y(x) = \sum_{n=1}^{M} a_n t_n k(x, x_n) + b \tag{4.7}$$

It can be shown with the Karush-Kuhn-Tucker conditions that each point will either satisfy $t_n y(x_n) = 1$ or $a_n = 0$. If $t_n y(x_n) = 1$, then the point is a support vector, else the point has no effect on the decision boundary, and can therefore be discarded. This creates the sparse representation, whereby only a subset of the points are needed to make predictions. Furthermore, due to the representation shown in Eq. (4.4), it is possible to model complex decision boundaries.
The optimisation problem given above forces each data-point to be correctly classified. However, it is sometimes preferable to allow a portion of the training data to fall into the margin or even incorrectly classify data, in favour of a larger margin. This is due to the fact that relaxing these constraints helps to avoid over-fitting, where data isn't easily separable. In order to achieve this, slip parameters ζ_n are introduced to the optimisation problem. This leads to a Lagrangian with an identical form to the one given in Eq. (4.4). However, this is now minimised subject to the constraints given in Eqs. (4.8) and (4.9).

$$0 \leq a_n \leq C \tag{4.8}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{4.9}$$

The parameter C in Eq. (4.8) controls the cost of allowing variables to fall into the margin. Therefore, increasing this parameter creates stricter constraints at the cost of smaller margins.

4.1 Design

In the creation of the SVM algorithm, we will be following closely to the guidelines set out in Hsu el al.[5]. Furthermore, the LIBSVM library[6] will be used to perform the optimisation shown in Eqs. (4.4) to (4.6) and the prediction step given by Eq. (4.7). This module gives us a choice between 4 different kernel functions given as follows:

- Linear: $x' \cdot x$,
- Polynomial: $(\gamma x' \cdot x + C_0)^d$,
- Radial basis: $\exp(-\gamma ||x' - x||^2)$,
- Sigmoid: $\tanh(\gamma x' \cdot x + C_0)$.

All four of these are tested separately in order to determine their effectiveness.
In order to properly test the algorithm, we begin by randomly splitting the data into a training set and a test set. This is done in a 60/40 ratio. We then need a method to turn the various parameters needed for the kernel functions. This is done with the use of a coarse to fine grid-search as proposed in Hsu et al. Each parameter is iterated in powers of two in a nested loop. Once the entire search-space is iterated, a new search space is created in the grid which achieved the best results and the step size is reduced. This is done multiple times until a suitable accuracy is achieved.
In order to perform this grid search, a method is required to test the accuracy of the algorithm. Since the test set should not be used when training, the training data set needs to be split to create a second training dataset and a validation dataset. One possible method for doing so is named v fold cross-validation. The dataset is split into v subsets. From this, v-1 subsets are used to train the algorithm and the remaining subset is used to calculate the validation error. This can be done v times, each time leaving out a different subset, and the validation error can be accumulated. This provides insight as to

how the algorithm would work on the testing dataset. The issue with this approach is when there is few items to train with. In this case, many of the cross-validation errors will be at a minimum, making it impossible to choose between various parameters. Due to this, it was decided to randomly split the training dataset into the second training set and the validation set. This is done in a similar fashion to how the data is split into the training and testing dataset. A ratio of 70/30 is used for this process. This is then done 40 times to minimise the possibility of parameters scoring the same cross-validation error. Due to the large number of cross-validation steps, it is important to stop testing a parameter if it is yielding poor results. This is done to speed up the search. Another motivation for this speed-up, is that parameters which are most incorrect take the longest to converge and often don't converge in the given number of iterations. This can be accomplished by allowing 3 runs of the cross-validation step. Thereafter, if the average of the cross-validation error drops below a threshold, the average is returned as is. It was decided to set this threshold as 85%. The code shown in Fig. 17 is used to calculate the cross-validation error.

```
def getVfoldCrossValidation(param, XTrain, yTrain, n=40,
 splitRatio=0.7, prewnPercentage = 85):
errorAccumulator = 0
"""used for shuffeling the data"""
ri=np.arange(len(yTrain))

for i in range(1,n+1):
  trainx, trainy, testx, testy =
        IrusDataReader.ShuffelAndSplit(XTrain,yTrain,ri,splitRatio)
  prob = svmutil.svm_problem(trainy,trainx.tolist())
  m = svmutil.svm_train(prob, param, '-q')
  p_label, p_acc, p_val =
        svmutil.svm_predict(testy,testx.tolist(), m, '-q')
  errorAccumulator +=p_acc[0]
  if i > 3 and errorAccumulator < prewnPercentage*i:
        return n*errorAccumulator/i
return errorAccumulator
```

Figure 17. Code used to calculate the cross validation error

Due to the vast number or parameters which need to be optimised, it was decided to implement the grid search in a multi-threaded approach. This is done by letting each thread calculate the cross-validation error of a particular area in the search grid. A list of all the cross-validation errors and the parameters used to achieve them is then stored in a list. This is then used to determine the centre point for the next grid search. The upper and lower bounds for the next grid search is set to the centre plus and minus the step size. Figure 18 shows the code used to implement this. Note that the parameter d must be an integer. Due to this, it is optimised in the initial parse through the grid-search by setting the initial range of search parameters to integers between 2 and 5 and then leaving it constant for the remainder of the search. The parameter C, from Eq. (4.8), is tested for each value in: $2^{-6}, 2^{-5}, 2^{-4}, ..., 2^{7}$. This allows us to explore the effects of C separately to the rest of the parameters. Once the optimal parameters are found, the algorithm is trained on the entire training dataset. This model is then used to classify the testing dataset. The error is then calculated with Eq. (4.10). This entire process is repeated 50 times with different training and testing datasets, in order to determine how consistent this algorithm is. The code used to determine the testing sets accuracy for a given trial is given in Fig. 19

$$error = \frac{1}{N}\sum_{i=1}^{N}\{t_i - \text{sign}(y(\mathbf{x}_i)) = 0\} \tag{4.10}$$

```
prevstep1=1
prevstep2=2
for stepPow in range(12):
"""
Perform the mapping from the parameters to the cross validation errors
Note that this function also takes 2^param for gamma and C0
"""
  scores,values = zip(*_pool.map(getscorepartial,product(*params)))
"""
Get the parameters with the best score
"""
  bvi=np.argmax(scores)
  besterrparams=values[bvi]
"""
Reduce the stepsize and create a new grid around the new parameter
"""
  prevstep1 = nextstep1
  nextstep1 = prevstep1/2
  prevstep2 = nextstep2
  nextstep2 = prevstep2/2
  params[1]=[besterrparams[1]]"""d"""
  params[2]=np.arange(besterrparams[2]-prevstep1
                      ,besterrparams[2]+prevstep1+nextstep1
                      ,nextstep1)"""Gamma"""
  params[3]=np.arange(besterrparams[3]-prevstep2
                      ,besterrparams[3]+prevstep2+nextstep2
                      ,nextstep2)"""C0"""
```

