Supervised Learning
Coursework 1

Raza Rizvi, Klaudia Ludwisiak
16100325, 16117698

January 16, 2017

Exercise 1

For the following question I have created multiple MATLAB functions to avoid repetition and improve readability of my code. These functions have been listed below:

1. **createRandomData.m** - *Creates random data and splits it into training and test sets defined by user inputs.*
2. **estimate_w.m** - *Calculates optimal weights given training data.*
3. **myMSE.m** - *calculates the Mean Square Error. This function first calculates the predicted y and then the MSE.*

Part(a)

* Please see 'Section 1: Part a' of Exercise1.m for the code.

I used the function createRandomData.m to generate the required dataset as well as splitting the data into the correct training and subsets. The function takes the inputs of dataset size, number of dimensions and training set size. The function then outputs the results in cell arrays. So I unpacked the results into variables which are easier to use.

Part (b)

* Please see Section 2: Part b of Exercise1.m for the code.

I created functions *estimate_w.m* and *myMSE.m* to answer this question.

Training MSE:	0.8316
Test MSE:	0.951

Part (c)

* Please see Section 3: Part c of Exercise1.m for the code.

Since I had already created the necessary functions; to answers this question, I simply changed input arguments.

Training MSE:	0.666
Test MSE:	1.453

As one would expect with such a small training set, the test error has increased in comparison with part (b).

Part (d)

* Please see Section 4: Part d for Exercise1.m for the code.

-	10	100
Training MSE:	0.8595	1.004
Test MSE:	1.1355	1.0114

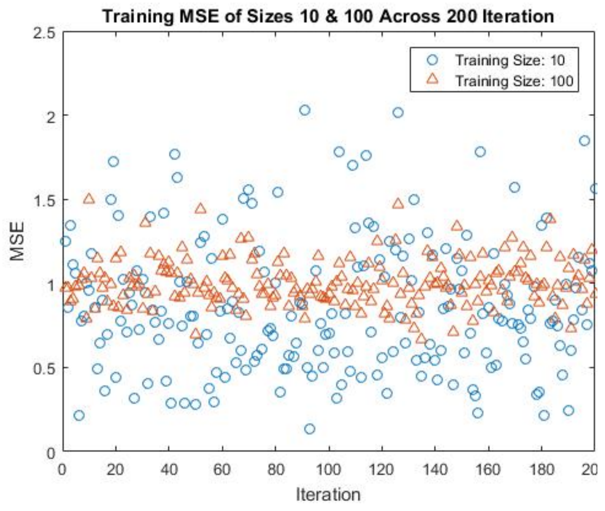


Figure 1: Training MSE for size 10 & 100

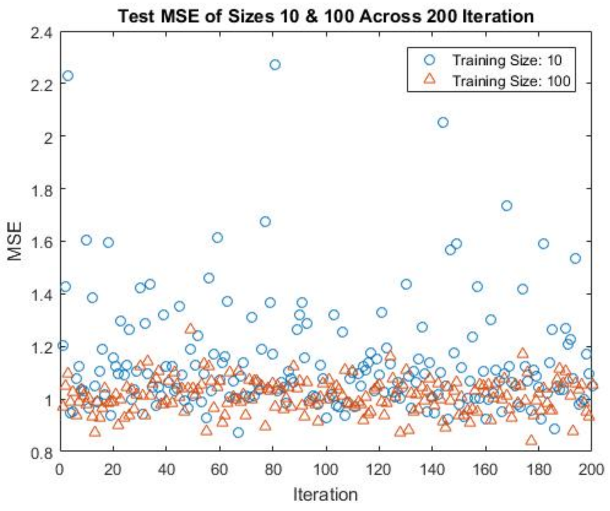


Figure 2: Test MSE for size 10 & 100

The table above gives the results of this experiment. From here it is clear that the test MSE is much higher for a training set size of 10 than 100. A small training set does not contain enough information to accurately explain the underlying data behaviour. With such few points the model 'overfits' the training data and as we would expect the test error becomes larger.

The training MSE is lower for size 10. This is because of with such few points, a line that fits the data is likely to be closer to all the points then if there were many more identically distributed points.

The average MSE's by themselves do not fully describe the problem with a small training set. The figures above show how the training and test MSE change over 200 runs. It is clear that with a small training set the spread of the MSE is high. This shows that not only does a larger training set give a better model but it also provides a more stable solution.

Exercise 2

I have reused the functions I created for Exercise 1 throughout this question.

Part (a)

* Please see Section 1: Part a of Exercise2.m

Again, for this question I used my MATLAB function *createRandomData.m* and used the inputs $600, 10, 100$ for dataset size, number of dimensions and training set size respectively.

Part (b)

Repeat (b) from Ex1:

Training MSE:	0.9159
Test MSE:	1.0610

Repeat (c) from Ex1:

Training MSE:	2.73×10^{-28}
Test MSE:	177.87

Repeat (d) from Ex1:

-	10	100
Training MSE:	9.26×10^{-23}	0.9072
Test MSE:	1027.6	1.122

Part (c)

