GI01: Supervised Learning – In-Course Assessment 1 15.01.2016

Group Members

Goodacre, John – MSc Machine Learning
Riedel, Benjamin – MSc Computational Statistics and Machine Learning

Table of Contents and Contributions

Report compilation – Riedel, Benjamin

Important Note

All of the regressions and mean squared error calculations of Part 1, Exercise 1 through Exercise 7 were performed using a versatile MATLAB function (see Appendix A, page 20, `regression.m` function) fundamentally based on ridge regression.

Accordingly, the regressions and mean squared error calculations of Exercise 1 and Exercise 2 in their final form were not carried out using the standard least squares equations as requested. Nevertheless, the report at hand and the MATLAB script `PartIRegression.m` in Appendix A should hopefully be sufficient indication of our understanding of the subject matter.

For the sake of clarity, results of single trials are not included in this report unless they were specifically required.

Part I

Exercise 1 (LSR – effect of the training set size)

See `MATLAB` script `PartIRegression.m` in Appendix A, page 16.

Part a., b., c. and d.

See Table 1 below.

Table 1: Average effect over 200 trials of the training set size on training and test set error

	Training set $n = 100$	Training set $n = 10$
$\overline{\mathrm{MSE}}_{train}$	0.9937	0.8813
$\overline{\mathrm{MSE}}_{test}$	1.0191	1.1254

The mean squared error on the training set is lower than that on the test set. Furthermore, on average, the mean squared error on the training set is (marginally) lower for the smaller training set size, whereas the mean squared error on the test set is (marginally) lower for the larger training set size.

The least squares regression method is set up to minimize the mean squared error on the training set, not on the test set. Hence, the training set error is expected to be lower than the test set error. The performance on the test set depends on how well the regression function generalizes to unseen data. The training set error is higher for the larger training set size because the regression has less flexibility in fitting to the individual data points than in the case of the smaller training set. However, the solution associated with the larger training set in turn generalizes better to unseen data and thus its test set performance is better than that of the smaller training set.

Exercise 2 (LSR – effect of the dimensionality)

See `MATLAB` script `PartIRegression.m` in Appendix A, page 17.

Part a. and b.

See Table 2 below.

Table 2: Average effect over 200 trials of the dimensionality on the training and test set error

	Training set $n = 100$	Training set $n = 10$
$\overline{\mathrm{MSE}}_{train}$	0.9042	0.000
$\overline{\mathrm{MSE}}_{test}$	1.1198	6459.8391

Part c.

For a small training set size given relatively high dimensional data, the test set performance is poor despite the mean squared error on the training set being very small. The reason for this is that the regression function will fit the random noise too much, while the interesting part of the signal is too small. This result is an example of the concept of overfitting.

Exercise 3 (Ridge regression)

Part a.

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}}\, \gamma \mathbf{w}^{\mathrm{T}}\mathbf{w} + \frac{1}{l}\sum_{i=1}^{l}(\mathbf{x}_i{}^{\mathrm{T}}\mathbf{w} - y_i)^2$$

$$= \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{l}(\mathbf{y}-\mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y}-\mathbf{X}\mathbf{w}) + \gamma \mathbf{w}^{\mathrm{T}}\mathbf{w}$$

Since

$$\nabla\left[\frac{1}{l}(\mathbf{y}-\mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y}-\mathbf{X}\mathbf{w}) + \gamma \mathbf{w}^{\mathrm{T}}\mathbf{w}\right] = -\frac{1}{l}2\mathbf{X}^{\mathrm{T}}(\mathbf{y}-\mathbf{X}\mathbf{w}) + 2\gamma\mathbf{w}$$

at minimum:

$$0 = -\frac{1}{l}2\mathbf{X}^{\mathrm{T}}(\mathbf{y}-\mathbf{X}\mathbf{w}^{*}) + 2\gamma\mathbf{w}^{*}$$

$$0 = -\mathbf{X}^{\mathrm{T}}\mathbf{y} + \mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w}^{*} + l\gamma\mathbf{w}^{*}$$

$$\Rightarrow \mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w}^{*} + l\gamma\mathbf{w}^{*} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

$$(\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I})\mathbf{w}^{*} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

$$\Rightarrow \mathbf{w}^{*} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

Part b.

If $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is positive definite, $\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I}$ is clearly positive definite as we are simply adding positive numbers to the diagonal of a non-singular matrix.

More specifically, $l\gamma\mathbf{I}$ is positive definite as it is a diagonal matrix with positive elements along the diagonal since $l\gamma > 0$. Furthermore, if $\mathbf{A}$ and $\mathbf{B}$ are both positive definite, then their sum is positive definite – that is, $\mathbf{v}^{\mathrm{T}}(\mathbf{A}+\mathbf{B})\mathbf{v} = \mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} + \mathbf{v}^{\mathrm{T}}\mathbf{B}\mathbf{v} > 0$.

Therefore, as stated above, $\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I}$ is positive definite if $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is positive definite.

In the case where $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is singular, one can apply singular value decomposition as follows.

$$\begin{aligned}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} &= (\mathbf{V}\mathbf{D}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}})^{-1} \\ &= (\mathbf{V}\mathbf{D}^{2}\mathbf{V}^{\mathrm{T}})^{-1} \\ &= \mathbf{V}\mathbf{D}^{-2}\mathbf{V}^{T}\end{aligned}$$

Here, the diagonal matrix $\mathbf{D}$ is not invertible and has one or more zero values.

Adding the regularisation parameter, however, leads to following observation.

$$\begin{aligned}(\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I})^{-1} &= (\mathbf{V}\mathbf{D}^{2}\mathbf{V}^{\mathrm{T}} + l\gamma\mathbf{I})^{-1} \\ &= \mathbf{V}(\mathbf{D}^{2} + l\gamma\mathbf{I})^{-1}\mathbf{V}^{\mathrm{T}}\end{aligned}$$

Now, the difference to above is that positive values are added to the diagonal elements of $\mathbf{D}^{2}$. This ensures that all the eigenvalues of the resultant matrix are now strictly greater than zero and thus $\mathbf{X}^{\mathrm{T}}\mathbf{X} + l\gamma\mathbf{I}$ is positive definite.

Exercise 4 (Effect of the regularisation parameter)

See `MATLAB` script `PartIRegression.m` in Appendix A, page 17.

Part a. and b.

See Figure 1 below.

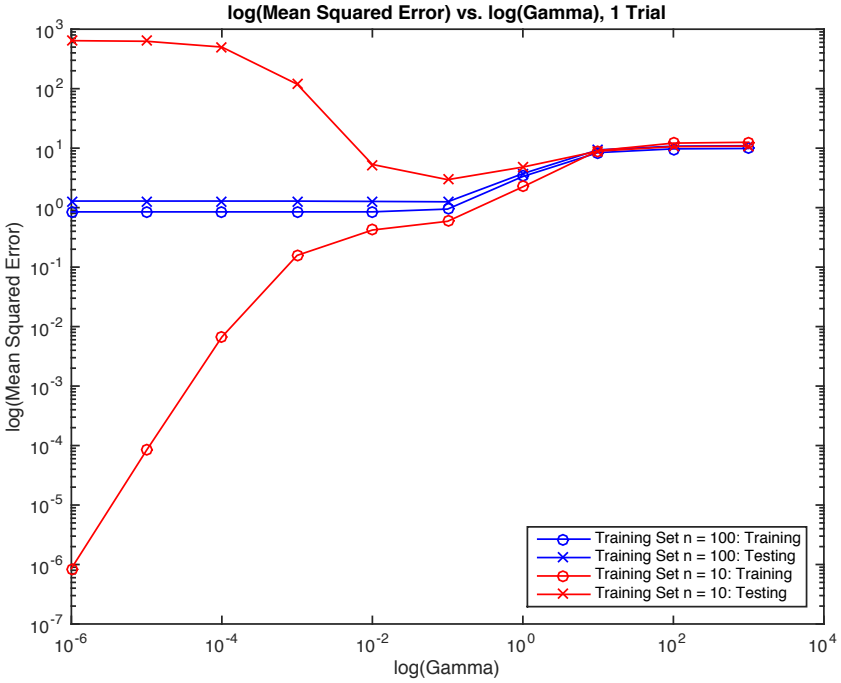

Figure 1: Effect of the regularisation parameter on training and test set error

Part c.

See Figure 2 below.

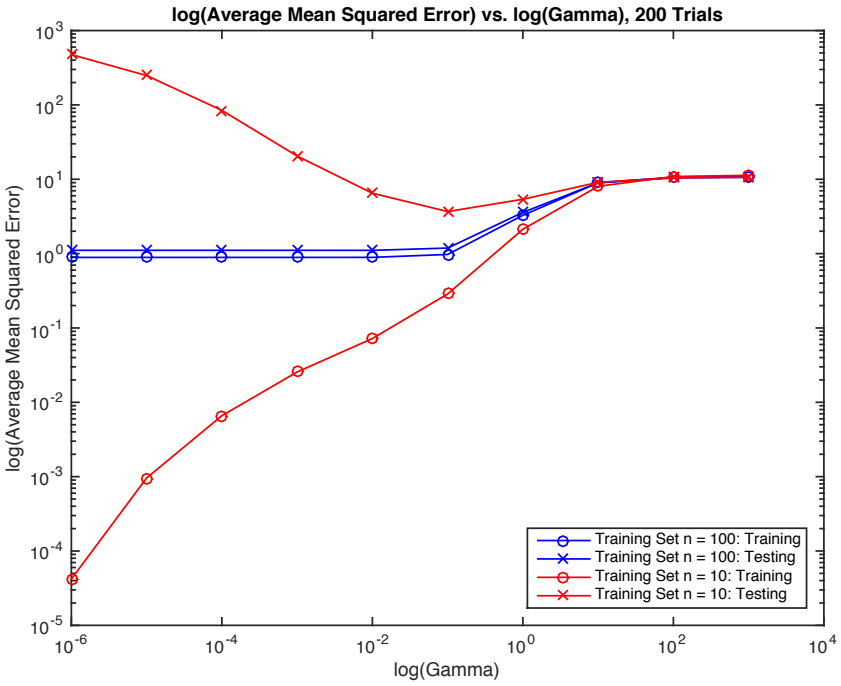

Figure 2: Average effect over 200 trials of regularisation parameter on training and test set error

The results suggest that evaluating the training set error is not a viable method for regularisation parameter selection with the goal of minimizing test set error, esp. given a small training set size or differing training set sizes. For example, in the case of the $n = 10$ training set size, selecting the level of the regularisation parameter minimizing training set error would lead to very poor test set performance.