Figure 18. Code used for the coarse to fine grid search

```
templateParameter = "-t {0} -d {1} -g {2} -r {3} -c {4} -q"
prob = svmutil.svm_problem(yTrain,XTrain.tolist())
parameterString = templateParameter.format(*[besterrparams[0],
  besterrparams[1],2**besterrparams[2],2**besterrparams[3],
  besterrparams[4]])
parameterStructure = svmutil.svm_parameter(pstring)

m = svmutil.svm_train(prob, parameterStructure)
p_label, p_acc, p_val = svmutil.svm_predict(yTest,XTest.tolist(), m, '-q')
```

Figure 19. Code used to calculate the cross validation error

4.2 Iris Dataset

In order to test this algorithm, we will use the Iris dataset [7]. This consists of determining the class of an iris plant given various measurements. This dataset contains 150 samples, consisting of 3 different classes (50 of samples of each). The 3 classes are: Versicolor, Virginica, Setosa. There are 4 attributes for each of these samples. This dataset will be split into combinations of 2 labels in order keep the classification problem simple. Projections of the data on to various planes is shown in Figures 20 to 22. From these figures, it is possible to see that Setosa is linearly separable from both Versicolor and Virginica. However, Versicolor and Virginica are not easily separable.

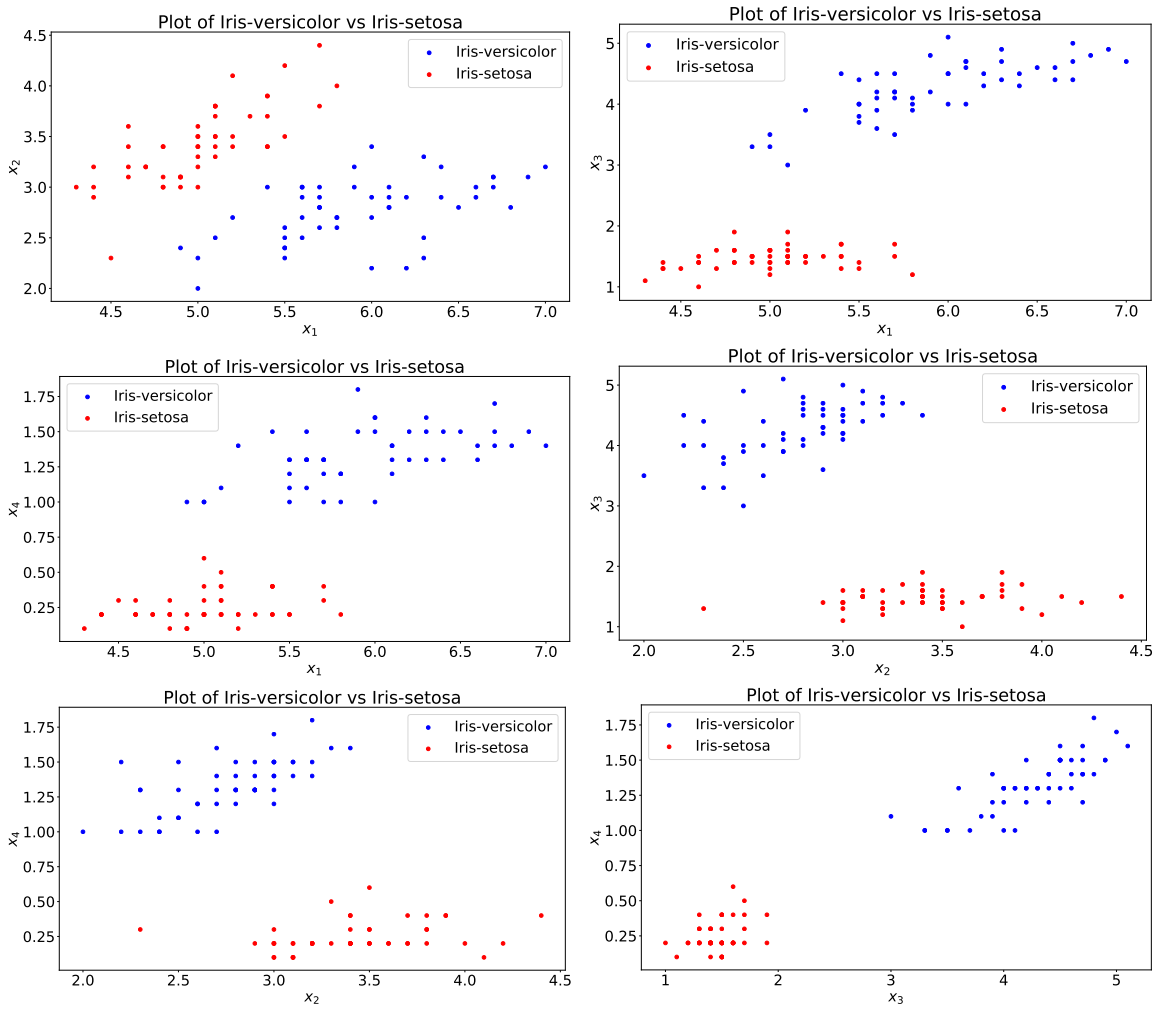


Figure 20. Cross sections of the Versicolor vs Setosa

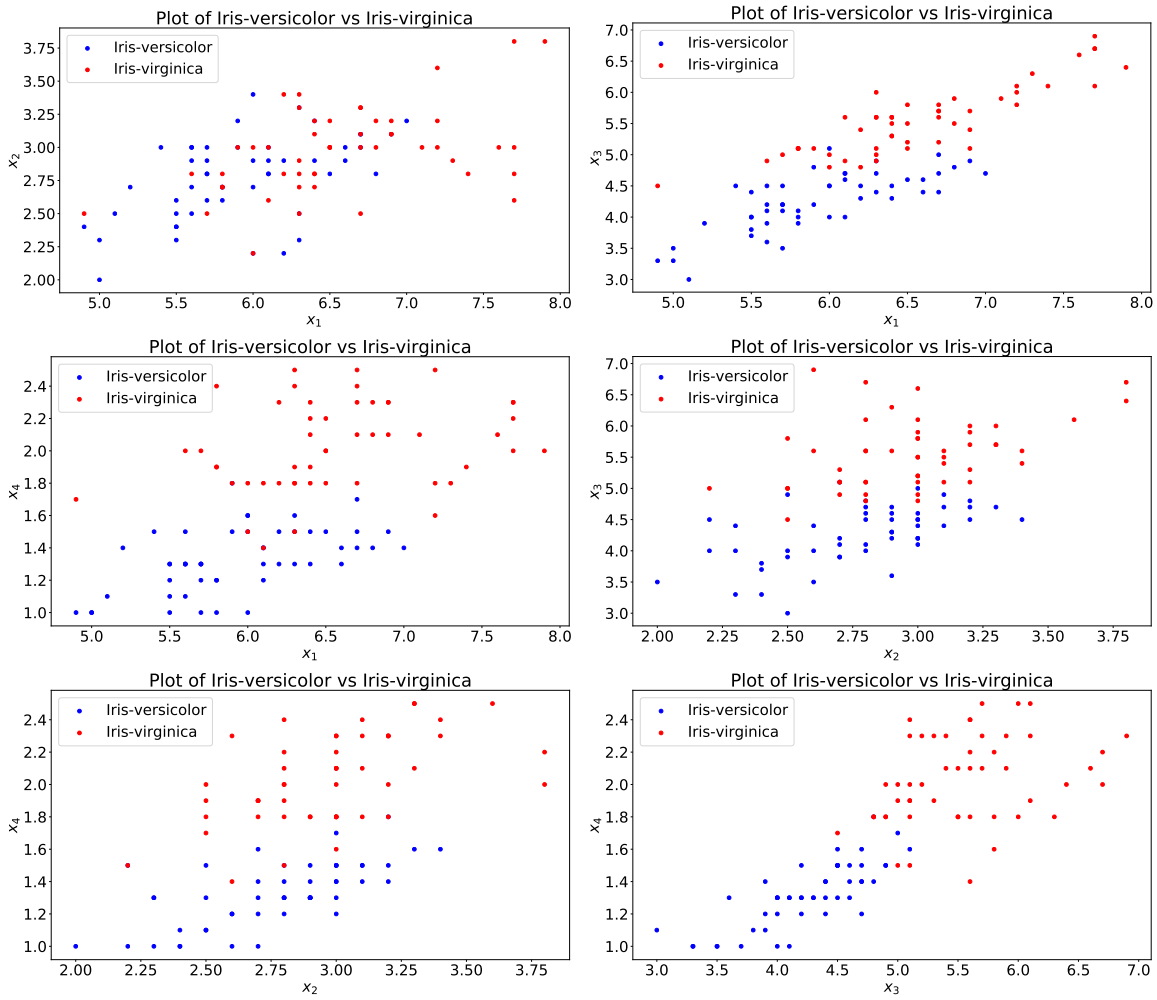


Figure 21. Cross sections of the Versicolor vs Virginica

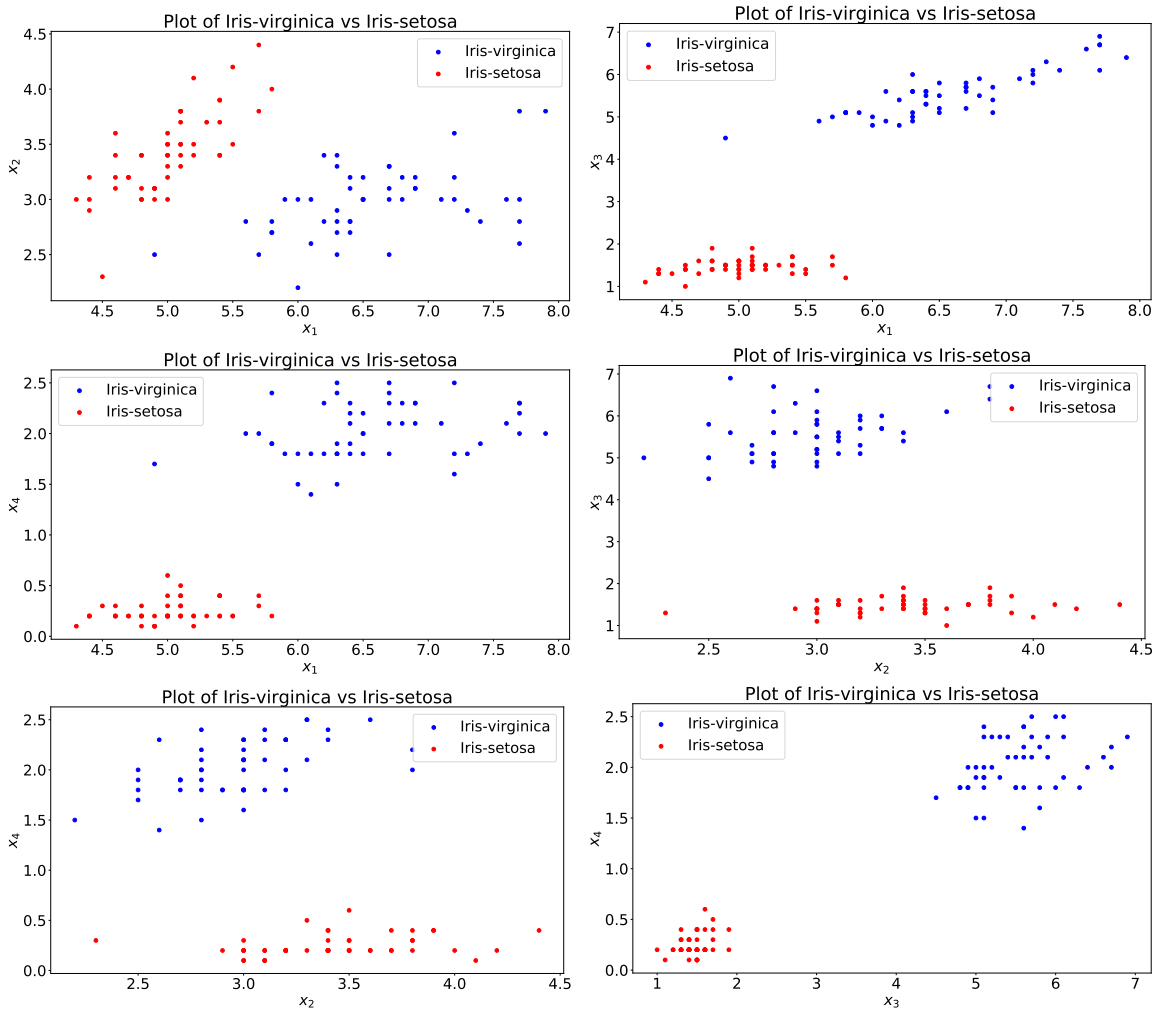
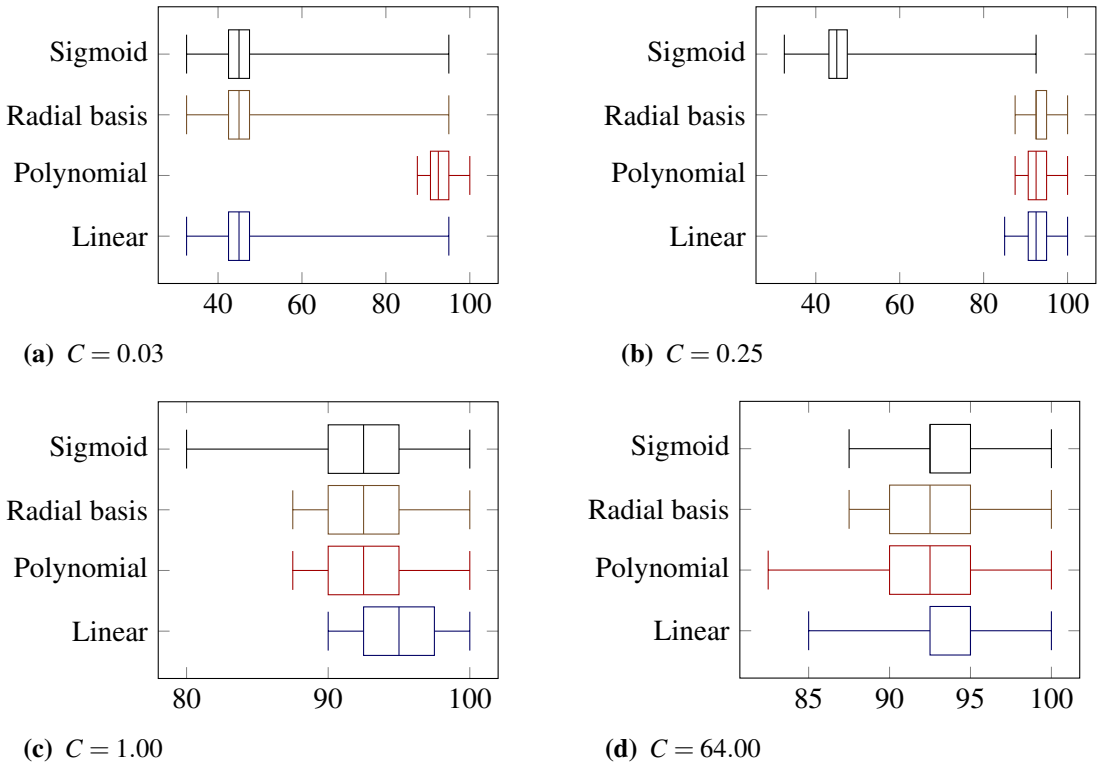