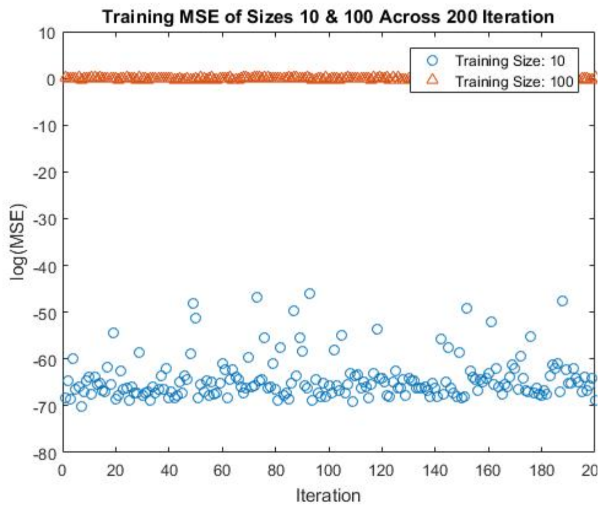


Figure 3: Training MSE for size 10 & 100

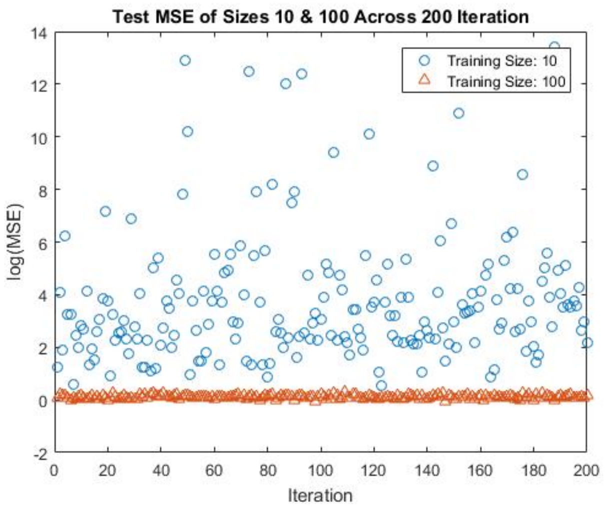


Figure 4: Test MSE for size 10 & 100

For comparison I've included the same graphs as in question 1 but with a log scale. It becomes very clear what affect a smaller training set can have. For training set of size 10 the spread of the Test MSE is unacceptably large by any standard this is due to over-fitting of the training set.

Exercise 3

3.1

Prove that solving the optimisation problem (6) yields:

$$w^* = (X'X + \gamma lI)^{-1}X'y$$

Where equation (6) is,

$$w_* = argmin_w \underbrace{\gamma w'w + \frac{1}{l}\sum_{i=1}^{l}(x_i'w = y_i)^2}$$

Using the standard matrix notation $X = (x_1, x_2, ..., x_l)'$ we can rewrite the the portion of the equation after the argmin (emphasized by the underbrace above).

$$\begin{aligned} &\gamma w'w + \frac{1}{l}(Xw - y)'(Xw - y) \\ &= l\gamma w'w + w'X'Xw - 2X'y + y'y \end{aligned}$$

Taking derivatives w.r.t. w and setting it equal to zero,

$$\begin{aligned} 2l\gamma w^* + 2X'Xw^* - 2X'y &= 0 \\ l\gamma w^* + X'Xw^* &= X'y \\ w^*(X'X + \gamma lI) &= X'y \\ w^* &= (X'X + \gamma lI)^{-1}X'y \end{aligned}$$

Of course, this solution is only valid if $X'X + \gamma lI$ is positive definite, which is proved below.

3.1

Prove that $X'X + \gamma lI$ is positive definite matrix

Since we have already shown the equivalence between the complexity function, $w_* = argmin_w \gamma w'w + \frac{1}{l}\sum_{i=1}^{l}(x_i'w = y_i)^2$ and $w^* = (X'X + \gamma lI)^{-1}X'y$. We can see that the convexity of the complexity function implies that $X'X + \gamma lI$ is positive definite.

However, it is easier to show this through through the properties of the matrix. A $n \times n$ symmetric matrix M is said to be positive definite if for every non zero column vector z we have,

$$z'Mz > 0$$

Therefore,

$$z'(X'X + \gamma lI)z = (Xz)'(Xz) + \gamma z'z \geq \gamma ||z||_2^2 > 0 \tag{1}$$

The inequality holds since $X'X$ is positive semi-definite and z is non zero.

Exercise 4

Please note: In this question I have used λ instead of γ for the regularization parameter. Since I have used this notation in the code I have decided to keep it here.

For this question we have reused functions that we created for question 1 and 2. We have also created a new function *ridgesolution.m* which takes as inputs: a vector of observations, a data set and lambda, then outputs w^*. Using functions throughout this code means that much of the code for the following questions are identical barring inputs to the function. At times we sacrificed marginal improvements in speed to improve readability of code.
Function used for ridge regression questions:

1. **createRandomData.m** - *Creates random data and splits it into training and test sets defined by user inputs.*

2. **ridgesolution.m** - *Calculates optimal weights given training data, observations and λ.*

3. **myMSE.m** - *calculates the Mean Square Error. This function first calculates the predicted y and then the MSE.*

(a)

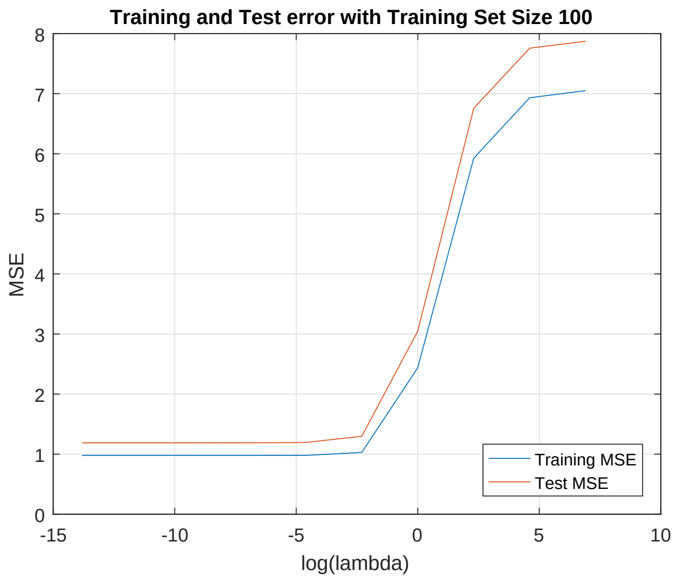


Figure 5: Test MSE for size 10 & 100

Part (a) uses 600 data points with a training set of size 100. The required MSE's are in the table below.

Table 1: **Training set size 100:** Training and Test MSE for each lambda

λ	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{-0}	10^{1}	10^{2}	10^{3}
Training MSE	0.885	0.885	0.885	0.885	0.887	1.012	4.596	13.719	16.364	16.675
Test MSE	1.106	1.106	1.106	1.105	1.105	1.225	4.832	13.393	15.785	16.065

(b)

Part (b) uses a training set of size 10 and a test set of size 500. Results are given in the table below.

