Supervised Learning: Assignment 1

Mahshid Alimi, Antonio Remiro Azócar, MSc Machine Learning

16 January 2017

Code (README)

Part I

Exercise 1a: `exec1_a.m`
Exercise 1b: `exec1_b.m`
Exercise 1c: `exec1_c.m`
Exercise 1d: `exec1_d.m`
Exercise 2a: `exec2_a.m`
Exercise 2b: `exec2_b.m`
Exercise 4: `exec4.m`
Exercise 5: `exec5.m`
Exercise 6: `exec6_7.m`, `MSE.m`, `w_ridge.m`
Exercise 7: `exec6_7.m`
Exercise 9: `exec9.m`, `MSE.m`, `w_calc.m`
Exercise 10: `exec10.m`, `Kernel_m.m`, `kridgereg.m`, `dualcost.m`, `MSE.m`

Part II

Exercise 4: `accuracy.m`, `winnow_pred.m`, `winnow_w.m`, `perceptron_pred.m`, `perceptron_w.m`, `least_squares_w.m`, `least_squares_pred.m`, `secondpart.m`

Part I

Exercise 1

(a) The procedure is commented in the code below.

```
clear all;
%
dim = 1; % dimensions
ntot = 600;
ntrain = 100;
ntest = 500;
%
w = randn(1,dim); % pick random value for w from st normal dist.
x = randn(ntot,dim); % 600 samples drawn from st normal dist.
n = randn(ntot,1); % noise generated from st normal dist.
y = x*w' + n; % noisy random dataset generated
%
% split dataset into training set size 100, test set size 500
x_train = x(1:ntrain,:);
y_train = y(1:ntrain);
x_test = x(ntrain+1:end,:);
y_test = y(ntrain+1:end);
```

(b) The code below is utilised.

```
% Exercise 1b
%
clear all
%
dim = 1; % dimensions
ntot = 600;
ntrain = 100;
ntest = 500;
%
w = randn(1,dim); % pick random value for w from st normal dist.
x = randn(ntot,dim); % 600 samples drawn from st normal dist.
n = randn(ntot,1); % noise generated from st normal dist.
y = x*w' + n; % noisy random dataset generated
%
% split dataset into training set size 100, test set size 500
x_train = x(1:ntrain,:);
y_train = y(1:ntrain);
x_test = x(ntrain+1:end,:);
y_test = y(ntrain+1:end);
%
% using eqn 5, estimate weights based on training set
train_weights = mldivide((x_train'*x_train), x_train'*y_train);
%
% using eqn 1, calculate predictive values
y_train_predict = x_train*train_weights;
y_test_predict = x_test*train_weights;
%
% using eqn 3, compute MSE on train and test sets
MSE_train = ((train_weights'*x_train'*x_train*train_weights)...
                -(2*y_train'*x_train*train_weights)+(y_train'*y_train))/ntrain
                ;
MSE_test = ((train_weights'*x_test'*x_test*train_weights)...
                -(2*y_test'*x_test*train_weights)+(y_test'*y_test))/ntest;
```

- To calculate the trained weight $\mathbf{w}^*$, equation $\mathbf{w}^* = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$ is utilised where, $\mathbf{X}$, $\mathbf{y}$ are training inputs. A predictive value is obtained via $y_i = \mathbf{x}_i\mathbf{w}$.
- The expression $\frac{1}{l}(\mathbf{w}'\mathbf{X}'\mathbf{X}\mathbf{w} - 2\mathbf{y}'\mathbf{X}\mathbf{w} + \mathbf{y}'\mathbf{y})$ has been used to calculate the train/test error, where $\mathbf{w}$ is the trained weight, $\mathbf{X}$, $\mathbf{y}$ correspond to train/test inputs and l are the number of test/train examples.

Over one example run, we obtain $MSE_{train} = 1.058, MSE_{test} = 1.116$.

(c) We utilise the same procedure as in **(b)**, this time with values, `ntot=510`, `ntrain=10`, `ntest=500`. Over one example run, we obtain $MSE_{train} = 0.738$, $MSE_{test} = 0.997$.

(d) The previous subsections are repeated looping over 200 `trials`, each time selecting a different random weight vector and a different random dataset. For each trial, `MSE_test` and `MSE_train` are appended to arrays `tot_MSE_test` and `tot_MSE_train` correspondingly; total error sums are later divided by the number of trials to calculate `avg_MSE_train` and `avg_MSE_test`. Over 200 trials, the following table of results (Table 1) is obtained for both 10/500 and 100/500 train/test splits using a dimensionality of 1:

Training Examples	MSE_{train}	MSE_{test}
10	0.925 ± 0.461	1.134 ± 0.234
100	0.996 ± 0.138	0.999 ± 0.070

Table 1: Recorded training and test MSEs. The number of test samples is set to 500. `dim=1`

The test error tends to be larger than the training error. This is expected; the weight parameters have been learned from the training set and do not fully generalise to the rest of the dataset. This effect is exacerbated when the training set is small. A larger training set size gives a lesser test error. The reasons why should

be evident; the more training data, the greater the sample size and the more information we have about the full dataset.

The procedure is implemented with the code below.

```
% Exercise 1d
%
clear all
%
dim = 1; % dimensions
trials = 200;
ntot = 600;
ntrain = 100;
ntest = 500;
tot_MSE_train = [];  % keeps track of training errors
tot_MSE_test = []; % keeps track of test errors
%
for i=1:trials
    w = randn(1,dim); % pick random value for w from st normal dist.
    x = randn(ntot,dim); % samples drawn from st normal dist.
    n = randn(ntot,1); % noise generated from st normal dist.
    y = x*w' + n; % noisy random dataset generated
    %
    % split dataset into training set, test set.
    x_train = x(1:ntrain,:);
    y_train = y(1:ntrain);
    x_test = x(ntrain+1:end,:);
    y_test = y(ntrain+1:end);
    %
    % using eqn 5, estimate weights based on training set
    train_weights = mldivide((x_train'*x_train), x_train'*y_train);
    %
    % using eqn 1, calculate predictive values
    y_train_predict = x_train*train_weights;
    y_test_predict = x_test*train_weights;
    %
    % using eqn 3, compute MSE on train and test sets
    MSE_train = ((train_weights'*x_train'*x_train*train_weights)...
                 -(2*y_train'*x_train*train_weights)+(y_train'*...
                 y_train))/ntrain;
    MSE_test = ((train_weights'*x_test'*x_test*train_weights)...
                -(2*y_test'*x_test*train_weights)+(y_test'...
                *y_test))/ntest;
    % append errors for each trial to total
    tot_MSE_train = [tot_MSE_train MSE_train];
    tot_MSE_test = [tot_MSE_test MSE_test];
end
%
% calculate averages and st.dev
avg_MSE_train = sum(tot_MSE_train)/trials;
stdev_MSE_train = std(tot_MSE_train);
avg_MSE_test = sum(tot_MSE_test)/trials;
stdev_MSE_test = std(tot_MSE_test);
```

Exercise 2

(a) The code is the same as that in **(1a)** but generating a dataset with `dim=10`. The file `exec2_a.m` is in the folder submitted online.