Exercise 5 (Tuning the regularisation parameter using a validation set)

See `MATLAB` script `PartIRegression.m` in Appendix A, page 18.

Part a. and b.

See Table 3, Figure 3 and Figure 4 below.

Table 3: Training and test set error using regularisation parameter selected during validation

	Training set $n = 100$	Training set $n = 10$
$\mathrm{MSE}_{train}(\gamma_{\min \mathrm{MSE}_{validation}})$	0.8482	9.0931
$\mathrm{MSE}_{test}(\gamma_{\min \mathrm{MSE}_{validation}})$	1.2897	8.9112

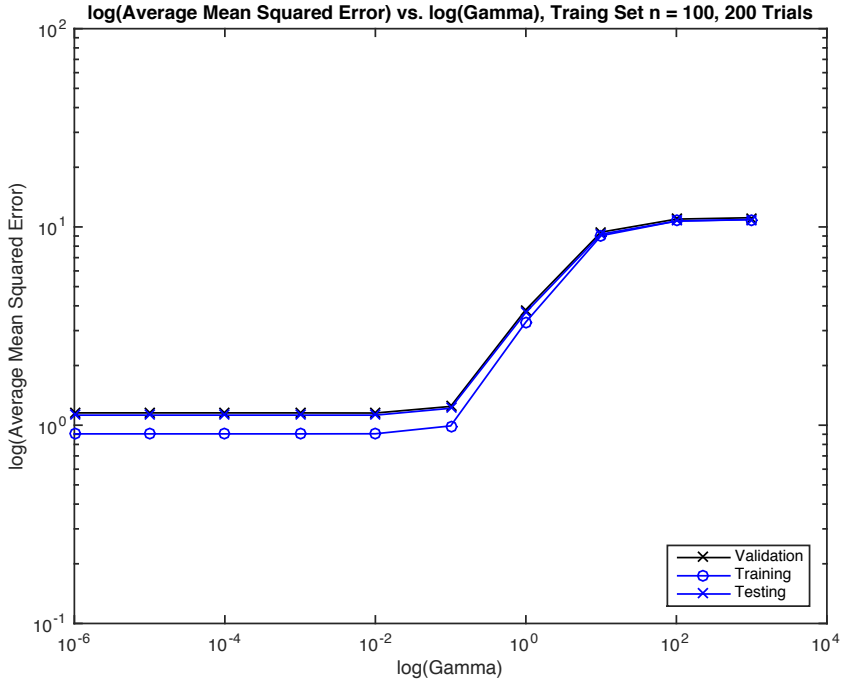

Figure 3: Average effect over 200 trials of regularisation parameter on validation, training and test set error (training set size $n = 100$)

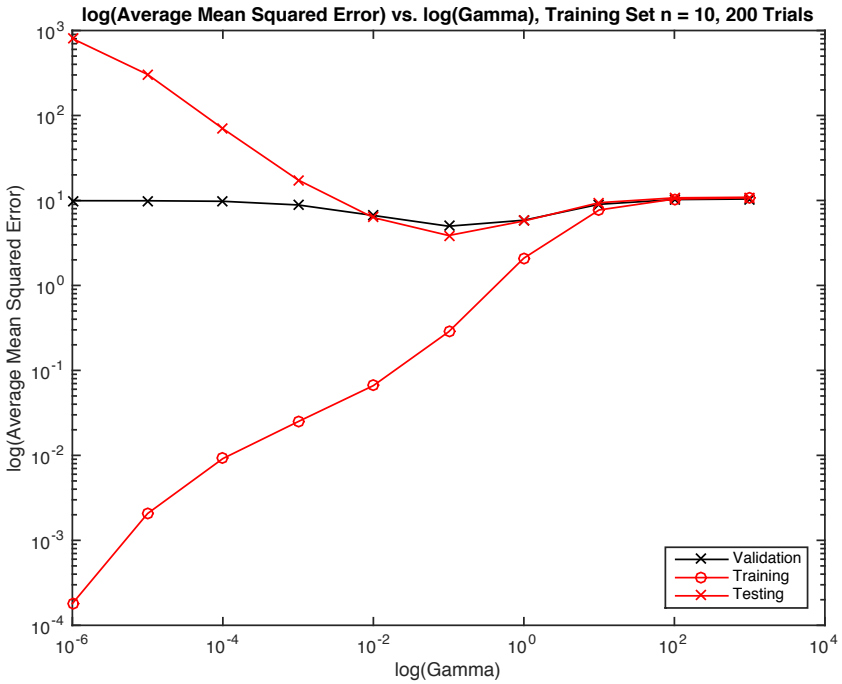

Figure 4: Average effect over 200 trials of regularisation parameter on validation, training and test set error (training set size $n = 10$)

Part c. and d.

See Table 4 below.

Table 4: Average over 200 trials of regularisation parameter selected during validation

	$\bar{\gamma}^{(100)}$	$\bar{\gamma}^{(10)}$
10 dimensional data set	0.0360	97.0496
1 dimensional data set	76.9256	296.4687

The average selected regularisation parameter is larger for the smaller training set size of $n = 10$, both for the 10 dimensional and 1 dimensional data sets. The reason for this is that – as seen above – the risk of overfitting is higher for the smaller training set. Therefore, the complexity of the respective weight vector must be restricted more to limit susceptibility to the influence of noise and to thereby improve generalizability.

Compared to the case of 10 dimensional data, the average selected regularisation parameter is higher for 1 dimensional data. The reason is analogous to above: the flexibility of the regression function is higher in 1 dimension, as is the risk of overfitting. The regularisation parameter hence has to be larger in the 1 dimensional case in order to counteract this higher level of risk.

Exercise 6 (Tuning the regularisation parameter using cross-validation)

See `MATLAB` script `PartIRegression.m` in Appendix A, page 19.

Part a. and b.

See Figure 5 and Figure 6 below.

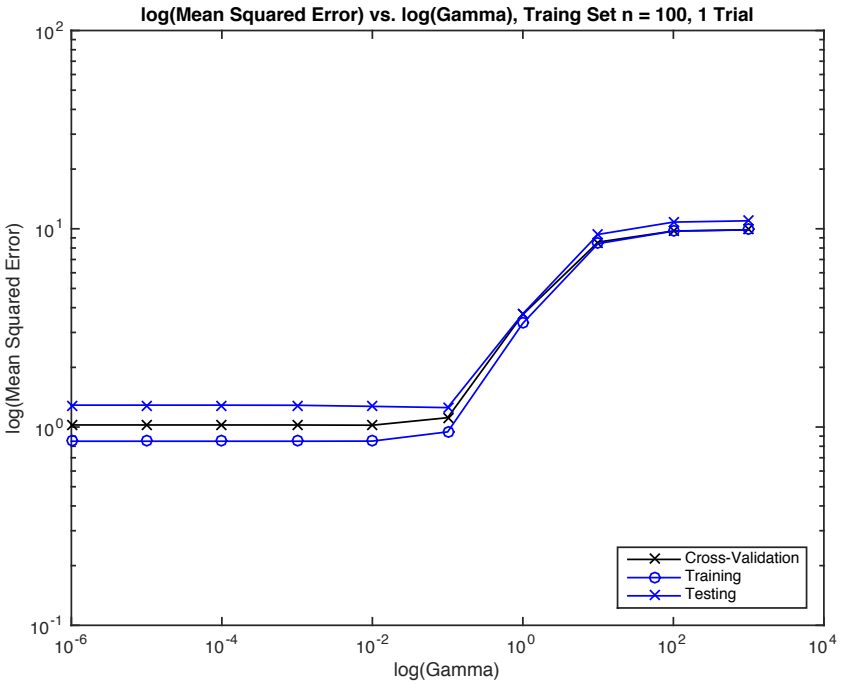

Figure 5: Effect of the regularisation parameter on cross-validation, training and test set error (training set size $n = 100$)

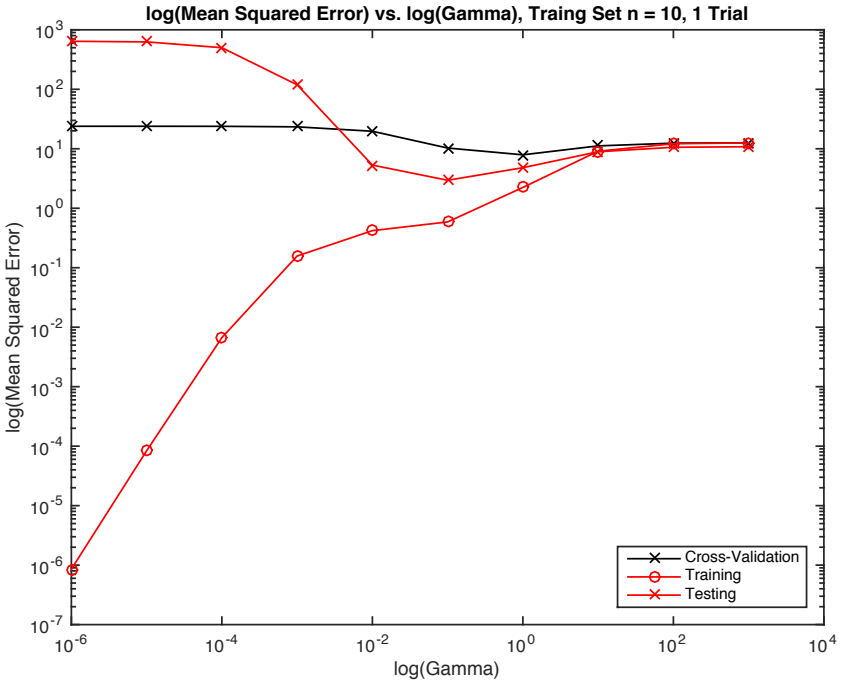

Figure 6: Effect of the regularisation parameter on cross-validation, training and test set error (training set size $n = 10$)

Exercise 7 (Comparison of regularisation parameter tuning methods)

See MATLAB script `PartIRegression.m` in Appendix A (page 19) and Table 5 below.