Table 2: **Training set size 10:** Training and Test MSE for each lambda

λ	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{-0}	10^{1}	10^{2}	10^{3}
Training MSE	1.34E-08	1.33E-06	1.23E-04	0.0063	0.0486	0.173	1.777	9.65	13.40	13.90
Test MSE	47.41	47.12	44.39	28.40	9.584	4.431	4.245	6.912	8.313	8.502

We decided to include a graph (figure 6) of log(MSE) to show that the training MSE increases as lambda increases. Without taking logs (figure 7) it appears that the training MSE is constant for a range of lambdas. We can see here that the behaviour of the MSE's is remarkably different when compared to a larger training set. When the regularization parameter is small, the penalty

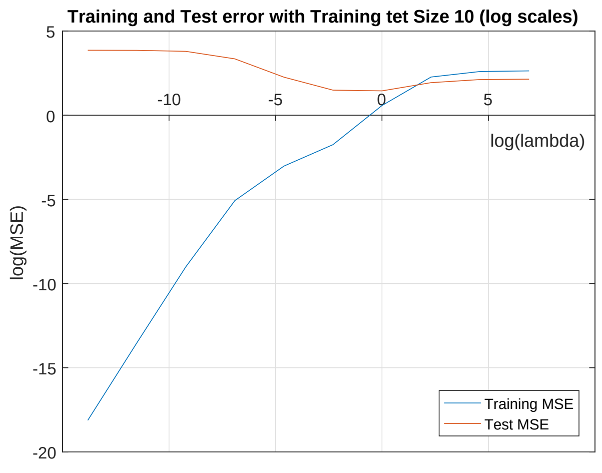


Figure 6: Training and Test log(MSE)

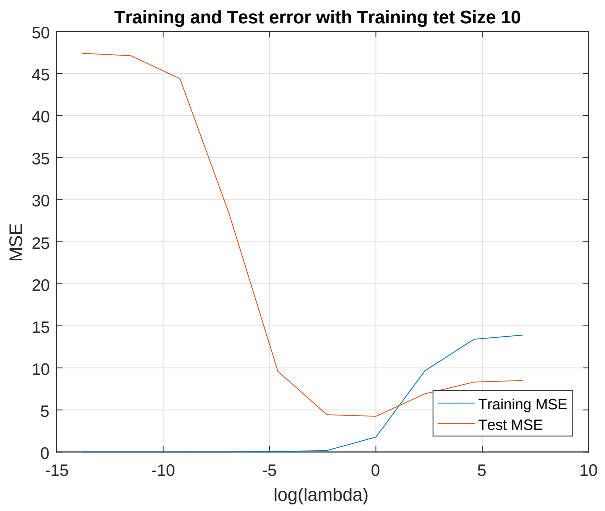


Figure 7: Training and Test MSE

on the weights is small. Clearly for this small training data set a larger regularization parameter - i.e. a larger penalty for larger weights gives better results. However as this is only one run, it is hard to make any concrete statements which values of the regularization parameter are the best.

(c)

Table 3: **Training Set 100** - Average MSE over 200 runs

λ	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{-0}	10^{1}	10^{2}	10^{3}
Training MSE	0.912	0.912	0.912	0.912	0.914	1.005	3.456	9.484	11.247	11.456
Test MSE	1.105	1.105	1.105	1.105	1.104	1.197	3.823	9.638	11.234	11.421

Table 4: **Training Set 10** - Average MSE over 200 runs

λ	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{-0}	10^{1}	10^{2}	10^{3}
Training MSE	0.002	0.006	0.013	0.027	0.065	0.282	2.234	8.384	11.382	11.803
Test MSE	1665	333	63.1	17.4	5.88	3.741	5.757	9.653	11.063	11.248

From the table above we can see that the regularization parameter which leads to the lowest test error for training set size of 100 is $\lambda = 10^{-2}$, whereas the regularization parameter which leads to the lowest test error with training set size of 10 is $\lambda = 10^{-1}$. Comparing these results to question 2 (regression without regularization) we can we that if we set λ to it's optimal value, for training set size 100, the performance is similar. Without the regularization parameter we have a test error of 1.122 and with the optimal λ we have 1.104. However, for the training set of size 10, we can see a real improvement in the regression performance. The test error falls from 1027 without the regularization parameter to 3.741 with the parameter set to the optimal value. The table below summarizes these results.

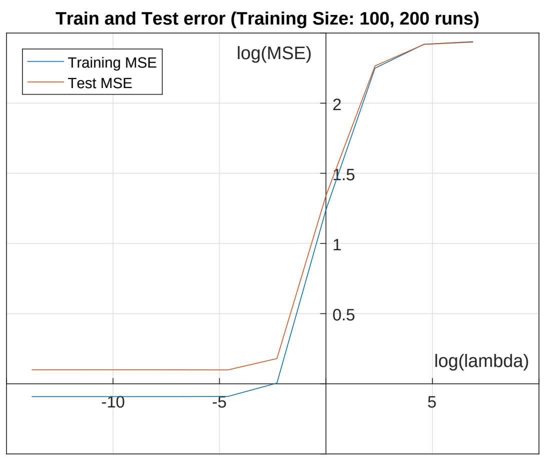


Figure 8: 4(c):Training and Test log(MSE)

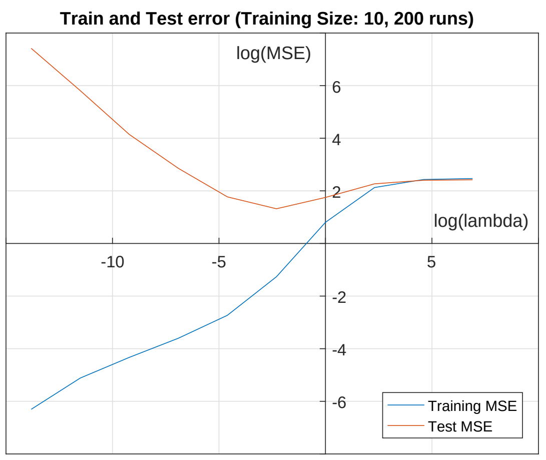


Figure 9: 4(c): Training and Test log(MSE)

Table 5: Summary of test errors with and without the regularization parameter, λ

	Training size 100		Training size 10	
	No λ	Optimal λ	No λ	Optimal λ
Test error	1.1220	1.1040	1027.6000	3.7410

From our result, we can be certain that the training error is not a good indicator of how well the model will perform on the test set. It is clear that to find the optimal lambda, without using the test set, we must introduce a validation set and find which lambda gives lowest validation error. Once we have found this lambda we then have a trained model which will perform much better than if we had just minimised on the error on the training set.