(b) The tasks/code from **(1b)**,**(1c)**, **(1d)** are repeated but with 10-dimensional datasets. The file `exec2_b.m` is in the folder submitted online, highlighting a replication of **(1d)** for 10 dimensions.

For 100 training examples and 500 test examples, we obtain $MSE_{train} = 0.939, MSE_{test} = 1.192$, over one example run. For 10 training examples and 500 test examples, we obtain $MSE_{train} = 1.047 \times 10^{-29}$ and $MSE_{test} = 3.56$ over one example run, this time using expression $\frac{1}{l}\sum_{i=1}^{l}(\mathbf{x}_i\mathbf{w}' - y_i)^2$. The process is repeated, this time looping over 200 `trials`, each time selecting a different random weight vector and a different random dataset. Table 2 is obtained for both 10/500 and 100/500 train/test splits using a dimensionality of 10.

Training Examples	MSE_{train}	MSE_{test}
10	$1.404 \times 10^{-23} \pm 1.946 \times 10^{-22}$	$1.291 \times 10^{3} \pm 5.802 \times 10^{3}$
100	0.884 ± 0.131	1.103 ± 0.088

Table 2: Recorded training and test MSEs. The number of test samples is set to 500.

(c) The test error tends to be larger than the training error for the reasons detailed in Exercise 1. The weight parameters have been learned from the training set and do not fully generalise to the rest of the dataset - overfit. The phenomenon of overfitting - the regression function fitting the noise too much - worsens with a problem of higher dimensionality, as seen in the results. It is exacerbated when using a small training sample size, as can be seen from the considerably larger test error when using 10 training examples. In this case, the model adapts itself too closely to the training data and does not generalise well. With 10 dimensions, the model is underdetermined. This is also reflected in the larger variance/standard deviation of the error/predictions using 10 training samples.

Exercise 3

(a) In matrix form, the loss function in optimisation problem (6) can be written as,

$$L = \gamma \mathbf{w}'\mathbf{w} + \frac{1}{l}(\mathbf{y} - \mathbf{X}\mathbf{w})'(\mathbf{y} - \mathbf{X}\mathbf{w}).$$

Taking the derivative of the loss function and setting it equal to zero gives,

$$\frac{\partial L}{\partial \mathbf{w}^*} = 2\gamma l \mathbf{w}^* - 2\mathbf{X}'\mathbf{y} + 2\mathbf{X}'\mathbf{X}\mathbf{w}^* = \mathbf{0},$$

with $\mathbf{w}^*$ as the solution of the problem. Rearranging the previous equation results in,

$$\mathbf{X}'\mathbf{X}\mathbf{w}^* + \gamma l \mathbf{I}\mathbf{w}^* = \mathbf{X}'\mathbf{y},$$
$$\rightarrow \mathbf{w}^* = (\mathbf{X}'\mathbf{X} + \gamma l \mathbf{I})^{-1}\mathbf{X}'\mathbf{y}.$$

(b) For any matrix $\mathbf{X}$, the matrix $\mathbf{X}'\mathbf{X}$ is symmetric; this is easy to prove: $(\mathbf{X}'\mathbf{X})'$=$\mathbf{X}'(\mathbf{X}')'$= $\mathbf{X}'\mathbf{X}$. The spectral theorem states that for any symmetric matrix, we have a basis of eigenvectors and every eigenvalue is real. $\mathbf{X}'\mathbf{X}$ is therefore positive semidefinite, with real non-negative eigenvalues and $\mathbf{y}'\mathbf{X}'\mathbf{X}\mathbf{y} = (\mathbf{X}\mathbf{y})'\mathbf{X}\mathbf{y} \geq 0$ for $\mathbf{y} \in \mathbb{R}^n$. Hence, the condition $\gamma > 0$ and $l > 0$ is sufficient to guarantee $(\mathbf{X}'\mathbf{X} + \gamma l \mathbf{I})$ being positive definite, therefore non-singular, and a ridge solution exists independently of the value of $\mathbf{X}$.

Exercise 4

The computations in this program are performed using the program `exec_4.m` presented below.

```
clear all;

% generate dataset like that of Q2

ntot = 600; % number of data points
dim = 10; % number of dimensions
ntrain = 100; % number of training examples
ntest = ntot - ntrain; % number of test examples

a = -6:1:3; % range of powers of ten used
for i=1:length(a)
```

```
    gamma(i) = 10^a(i);  % gamma generation
end
gamma_size = numel(gamma);

runs = 200;

train_errors = zeros(1, gamma_size); % keeps track of training errors
test_errors = zeros(1, gamma_size); % keeps track of test errors


for i=1:runs

    w = randn(1,dim); % pick random value for w from st normal dist.
    x = randn(ntot,dim); % x samples drawn from st normal dist.
    n = randn(ntot,1); % noise drawn from st normal dist.
    y = x*w' + n; % noisy random dataset generated
    % original train/test split
    x_train = x(1:ntrain,:);
    y_train = y(1:ntrain);
    x_test = x(ntrain+1:end,:);
    y_test= y(ntrain+1:end);
    for j=1:gamma_size
       % obtain alpha parameters for dual ridge regression
       alpha = mldivide((x_train*x_train' + gamma(j)...
                                *ntrain*eye(ntrain)),y_train);
       w_learned = x_train' * alpha; % learned weights from train
       % predictions
       y_test_predict = x_test * w_learned;
       y_train_predict = x_train * w_learned;
       % errors
       MSE_train = (norm(y_train_predict - y_train)^2) /ntrain;
       MSE_test = (norm(y_test_predict - y_test)^2) /ntest;
       % tracks errors over different parameter values
       train_errors(:,j) = train_errors(j) + MSE_train;
       test_errors(:,j) = test_errors(j) + MSE_test;
    end
end

% computing average values
avg_train_errors = train_errors./runs;
avg_test_errors = test_errors./runs;

% plots
figure
semilogx(gamma,avg_train_errors, '-.or')
hold on
semilogx(gamma, avg_test_errors, ':+k')
hold off
h = legend('training', 'test','Location','northwest');
set(h, 'FontSize', 14);
```

(a) Setting `runs` to 1, RR is performed for the data generated in Exercise 2a using different regularisation parameters. The errors over one run are plotted as a function of the regularisation parameter in Figure 1.