Table 5: Performance comparison based on 200 trials of different regularisation parameter tuning methods

	Training set $n = 100$		Training set $n = 10$	
Method	$\overline{\mathrm{MSE}}_{test}$	σ	$\overline{\mathrm{MSE}}_{test}$	σ
Min. training error	1.1089	0.0844	681.2132	3192.9952
Min. validation error	1.1378	0.0976	16.8813	66.3577
Min. cross-validation error	1.1120	0.0838	5.9280	7.7744

Exercise 9 (Baseline versus full linear regression)

See MATLAB script `PartIKernel.m` in Appendix B, page 22.

Part a., b. and c.

See Table 7 in Exercise 10, Part d. below.

Exercise 10 (Kernel ridge regression)

See MATLAB script `PartIKernel.m` in Appendix B, page 23.

Part a. and b.

See functions `kridgereg.m` and `dualcost.m` in MATLAB script `PartIKernel.m` in Appendix B, page 25.

It should briefly be noted that the function `kridgereg.m` using the left division operator "\" correctly computes the dual weight vector α^*. As indicated in the MATLAB help text associated with the left division operator / `mldivide.m function`, $\mathbf{A}\backslash\mathbf{B}$ is the matrix division of matrix $\mathbf{A}$ into matrix $\mathbf{B}$ and is roughly equivalent to $\mathbf{A}^{-1}\mathbf{B}$, albeit using a different computational approach. Moreover, $\mathbf{X} = \mathbf{A}\backslash\mathbf{B}$ constitutes the solution to the equation $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A}$ is a l-by-l matrix and $\mathbf{B}$ is a column vector of l components – as is the case for the equation $\alpha^*(\mathbf{K} + \gamma l\mathbf{I}_l) = \mathbf{y} \Rightarrow \alpha^* = (\mathbf{K} + \gamma l\mathbf{I}_l)^{-1}\mathbf{y}$ where $(\mathbf{K} + \gamma l\mathbf{I}_l)$ and $\mathbf{y}$ respectively correspond to $\mathbf{A}$ and $\mathbf{B}$.

Part c.

See Figure 7 and Table 6 below.

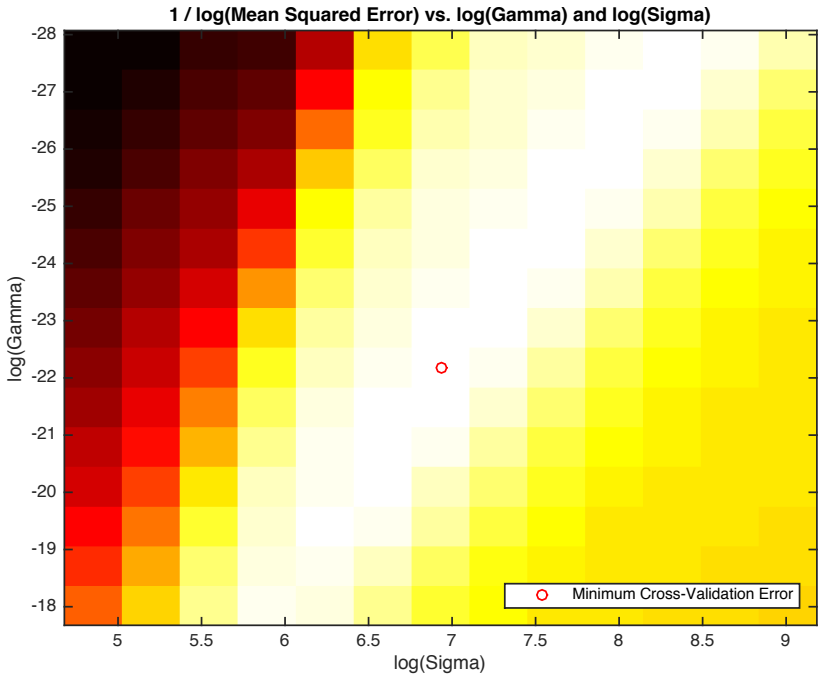

Figure 7: Cross-validation error as a function of regularisation and variance parameters

Table 6: Train and test error of kernel ridge regression for best performing regularisation and variance parameters

	Kernel ridge regression
$\mathrm{MSE}_{train}(\gamma,\sigma)_{\min \mathrm{MSE}_{validation}}$	6.1374
$\mathrm{MSE}_{test}(\gamma,\sigma)_{\min \mathrm{MSE}_{validation}}$	14.0776

Part d.

See Table 7 below.

Table 7: Performance comparison based on 20 random train / test splits of different regression methods

Method	$\overline{\mathrm{MSE}}_{train}$	σ	$\overline{\mathrm{MSE}}_{test}$	σ
Naïve regression	84.6329	5.3397	84.2251	10.5829
Linear regression (attribute 1)	70.4115	3.8695	75.2671	7.9041
Linear regression (attribute 2)	72.7949	4.8714	75.1957	9.7808
Linear regression (attribute 3)	65.6118	4.1402	63.0964	8.1468
Linear regression (attribute 4)	82.0045	3.0899	82.1555	6.5055
Linear regression (attribute 5)	68.9702	5.1644	69.3575	10.2884
Linear regression (attribute 6)	44.3658	3.6634	42.5641	7.1013
Linear regression (attribute 7)	71.0050	6.8040	75.7200	13.3939
Linear regression (attribute 8)	81.5292	3.7289	74.9711	7.1731
Linear regression (attribute 9)	73.3256	3.7184	70.1969	7.2080
Linear regression (attribute 10)	64.6243	6.1497	68.7761	12.1070
Linear regression (attribute 11)	60.9878	3.5307	66.4633	7.0891
Linear regression (attribute 12)	74.2591	5.2996	76.7468	10.3655
Linear regression (attribute 13)	37.3411	2.5560	41.1314	5.0414
Linear regression (all attributes)	24.1514	1.8487	25.6877	3.4909
Kernel ridge regression	8.2005	1.7230	14.1692	4.5721

Part II

Question 1 (Using linear regression with a Gaussian kernel to simulate 1-nearest neighbour classifier)

Given the test observation $\mathbf{x}_{test}$, linear regression with the Gaussian kernel $K_\beta(\mathbf{x}, \mathbf{t}) = \exp(-\beta\|\mathbf{x} - \mathbf{t}\|^2)$ can be used to simulate a 1-nearest neighbour classifier by increasing the value of β until the kernel $K_\beta(\mathbf{x}_{test}, \mathbf{x}_i)$ is equal to (or at least tends to) 0 for all $i = 1, \dots, n$ except for one observation – say observation $\mathbf{x}_k$.

In this case, the following equation to perform linear regression with a Gaussian kernel

$$y_{test} = \frac{\sum_{i=1}^{n} K_\beta(\mathbf{x}_{test}, \mathbf{x}_i) y_i}{\sum_{i=1}^{n} K_\beta(\mathbf{x}_{test}, \mathbf{x}_i)}$$

where y_{test} is the predicted label for the test observation $\mathbf{x}_{test}$ and y_i is the observed label for observation $\mathbf{x}_i$, will simplify to

$$y_{test} \cong \frac{K_\beta(\mathbf{x}_{test}, \mathbf{x}_k) y_k}{K_\beta(\mathbf{x}_{test}, \mathbf{x}_k)}$$

Hence, the predicted label y_{test} for the test observation $\mathbf{x}_{test}$ will be (roughly) equal to the label y_k of the nearest neighbour $\mathbf{x}_k$ in terms of the Gaussian kernel.

Question 2 (Using a perceptron to learn a linear classifier with a bias)

Part a.

Given that

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + b = (\mathbf{w}, b)^{\mathrm{T}}(\mathbf{x}, 1) = \mathbf{w}'^{\mathrm{T}}\mathbf{x}'$$

the perceptron separating through the origin may be changed to one that separates with a bias b by adding a value of one to $\mathbf{x}$.

Part b.

The Novikoff conditions for the perceptron specify that all of the data points $\mathbf{x}$ must lie within a ball of radius R (clearly, this can be of high dimensionality). In addition, the distance of the separating perceptron hyperplane to the data points on either side must be at least γ.

At time t, the Novikoff condition on the number of mistakes M_t should be as follows.

$$M_t \leq \left(\frac{R}{\gamma}\right)^2$$

Incorporating the bias term in the perceptron as outlined above will change the Novikoff condition on the number of mistakes M_t incurred by the algorithm. Specifically, the bound on the number of mistakes may increase or decrease depending on the dataset.

The following two simple examples nicely illustrate this point.

Example 1

Let data points in the (x, y) plane be labelled as:

- Negative label: $(-1/n, 0)$, $(-1/n, 1)$
- Positive label: $(2,0)$, $(2,1)$

A perceptron without bias would in this case have $\gamma = 1/n$ and $R = \sqrt{5}$. For a very large n or data points very close to the separating line, the Novikoff bound would be very large, i.e. $M \leq n^2\sqrt{5}$.

However, if a bias term is added to the perceptron, the data can be clearly separated by a plane with $\gamma = 1$, whereas R only increases marginally to a value of 3 (i.e. the norm of $(2, 2, 1)$). Hence, the Novikoff bound has decreased.

Example 2

Now, let data points in the (x, y) plane have the following labels:

- Negative label: $(-1/2, 0)$, $(-1/2, 1)$
- Positive label: $(1,0)$, $(1,1)$

Here, $\gamma = 1/2$, $R = \sqrt{2}$ and $M \leq 2\sqrt{2}$ for a perceptron without a bias term.

If a bias term is introduced by adding one dimension, however, γ will increase to $\sqrt{5/4}$, R will increase to $\sqrt{3}$ and thus the Novikoff bound has increased. In other words, since $\gamma R < 1$ in this case, increasing the dimensionality by adding the bias term leads to a larger M (using Pythagoras' theorem to calculate the new γ).

As already indicated, introducing the bias term therefore has a data dependent effect on the Novikoff bound.