Figure 22. Cross sections of the Virginic vs Setosa

4.3 Results

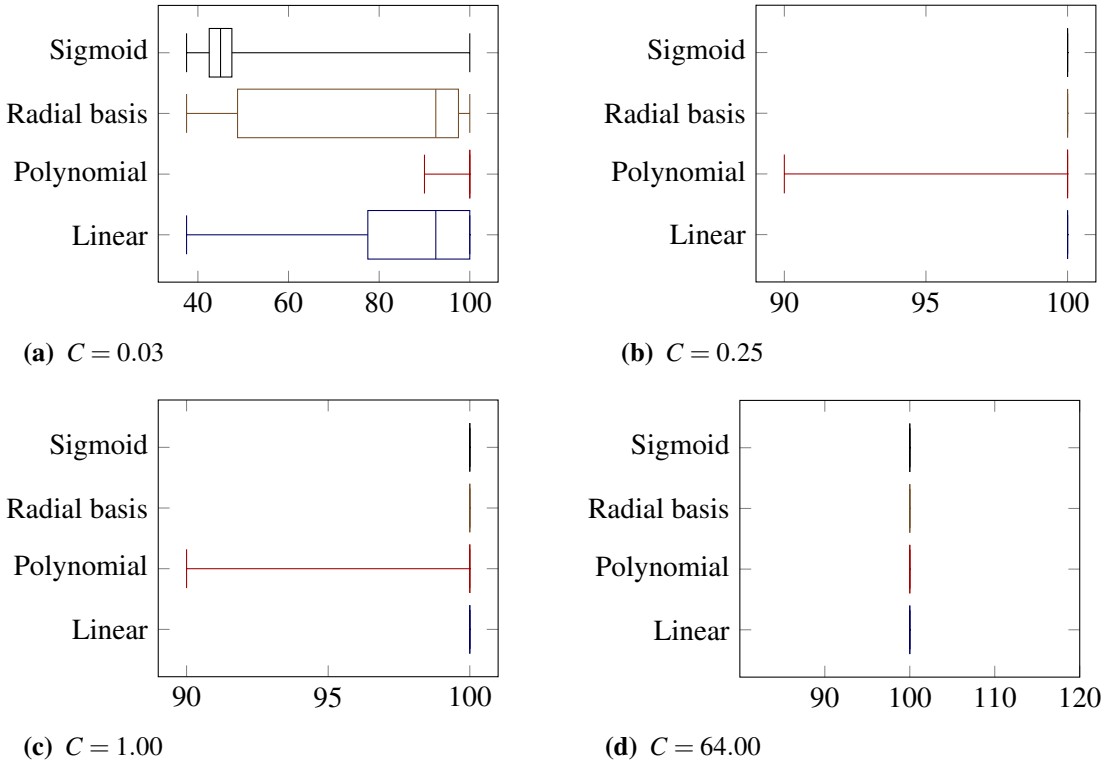
Figures 23 to 25 summarises the results achieved for the various problems using various C values and various functions.



(a) $C = 0.03$ **(b)** $C = 0.25$

(c) $C = 1.00$ **(d)** $C = 64.00$

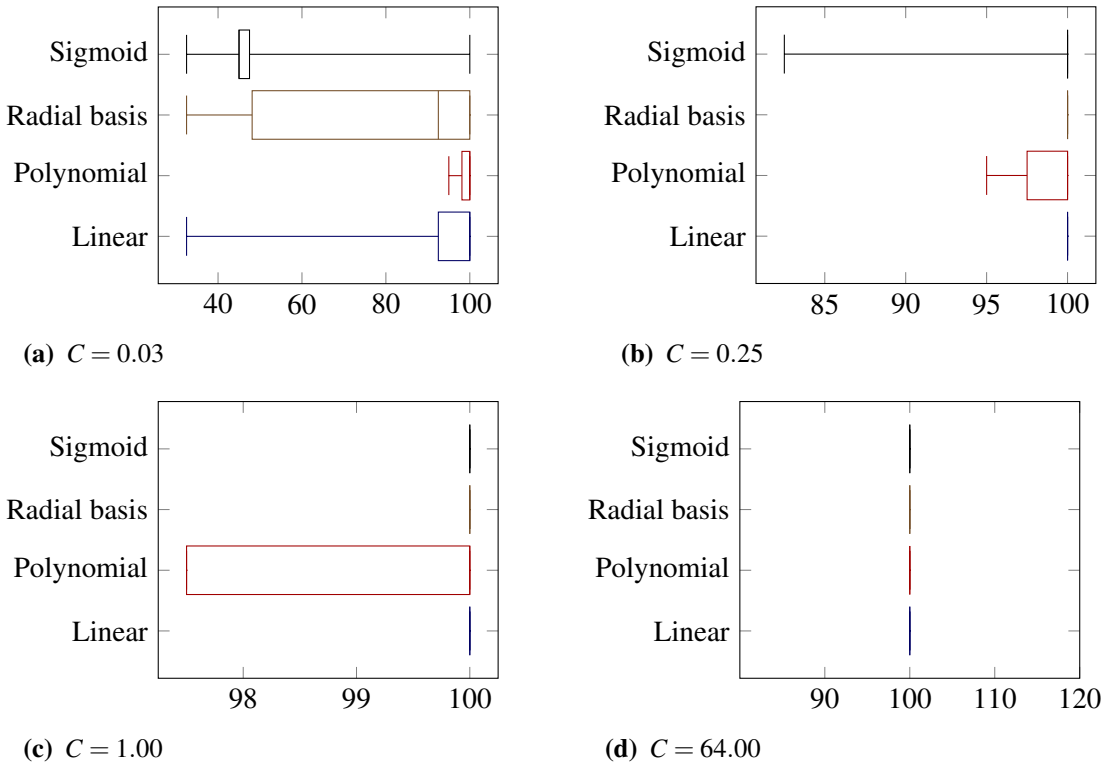
Figure 23. Summary of accuracies achieved for Versicolor vs Virginica