(b) This time using only 10 training examples and 500 test examples, the errors over one run are plotted as a function of the regularisation parameter in Figure 2.

(c) `runs` is set to 200 and parts **(a)** and **(b)** are repeated computing the average errors over such number of runs. The average errors over 200 runs are plotted a a function of the regularisation parameter using 100 training examples (Figure 3) and 100 training examples (Figure 4).

The results suggest that there is an 'optimal' γ parameter over which the error can be minimised. We can split our data into three (training/validation/test splits) instead of two. The purpose of using the validation set is to find the error-minimising γ. A suggested method is proposed below:

```
for trial in trials
    generate random data
    test/val/train/split
    for each gamma
        train - calculate train set weights
        compute val set predictions/error
    pick gamma with lowest error
    store gamma
    compute test set error
    store test set error
output mean gamma
output mean test error
```

We inspect Figures 1-4. We can observe that when using 100 training examples, the regularisation parameter γ tends to very low. On the other hand, when using 10 examples it tends to be higher (around 0.1). This has a simple explanation. A system with less training examples, particularly if the dimensionality of the problem is high, will tend to overfit/be overdetermined. Hence, the regularisation parameter will be higher to compensate for this effect.

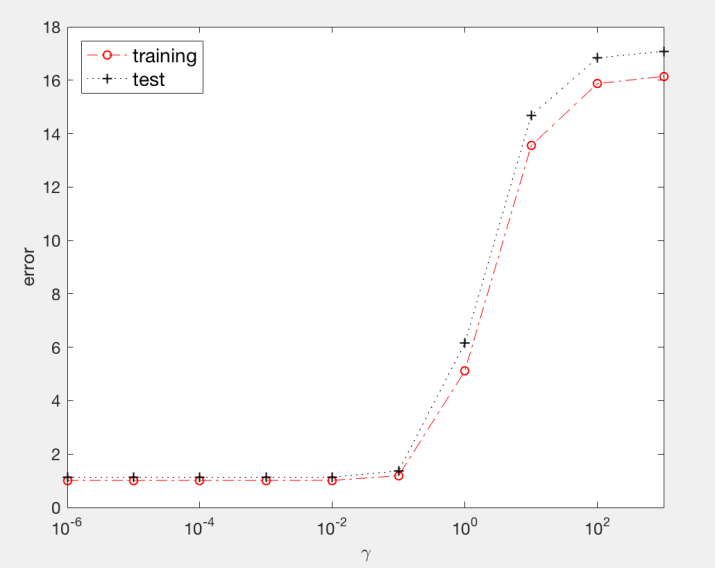

Figure 1: Training/test error plotted against γ (log scale) for 1 trial (100 training examples).

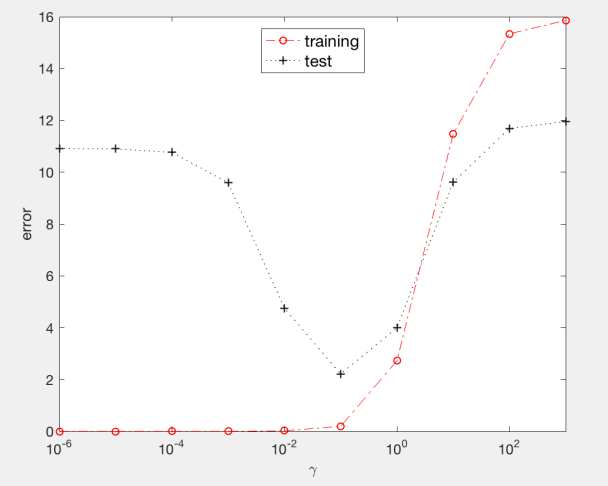

Figure 2: Training/test error plotted against γ (log scale) for 1 trial (10 training examples).

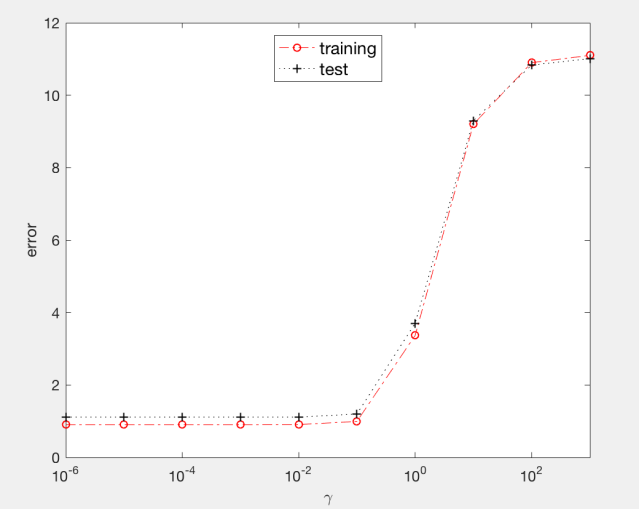

Figure 3: Training/test MSEs plotted against γ (log scale) for 200 trials (100 training examples).

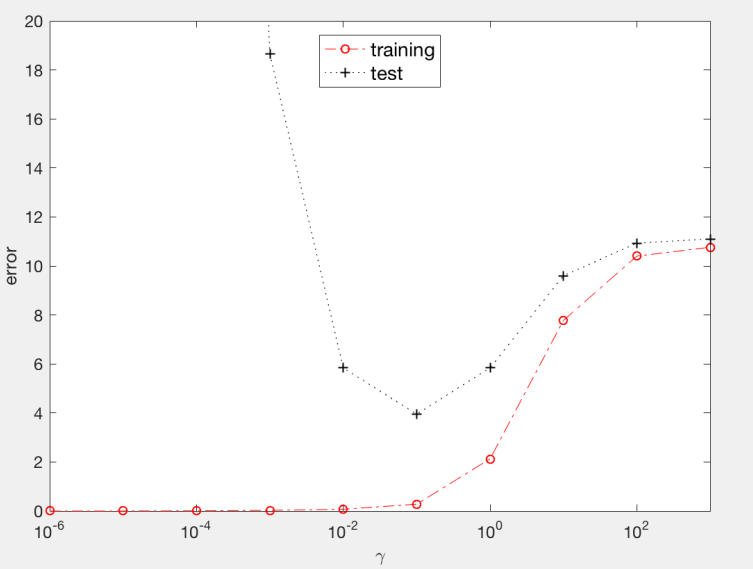

Figure 4: Training/test MSEs plotted against γ (log scale) for 200 trials (10 training examples).