Question 5

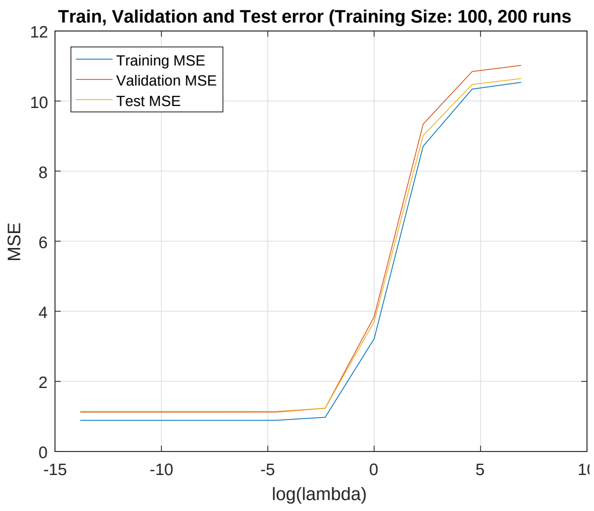
In this exercise we have used many of the function created previously as well as introducing a new function: datasplit.m

1. **datasplit.m** - *Take as inputs a matrix and a scaler and then outputs two matrices split by the scaler. This function has been used throughout question 5 to split the training set into training and validation sets.*

Average test error: 1.134
The graph in figure 10 shows that the validation error is very similar to the test set error for each λ. This shows choosing a λ that minimises validation error will give a good result for a λ which minimises the test set error.

(b)

Average test error: 14.57
Which such a small training set the benefits of ridge regression with a validation set can be realised.



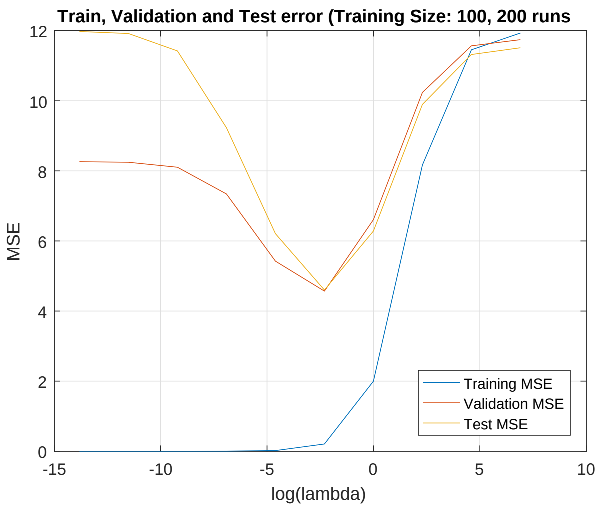


Figure 10: 5(a):Training, Validation,Test MSE

Figure 11: 5(b): Training, Validation, Test MSE

We can see that despite the validation set being of size 2, it follows the behaviour of the test error. In particular, for the average of the 200 iterations, the validation error and the test error both achieve their minimums for the same lambda (figure 11).

(c)

Please note: *I have been calling the regularization parameter λ instead of γ, which is the term used in the question. For this question I will use γ since it was explicitly asked.*

Table 6: Average values for γ

	True value	approx	mode
γ^{10}	107.2288	$10^{2.0303}$	10^{-1}
γ^{100}	0.0370	$10^{-1.4984}$	10^{-6}

Table 6 summarises the data for the average γ. We can see that γ^{100} is smaller than γ^{10}. If we look at the graphs in figure 10 and 11 we can see that neither of the average γ's are at the lowest point for the validation error. We believe part of the reason why this is happening is because the average is the wrong statistic to take as it is influenced by extreme values. The standard deviation for γ for a training set size of 10 is 313.41. Taking the average will not give a good representation of how γ behaves across the 200 iterations. Using the mode, i.e out of the 200 iterations which γ was most often the lowest, gives a better approximation of the correct γs. We can see the modal γ's are closer to the minimum average validation error on the graph.

However this does explain why $\gamma^{100} < \gamma^{10}$. The reason for this is that training on small amounts of data will lead to over-fitting causing high validation error (and very low training error). The regularization parameter restricts weights such that large weights are penalised. For a small dataset, having a higher penalty will result in less over-fitting and thus lower validation error.

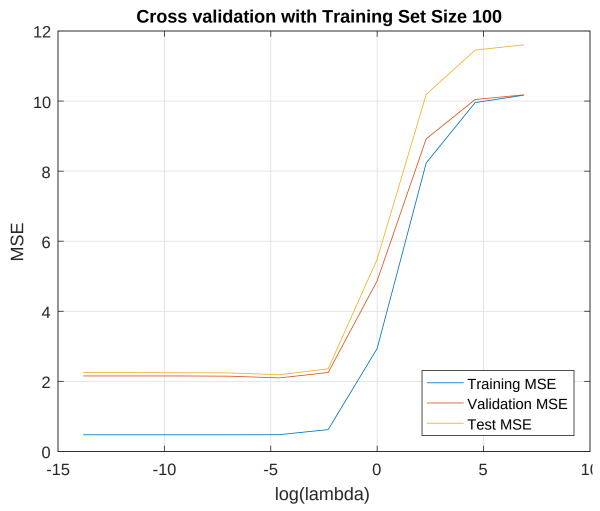


Figure 12: 6(a) Training size 100

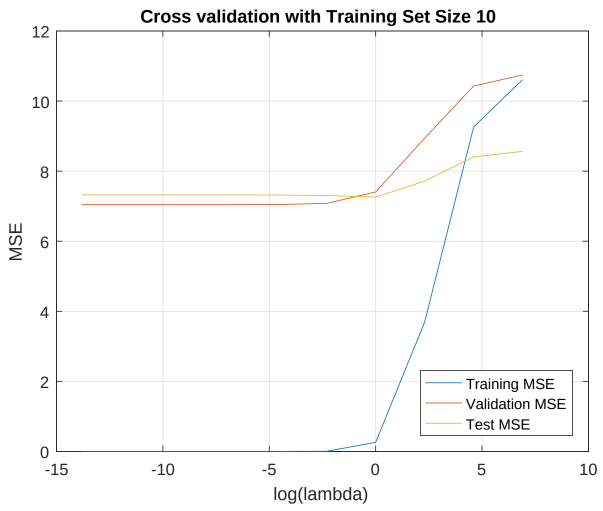


Figure 13: 6(b): Training size 10

(d)

The results repeating this question on exercise 1 data are given in table 7.