Question 3 (Kernel modification)

Part a.

For $K_c(\mathbf{x}, \mathbf{z}) = c + \sum_{i=1}^{n} x_i z_i$ where $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$ and $c \in \mathbb{R}$ to be a valid, positive semidefinite kernel, the function must satisfy Mercer's theorem: that is, there exists a feature map $\boldsymbol{\phi} : \mathbb{R}^n \to \mathcal{W}$ for Hilbert space $\mathcal{W}$ such that $K(\mathbf{x}, \mathbf{z}) = \langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{z}) \rangle$, where $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ and $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$.

Let a feature map be given by $\boldsymbol{\phi}(\mathbf{x}) = (\sqrt{c}, x_1, x_2, \dots, x_n)$.

Then, $\langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{z}) \rangle = (\sqrt{c}, x_1, x_2, \dots, x_n)^{\mathrm{T}} (\sqrt{c}, z_1, z_2, \dots, z_n) = c + \sum_{i=1}^{n} x_i z_i = K_c(\mathbf{x}, \mathbf{z})$.

For this to be valid, $c > 0$ and thus $K_c(\mathbf{x}, \mathbf{z})$ is a valid kernel only for such values.

Part b.

$K_c(\mathbf{x}, \mathbf{z})$ is non-homogeneous first order polynomial kernel.

The feature map essentially adds another dimension / bias to the feature vector – that is $\boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^{n+1}$.

Having said this, implemented as part of the kernel $K_c(\mathbf{x}, \mathbf{z})$ in linear regression, the constant c appears to act as a regularization parameter similar to the γ value in ridge regression.

That is, as the value of c is increased, the elements of the optimal dual weight vector α^* are increasingly weighed down. As $c \to \infty$, the elements of the dual weight vector all seem to tend towards 0. See Figure 8 below for an exemplary graphical representation of said effect of c.

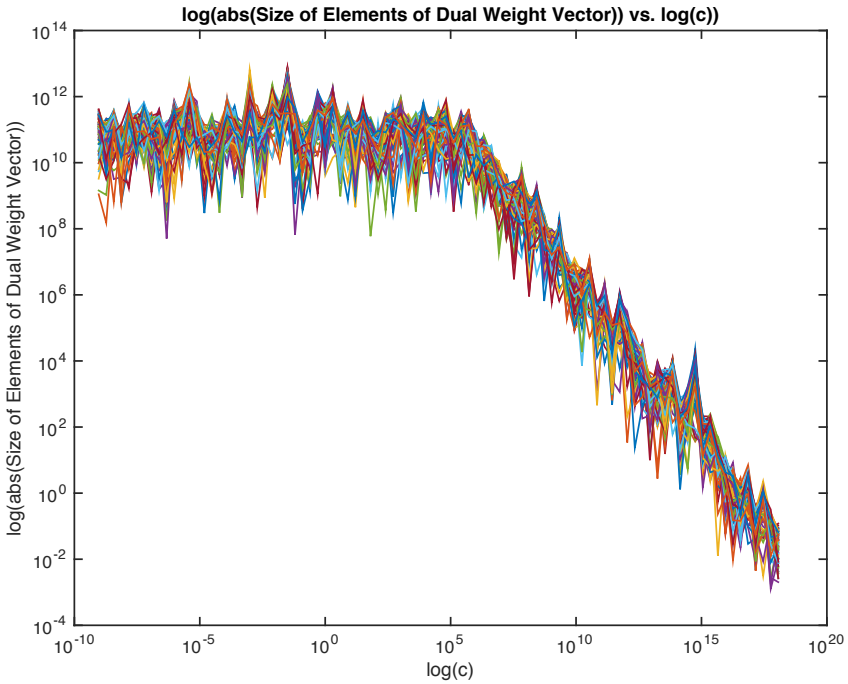

Figure 8: Effect of regularization parameter c on size of elements of the optimal dual weight vector α^* (data set: Boston housing; setup: least squares regression using kernel $K_c(\mathbf{x}, \mathbf{z}) = c + \sum_{i=1}^{n} x_i z_i$, incl. all 13 attributes, average over 100 random training sets of size $n = 100$)

Question 4 (Sparse learning)

Part a.

See `MATLAB` script `PartIISparseLearning.m` in Appendix C (page 26) and Figure 9 as well as Figure 10 below.

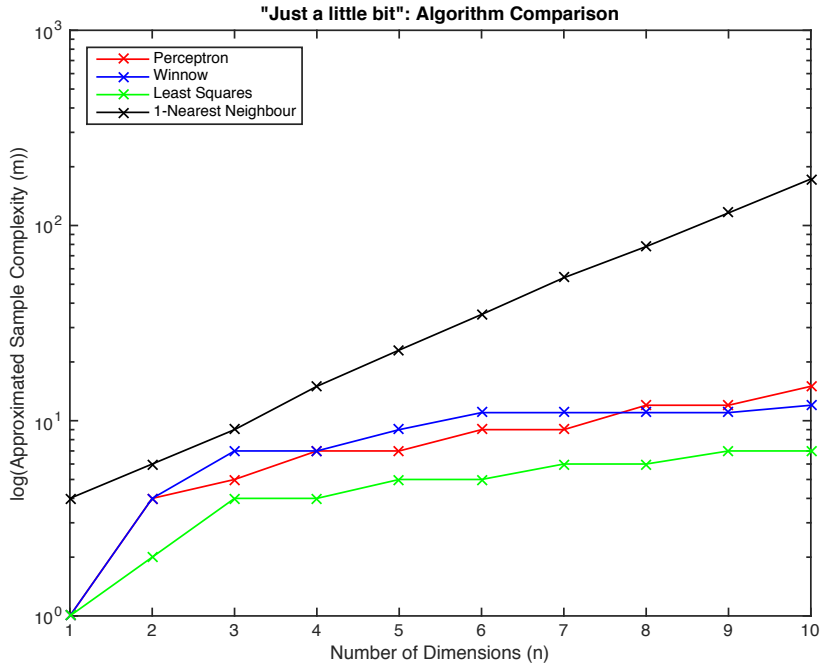

Figure 9: Performance comparison of classification algorithms in terms of approx. sample complexity as a function of number of dimensions; $1{,}000$ training set samples and full test set

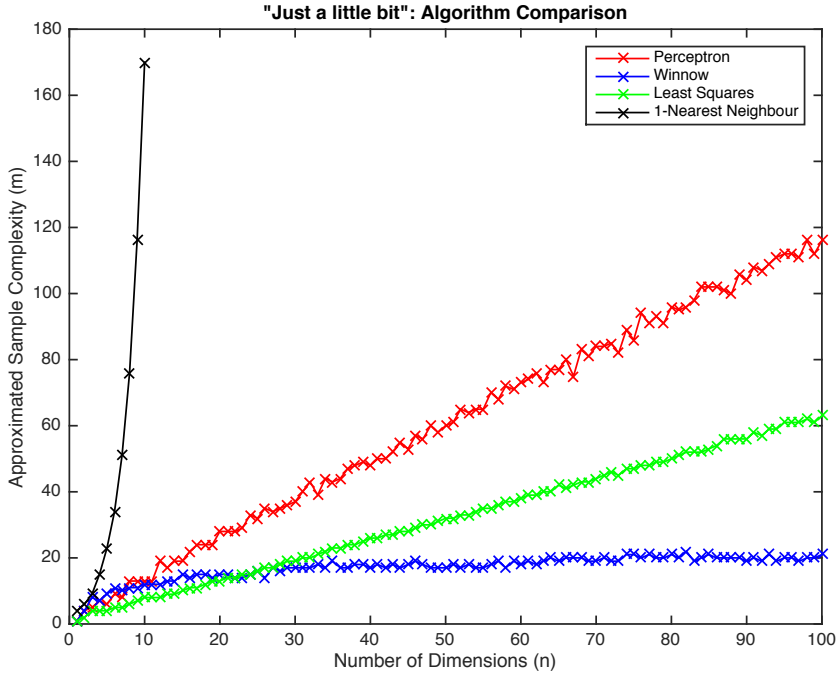

Figure 10: Performance comparison of classification algorithms in terms of approx. sample complexity as a function of number of dimensions; 100 training and test set samples with test set size equal to 100 observations

Part b.

i. What we did

The method employed for separately approximating the sample complexity of the classification algorithms is based on nested (pseudo) random sampling. Please refer to `MATLAB` script `PartIISparseLearning.m` in Appendix C (page 26) in reading the below.

At first, the sample generalization error for a specified number of dimensions n and training observations m is estimated for number of dimensions n larger than variable `complimitn`. Below the value specified by this variable, the entire set $\mathbf{x} \in \{-1,1\}^n$ is generated and tested upon, thus entailing an exact calculation of the generalization error given a specific training set. In the script submitted, the value is set to 10, however, an exact computation of the generalization error given a specific training set is computationally feasible up to a value of 25.

Above the value of `complimitn`, a specified number of pseudo random `testsamples` of size `testsamplesize` drawn from $\mathbf{x} \in \{-1,1\}^n$ are used to approximate the generalization error given a specific training set: the generalization error is calculated for each distinct test sample and then an average is calculated over the number of `testsamples`.

Secondly, given the number of dimensions n and the training observations m, the above is in turn repeated for a specified number of `trainsamples` (pseudo) randomly drawn from $\boldsymbol{S} \in \{-1,1\}^{n^m}$ and the generalization error then further averaged over these different training sets to estimate the expected generalization error.

Holding the number of dimensions n constant, the above two steps are repeated for an increasing number of training observations m until the expected generalization error is equal to or falls below the specified threshold value of 10%. The number of observations m for which this first happens is then used as the estimate for the sample complexity of the classification algorithm at hand.

ii. Why we did it

The problem of calculating the expected generalization error and the sample complexity exactly very quickly becomes computationally intractable. The visual intuition is that we are learning a hypercube of dimension n. Half of the points on the hypercube will have a label of 1 and the other half a label of -1.

Clearly this hypercube will have 2^n points. So even at dimension $n = 100$ it becomes impossible to check the algorithm against every point's classification, let alone train the algorithm using every one of the possible 2^{n^m} training sets.