Exercise 5

The computations in this program are performed using the program `exec5.m`, presented below.

```
clear all;

% generate dataset like that of Q2

ntot = 600; % number of data points
dim = 10; % number of dimensions
ntrain_original = 100; % number of training examples (original)
ntest = ntot - ntrain_original; % number of test examples

% 80/20 train/validation split
ntrain_new = floor(0.8 .* ntrain_original); % ntrain2 is the new training set
    size
nval = ntrain_original - ntrain_new;

a = -6:1:3; % range of powers of ten used
```

```
for i=1:length(a)
    gamma(i) = 10^a(i);  % gamma generation
end
gamma_size = numel(gamma);

runs = 200;

train_new_errors = zeros(1, gamma_size); % keeps track of training errors
val_errors = zeros(1, gamma_size); % keeps track of validation errors
test_errors = zeros(1, gamma_size); % keeps track of test errors
tot_min_gamma = [];  % keeps track of gamma values.
tot_definitive_test_err = []; % keeps track of definitive test error.

for i=1:runs
    min_val_err = 10^8; % dummy variable for minimum val error calculation
    w = randn(1,dim); % pick random value for w from st normal dist.
    x = randn(ntot,dim); % x samples drawn from st normal dist.
    n = randn(ntot,1); % noise drawn from st normal dist.
    y = x*w' + n; % noisy random dataset generated
    % original train/test split
    x_train_original = x(1:ntrain_original,:);
    y_train_original = y(1:ntrain_original);
    x_test = x(ntrain_original+1:end,:);
    y_test= y(ntrain_original+1:end);
    % split original train into train and val
    x_train_new = x(1:ntrain_new, :);
    y_train_new = y(1:ntrain_new, :);
    x_val = x(ntrain_new+1:ntrain_original, :);
    y_val = y(ntrain_new+1:ntrain_original, :);
    for j=1:gamma_size
       % obtain alpha parameters for dual ridge regression
       alpha = mldivide((x_train_new*x_train_new' + gamma(j)...
                                *ntrain_new*eye(ntrain_new)),y_train_new);
       w_learned = x_train_new' * alpha; % learned weights from train
       % predictions
       y_val_predict = x_val * w_learned;
       y_train_new_predict = x_train_new * w_learned;
       y_test_predict = x_test * w_learned;
       % errors
       MSE_val = (norm(y_val_predict - y_val)^2) / nval;
       MSE_train_new = (norm(y_train_new_predict - y_train_new)^2)/ntrain_new
           ;
       MSE_test = (norm(y_test_predict - y_test)^2) /ntest;
       % tracks errors over different parameter values
       train_new_errors(:,j) = train_new_errors(j) + MSE_train_new;
       val_errors(:,j) = val_errors(j) + MSE_val;
       test_errors(:,j) = test_errors(j) + MSE_test;
       % keeps track of optimal gamma for a particular run
       if MSE_val < min_val_err;
            min_val_err=MSE_val;
            min_gamma = gamma(j);
       end
    end
    % append to gamma tracker
    tot_min_gamma = [tot_min_gamma min_gamma];
    % use both train and validation sets to train weights for test error
    alpha_new = mldivide((x_train_original*x_train_original' + min_gamma...
                           *ntrain_original*eye(ntrain_original)),
                              y_train_original);
```

```
    w_learned_new = x_train_original' * alpha_new; % learned weights from
        train
    % test predictions using optimal gamma
    y_test_predict_new = x_test * w_learned_new;
    % keeps track of final test errors
    definitive_test_error = (norm(y_test_predict_new - y_test)^2) / ntest;
    [tot_definitive_test_err] = [tot_definitive_test_err
        definitive_test_error];
end

% computing average values
avg_train_new_errors = train_new_errors./runs;
avg_val_errors = val_errors./runs;
avg_test_errors = test_errors./runs;
avg_gamma = sum(tot_min_gamma)/runs;
avg_definitive_test_error = sum(tot_definitive_test_err)/runs;
std_test_error = std(tot_definitive_test_err);

% plots
figure
semilogx(gamma,avg_train_new_errors, '-.or')
hold on
semilogx(gamma, avg_val_errors, '--xb')
semilogx(gamma, avg_test_errors, ':+k')
hold off
h = legend('training', 'validation','test','Location','northwest');
set(h, 'FontSize', 14);
```

(a) The procedure is implemented using the code `exec_5a.m`. The same data generation process is used as that of **2a**. This time, however, data is generated for each of 200 runs. For each run, we loop over the ten different values of γ, storing training errors, validation errors and test errors. Note that an 80:20 ratio is used to split the original training set into a new training set and a validation set. The computed average errors over 200 runs are plotted as a function of the regularisation parameter (log scale) in Figure 5.

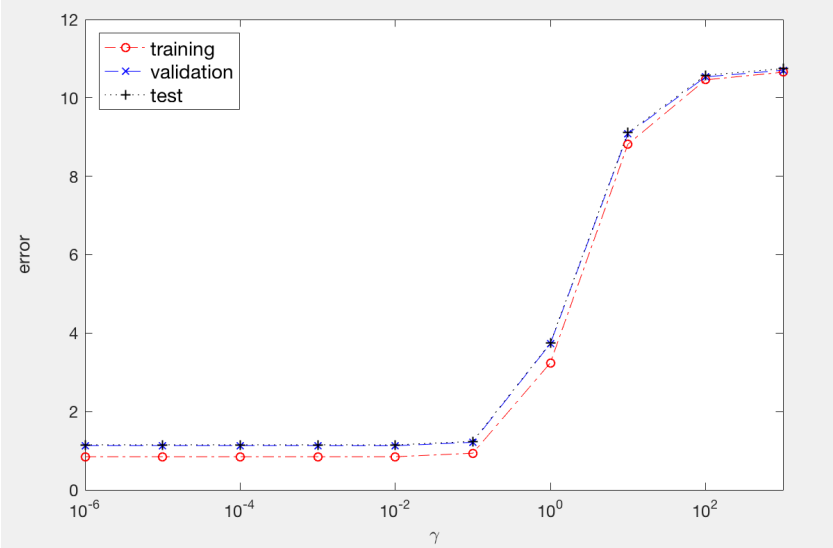

Figure 5: Training/val/test MSEs plotted against γ (log scale) for 200 trials, 80/20 train/val split.