Table 7: Results for 5(d)

	100	10
Avg Test Error	1.038	1.241
Avg Lambda	85.923	241.6

Exercise 6

For this Exercise I did not use any additional functions. However I will briefly outline how I implemented the cross validation to help the reader understand the code.

Using the randomdata.m function I create a training matrix, $\mathbf{X}$ of size say (100×10). I then take the first fifth of the matrix (i.e rows 1 to 20) and set it equal to the validation matrix, **vali** (size: 20×10). The remaining rows (rows 21 to 100) are set equal to the 'new training matrix', **new_X**. I then perform my ridge regression on the training set and calculate the validation error. Once this is done I set rows 1 to 80 of $\mathbf{X}$ equal the training matrix,**new_X**, and rows 81 to 100 to the validation matrix, **vali** . That is, the original training matrix becomes $\mathbf{X}$ = **new_X**; **vali**]. I then repeat this process 4 more times. Using this method means the first 20 rows which become the new training matrix were rows 41 to 60 of $\mathbf{X}$ in the previous iteration. This performs a valid cross validation since the data is random.

(a) - (b)

The requested graphs can be seen in figure 12 and 13. Since this question did not ask to calculate the test error with the optimal λ did not include this in the code.

Exercise 7

The results are summarised in the table below.

Table 8: Summary of test errors using different methods of selecting regularization parameter

	Training size: 100		Training size: 10	
	Mean	STD	Mean	STD
Minimising Training Error	1.108	0.0816	615.01	2980.4
Minimising Validation Error	1.153	0.1408	33.11	165.61
Cross Validation	1.107	0.0876	5.535	5.571

From the table above we can see that the for a training set size of 100 the three methods produce similar results. Minimising the validation error without cross validation causes a slightly higher standard deviation then the other methods. This is likely due to having a smaller training and validation set. In fact the table above suggest that with a training set size of 100, minimising the training error produces sufficient results.

The real power of introducing the validation set and cross validation can be seen through the training set of size 10. We can see that the standard deviation falls by a factor of 10 when choosing minimising on the validation set. When using cross validation the average test error falls to single digits.

Section 3

In this section we apply kernel methods and linear regression to the Boston housing data set. In this classic dataset we are given 506 data instances of 14 variables (detailed in table below). The last column (14) describes the median value of owner-occupied homes in \$1000's, which we will set as our dependent variable and predict its value basis the remaining 13 variables (regressors).

variable	description
1. CRIM	per capita crime rate by town
2. ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS	proportion of non-retail business acres per town
4. CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX	nitric oxides concentration (parts per 10 million)
6. RM	average number of rooms per dwelling
7. AGE	proportion of owner-occupied units built prior to 1940
8. DIS	weighted distances to five Boston employment centres
9. RAD	ndex of accessibility to radial highways
10. TAX	full-value property-tax rate per $10,000
11. PTRATIO	pupil-teacher ratio by town
12. BK	proportion of blacks by town
13. LSTAT	% lower status of the population
14. MEDV	Median value of owner-occupied homes in $1000's

Exercise 8 and 9

In Exercise 8 we have successfully downloaded the Boston dataset. When implementing the code, this data set should be located in the current folder together with other associated files.

In Exercise 9 we examined the prediction performance of linear regression models of varying complexity and evaluated this using the mean squared error (MSE). In Part 1 we performed Naive regression using only the dependent variable and a bias term. In Part 2 we run Linear Regression (LR) with each of the 13 regressors one at a time, also including a bias term. In part 3 we performed Linear regression again but this time with all 13 regressors and a bias included in the model. In all Parts we trained on 2/3 of the data set and tested on 1/3 of the data. The table below summarizes our results.

Model	MSE train	MSE test
Part 1 - Mean value	85.482	82.461
Part 2 - LR, 1 variable - min MSE	38.115	39.606
Part 2 - LR, 1 variable - max MSE	81.635	82.897
Part 2 - LR, 1 variable - mean MSE	66.252	68.752
Part 3 - All variables	22.068	22.939

Unsurprisingly, the Naive regression used in Part 1 yields the highest MSE values as model is too simplistic and does not make use of the additional information available. Using Linear Regression (LR) in part 2 yields improved results as compared with part 1. The table above displays not only the mean MSE across the 13 regression models fitted one at a time but also the best (min MSE) and worst (max MSE) variables. Minimum MSE is reached for for var 13 indicating that the % lower status of the population (LSTAT) is the single most important determinant of the median house price in Boston. The second lowest error is achieved by variable 6; average number of rooms per dwelling (RM). Indicating that if we wanted to use LR model with only a few regressors we should focus on attribute 13 first, followed by attribute 6 and so on in ascending order of the MSE values. Max MSE is reached for attribute 4 , thus this should be used last. As opposed to the other attributes, attribute 4 (Charles River dummy variable,) is categorical and this explains why using the simple LR model yields highest MSE values. The last row of the table above depicts the much improved prediction results achieved when using a robust LR model, including all 13 attributes.