(a) $C = 0.03$ **(b)** $C = 0.25$

(c) $C = 1.00$ **(d)** $C = 64.00$

Figure 24. Summary of accuracies achieved for Versicolor vs Setosa



(a) $C = 0.03$ **(b)** $C = 0.25$ **(c)** $C = 1.00$ **(d)** $C = 64.00$

Figure 25. Summary of accuracies achieved for Virginica vs Setosa

Table 5 summarises the mean accuracies achieved.

Problem	C	**Mean accuracies**			
		Linear	Polynomial	Radial basis	Sigmoid
Versicolor vs Virginica	0.03	47.40	93.40	47.40	46.75
	0.25	93.10	93.50	93.60	47.40
	1.00	94.50	93.10	93.40	91.55
	64.00	93.65	92.65	92.95	93.85
Versicolor vs Setosa	0.03	85.35	99.80	77.75	53.35
	0.25	100.00	99.45	100.00	100.00
	1.00	100.00	99.50	100.00	100.00
	64.00	100.00	100.00	100.00	100.00
Virginica vs Setosa	0.03	92.95	99.30	78.40	53.05
	0.25	100.00	98.65	100.00	99.65
	1.00	100.00	99.05	100.00	100.00
	64.00	100.00	100.00	100.00	100.00

Table 5. Mean accuracies of the various parameters

Table 6 shows a small sample of converged parameters which achieved low cross-validation errors.

Problem	Function	Converged Parameters			
		d	γ	C_0	C
Versicolor vs Virginica	Linear	-	-	-	8.00
	Polynomial	2.00	1.18	40.91	0.06
	Radial basis	-	0.62	-	4.00
	Sigmoid	-	0.24	0.13	64.00
Versicolor vs Setosa	Linear	-	-	-	0.12
	Polynomial	2.00	0.25	0.02	4.00
	Radial basis	-	0.25	-	0.25
	Sigmoid	-	0.09	0.09	1.00
Virginica vs Setosa	Linear	-	-	-	0.12
	Polynomial	2.00	0.28	0.05	1.00
	Radial basis	-	0.25	-	0.25
	Sigmoid	-	0.09	0.09	1.00

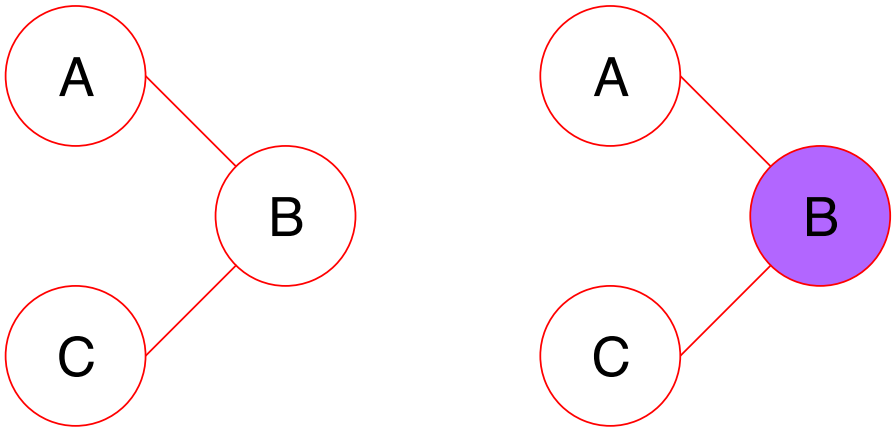
Table 6. Table showing the parameters with the lowest validation errors

4.4 Discussion

The SVM was able to achieve high accuracies for all the problems tested, provided that all the parameters were properly tuned. When looking at Figures 24 and 25, we see that a consistent accuracy of 100% is achieved provided that the parameter C is sufficiently large. This is due to the fact that the data is linearly separable, as there is a large gap between opposing classes. This allows most simple functions to easily split the data, without the need for any data to fall within the margin. This is confirmed in Table 6, where the parameters have settled to small values corresponding to simpler functions. The C value here is set to the first C value which achieves the highest cross-validation error. Due to this, these C values are relatively small and may be sub-optimal. This would be rectified with more training data or more iterations in the cross-validation step. For all the cases, we can see that when C is too small, the error contributed by each vector falling into the margin becomes negligible. Due to this, the SVM yields bad results. From Fig. 23, we can see that this Versicolor is more challenging to separate from Virginica. This is to be expected after one examines Fig. 21, as much of the opposing classes are overlapping. Due to this, intermediate values for C work the best as C values, which are too high, do not generalise very well and C values, which are too low, cannot capture the intricacies of the separation boundary.

5 Part C: Graphical Models

Graphical models form a powerful tool which can be used to convey dependencies between random variables. This allows one to easily identify the parameters which should be considered when marginalising and conditioning over variables, which forms the majority of operations in Bayesian machine learning. Graphical models can be split into two major categories, namely directed and undirected. Undirected graphical models are referred to as Markov random fields. Figure 26 demonstrates examples of two Markov random fields. Here nodes represent random variables and links represent dependencies between them.