Subsequently, RR is performed in the full original training set (100 samples). Note that in Figure 5, the test error corresponds to that computed using the weights trained using only the training set (after split). For this part of the question, once the optimal gamma (minimising validation error) is known, the test error is calculated using weights trained from the full (training + validation) set. We obtain an optimal $err_{test} = 1.156 \pm 0.195$.

(b) Part (**a**) is repeated but using an original training set of 10 examples (with the same 8/2 train/validation

proportion). The computed average errors are plotted as a function of the regularisation parameter in Figure 6. An optimal test error of $err_{test} = 39.7 \pm 402.7$ is obtained.

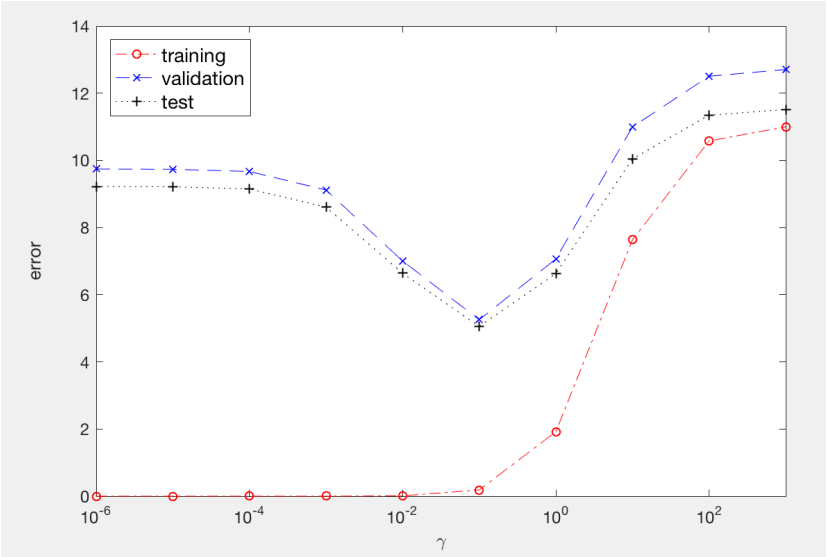

Figure 6: Training/val/test MSEs plotted against γ (log scale) for 200 trials, 8/2 train/val split.

(c) For part **(a)** (with a different run of the program), we obtain the value $\bar{\gamma}^{100} = 0.039 \pm 0.106$. For part **(b)**, we obtain $\bar{\gamma}^{10} = 95.5 \pm 293.8$. Again, this result is expected. A system with less training examples, particularly if the dimensionality of the problem is high, will tend to overfit/be overdetermined. Hence, the regularisation parameter will be higher to compensate for this effect.

(d) The exercise is repeated with a dimensionality of 1. For the split in part **(a)**, this gives the results $err_{test} = 1.044 \pm 0.116$, $\bar{\gamma}^{100} = 40.4 \pm 196.4$. For **(b)**, $err_{test} = 1.308 \pm 0.614$, $\bar{\gamma}^{10} = 256.2 \pm 436.3$.

Exercise 6

The code utilised for this exercise is the following, `exec6_7.m` - also utilised for exercise 7.

```
% Section 1 Exercise 6
clear all;
clc;
close all;

% The data set is taken out of Exercise 2, Part a:
% The data set has 600 samples and is 10 dimentional.
ntot = 600;
dim = 10;

% The first 100 samples in the data set is dedicated to the train set and
% the rest is allocated to the test set.
ntrain = 10;
ntest = ntot - ntrain;

% K_fold is the number of k-fold disjoint sets
K_fold = 5;

% Initialising an array to the parameter gamma and the number of its elements
    .
gamma = 10.^(-6:3);
gamma_size = numel(gamma);

%  w_learned is generated using the normal distribution.
```

```
w_learned = randn(dim,1);

% 600 x-variables are also randomly generated from the normal distrubtion,
x = randn(ntot,dim);
% An array of noise is generated from the normal distribution.
eps = randn(ntot,1);

% Random y_variables are generated using y = x*w' + eps.
for i = 1:ntot
    y(i,:) = x(i,:)*w_learned + eps(i,:);
end

% These traing sets include the first 80 samples of the data set.
x_train = x(1:8,:);
y_train = y(1:8);

% The validation sets include the range 81 to 100th sample of the data set.
x_val = x(9:ntrain,:);
y_val = y(9:ntrain);

% The original training sets include the first 100 samples from the
% data set. The training and the validation sets both make up the original
% trining set.
x_train_original = x(1:ntrain,:);
y_train_original=y(1:ntrain);

% The test data set includes the 101 to 600th sample.
x_test = x(ntrain+1:end,:);
y_test = y(ntrain+1:end);

% The training set is sub-divided into smaller validation sets.
fold_sub = ntrain/K_fold;

% In the following for loop, a validation set is picked and the rest of the
% original training set is then assigned to the training set for the x and
% y variables.
% At each iteration, a different validation set is picked and the process
% is repeated.

for k = 1:K_fold
    x_train_fold = x_train_original(1:fold_sub,:);
    x_val_fold = x_train_original(fold_sub+1:end,:);
    y_train_fold = y_train_original(1:fold_sub,:);
    y_val_fold = y_train_original(fold_sub+1:end,:);
    x_train_original = circshift(x_train_original,[fold_sub 0]);
    y_train_original = circshift(y_train_original,[fold_sub 0]);
end


% The aim of this for loop is to predict the value of y and calculate its
% Mean Squared Error from the actual value of y. This process is repeated
% 200 times.
mse_test_original = zeros(1,gamma_size);
mse_train_original = zeros(1,gamma_size);
mse_val = zeros(1,gamma_size);
iteration = 200;
train_error = zeros(1,iteration);