We are thus reliant upon approximation and we choose to do so through (pseudo) random sampling. The issue with this approach is clearly that for increasing number of dimensions n and observations m, the training sample employed and the points used to test upon are a small and exponentially decreasing proportion of, respectively, the possible training sets and test points. This is the curse of dimensionality in practice.

In order not to have to rely on what could be unusual and / or unrepresentative single samples, we decided to draw a considerable amount of (pseudo) random samples for both the training sets (of size m) and of test points (of specified size). We believed that the average over these samples provide us with a reasonably unbiased as well as accurate estimate of the (expected) generalization error and thus the sample complexity.

We chose the number of training and test set samples as well as the size of the latter based on two considerations: (i) the accuracy of at least the test set sampling as compared to the full test set generated up to 25 dimensions (see above) and (ii) the limits of our computational power.

Nevertheless, we are aware that no matter how many training samples and samples of test points of specified size we are computationally able to work with, we will never be given a guarantee that – at high dimensionality – our multiple samples are in fact representative over the entire sets of possible training samples and test points.

For other problems this caveat of our approximation approach would be a considerable issue. However, the "just a little bit" problem appears to have an approximation accommodating structure and we are pleasantly surprised by the clarity and stability of our test results. The graphs showed consistency as dimensions n increased and on repeated runs, thus suggesting that the inevitably small proportion of the training and test spaces we were able to explore seemed to give representative results.

Part c.

Perceptron algorithm: $m = O(n)$

Winnow algorithm: $m = O(\log(n))$

Least squares algorithm: $m = O(n)$

1-nearest neighbour algorithm: $m = O(\exp(n))$

The sample complexity of both the perceptron and least squares algorithm appear to grow linearly as a function of n as $n \to \infty$. However, given the setup at hand, the approximated linear coefficient α associated with the perceptron's sample complexity growth appears to be higher than that of the least squares algorithm. For the former $\alpha \cong 1.2$ and for the latter $\alpha \cong 0.6$.

The winnow algorithm's sample complexity growth, on the other hand, seems to be logarithmic in n as $n \to \infty$. This finding was supported by simulation up to $n = 100$ dimensions.

Lastly, the 1-nearest neighbour's sample complexity increases exponentially as a function of n as $n \to \infty$. Up to $n = 15$ dimensions, this result was corroborated through simulation.

The reason sample complexity m increases with the number of dimensions n is due to the curse of dimensionality. In short, this means that as the number of (irrelevant) dimensions n increases, the sample size m is effectively reduced. Hence, the number of observations m required to obtain an adequate generalization error – i.e. separate the relevant signal from the noise – grows rapidly as the number of (irrelevant) dimensions n increase.

In conclusion, it appears that for the "just a little bit" problem at hand, the classification algorithms may be ordered in decreasing level of performance (in terms of sample complexity as the number of dimensions increase) as follows: winnow, least squares, perceptron and 1-nearest neighbour.

Part d.

Clearly, a good lower bound on the computational complexity of the 1-nearest neighbour algorithm is related to the golden ratio – ubiquitous and hidden in all sorts of places. Therefore, $f(n) = \varphi^n$, where φ is the golden ratio $\frac{1+\sqrt{5}}{2}$.

Appendix A – MATLAB script PartIRegression.m for Part I, Exercise 1 to Exercise 7