(a) No observed variables **(b)** Known value of B

Figure 26. Examples of Markov random fields

From Fig. 26a, we see that A is dependant on B alone and so is C. However, B is dependant on both A and C. In Fig. 26b the value of B is observed and therefore, it is filled in. An important concept for Markov random fields is cliques, which are groups of fully connected nodes in the graph. In Fig. 26, there are only 2 cliques consisting of A,B and B,C. Maximal cliques are any cliques which do not form part of a larger clique. In Fig. 26, both of the cliques are maximal cliques. However, if there was a link between A and C, there would be one maximal clique consisting of A,B,C. The joint distribution over the variables A,B and C is shown in Eq. (5.1). Here ψ is known as an potential function and Z is a normalisation constant. The potential function is usually set to a function which is positive for all values of x. This ensures that all the probabilities produced by this function is positive. An example of such a function if $\psi(\mathbf{x}) = \exp(-E(\mathbf{x}))$, where E is an energy function.

$$\begin{aligned} P(\mathbf{x}) &= \frac{1}{Z}\prod_{C}\psi_C(\mathbf{x}_C) \\ P(A,B,C) &= \frac{1}{Z}\psi_{A,B}(A,B)\psi_{B,C}(B,C) \end{aligned} \tag{5.1}$$

To explore the concept of a Markov random field, an image de-noising application will be built. This model will follow the graphical design provided in Bishop[1].

5.1 Design

In order to design an image filter, we must first discuss how noise is added to an image. For this assignment, we will assume that an image is made of pixels which have values of either 1 or -1. From this image, noise can be added by randomly flipping the signs of a fraction of the pixels in the image. It is clear from this that the actual value of a pixel $t_{i,j}$ is strongly related to the pixels value in the noisy image $y_{i,j}$. Furthermore, it is save to assume that pixels in the de-noised image$x_{i,j}$ are strongly correlated with neighbouring pixels. This can be easily modelled with the use of Markov random fields as shown in Fig. 27.

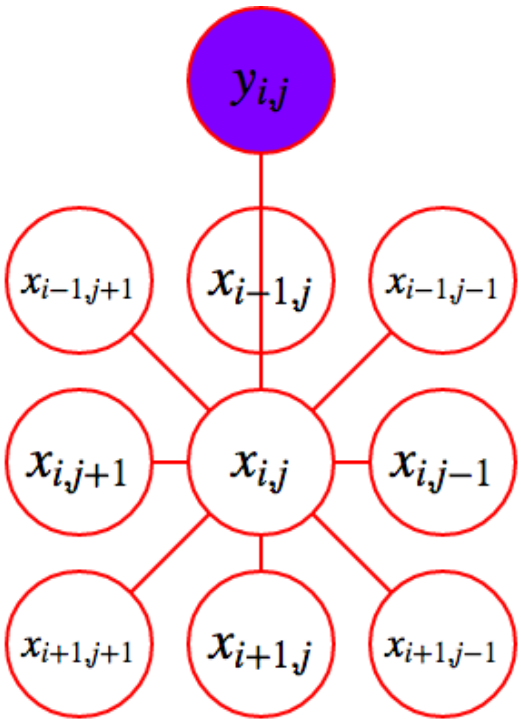


Figure 27. Markov random field of the image filter

It is also possible to add a prior to this, whereby we can bias the prevalence of one pixel value over another. The energy function for this model is given in Eq. (5.2), where I denotes summing over the image and A denotes the adjacent pixels to the current pixel. This summation is replaced with a window function W, which takes the weighted sum of neighbouring pixels, with respect to a kernel matrix $\mathbf{K}$.

$$\begin{aligned} E(x,y) &= h\sum_{i,j}^{I} x_{i,j} - \beta \sum_{i,j}^{I} x_{i,j} \sum_{k,l}^{A} x_{k,l} - \eta \sum_{i,j}^{I} x_{i,j} y_{i,j} \\ &= h\sum_{i,j}^{I} x_{i,j} - \beta \sum_{i,j}^{I} x_{i,j} W(x,i,j) - \eta \sum_{i,j}^{I} x_{i,j} y_{i,j} \end{aligned} \tag{5.2}$$

Setting $P(x,y) = \frac{1}{Z}\exp(-E(x,y))$, we seek to maximise the value of $P(x|y)$. Noting that $P(x|y) = \frac{P(x,y)}{P(y)}$, we see that maximising $P(x|y)$ is equivalent to $P(x,y)$, which is equivalent to minimising $E(x,y)$. This can be done in an iterative process, named iterated conditional modes (ICM)[8]. This is done by sequentially looping over individual pixels in the image and checking if the overall energy decreases when the pixel is flipped. This is done multiple times until either no pixels are flipped or until a

maximum iteration is reached. This can be made efficient by noting that the energy contribution of a pixel is affected by only a subset of the entire images pixels. Therefore, the change in energy caused by a pixel flip can be calculated very quickly.

$$
\begin{aligned}
\Delta E(x,y) &= E(x_{before},y) - E(x_{after},y) \\
&= h\sum_{i,j}^{I} x_{i,j,after} - \beta\sum_{i,j}^{I} x_{i,j,after}W(x,i,j) - \eta\sum_{i,j}^{I} x_{i,j,after}y_{i,j} - \\
&\left(h\sum_{i,j}^{I} x_{i,j,before} - \beta\sum_{i,j}^{I} x_{i,j,before}W(x,i,j) - \eta\sum_{i,j}^{I} x_{i,j,before}y_{i,j} \right) \\
&= h(x_{i,j,after} - x_{i,j,before}) - \beta W(x,i,j)(x_{i,j,after} - x_{i,j,before}) - \eta y_{i,j}(x_{i,j,after} - x_{i,j,before}) \\
&= h(-x_{i,j,before} - x_{i,j,before}) - \beta W(x,i,j)(-x_{i,j,before} - x_{i,j,before}) - \eta y_{i,j}(-x_{i,j,before} - x_{i,j,before}) \\
&= -2hx_{i,j,before} + 2\beta W(x,i,j) - x_{i,j,before} + 2\eta y_{i,j}x_{i,j,before} \\
\frac{\Delta E(x,y)}{2} &= -hx_{i,j,before} + \beta W(x,i,j)x_{i,j,before} + \eta y_{i,j}x_{i,j,before}
\end{aligned}
\tag{5.3}
$$

Algorithm 1 gives pseudo-code for how this algorithm can be implemented using $\frac{\Delta E(x,y)}{2}$.

Algorithm 1 Pseudo-code for the ICM algorithm

```
1:  procedure ICM(I_y, K)
2:      I_x ← copy of I_y
3:      while Not Converged do
4:          for i,j in shape(I_x) do
5:              if ΔE(x_{i,j}, y_{i,j}, K)/2 < 0 then
6:                  x_{i,j} ← −x_{i,j}
7:              end if
8:          end for
9:      end while
10: return I_x
11: end procedure
```

The code shown in Algorithm 1 is highly inefficient as the entire image needs to be processed on each run. This can be avoided by tracking the pixels which are in the vicinity of a flipped pixel, determined by the kernel K. These can then be processed on the next iteration. Note that it is possible that these algorithms converge to different values as both do not grantee a global maximum. Therefore, differences are introduced due to a change in the order of operations. The pseudo-code for this improved version is given in Algorithm 2. The implemented version of this algorithm keeps track of 2 sets, one of which for the current loop and one for the next loop. This was done in order to show the progress of the algorithm for each iteration.