```

```
for j = 1:iteration

%     Ridge regression is implementend:
    for k = 1:K_fold
    x_train_fold = x_train_original(1:fold_sub,:);
    x_val_fold = x_train_original(fold_sub+1:end,:);
    y_train_fold = y_train_original(1:fold_sub,:);
    y_val_fold = y_train_original(fold_sub+1:end,:);
    x_train_original = circshift(x_train_original,[fold_sub 0]);
    y_train_original = circshift(y_train_original,[fold_sub 0]);


    for l = 1:gamma_size


        % Ridge regression is done on the training folds and the respective
          MSE is
        % calculated.
             w_fold = w_ridge(x_train_fold,y_train_fold,gamma(l));
             mse_train_fold(k,l) = MSE(x_train_fold,y_train_fold,w_fold);
             mse_val_fold(k,l) = MSE(x_val_fold,y_val_fold,w_fold);
             mse_test_fold(k,l) = MSE(x_test, y_test, w_fold);

    end
            % The avarage MSE for the training and validation folds is then
                calculated
         mse_train_original = sum(mse_train_fold)./K_fold;
         mse_val_avg = sum(mse_val_fold)./K_fold;
         mse_test_avg = sum(mse_test_fold)./K_fold;

    end


    % The optimal gamma that gives minimum five fold validation error
    [N3,D3] = min(mse_val_avg);
    kfold_val_error(j) = gamma(D3);

    % MSE using optimum gamma for five fold validation set:
    w_learned = w_ridge(x_train_original,y_train_original,kfold_val_error(j))
        ;
    mse_opt(j) = MSE(x_test,y_test,w_learned);

end



% A graph of log of gamma against the MSEs' is then plotted.


 hold on
 plot(log(gamma),mse_train_original)
 plot(log(gamma),mse_val_avg)
 plot(log(gamma),mse_test_avg,'green')
 hold off;

  title('Main Training set of 10 Samples')
legend('MSE Main Training Set','MSE Test Set','MSE Validation Set')
 xlabel('log Gamma');
 ylabel('MSE');
```

```
139
140 % Question 7) part c)
141     if ntrain == 100
142         mse_optimum_train_avg_hundred = mean(mse_train_original)
143         mse_optimum_val_avg_hundred = mean(mse_val_avg)
144         mse_optimum_kfold_avg_hundred = mean( mse_test_avg)
145     elseif ntrain == 10
146         mse_optimum_train_avg_ten = mean(mse_train_original)
147         mse_optimum_val_avg_ten = mean(mse_val_avg)
148         mse_optimum_kfold_avg_ten = mean( mse_test_avg)
149     end
```

(a) The results presented in Figure 7 are obtained.

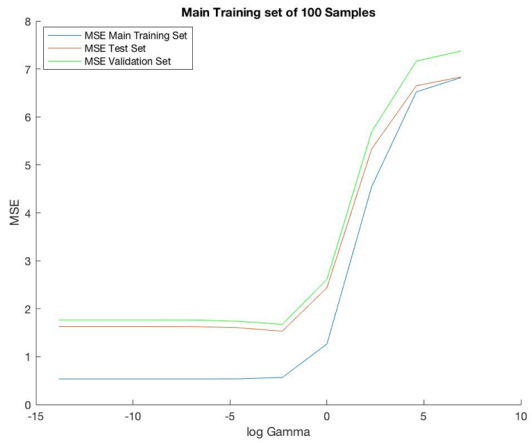

Figure 7: MSE plotted against $\log \gamma$ for part **(a)**

As expected the the training set has the lowest of MSEs and the the validation set has the highest of MSEs for 100 training points.
(b) The results presented in Figure 8 are obtained.

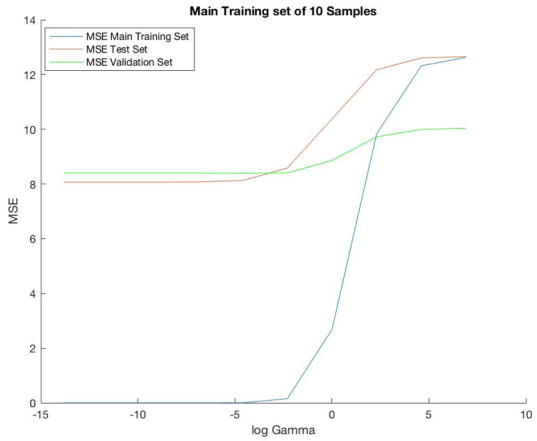

Figure 8: MSE plotted against $\log \gamma$ for part **(b)**

As illustrated in 10 sample Figure 8, despite the test and validation sets not following the same trend, their minimum MSE value is very close. The training set looks very similar to that of the 100 point sample. In **(a)** and **(b)**, the cross validation method yields more reliable outputs as it finds the optimal γ value on the validation set and uses it to predict the y-value of test set.
This exercise makes use of the following functions: `MSE.m` to calculate the mean square error.

```
% The Means Squared Error (MSE) is often used after regression operations,
% for training or test sets.
% The following fucntion is created which will be called.

function mse = MSE(x,y,w)

len = size(x,1);
for i = 1:len
    n_mse(i) = ((x(i,:)*w) - y(i,:)).^2;
end
mse = (1./len) .* sum(n_mse);

end


% MSE, = (1/number of rows) * sum((x*w-y).^2)
% where,
% x is the training/test data set x-values
% y is the training/test data set y-values
% w is a result of w_calc function
```

`w_ridge.m` to calculate the Ridge regression weights learned from the training examples.

```
function fcn = w_ridge(x,y,gamma)
d = size(x,2);
l = size(x,1);
fcn = mldivide((x'*x)+(gamma*d*eye(d)) , (x'*y));
end
```

Exercise 7

Table 3: Test errors across different methods

	Training size: 100		Training size: 10	
	Mean	STD	Mean	STD
Training Error	1.162	0.072	124.0	264.8
Validation Error	1.303	0.132	34.9	179.2
Cross Validation	1.107	0.088	5.85	5.87

In this exercise, the code `exec6_7.m` is utilised. As illustrated in Table 3, the general the MSE improves from exercise 4 to 6. All the model have a lower error on 100 training sample data points than 10 sample data points. The standard devastation is higher for 10 sample than for 100 sample training set as we are less likely to overfit noise. Intuitively, the regulation parameter should decrease in value from the train error to the cross-validation model. This premise is true since a higher regularisation parameter is needed for smaller sets to avoid overfitting. Unsurprisingly, train error gives out a simplistic estimate of the generalisation error. This is due to using the training set to predict the y-value of the test set ,which as expected would give lower MSE as the model does not use the test set to learn weights. The validation model will evidently perform poorly and output a large MSE when 2 and 8 sample points are used as the validation set and training set respectively as merely 2 data samples will not be adequate for the prediction of the regressors. This reasoning is further proved where MSE is considerably improved for training sample of 80 and validation set of 20. The validation model predicts the y-value on the test set and hence by comparing the train error and validation error on 100 training sample, it is deduced that the MSE is slightly higher for the validation set. The cross validation model outputs smaller MSEs on the 10 training sample run. There is 5 different folds to the validation sample set and at each fold a different set of validation set is picked and the rest are assigned to the training set. This means that the model predicts the regressor on the 10 sample sets in total and hence is expected to output lower MSE.