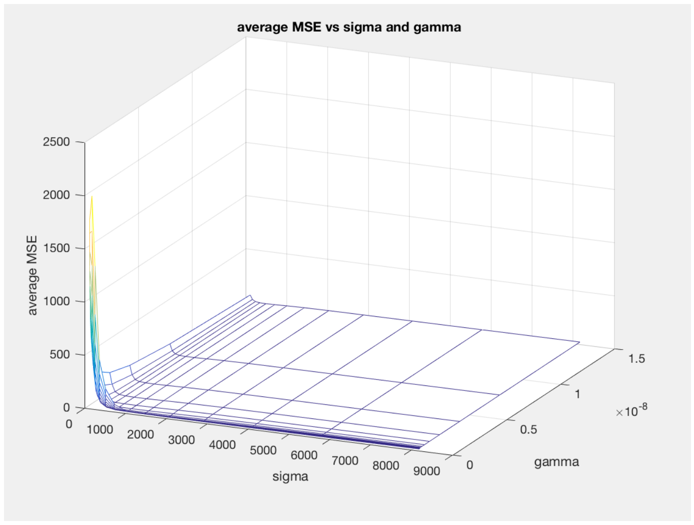


Figure 14: Exercise 10(a)

This improvement, is expected as now the full extent of available information is used for prediction. For further detail of results refer to the table in Exercise 10.

Exercise 10

In This exercise we performed Gaussian Kernel Ridge Regression (KRR) on the Boston dataset and compared with LR results obtained in exercise 9. As before, we trained our model on 2/3 of data and tested on the remaining 1/3.By nature, linear regressions models are limited when predicting on non-linear data sets and Kernels can be used to mitigate this. Applying a Kernel function (here the Gaussian Kernel) to the non-linear data set 'lifts' the analysis into a higher dimension feature space where a linear regression model can be fitted and will amount to a non-linear regression function when related back to the original input space.

In Part A we create a function called kridgereg.m to perform kernel ridge regression, this can be found in Appendix. Functions - Boston. The function takes as inputs the kernel matrix K, y vector, and scalar ridge parameter γ and return a vector α of dual weights. Left division was used in Matlab as it is computationally significantly less expensive then the Matalb inverse function. The relevant equations are:

Kernel Matrix element : K(xi,xj) = exp(-(‖xi−xj‖^2)/ 2σ2)

Dual weight vector: α∗ = ((K + γlI(l))^−1)y

This implementation correctly computes alpha since we are making use of duality and the kernel trick. The dual formulation of the weight vector expressed is : α∗ = ((XX' + γlI(l))^−1)y. Additionally, we observe that the kernel matrix is equivalent to the inner product of the non-linear mappings of the data into the feature space (K=XX'). Thus, the dual weight vector can be calculated using the kernel matrix and this is known as the kernel trick.

In Part B we create a function called dualcost.m to calculate the Mean Squared Error (MSE) as per equation: (1/l)(Ktestα − y)´(Ktestα − y). Thus, the function takes the kernel matrix K, y vector, and dual weight vector α as inputs and returns the corresponding MSE.

In part C we brought together functions created in Parts A and B to implement KRR on the Boston data set. The model was run for a range of ridge regression parameters γ and variance parameters (σ) used to construct the gaussian kernel. Cross-validation with 5 folds was also used to obtain average MSE results over the 5 folds, yielding a more robust solution. In Part C(1) the average MSE was plotted agains the different γ and σ parameters as can be seen on the figure below. The bets parameters γ and σ were identified as those, which minimized the test MSE. Best gamma was found at 2^-40 and bets sigma at 2^12.5. The corresponding Mean test MSE was 13.04 and on the training MSE was 7.75. The low result on the training MSE is hardly surprising as this was used to build the model.

In part D we repeated parts A to C 20 times to get average results over variations of the shuffled data. The corresponding mean average test MSE was 13.043 (standard deviation 1.6386). A detailed summary of all the results including those from Exercise 9, can be seen in the table below. It should also be noted that the 'best' γ and σ parameter values were not always consistent over the 20 iterations. In most of the cases minimum MSE was indeed achieved at γ = 2^-40 and σ = 2^12.5. However a few instances of different 'best' parameters existed. It can be observed that mean MSE values and corresponding standard deviations are in all cases generally lower in the training set as compared to the training set. The lower error results on the training set are expected as this were used to construct and adjust the model. In a few instances, the MES on the test set is actually lower than that on the train set, however this is compensated by the much higher standard deviation on the results. Signifying overall worse predicion than on the training set. It can also be seen that the error reduces as the complexity of the model increases, with highest errors achieved in Exercise 9 part 1 when applying Naive Regression and best results achieved by the kernel ridge regression implemented in Exercise 10. KRR generally performed almost twice as well as the robust LR applied in Exercise 9 part 3.

Model	MSE train	MSE test
Ex.9 Part 1 - Mean value	85.48 ± 4.71	82.46 ± 9.49
Ex.9 Part 2 - LR, variable 1	70.00 ± 5.63	76.22 ± 11.32
Ex.9 Part 2 - LR, variable 2	74.333 ± 4.3687	72.15 ± 8.77
Ex.9 Part 2 - LR, variable 3	63.61 ± 4.7913	67.21 ± 9.49
Ex.9 Part 2 - LR, variable 4	81.64 ± 5.87	82.90 ± 12.00
Ex.9 Part 2 - LR, variable 5	70.08 ± 4.16	67.21 ± 8.14
Ex.9 Part 2 - LR, variable 6	43.40 ± 4.73	44.60 ± 9.72
Ex.9 Part 2 - LR, variable 7	70.90 ± 4.50	75.80 ± 9.18
Ex.9 Part 2 - LR, variable 8	78.20 ± 5.33	81.41 ± 10.84
Ex.9 Part 2 - LR, variable 9	69.64 ± 6.07	77.44 ± 12.13
Ex.9 Part 2 - LR, variable 10	65.72 ± 4.93	66.82 ± 9.70
Ex.9 Part 2 - LR, variable 11	62.26 ± 3.26	63.86 ± 6.70
Ex.9 Part 2 - LR, variable 12	73.45 ± 3.96	78.55 ± 7.94
Ex.9 Part 2 - LR, variable 13	38.12 ± 2.80	39.61 ±5.64
Ex.9 Part 3 - All variables	22.07 ± 1.74	22.94 ±3.73
Ex.10 KRR - All variables	7.75 ± 1.40	13.04 ± 1.64

Part II

1.