Algorithm 2 Pseudo-code for the improved ICM algorithm

procedure ICM(I_y, K)
    $I_x \leftarrow$ copy of I_y
    $S \leftarrow \{\}$
    for i,j in shape(I_x) **do**
        if $\frac{\Delta E(x_{i,j}, y_{i,j}, K)}{2} < 0$ **then**
            $x_{i,j} \leftarrow -x_{i,j}$
            $S \leftarrow S \bigcup \{$set of all indicies in vacinity of i,j$\}$ ▷ Note that this operation only keeps unique entries in S
        end if
    end for
    while S is not empty and no other termination conditions are met **do**
        i,j=pop(s)
        if $\frac{\Delta E(x_{i,j}, y_{i,j}, K)}{2} < 0$ **then**
            $x_{i,j} \leftarrow -x_{i,j}$
            $S \leftarrow S \bigcup \{$set of all indicies in vacinity of i,j$\}$
        end if
    end while
return I_x
end procedure

5.2 Procedure

The ICM function is tested on the image shown in Fig. 28b. Here random noise has been added to the image shown in Fig. 28a. This is done as discussed in Section 5.1. A particle swarm algorithm, which was previously developed, is then used to optimise the relevant parameters for each kernel function. This is done by minimising the number of incorrectly classified pixels using the original image. It is unusual to use the test data in the optimisation of parameters. However, in this case, we are not concerned with the optimisation algorithm. Rather, we are interested in how well the ICM algorithm can perform given optimal conditions. In order to achieve fast convergence, η was set to 1 and the absolute value of β was taken. The error is then calculated as the total Number of incorrect pixels/Total number of pixels.

(a) Original image **(b)** Noisy image

Figure 28. Data used to test the image de-noising

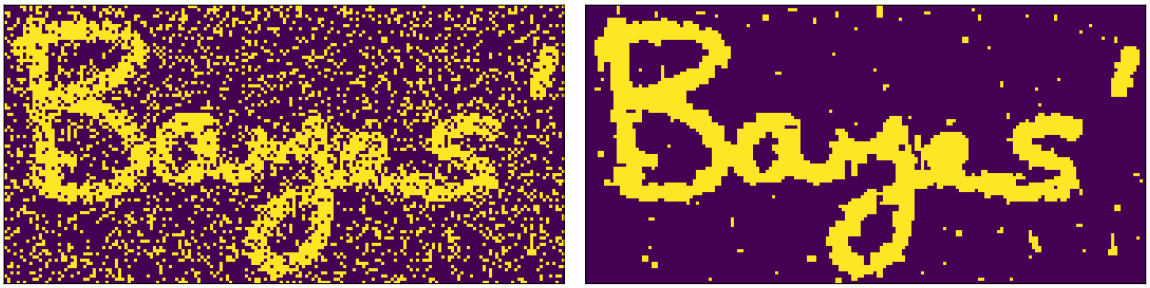
It was decided to test a variety of kernels in order to determine the effects of various shapes an weight configurations. These kernel configurations are summarised in Table 7.

Kernel Name	Kernel Function
Vertical	$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$
Horizontal	$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
Plus Shaped	$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$
N×N All Ones	$\begin{pmatrix} \dots & 1 & 1 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & 1 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & 1 & 1 & 1 & \dots \end{pmatrix}$
3×3 Adaptive Parameters	$\begin{pmatrix} a & 1 & a \\ 1 & 0 & 1 \\ a & 1 & a \end{pmatrix}$, where a is an adaptive parameter.
Diamond	$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$
5×5 Adaptive Parameters	$\begin{pmatrix} p_1 & p_2 & p_3 & p_2 & p_1 \\ p_4 & p_5 & p_6 & p_5 & p_4 \\ p_7 & p_8 & 0 & p_8 & p_7 \\ p_4 & p_5 & p_6 & p_5 & p_4 \\ p_1 & p_2 & p_3 & p_2 & p_1 \end{pmatrix}$, where p_1 to p_8 are Adaptive Parameters.
N×N Gaussian	$\frac{1}{Z}\begin{pmatrix} \mathcal{N}([-w,-w]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([0,w]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([w,w]\|\mathbf{0},w\mathbf{I}_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{N}([-w,0]\|\mathbf{0},w\mathbf{I}_2) & \dots & 0 & \dots & \mathcal{N}([w,0]\|\mathbf{0},w\mathbf{I}_2) \\ \mathcal{N}([-w,1]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([0,1]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([w,1]\|\mathbf{0},w\mathbf{I}_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{N}([-w,w]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([0,w]\|\mathbf{0},w\mathbf{I}_2) & \dots & \mathcal{N}([w,w]\|\mathbf{0},w\mathbf{I}_2) \end{pmatrix}$, where $w = \lfloor \frac{N}{2} \rfloor$ and Z is a normalisation constant.

Table 7. Table of kernels tested

5.3 Results

Figure 29 shows the result of each step of the ICM algorithm using the "Plus Shaped" kernel.



(a) Step 0

(b) Step 1

(c) Step 2

(d) Step 3

(e) Step 4

(f) Step 5

(g) Step 6

(h) Step 7

Figure 29. Steps of the ICM algorithm with the plus shaped kernel

Figure 30 shows the result of each step of the ICM algorithm using the 5×5 Adaptive Parameters kernel.

(a) Step 0

(b) Step 1

(c) Step 2

(d) Step 3

(e) Step 4

(f) Step 5

(g) Step 6

(h) Step 7

Figure 30. Steps of the ICM algorithm with the 5×5 Adaptive Parameters

Figure 31 shows the final results of the Gaussian kernel for various window sizes.

(a) $N = 3$

(b) $N = 5$

(c) $N = 7$

(d) $N = 9$

(e) $N = 11$

(f) $N = 13$

(g) $N = 15$

(h) $N = 17$

Figure 31. Results of the N×N Gaussian kernel for various N sizes

Figure 32 shows the final results of the "All Ones" kernel for various window sizes.