Exercises 8, 9

The dataset from "Boston Housing" was downloaded and imported into Matlab. Generally, the Mean Squared Error (MSE) is lower on the training set as anticipated. For the purposes of this exercise, the training set is taken to be the two thirds of the dataset and the model is tested on the remaining third.

(a) The results obtained for 20 runs are shown below:

Method	MSE_{train}	MSE_{test}
Naive Regression	82.7 ± 6.0	88.1 ± 12.1

Table 4: Recorded training and test MSEs using naive regression in **(a)**

.

(b) The results obtained for 20 runs are shown in Table 5.

Attribute	MSE_{train}	MSE_{test}
Attribute 1	71.7 ± 5.5	72.5 ± 10.8
Attribute 2	72.9 ± 5.4	77.2 ± 9.8
Attribute 3	61.9 ± 5.0	67.8 ± 11.2
Attribute 4	80.3 ± 5.3	84.6 ± 11.4
Attribute 5	67.7 ± 5.3	71.5 ± 10.6
Attribute 6	42.4 ± 3.7	42.1 ± 7.5
Attribute 7	72.8 ± 5.2	74.0 ± 12.5
Attribute 8	77.7 ± 6.1	82.1 ± 12.2
Attribute 9	70.9 ± 5.3	73.5 ± 11.8
Attribute 10	64.4 ± 5.5	68.4 ± 10.2
Attribute 11	63.4 ± 4.7	64.2 ± 9.3
Attribute 12	74.5 ± 5.8	77.3 ± 11.2
Attribute 13	36.8 ± 2.7	39.9 ± 5.3

Table 5: Recorded training and test MSEs over different attributes.

(c) The results obtained for 20 runs are shown in Table 6.

	MSE_{train}	MSE_{test}
Best single attribute	36.8 ± 2.7	39.9 ± 5.3
Worst single attribute	80.3 ± 5.3	84.6 ± 11.4
All attributes	21.5 ± 1.7	24.2 ± 3.6

Table 6: Recorded training and test MSEs over best/worst single attribute and all attributes.

Analysis: The **naive regression** method regresses the dependent variable on a vector of ones which results into the MSE of the y-variable. It only takes into account the actual dependent variables and the bias term. This is the most basic form of regression and appropriately given the name "Naive". As expected, Naive regression performs poorly in comparison to linear ridge regression and KRR when used to predict the y-variable (it merely computes the MSE between train and test y variables.

Linear ridge regression: y-variable values are predicted by learning a weight vector (accounting for a regularisation parameter) from the training set and $y = \mathbf{w}^T\mathbf{x}$. The MSE represents the predictive accuracy. Regression is performed using a one dimensional component, hence referred to as single attribute. It takes into account each attribute at a time plus the added bias term. The resulting output is the independent prediction made by each single attribute. Attribute 4 results in the worst single attribute prediction ($MSE_{test} = 84.6 \pm 11.4$) while Attribute 13 gives the best single attribute prediction ($MSE_{test} = 39.9 \pm 5.3$).

By comparing the tabulated results of part a and b, it is evident that for this dataset, linear Ridge regression is a more reliable model function than Naive regression, yielding lower MSE results.

On part c, the regression is performed using all dependent attributes where the updated x-component is 13-dimentional. Each dimension consists of the values of each of 13 attributes. Furthermore, a bias term is included which adds a column of ones to the dataset. "All attribute" regression appears to outperform

the methods used in part a and b with $MSE_{test} = 24.2 \pm 3.6$. In part c, regression considers the whole database as opposed to individual attributes at a time.

Code for exercise 9 is submitted as `exec9.m` and presented below.

```
% Supervised Learning 1) Section 1) Exercise 9) Part a)

load ('boston.mat')

% Number of rows:
n_row = (boston,1);

% Each row of the "Boston" dataset represents a record.
% As requested from the Exercise,the training set needs tob 2/3 of the data base.
% Hence the training set number is:
% "2/3*number of records(506)=337.3"
% This needs to be rounded down as number of training sets needs to be an integer.
% The function 'floor' is used to to round down the value of the result:
n_train_set = floor(2/3.*n_row);

% "set_x" represents the x values and as requested from the question,
% It is a vector of ones.
set_x = ones(n_row,1);

% Linear regression is used to predict with mean of the y-value from training set.
% w is a vector which is worked out as follows:
% w = (x'*x) \ (x'*y)
% Since x_set is 1, expression (x'*x) will be the number of elements within
% vector of x which is 506.
% Furthermore,(x'*y) gives [y1+y2+..+y506] as each element in x is 1.
% Therefore, learned w is merely same as calculating the mean.
set_y = boston(:,14);

% At each iteration, at random, 2/3 of the data points is chosen for the
% training set.
% The following for loop demonstrates the prediction of y which as
% explained is the same as the learned w. This is then itrated 20 times.

MSE_train_part_a=zeros(1,20);
MSE_test_part_a=zeros(1,20);

for i = 1:20

    % The data_sample function is built to get a random permuation from
    % the data base.
    % The function randperm(n_row) returns vector of "ones" to the shuffled
    % n_row.
    data_sample = randperm(n_row);

    % These elements are then used to as indices for "set_x" on the training set.
    % The function below returns an array where "set_x" is its elements.
    x_train = set_x(data_sample(1:n_train_set),:);

    % This procedure is repeated on the rest of the database for "x_test".
    x_test = set_x(data_sample(n_train_set+1:end),:);

    % Similar prodecure is done for "y_train" and "y_test".
```

```
    y_train = set_y(data_sample(1:n_train_set),:);
    y_test = set_y(data_sample(n_train_set+1:end),:);

    % As the name suggests w_learned is the learned w vector and it is used
    % The function w_calc is explained  in details in its own file.
    % It essensially calculates w using (X'X)\(X'y).
    w_learned_a(i) = w_calc(x_train,y_train);

    % MSE_train_part_a and test_MSE_part_a are the mean square error
    % for train and test sets.
    MSE_train_part_a(i) = MSE(x_train,y_train,w_learned_a(i));
    MSE_test_part_a(i) = MSE(x_test,y_test,w_learned_a(i));
end

% The avarage value for each array is then taken.
avg_train_MSE_part_a = mean(MSE_train_part_a);
avg_test_MSE_part_a = mean(MSE_test_part_a);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Supervised Learning 1) Section 1) Exercise 9) Part b)


%  For each iteration and aeach of the 13 attributes regression is
%  performed and MSE is calculated.
%  To operate this, 2 for loops are created  one to
%   loop through the attributes (which are columns in the boston matrix),
%   and the other loop for the simulations.

% Number of columns:

n_col =(boston,2);

% The bias is merely a colomn vector of 1.
bias = ones(n_row,1);

MSE_train_part_b=zeros(1,20);
MSE_test_part_b=zeros(1,20);

% This for loops goes through the attributes
for i = 1:n_col-1

   % This for loop iterates:
   for k = 1:20
       % set_xb will be the entire column for each attribute.
       set_xb = boston(:,i);
       % set_y stays the same and is the last column .
       set_y = boston(:,14);
       % x and y values fro training/test sets are generated randomly.
       x_train_b = [set_xb(data_sample(1:n_train_set),:) bias(1:n_train_set)];
       x_test_b = [set_xb(data_sample(n_train_set+1:end),:) bias(n_train_set+1:end)];
       y_train_b = set_y(data_sample(1:n_train_set),:);
       y_test_b = set_y(data_sample(n_train_set+1:end),:);
       % learned value is calclated using w_calc.
       w_learned_b = w_calc(x_train_b,y_train_b);
```

```
        % MSE is then taken:
        MSE_train_part_b(i,k) = MSE(x_train_b,y_train_b,w_learned_b);
        MSE_test_part_b(i,k) = MSE(x_test_b,y_test_b,w_learned_b);
    end

    % for each iteration the avarage MSE is caluclated
    avg_train_MSE_part_b(i) = mean(MSE_train_part_b(i,:));
    avg_test_MSE_part_b(i) = mean(MSE_test_part_b(i,:));
end




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Supervised Learning 1) Section 1) Exercise 9) Part c)


set_y = boston(:,14);
for i = 1:20

    x_train_c = set_x_c(data_sample(1:n_train_set),:);
    x_test_c = set_x_c(data_sample(n_train_set+1:end),:);
    y_train_c = set_y(data_sample(1:n_train_set),:);
    y_test_c = set_y(data_sample(n_train_set+1:end),:);
    w_learned_c = w_calc(x_train_c,y_train_c);
    MSE_train_part_c(i) = MSE(x_train_c,y_train_c,w_learned_c);
    MSE_test_part_c(i) = MSE(x_test_c,y_test_c,w_learned_c);
end
avg_train_MSE_part_c = mean(MSE_train_part_c);
avg_test_MSE_part_c = mean(MSE_test_part_c);
```