How should one choose beta so that the learned linear classifier simulates a 1-Nearest Neighbour classifier? Explain your reasoning.

Kernel measures the similarity between points. In fact we can consider the Gaussian Kernel as a covariance matrix since all it's values are between 0 and 1. Explicitly, this means that for a $m \times n$ training matrix we get a $n \times n$ (symmetric) covariance matrix, K. This means the, $K_{i,j}$ is the covariance of the $i^{th}\&j^{th}$ data point ($K_{i,j} \in [0, 1]$). The covariance of two points is defined by the distance between two points, as seen from the equation below.

$$K^2(x, t) = exp(-\beta||x - t||^2)$$

If $x\&t$ are close than $||x - t|| \to 0$ and therefore $exp(-\beta||x - t||^2) \to 1$. Similarly if x and t and far apart then $||x - t|| \to \infty$ and therefore $exp(-\beta||x - t||^2) \to 0$. The role of β is as the length-scale parameter. As beta gets larger the covariance of points that are close together approaches zero. Therefore to mimic the 1-nearest-neighbour we must chose a suitably large β such that only points in the immediate vicinity of a data points have a non-zero entry in the covariance matrix.

Choosing the β to simulate the 1-Nearest-Neighbour algorithm will also be highly dependent on the density of the data. If the data points are every dense (i.e. many points are close together) then beta must be set large so that multiple points in each row of the covariance matrix do not have non-zero value (other than points on the diagonal).

In an ideal scenario, we would chose a β such that only one value (other than the 1's of the diagonal) are non-zero for each row. This would mean that for each data point only one other point is considered when training.

2

I used John-Shawe Taylors book 'Support Vector Machines' as a reference throughout this question (Chapter 2.1, page 14).

(a)

The update for the bias is:

$$b_t = b_{t-1} + y_i R^2$$

Where $R = max_{1<i<l}||x_i||$ This can be seen if we rewrite x and w as,

$$\hat{x_i} = \begin{bmatrix} x_i \\ R \end{bmatrix}, \hat{w_t} = \begin{bmatrix} w_t \\ b/R \end{bmatrix}$$

and so the new update becomes,

$$\hat{w}_{t+1} = \hat{w}_t + y_i\hat{x}_i \implies \begin{bmatrix} w_{t+1} \\ b_{t+1}/R \end{bmatrix} = \begin{bmatrix} w_t \\ b_t/R \end{bmatrix} + y_i \begin{bmatrix} x_i \\ R \end{bmatrix}$$

and therefore our updates become,

$$w_{t+1} = w_t + y_i xi$$

$$b_{t+1}/R = b_t/R + y_i R \implies b_{t+1} = b_t + y_i R^2$$

(b)

Incorporating the bias term will **increase** Novikoff bound. If we are using the bias update as I have describe in the part (a), the bound will increase by a factor of 4. This proof is outlined in detail in John-Shawe Taylors book, support Vector Machines and other Kernel-based method (page 14). Since the question did not ask to state a proof to why the bound will increase, I will avoid repeating his work here.

Initially, this proof does seem counter-intuitive; one would expect that introducing a new parameter would cause the bound the fall.

3

(a)

Define the kernel $K_0(x,x) = \sum_{i=1}^{n} x_i x_i = XX^T$. Any matrix of the form XX^T is positive semi-definite. This means means we can rewrite the kernel matrix as,

$$K_c(x,x) = C + XX^T$$

Where C is a matrix of size XX^T with each entry equal to c. For any positive vector a we have,

$$K_c(x,x) = a^T(C + XX^T)a = a^T Ca + \underbrace{a^T XX^T a}_{\geq 0}$$

From the above we can see that the kernel will be semi positive definite if c$\geq$ 0.

(b)

c does not actually influence the correctness of the solution. The inner product of the kernel moves that data into a different dimension, whereas the addition of the scalar c will only translate the data (without any other transformation). That is, if the correct solution and be achieved without the scalar addition, the addition of the scaler will not impact the solution.

4

Exercise 4 Part A -'Just a little bit problem'

(a) - (b)

In this problem we consider the sample complexity of the perceptron, winnow, least squares, and 1-nearest neighbors algorithms for a given data set. The data (X) is sampled uniformly at random from $\{-1, 1\}$^n and has dimensions of m by n where m are the data instances and n are columns of features. The labels (Y) are chose as the 1st column of X, so for each data instance the label is the first X value. For each of the 4 algorithms we find the sample complexity and corresponding generalization error on a data set of varying size. The sample complexity is he minimum number of examples (m) to incur no more than 10% generalization error. The results are summarized by plots of the mean generalization error for different values on m and n (below). For further detail about the algorithm implementation refer to the published code and functions in the sections that follow.