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 GI01: Supervised Learning - Coursework 1                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          Part I - Linear regression                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Program to perform various regressions to evaluate effect of training
% set size, dimensionality and regularisation parameter as well as to
% compare different regularisation parameter tuning methods
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialisation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
clc; % Clear command window
clear all; % Clear all variables etc.
close all; % Close all figures / windows
s = RandStream('mt19937ar', 'Seed', 1); % Create stream of random numbers
RandStream.setGlobalStream(s); % Set stream as global stream
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
nobs = 600; % Set number of observations
ntrain_a = 100; % Set size of training set
ntrain_b = 10; % Set alternative size of training set
ntest = 500; % Set size of test set
ntrials = 200; % Set number of trials
ndim_a = 1; % Set number of dimensions
ndim_b = 10; % Set alternative number of dimensions
gamma = logspace(-6, 3, 10); % Set range of regularisation parameter
factortrainsplit = 5; % Set factor for split of training set
crossvalfolds = 5; % Set number of folds for cross-validation
%
% Break if number of folds for cross-validation is larger than factor for
% split of training set
%
if crossvalfolds > factortrainsplit;
    break;
end
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 1 (LSR - effect of the training set size)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. and b. LSR - training set n = 100, 1 dimension
%
[X, y] = createdata (nobs, ndim_a);
[msetrain_a_1, msetest_a_1] = regression (X, y, ntrain_a, ntest);
%
% c. LSR - training set n = 10, 1 dimension
%
[msetrain_b_1, msetest_b_1] = regression (X, y, ntrain_b, ntest);
%
% d. LSR - training sets n = 100 and n = 10, 1 dimension, 200 trials
%
[msetrain_a_1d, msetest_a_1d, msetrain_b_1d, msetest_b_1d] = deal([]);
%
for i = 1 : ntrials;
    [Xloop, yloop] = createdata (nobs, ndim_a);
    [msetrain_a_1d(i), msetest_a_1d(i)] = regression ...
        (Xloop, yloop, ntrain_a, ntest);
    [msetrain_b_1d(i), msetest_b_1d(i)] = regression ...
        (Xloop, yloop, ntrain_b, ntest);
end;
%
res_1d = [mean(msetrain_a_1d), mean(msetest_a_1d); ...
    mean(msetrain_b_1d), mean(msetest_b_1d)];
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 2 (LSR - effect of the dimensionality)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. and b. LSR - training set n = 100, 10 dimensions
%
[X2, y2] = createdata (nobs, ndim_b);
[msetrain_a_2, msetest_a_2] = regression (X2, y2, ntrain_a, ntest);
%
% c. LSR - training set n = 10, 10 dimensions
%
[msetrain_b_2, msetest_b_2] = regression (X2, y2, ntrain_b, ntest);
%
% d. LSR - training sets n = 100 and n = 10, 10 dimensions, 200 trials
%
[msetrain_a_2d, msetest_a_2d, msetrain_b_2d, msetest_b_2d] = deal([]);
%
for i = 1 : ntrials;
    [Xloop, yloop] = createdata (nobs, ndim_b);
    [msetrain_a_2d(i), msetest_a_2d(i)] = regression ...
        (Xloop, yloop, ntrain_a, ntest);
    [msetrain_b_2d(i), msetest_b_2d(i)] = regression ...
        (Xloop, yloop, ntrain_b, ntest);
end;
%
res_2d = [mean(msetrain_a_2d), mean(msetest_a_2d); ...
    mean(msetrain_b_2d), mean(msetest_b_2d)];
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 4 (Effect of the regularisation parameter)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. RR - training set n = 100, 10 dimensions, gamma = 10^6 : 10^3
%
[msetrain_a_4, msetest_a_4] = regression (X2, y2, ntrain_a, ntest, gamma);
%
% b. RR - training set n = 10, 10 dimensions, gamma = 10^6 : 10^3
%
[msetrain_b_4, msetest_b_4] = regression (X2, y2, ntrain_b, ntest, gamma);
%
figure;
loglog(gamma, msetrain_a_4, 'bo-', gamma, msetest_a_4, 'bx-', gamma, ...
    msetrain_b_4, 'ro-', gamma, msetest_b_4, 'rx-');
title('log(Mean Squared Error) vs. log(Gamma), 1 Trial')
ylabel('log(Mean Squared Error)')
xlabel('log(Gamma)')
legend('Training Set n = 100: Training', 'Training Set n = 100: Testing', ...
    'Training Set n = 10: Training', 'Training Set n = 10: Testing', ...
    'Location', 'southeast')
%
% c. RR - training sets n = 100 and n = 10, 10 dimensions,
% gamma = 10^6 : 10^3, 200 trials
%
[msetrain_a_4c, msetest_a_4c, msetrain_b_4c, msetest_b_4c] = ...
    deal([]);
%
for i = 1 : ntrials;
    [Xloop, yloop] = createdata (nobs, ndim_b);
    [msetrain_a_4c(i, :), msetest_a_4c(i, :)] = regression ...
        (Xloop, yloop, ntrain_a, ntest, gamma);
    [msetrain_b_4c(i, :), msetest_b_4c(i, :)] = regression ...
        (Xloop, yloop, ntrain_b, ntest, gamma);
end;
%
res_4c = [mean(msetrain_a_4c); mean(msetest_a_4c); ...
    mean(msetrain_b_4c); mean(msetest_b_4c)];
%
figure;
loglog(gamma, res_4c(1, :), 'bo-', gamma, res_4c(2, :), 'bx-', ...
    gamma, res_4c(3, :), 'ro-', gamma, res_4c(4, :), 'rx-');
title('log(Average Mean Squared Error) vs. log(Gamma), 200 Trials')
ylabel('log(Average Mean Squared Error)')
xlabel('log(Gamma)')
legend('Training Set n = 100: Training', 'Training Set n = 100: Testing', ...
    'Training Set n = 10: Training', 'Training Set n = 10: Testing', ...
    'Location', 'southeast')
%
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 5 (Tuning the regularisation parameter using a validation set)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. RR - training set n = 100, 10 dimensions, gamma = 10^6 : 10^3,
% training set split factor = 5
%
[msetrain_a_5, msetest_a_5, mseval_a_5, ~, minmsetrain_a_5, minmsetest_a_5] ...
    = regression (X2, y2, ntrain_a, ntest, gamma, factortrainsplit);
%
% b. RR - training set n = 10, 10 dimensions, gamma = 10^6 : 10^3,
% training set split factor = 5
%
[msetrain_b_5, msetest_b_5, mseval_b_5, ~, minmsetrain_b_5, minmsetest_b_5] ...
    = regression (X2, y2, ntrain_b, ntest, gamma, factortrainsplit);
%
% a. and b. RR - training sets n = 100 and n = 10, 10 dimensions,
% gamma = 10^6 : 10^3, training set split factor = 5, 200 trials
%
[msetrain_a_5ab, msetest_a_5ab, mseval_a_5ab, mingammaval_a_5ab, ...
    msetrain_b_5ab, msetest_b_5ab, mseval_b_5ab, mingammaval_b_5ab] = ...
    deal([]);
%
for i = 1 : ntrials;
    [Xloop, yloop] = createdata (nobs, ndim_b);
    [msetrain_a_5ab(i, :), msetest_a_5ab(i, :), mseval_a_5ab(i, :), ...
        mingammaval_a_5ab(i)] = regression ...
        (Xloop, yloop, ntrain_a, ntest, gamma, factortrainsplit);
    [msetrain_b_5ab(i, :), msetest_b_5ab(i, :), mseval_b_5ab(i, :), ...
        mingammaval_b_5ab(i)] = regression ...
        (Xloop, yloop, ntrain_b, ntest, gamma, factortrainsplit);
end;
%
res_a_5ab = [mean(msetrain_a_5ab); mean(msetest_a_5ab); mean(mseval_a_5ab)];
%
figure;
loglog(gamma, res_a_5ab(3, :), 'kx-', gamma, res_a_5ab(1, :), 'bo-', ...
    gamma, res_a_5ab(2, :), 'bx-');
title('log(Average Mean Squared Error) vs. log(Gamma), Traing Set n = 100, 200
Trials')
ylabel('log(Average Mean Squared Error)')
xlabel('log(Gamma)')
legend('Validation', 'Training', 'Testing', 'Location', 'southeast')
%
res_b_5ab = [mean(msetrain_b_5ab); mean(msetest_b_5ab); mean(mseval_b_5ab)];
%
figure;
loglog(gamma, res_b_5ab(3, :), 'kx-', gamma, res_b_5ab(1, :), 'ro-', ...
    gamma, res_b_5ab(2, :), 'rx-');
title('log(Average Mean Squared Error) vs. log(Gamma), Training Set n = 10, 200
Trials')
ylabel('log(Average Mean Squared Error)')
xlabel('log(Gamma)')
legend('Validation', 'Training', 'Testing', 'Location', 'southeast')
%
% c. Average regularisation parameter - training sets n = 100 and n = 10
%
meangamma_5ab = [mean(mingammaval_a_5ab), mean(mingammaval_b_5ab)];
%
% d. RR - training sets n = 100 and n = 10, 1 dimension, gamma = 10^6 : 10^3,
% training set split factor = 5, 200 trials
%
[msetrain_a_5d, msetest_a_5d, mseval_a_5d, mingammaval_a_5d, ...
    msetrain_b_5d, msetest_b_5d, mseval_b_5d, mingammaval_b_5d] = ...
    deal([]);
%
for i = 1 : ntrials;
    [Xloop, yloop] = createdata (nobs, ndim_a);
    [msetrain_a_5d(i, :), msetest_a_5d(i, :), mseval_a_5d(i, :), ...
        mingammaval_a_5d(i)] = regression (Xloop, yloop, ntrain_a, ntest, ...
        gamma, factortrainsplit);
    [msetrain_b_5d(i, :), msetest_b_5d(i, :), mseval_b_5d(i, :), ...
        mingammaval_b_5d(i)] = regression (Xloop, yloop, ntrain_b, ntest, ...
        gamma, factortrainsplit);
end;
%
meangamma_5d = [mean(mingammaval_a_5d), mean(mingammaval_b_5d)];
%
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 6 (Tuning the regularisation parameter using cross-validation)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. RR - training set n = 100, 10 dimensions, gamma = 10^6 : 10^3,
% training set split factor = 5, cross-validation folds = 5
%
[msetrain_a_6, msetest_a_6, msecrossval_a_6] = regression ...
    (X2, y2, ntrain_a, ntest, gamma, factortrainsplit, crossvalfolds);
%
figure;
loglog(gamma, msecrossval_a_6, 'kx-', gamma, msetrain_a_6, 'bo-', ...
    gamma, msetest_a_6, 'bx-');
title('log(Mean Squared Error) vs. log(Gamma), Traing Set n = 100, 1 Trial')
ylabel('log(Mean Squared Error)')
xlabel('log(Gamma)')
legend('Cross-Validation', 'Training', 'Testing', 'Location', 'southeast')
%
% b. RR - training set n = 10, 10 dimensions, gamma = 10^6 : 10^3,
% training set split factor = 5, cross-validation folds = 5
%
[msetrain_b_6, msetest_b_6, msecrossval_b_6] = regression ...
    (X2, y2, ntrain_b, ntest, gamma, factortrainsplit, crossvalfolds);
%
figure;
loglog(gamma, msecrossval_b_6, 'kx-', gamma, msetrain_b_6, 'ro-', ...
    gamma, msetest_b_6, 'rx-');
title('log(Mean Squared Error) vs. log(Gamma), Traing Set n = 10, 1 Trial')
ylabel('log(Mean Squared Error)')
xlabel('log(Gamma)')
legend('Cross-Validation', 'Training', 'Testing', 'Location', 'southeast')
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 7 (Comparison of regularisation parameter tuning methods)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% RR - training sets n = 100 and n = 10, 10 dimension, gamma = 10^6 : 10^3,
% training set split factor = 5, cross-validation folds = 5, 200 trials
%
[msetest_train_a_7, msetest_train_b_7, msetest_val_a_7, ...
    msetest_val_b_7, msetest_crossval_a_7, msetest_crossval_b_7] ...
    = deal([]);
%
for i = 1 : ntrials;
    %
    % Create data
    [Xloop, yloop] = createdata (nobs, ndim_b);
    %
    % Minimize training error
    [~, ~, ~, ~, ~, msetest_train_a_7(i, :)] = regression ...
        (Xloop, yloop, ntrain_a, ntest, gamma);
    [~, ~, ~, ~, ~, msetest_train_b_7(i, :)] = regression ...
        (Xloop, yloop, ntrain_b, ntest, gamma);
    %
    % Minimize validation error
    [~, ~, ~, ~, ~, msetest_val_a_7(i, :)] = regression ...
        (Xloop, yloop, ntrain_a, ntest, gamma, factortrainsplit);
    [~, ~, ~, ~, ~, msetest_val_b_7(i, :)] = regression ...
        (Xloop, yloop, ntrain_b, ntest, gamma, factortrainsplit);
    %
    % Minimize 5-fold cross validation error
    [~, ~, ~, ~, ~, msetest_crossval_a_7(i, :)] = regression (Xloop, ...
        yloop, ntrain_a, ntest, gamma, factortrainsplit, crossvalfolds);
    [~, ~, ~, ~, ~, msetest_crossval_b_7(i, :)] = regression (Xloop, ...
        yloop, ntrain_b, ntest, gamma, factortrainsplit, crossvalfolds);
    %
end;
%
res_a_7 = [mean(msetest_train_a_7), mean(msetest_val_a_7), ...
    mean(msetest_crossval_a_7); std(msetest_train_a_7), ...
    std(msetest_val_a_7), std(msetest_crossval_a_7)];
%
res_b_7 = [mean(msetest_train_b_7), mean(msetest_val_b_7), ...
    mean(msetest_crossval_b_7); std(msetest_train_b_7), ...
    std(msetest_val_b_7), std(msetest_crossval_b_7)];
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to create data set y = X * w + n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [X, y] = createdata (nobs, ndim)
%
% Create data set
w = randn(ndim, 1); % Create weight vector with ndim rows
X = randn(nobs, ndim); % Create a nobs * ndim matrix X of random draws from
% standard normal distribution
n = randn(nobs, 1); % Create noise vector with nobs rows
y = (X * w) + n; % Calculate y as vector with nobs rows
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Versatile regression function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [msetrain, msetest, meanmsecrossval, mingammacrossval, ...
    minmsetrain, minmsetest] = regression ...
    (X, y, ntrain, ntest, gamma, factortrainsplit, crossvalfolds)
%
% Set defaults for variables gamma, factortrainsplit and crossvalfolds
if ~exist('gamma', 'var')
    gamma = 0;
end
%
if ~exist('factortrainsplit', 'var')
    factortrainsplit = 1;
end
%
if ~exist('crossvalfolds', 'var')
    crossvalfolds = 1;
end
%
% Split data into training and test sets
[Xtrain, Xtest] = deal(X(1 : ntrain, :), X(ntrain + 1 : ntrain + ntest, :));
[ytrain, ytest] = deal(y(1 : ntrain, :), y(ntrain + 1 : ntrain + ntest, :));
%
% Set indices for cross-validation
crossvalinds = kron(1 : factortrainsplit, ...
    ones(1, (ntrain / factortrainsplit)));
%
% Create empty matrix for efficiency
[msetrain, msetest, msecrossval] = deal([]);
%
% Perform regression
%
% Regularization
for j = 1 : length(gamma);
    %
    % Cross-validation
    for k = 1 : crossvalfolds;
        testind = (crossvalinds == k);
        if sum(testind) == ntrain;
            trainind = testind;
        else
            trainind = ~testind;
        end;
        %
        % Calculate w*
        wstrcv = ((Xtrain(trainind, :)' * Xtrain(trainind, :)) + ...
            (gamma(j) * size(Xtrain(trainind, :), 1) .* ...
            eye(size(Xtrain(trainind, :), 2)))) \ ...
            (Xtrain(trainind, :)' * ytrain(trainind, :));
        %
        % Calculate MSE cross-validation
        msecrossval(j, k) = ((wstrcv' * (Xtrain(testind, :)' * ...
            Xtrain(testind, :)) * wstrcv) - (2 * ytrain(testind, :)' * ...
            Xtrain(testind, :) * wstrcv) + (ytrain(testind, :)' * ...
            ytrain(testind, :))) / sum(testind);
        %
```