(a) $N = 3$

(b) $N = 5$

(c) $N = 7$

(d) $N = 9$

(e) $N = 11$

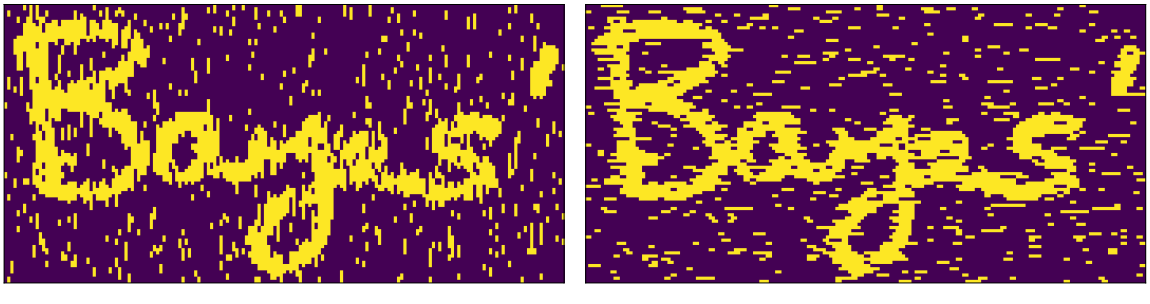
(f) $N = 13$

(g) $N = 15$

(h) $N = 17$

Figure 32. Results of the N×N All Ones Kernel for various N sizes

Figure 33 shows the final results for the other kernels tested.



(a) Vertical

(b) Horizontal

(c) 3×3 Adaptive Parameters

(d) Diamond

Figure 33. Results for the remaining kernels

The final error obtained for each kernel is summarised in Table 8.

Kernel Name	**Error (%)**	**Kernel Name**	**Error (%)**
Vertical	10.78	3×3 Adaptive Parameters	2.84
Horizontal	10.62	5×5 Adaptive parameters	1.98
Plus Shaped	3.15	Diamond	2.25
3×3 All ones	3.45	3×3 Gaussian	2.95
5×5 All ones	2.56	5×5 Gaussian	2.28
7×7 All ones	4.08	7×7 Gaussian	2.52
9×9 All ones	5.80	9×9 Gaussian	3.23
11×11 All ones	6.99	11×11 Gaussian	3.79
13×13 All ones	8.28	13×13 Gaussian	4.48
15×15 All ones	8.92	15×15 Gaussian	5.12
17×17 All ones	9.37	17×17 Gaussian	5.67

Table 8. Table summarising the errors achieved

5.4 Questions

5.4.1 Question A

The factor hx_i is to add a prior probability to the class 1 or -1. If h is positive, the algorithm would prefer negative values over positive values as $P(x,y) = \exp(-(\sum_i hx_i))$ would increase as $\sum_i hx_i$ decreases. Setting this value to zero, the algorithm would assume a uniform prior distribution over the classes.
The factor $-\beta x_i x_j$ controls how much a pixels value is related to neighbouring pixels. If neighbouring pixels are the same value, $x_i x_j$ would yield a positive result. $P(x,y)$ would therefore increase as the number of neighbouring pixels belong to the same class. Setting this value to zero would cause the output image to converge to a mixture between the prior and y. Assuming that h is also zero, we would see no change to x as neighbouring pixels would not contribute to the value of a given pixel.
Finally, the factor $-\eta x_i y_i$ controls how much the noisy images pixels should contribute to the current images pixels. This is due to the fact that $x_i y_i$ is positive when $x_i = y_i$, making $P(x,y)$ increase when there are more pixels where $x_i = y_i$. Setting η to zero would cause the output image to form large groups of pixels as the noisy image has no effect and only neighbouring pixels and the prior over classes controls a given pixels value.

5.4.2 Question B

If one where to set the value of β to a value which is larger than η, the image would cluster into various groups of a single class. Since a large β value prefers that neighbouring pixels are in the same class, $P(x,y)$ would be at a maximum when there are the fewest number of classes. This means that the algorithm will try to convert the entire image to either one class or another. Since ICM algorithm is not guaranteed to converge to a global minimum, it often forms a number of large clusters.

5.4.3 Question C

When looking at the parameter η separately, we can see that $P(x,y) \propto \exp(-(-\eta x_i y_i))$. We would like $P(x,y)$ to increase if $x_i = y_i$. If $x_i = y_i$, then $x_i y_i \geq 0$ and so $\eta x_i y_i \geq 0$, provided $\eta \geq 0$. Hence $P(x,y)$ increases if $x_i = y_i$ provided $\eta \geq 0$. Following the same argument for $-\beta x_i x_j$, we can see that $P(x,y)$ increases if $x_i = x_j$ provided $\beta \geq 0$.

5.4.4 Question D

As this report explored, various kernel sizes and kernel functions can be used in order to improve the estimates of the original image. Furthermore, when viewing text, it is important that lines stay intact. Therefore, a fourth parameter could be added which considers multiple pixels in a continuous line. This could be done by adding a term similar to $-\beta_0 x_i x_j x_k$. This would require that all three pixels have the same value in order to increase $P(x,y)$.

5.4.5 Question E

No, $P(x,y)$ only accounts for a bias, neighbouring pixels and observed pixels in order to make its estimates. Much more information could be considered in order to make a more optimal probability estimate.

5.5 Discussion

The ICM algorithm is simple yet effective at de-noising images. When using simple kernels, as shown in Fig. 29, many artefacts remain of the noise. This is especially prevalent in the linear kernels, as shown in Fig. 33, which are unable to remove noise that is perpendicular to the direction of the kernel. The kernel which used adaptive parameters was able to achieve the best results. This makes sense as it was able to over-fit to the output image and learn which combination of parameters achieved the best results. When looking at Figs. 31 and 32 we can see that increasing the kernel size starts to distort edges as a pixels value becomes more dependant on data which is further away. Increasing the kernel size to the size of the image would simulate a large prior parameter h. From Table 8, we can see that, in general, kernels which decrease the contribution of of parameters the further they are, the better the algorithm performs.

6 Conclusion

Kernel methods provide us with the ability to shift the computation expense from the number of dimensions to the number of data-points. This allows us to shift data from an input space to a potentially infinite feature space. The resulting algorithms perform extremely well when not much training data is available. However, the trade-off is that large amounts of data need to be kept when making predictions. This can be seen in Gaussian processes, which are able to model complex functions with minimal amounts of data. SVM's are able to easily create complex boundaries between opposing classes provided that the parameters are properly tuned. However, these parameters can be cosily to tune, especially for complex kernel function. This is due to the fact that a grid search is used to tune these parameters. Finally, graphical methods provides an effective way to communicate dependencies between random variables. Once these are properly modelled, deriving the joint distribution becomes a trivial task. Once the joint distribution is known, it is easy to determine the marginal distribution.

7 References

[1] M. B. Christopher, *Pattern Recognition and Machine Learning.* Springer-Verlag New York, 2016.

[2] J. MacKay David, "A practical bayesian framework for backprop networks," *Neural computation*, 1992.

[3] C. E. Rasmussen and H. Nickisch, "Gaussian processes for machine learning (gpml) toolbox," *Journal of Machine Learning Research*, vol. 11, no. Nov, pp. 3011–3015, 2010.

[4] H. Nyquist, "Certain topics in telegraph transmission theory," *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

[5] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[6] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[8] J. Kittler and J. Föglein, "Contextual classification of multispectral pixel data," *Image and Vision Computing*, vol. 2, no. 1, pp. 13–29, 1984.