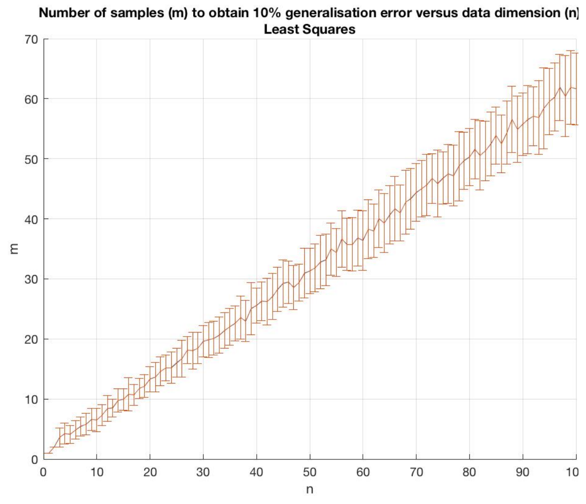


Figure 15: Linear Regression

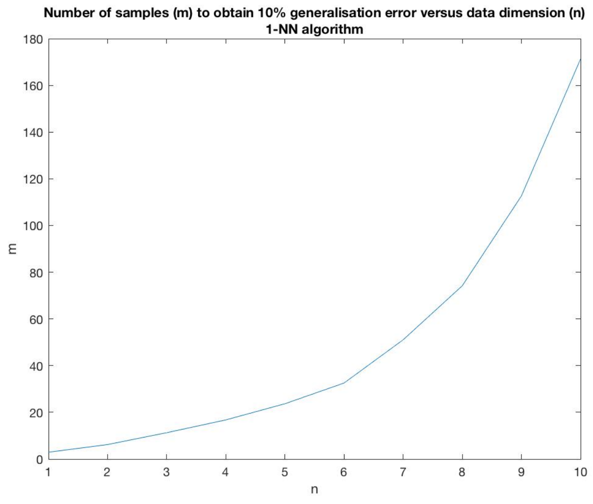


Figure 16: 1-NN

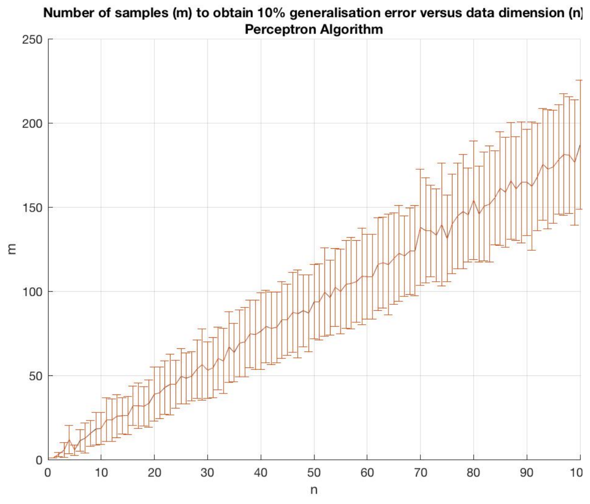


Figure 17: Perceptron

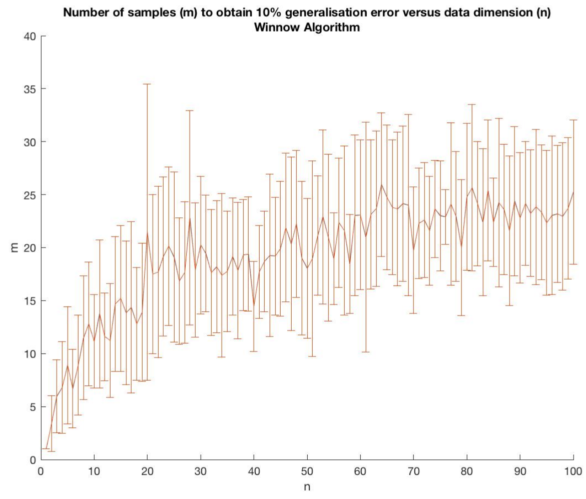


Figure 18: winnow

Inspecting the runtime and the amount of data instances needed to train the model (converge to a generalization error below 10%), it is evident that the 4 algorithms have different performance. Looking at the case when n=100; winnow gives the best performance in terms of the amount of data instances needed to train the model, Least Squares follows with roughly double the amount of points required, this is then followed by the perceptron, whereas 1-NN performs significantly worse. In fact k-NN is an algorithm which runs in polynomial time and it is impossible to obtain a graph for n=100 as this data size is too large. Therefore, the included graph is one for n=14, where the running of the algorithm was still feasible. It should be noted that for Least Squares the amount of data points needed (m) to converge for different values of n growers approximately linearly, whereas winnow exhibits non-linear performance (see graphs). Therefore, for the 'just a little bit' data, depending on the specific amount of features (n) it could be that LS outperforms winnow.

It should be noted that the 4 algorithms tested can be sub-categorized into two distinct classes of Machine learning (ML) algorithms based on the way they work. Perceptrons are linear binary classifiers; they work on training sets with just two classes and look for the optimal separation hyperplane between these through computing a weight vector and evaluating it against a threshold value. If the dot product of the weight vector and the data exceeds the threshold one of the two classes is assigned, conversely if this product is below the threshold the other class is assigned. Winnow is a spacial case of the perceptron where the data labels are limited to [0,1] and in our case this runs faster. Both perceptron and winnow can handle large amounts of features relatively easily. Least squares and k-NN on the other hand, are algorithms which use distances between data points to learn class labels . Least squares is a regression model, which finds the best-fitting curve to a given set of points by minimizing the sum of the square distance (residuals) of the points from the curve. The k- nearest neighbors (kNN) algorithm inspects an amount of k data points nearest to the currently selected data instance. In this assignment we implement the simples form of k-NN, where only one neighbor that is nearest the query example (1-NN). In this case there in no use for weighting the neighbors so our function parameter is set to d=1 and the kernel function is omitted. When examining all data points and their neighbors the algorithm searches through large portions of the data set, hence it does not perform well with high-dimensional and sparse data This is visible in our results and explains why k-NN exhibits the worst performance.

(b)

We actually attempted to compute the sample complexity and realised very quickly that it would not be possible for data of dimension greater than 10. For $x \in -1, 1$ there are $2^1 0$ possible combinations for the x. We decided that we will use the error metric,

$$\frac{\text{Number of mistakes}}{\text{Number of data points}}$$

Although at first glance this metric may seem very different to the sample complexity formula describe in the exercise, but since we can sampling randomly from the possible combinations of a data point $\mathbf{x}$, with enough data points the above metric will approximate the sample complexity.

(c)

We've made our estimates based on the graphs we plotted in part (a). The patterns produced by each algorithm give a good indication to their behaviour as $n \to \infty$. The linear regression classifier is $0(n)$. The preceptron is also $O(n)$. The Winnow classifier is $(log(n)$ and the 1-nearest-neighboour is (e^n).

From the graph we can we that the perceptron actualy has a steeper gradient than the linear regression. Suggesting that the linear regression is a better algorithm than the perceptron. The Winnow algorithm was the clearly the most efficient. The ordering of the algorithms according to time complexity are:

1. Winnow

2. Linear Regression

3. Perceptron

4. 1-Nearest-Neighbour

Although it is worth mentioning that despite 1-NN being by far the slowest it is the only algorithm which reaches within $2\times$ the Bayes Classifier as $n \to \infty$