```
    end;
    %
    % Calculate MSE training and test
    wstr = ((Xtrain' * Xtrain) + (gamma(j) * size(Xtrain, 1) .* ...
            eye(size(Xtrain, 2)))) \ (Xtrain' * ytrain);
    msetrain(j) = ((wstr' * (Xtrain' * Xtrain) * wstr) - ...
        (2 * ytrain' * Xtrain * wstr) + (ytrain' * ytrain)) / ntrain;
    msetest(j) = ((wstr' * (Xtest' * Xtest) * wstr) - ...
        (2 * ytest' * Xtest * wstr) + (ytest' * ytest)) / ntest;
end;
%
% Calculate average MSE cross-validation
meanmsecrossval = mean(msecrossval, 2);
%
% Identify regularization parameter that minimizes MSE cross-validation
if gamma == 0;
    minindmeancrossval = 1;
else
[~, minindmeancrossval] = min(meanmsecrossval);
mingammacrossval = gamma(minindmeancrossval);
end;
%
% Identify associated MSE training and test
minmsetrain = msetrain(minindmeancrossval);
minmsetest = msetest(minindmeancrossval);
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix B – MATLAB script `PartIKernel.m` for Part I, Exercise 8 to Exercise 10

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 GI01: Supervised Learning - Coursework 1                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    Part I - Boston housing and kernels                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Program to perform and compare various regressions, incl. Gaussian kernel
% ridge regression, on exemplary Boston housing (prices) data set
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialisation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
clc; % Clear command window
clear all; % Clear all variables etc.
close all; % Close all figures / windows
s = RandStream('mt19937ar','Seed',1); % Create stream of random numbers
RandStream.setGlobalStream(s); % Set stream as global stream
load('boston.mat'); % Load data from file boston.mat
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
nobs = length(boston); % Save number of observations
ndim = size(boston, 2) - 1; % Save number of dimensions
X = boston(:, 1 : ndim); % Save data along dimensions for all observations
y = boston(:, ndim + 1); % Save responses
traintestsplit = (2 / 3); % Set split of X for training set
factortrainsplit = 5; % Set factor for split of training set
crossvalfolds = 5; % Set number of folds for cross validation
ntrain = round((nobs * traintestsplit / factortrainsplit)) * factortrainsplit;
% Set size of training set
ntest = nobs - ntrain; % Set size of test set
ntrials = 20; % Set number of trials
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 9 (Baseline versus full linear regression)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. Naive regression
%
X_9a = ones(nobs, 1);
%
[msetrain_9a, msetest_9a] = deal([]);
%
for n = 1 : ntrials;
    [msetrain_9a(n), msetest_9a(n)] = regression9 ...
        (X_9a, y, nobs, ntrain, ntest);
end;
%
% b. Linear regression with single attribute
%
[msetrain_9b, msetest_9b] = deal([]);
%
% For every dimension
for i = 1 : ndim;
    Xloop = [ones(nobs, 1), X(:, i)];
    for n = 1 : ntrials;
        [msetrain_9b(n, i), msetest_9b(n, i)] = regression9 ...
            (Xloop, y, nobs, ntrain, ntest);
    end;
end;
%
% c. Linear regression with all attributes
%
[msetrain_9c, msetest_9c] = deal([]);
%
```

```
for n = 1 : ntrials;
    [msetrain_9c(n), msetest_9c(n)] = regression9 ...
        (X, y, nobs, ntrain, ntest);
end;
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exercise 10 (Kernel ridge regression)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% a. Create kridgereg.m
%
% See function kridgereg.m
%
% b. Create dualcost.m
%
% See function dualcost.m
%
% c. Kernel ridge regression
%
powersgamma = -40 : 1 : -26; % Set powers for range of gamma
gamma = repmat(2, 1, length(powersgamma)).^powersgamma; % Set range of gamma
powerssig = 7 : 0.5 : 13; % Set powers for range of sigma
sig = repmat(2, 1, length(powerssig)).^powerssig; % Set range of sigma
%
[msetrain_10c, msetest_10c, meanmsecrossvaltest_10c, ...
    indmingamma_10c, indminsig_10c] = kregression ...
    (X, y, nobs, ntrain, gamma, sig, factortrainsplit, crossvalfolds);
%
figure;
imagesc(log(sig), log(gamma), log(meanmsecrossvaltest_10c).^-1);
colormap hot;
hold on;
plot(log(sig(indminsig_10c)), log(gamma(indmingamma_10c)), 'ro');
title('1 / log(Mean Squared Error) vs. log(Gamma) and log(Sigma)');
zlabel('1 / log(Mean Squared Error)');
ylabel('log(Gamma)');
xlabel('log(Sigma)');
legend('Minimum Cross-Validation Error', 'Location', 'southeast')
%
% d. Kernel ridge regression, 20 trials with random training / test splits
%
[msetrain_10d, msetest_10d, indmingamma_10d, indminsig_10d] = deal([]);
%
for n = 1 : ntrials;
    [msetrain_10d(n), msetest_10d(n), ~, indmingamma_10d(n), ...
        indminsig_10d(n)] = kregression ...
        (X, y, nobs, ntrain, gamma, sig, factortrainsplit, crossvalfolds);
end;
%
res(:,:,1) = [mean(msetrain_9a), std(msetrain_9a); ...
    [mean(msetrain_9b); std(msetrain_9b)]'; ...
    [mean(msetrain_9c); std(msetrain_9c)]'; ...
    [mean(msetrain_10d); std(msetrain_10d)]'];
res(:,:,2) = [mean(msetest_9a), std(msetest_9a); ...
    [mean(msetest_9b); std(msetest_9b)]'; ...
    [mean(msetest_9c); std(msetest_9c)]'; ...
    [mean(msetest_10d); std(msetest_10d)]'];
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to perform simple regression with random train / test split
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [msetrain, msetest] = regression9 (X, y, nobs, ntrain, ntest)
%
% Create indices for random split of training and test sets
randind = randperm(nobs);
%
% Split data into training and test sets
[ytrain, ytest] = deal(y(randind(1 : ntrain), :), ...
    y(randind(ntrain + 1 : nobs), :));
[Xtrain, Xtest] = deal(X(randind(1 : ntrain), :), ...
    X(randind(ntrain + 1 : nobs), :));
%
```

```
% Calculate w*
wstr = (Xtrain' * Xtrain) \ (Xtrain' * ytrain);
%
% Calculate MSE train
msetrain = ((wstr' * (Xtrain' * Xtrain) * wstr) - ...
    (2 * ytrain' * Xtrain * wstr) + (ytrain' * ytrain)) / ntrain;
%
% Calculate MSE test
msetest = ((wstr' * (Xtest' * Xtest) * wstr) - ...
    (2 * ytest' * Xtest * wstr) + (ytest' * ytest)) / ntest;
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to perform kernel ridge regression
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [minmsetrain, minmsetest, meanmsecrossval, indmingamma, indminsig] ...
    = kregression (X, y, nobs, ntrain, gamma, sig, factortrainsplit, crossvalfolds)
%
% Create indices for random split of training and test sets
randind = randperm(nobs);
%
% Set indices for cross-validation
crossvalinds = kron(1 : factortrainsplit, ...
    ones(1, round((ntrain / factortrainsplit))));
%
% Create empty matrix for efficiency
msecrossval = deal([]);
%
% Perform kernel ridge regression
%
% Sigma tuning
for i = 1 : length(sig);
    %
    % Calculate kernel matrix
    kX = kmatrix(X, sig(i));
    %
    % Randomly split y and kX into training set
    ytrain = y(randind(1 : ntrain));
    kXtrain = kX(randind(1 : ntrain), randind(1 : ntrain));
    %
    % Regularization
    for j = 1 : length(gamma);
        %
        % Cross-validation
        for k = 1 : crossvalfolds;
            testind = (crossvalinds == k);
            if sum(testind) == ntrain;
                trainind = testind;
            else
                trainind = ~testind;
            end;
            %
            % Calculate dual weight vector
            adualw = kridgereg (kXtrain(trainind, trainind), ...
                ytrain(trainind), gamma(j));
            %
            % Calculate MSE cross-validation
            msecrossval(j, i, k) = dualcost (kXtrain(testind, ...
                trainind), ytrain(testind), adualw);
            %
        end;
    end;
end;
%
% Calculate average of MSE cross-validation
meanmsecrossval = mean(msecrossval, 3);
%
% Identify regularization parameter and sigma that minimize
% MSE cross-validation
[~, indminmeanmsecrossval] = min(meanmsecrossval(:));
[indmingamma, indminsig] = ind2sub(size(meanmsecrossval), ...
    indminmeanmsecrossval);
```

```
%
% Calculate MSE train / test using regularization parameter and sigma that
% minimize MSE cross-validation
%
% Calculate kernel matrix for minimizing sigma
kX = kmatrix(X, sig(indminsig));
%
% Randomly split y and X into training and test set
[ytrain, ytest] = deal(y(randind(1 : ntrain)), ...
    y(randind(ntrain + 1 : nobs)));
[kXtrain, kXtest] = deal(kX(randind(1 : ntrain), randind(1 : ntrain)), ...
    kX(randind(ntrain + 1 : nobs), randind(1 : ntrain)));
%
% Calculate dual weight vector
adualw = kridgereg (kXtrain, ytrain, gamma(indmingamma));
%
% Calculate MSE train
minmsetrain = dualcost (kXtrain, ytrain, adualw);
%
% Calculate MSE test
minmsetest = dualcost (kXtest, ytest, adualw);
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to calculate kernel matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [kX] = kmatrix (X, sigma)
%
kX = []; % Create empty matrix for efficiency of for loop
for i = 1 : size(X, 1); % For every row of X
    for j = 1 : size(X, 1); % For every row of X
        % Calculate kernel matrix
        kX(i, j) = exp(-norm(X(i,:) - X(j,:)) ^ 2 / (2 * sigma ^ 2));
    end;
end;
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to calculate dual weight vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [adualw] = kridgereg (K, y, gamma)
%
l = size(K, 1); % Determine length of kernel matrix
adualw = (K + (gamma * l * eye(l))) \ y; % Calculate dual weight vector
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to calculate mean squared error given dual weight vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [mse] = dualcost (K, y, adualw)
%
l = size(K, 1); % Determine length of kernel matrix
mse = (1 / l) * (K * adualw - y)' * (K * adualw - y); % Calculate MSE
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix C – MATLAB script `PartIISparseLearning.m` for Part II, Question 4

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 GI01: Supervised Learning - Coursework 1                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Part II - Question 4 - Sparse Learning                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Program to compute sample complexity for the 4 following classification
% algorithms: perceptron, winnow, least squares, 1-nearest neighbour
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialisation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
clc; % Clear command window
clear all; % Clear all variables etc.
close all; % Close all figures / windows
s = RandStream('mt19937ar','Seed',1); % Create stream of random numbers
RandStream.setGlobalStream(s); % Set stream as global stream
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
maxn = 100; % Set maximum number of dimensions
complimitn = 10; % Set limit on dimensions for exact computation of test set
Xtestmaster = dec2bin(0 : 2^complimitn - 1) - '0'; % Master test set
Xtestmaster_0 = Xtestmaster; % Save in separate variable {0, 1} ^ complimitn
Xtestmaster(Xtestmaster == 0) = -1; % Master test set {-1, 1} ^ complimitn
break
trainsamples = 100; % Set number of samples of training sets
testsamples = 100; % Set number of samples of test sets
testsamplesize = 100; % Set size of test set
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Pereceptron
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
m_perceptron = samplecomplexity ('perceptron', maxn, complimitn, ...
    Xtestmaster, trainsamples, testsamples, testsamplesize);
%
figure;
plot(1 : maxn, m_perceptron, 'rx-')
title('"Just a little bit": Perceptron Algorithm');
ylabel('Approximated Sample Complexity (m)');
xlabel('Number of Dimensions (n)');
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Winnow
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
m_winnow = samplecomplexity ('winnow', maxn, complimitn, ...
    Xtestmaster_0, trainsamples, testsamples, testsamplesize);
%
figure;
plot(1 : maxn, m_winnow, 'bx-')
title('"Just a little bit": Winnow Algorithm');
ylabel('Approximated Sample Complexity (m)');
xlabel('Number of Dimensions (n)');
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Least squares
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
m_leastsquares = samplecomplexity ('leastsquares', maxn, complimitn, ...
    Xtestmaster, trainsamples, testsamples, testsamplesize);
%
```

```
figure;
plot(1 : maxn, m_leastsquares, 'gx-')
title('"Just a little bit": Least Squares Algorithm');
ylabel('Approximated Sample Complexity (m)');
xlabel('Number of Dimensions (n)');
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1-nearest neighbor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
m_onenn = samplecomplexity ('onenn', maxn, complimitn, ...
    Xtestmaster, trainsamples, testsamples, testsamplesize);
%
m_onenn = [m_onenn, NaN(1, maxn - complimitn)];
figure;
plot(1 : maxn, m_onenn, 'kx-')
title('"Just a little bit": 1-Nearest Neighbour Algorithm');
ylabel('Approximated Sample Complexity (m)');
xlabel('Number of Dimensions (n)');
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Method comparison
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Up to computational limit for master test set / 1-nearest neighbour
figure;
semilogy(1 : complimitn, m_perceptron(1 : complimitn), 'rx-', ...
    1 : complimitn, m_winnow(1 : complimitn), 'bx-', ...
     1 : complimitn, m_leastsquares(1 : complimitn), 'gx-', ...
     1 : complimitn, m_onenn(1 : complimitn), 'kx-')
title('"Just a little bit": Algorithm Comparison');
ylabel('log(Approximated Sample Complexity (m))');
xlabel('Number of Dimensions (n)');
legend('Perceptron', 'Winnow', 'Least Squares', '1-Nearest Neighbour', ...
    'Location', 'northwest')
%
% Up to maximum number of dimensions
figure;
plot(1 : maxn, m_perceptron, 'rx-', 1 : maxn, m_winnow, 'bx-', ...
     1 : maxn, m_leastsquares, 'gx-', 1 : maxn, m_onenn, 'kx-')
title('"Just a little bit": Algorithm Comparison');
ylabel('Approximated Sample Complexity (m)');
xlabel('Number of Dimensions (n)');
legend('Perceptron', 'Winnow', 'Least Squares', '1-Nearest Neighbour', ...
    'Location', 'northeast')
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Algorithm dependent function to compute sample complexity defined as
% minimum number of observations (m) to incur no more than 10 percent
% generalisation error given number of dimensions (n)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [m] = samplecomplexity (method, maxn, complimitn, Xtestmaster, ...
    trainsamples, testsamples, testsamplesize)
%
% Reset maximum number of dimensions for 1-nearest neighbour
if strcmp(method, 'onenn');
    maxn = complimitn;
end;
%
% Create empty vectors for efficiency
[m, gerr, gerrtemp] = deal([]);
%
% For number of dimensions up to maxn
for n = 1 : maxn;
    %
    % Initialisation
    m(n) = 0; % Number of observations
    meangerr = 1; % Mean generalisation error
    %
```

```
% If all test points computationally feasible
if n <= complimitn;
    %
    % Save relevant subset of master test set
    Xtest = Xtestmaster(1 : 2^n, complimitn - (n - 1) : complimitn);
    ytest = Xtest(:, 1);
    %
    % While mean generalisation error is above 10 percent
    while meangerr > 0.1;
        %
        % Add another observation
        m(n) = m(n) + 1;
        %
        % Sample training sets to approximate expected generalisation error
        for train = 1 : trainsamples;
            %
            % Create training set
            [Xtrain, ytrain] = createdata(method, m(n), n);
            %
            % Compute weight vector / distances
            switch method;
                case 'perceptron';
                    w = perceptron(Xtrain, ytrain);
                case 'winnow';
                    w = winnow(Xtrain, ytrain);
                case 'leastsquares';
                    w = pinv(Xtrain) * ytrain;
                case 'onenn';
                   [~, w] = pdist2(Xtrain, Xtest, ...
                        'euclidean', 'Smallest', 1);
            end;
            %
            % Compute generalisation error for training sample
            gerr(train) = generalisationerror ...
                (method, ytrain, w, Xtest, ytest, n, 2^n);
            %
        end;
        % Compute mean generalisation error over training samples
        meangerr = mean(gerr);
        %
    end;
    %
% If all test points computationally infeasible, sample test sets
% to approximate full test set
else
    % While mean generalisation error is above 10 percent
    while meangerr > 0.1;
        %
        % Add another observation
        m(n) = m(n) + 1;
        %
        % Sample training sets to approximate expected generalisation error
        for train = 1 : trainsamples;
            %
            % Create training set
            [Xtrain, ytrain] = createdata(method, m(n), n);
            %
            % Compute weight vector
            switch method;
                case 'perceptron';
                    w = perceptron(Xtrain, ytrain);
                case 'winnow';
                    w = winnow(Xtrain, ytrain);
                case 'leastsquares';
                    w = pinv(Xtrain) * ytrain;
            end;
            %
            % Sample test sets to approximate full test set
            for test = 1 : testsamples;
                %
                % Create test set
                [Xtest, ytest] = createdata(method, testsamplesize, n);
                %
                % Compute distances
                if strcmp(method, 'onenn');
                    [~, w] = pdist2(Xtrain, Xtest, ...
                        'euclidean', 'Smallest', 1);
                end;
                %
```

```
                    % Compute generalisation error for test sample
                    gerrtemp(test) = generalisationerror ...
                        (method, ytrain, w, Xtest, ytest, n, testsamplesize);
                    %
                end;
                %
                % Compute generalisation error over test samples for train sample
                gerr(train) = mean(gerrtemp);
                %
            end;
            %
            % Compute mean generalisation error over train samples
            meangerr = mean(gerr);
            %
        end;
    end;
end;
%
return;
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Algorithm dependent function to create m by n data set X and associated y
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [X, y] = createdata (method, m, n)
%
switch method;
    case {'perceptron', 'leastsquares', 'onenn'};
        X = 2 * randi(2, m, n) - 3; % {-1, 1}
    case 'winnow';
        X = randi(2, m, n) - 1; % {0, 1}
end;
%
y = X(:, 1); % y_i = x_i(1)
%
return;
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to execute perceptron classification algorithm
% Inputs: m by n data set X (domain = {-1, 1}) and associated labels y
% Output: weight vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [w] = perceptron(X, y)
%
% Initialisation
maxits = 1000000000; % Set maximum number of iterations
its = 0; % Number of iterations
[m, n] = size(X); % Get number of observations and dimensions
w = zeros(n, 1); % Set weight vector equal to zero as per definition
wrong_un = m; % Set number of wrong classifications to m
%
% Iterate while wrong classifications > 0 and iterations < max. iterations
while (wrong_un > 0) && (its < maxits);
    %
    % Set number of wrong classifications to 0
    wrong_un = 0;
    %
    % For every observation
    for i = 1 : m;
        %
        % If misclassification for current observation
        if ((X(i, :) * w) * y(i) <= 0);
            wrong_un = wrong_un + 1;
            w = w + (y(i) * X(i, :))'; % Recompute weight
        end;
        its = its + 1;
    end;
```

```
end;
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to execute winnow classification algorithm
% Inputs: m by n data set X (domain = {0, 1}) and associated labels y
% Output: weight vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [w] = winnow(X, y)
%
% Initialisation
maxits = 1000000000; % Set maximum number of iterations
its = 0; % Number of iterations
[m, n] = size(X); % Get number of observations and dimensions
w = ones(n, 1); % Set weight vector equal to zero as per definition
wrong_un = m; % Set number of wrong classifications to m
%
% Iterate while wrong classifications > 0 and iterations < max. iterations
while (wrong_un > 0) && (its < maxits);
    %
    % Set number of wrong classifications to 0
    wrong_un = 0;
    %
    % For every observation
    for i = 1 : m;
        %
        % Receieve pattern
        if (X(i, :) * w) < n;
            yhat = 0;
        else
            yhat = 1;
        end;
        %
        % If prediction does not equal to label
        if yhat ~= y(i);
            wrong_un = wrong_un + 1;
            w = w .* (2 .^ ((y(i) - yhat) * X(i, :)))'; % Recompute weight
        end;
        its = its + 1;
    end;
end;
%
return
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Algorithm dependent function to calculate generalisation error
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
function [gerr] = generalisationerror ...
    (method, ytrain, w, Xtest, ytest, n, base)
%
switch method;
    case {'perceptron', 'leastsquares'};
        gerr = sum(sign(Xtest * w) ~= ytest) / base;
    case 'winnow';
        gerr = sum((Xtest * w >= n) ~= ytest) / base;
    case 'onenn';
        gerr = sum(ytrain(w, :) ~= ytest) / base;
end;
%
return;
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```