Exercise 9 also makes use of the following functions:
`MSE.m` to calculate the mean square error (presented in Exercise 6).
`w_calc.m` to learn the linear regression weights of the training data.

```
% As part of the linear regression operation,intead of performing the
% calculation of the learned w each time, the follwoing fucntion is created
% which will be called.

% As the name of the fucntion suggests, w_calc calculates the learned w
    vector.
% This is done by opertaing regression on the training se.

function fnc = w_calc(x,y)
fnc =mldivide((x.'*x),(x.'*y));
end

%  w = (X'X)^(-1) * (X'y)
%  where,
%  x is  the training/test set of x values.
%  y is the training/test set y-values.
```

Exercise 10

Gaussian Kernel regression yields the best prediction of the y-value among all models, adding non-linearity and not restricting itself to a linear space. Optimal values for σ and γ are picked by minimising the MSE

on the five-fold validation set. These optimal parameter values are later used on the test set in order to predict the y-value for the test set.

(a) The function `kridgereg.m` is presented below.

```
% This function is programmed to perform kernel ridge regression
% using alpha=y/(K+(gamma*l*I))

function fcn = kridgereg(k,y,gamma)
% Number of training points:
l = size(y,1);
% Kernel ridge regression:
fcn = mldivide((k + (gamma*l*eye(l))) , y);
end
```

(b) The function `dualcost.m` is presented below.

```
% the following function is created using:
% (1/l)(k_test*alpha - y)'(k_test*alpha - y)

function MSE=dualcost(k,y,alpha)
    [l,~]=size(y);
    MSE=1/l*(k*alpha-y)'*(k*alpha-y);
end
```

(c) Figures 9,10 show plots of the "cross-validation error" (mean over folds of validation error) as a function of γ and σ. Each has a different orientation to facilitate observing the minimum. Note that the σ and γ axes do not represent the actual value of the parameters; they represent the power of 10 by which they are multiplied.

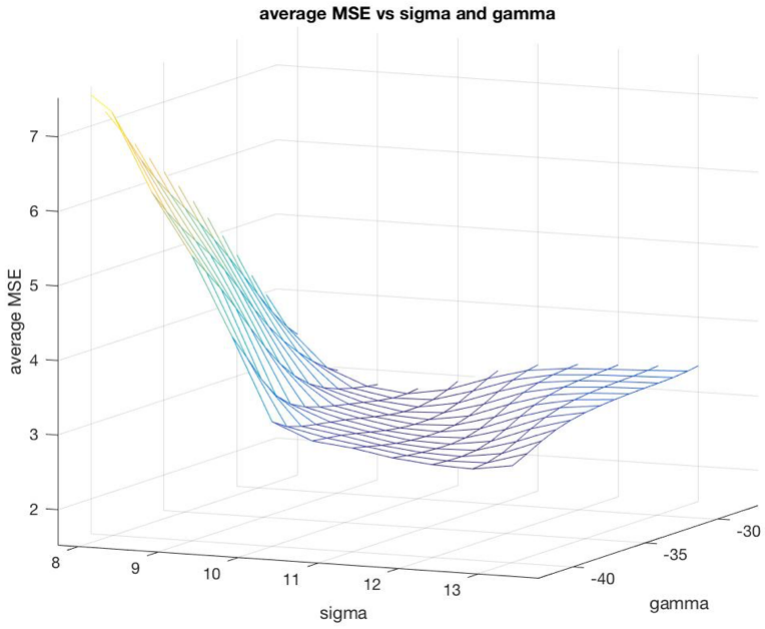

Figure 9: Plot of the "cross-validation error" (mean over folds of validation error) as a function of γ and σ (Orientation 1). Note that the σ and γ axes do not represent the actual value of the parameters; they represent the power of 10 by which they are multiplied.

For future reference, one can also plot the logarithm of the cross-validation error against the parameters. In that way, it is easier to see where the minimum is; this can be done since log is a monotonic function so extrema are maxed to extrema of the same kind.

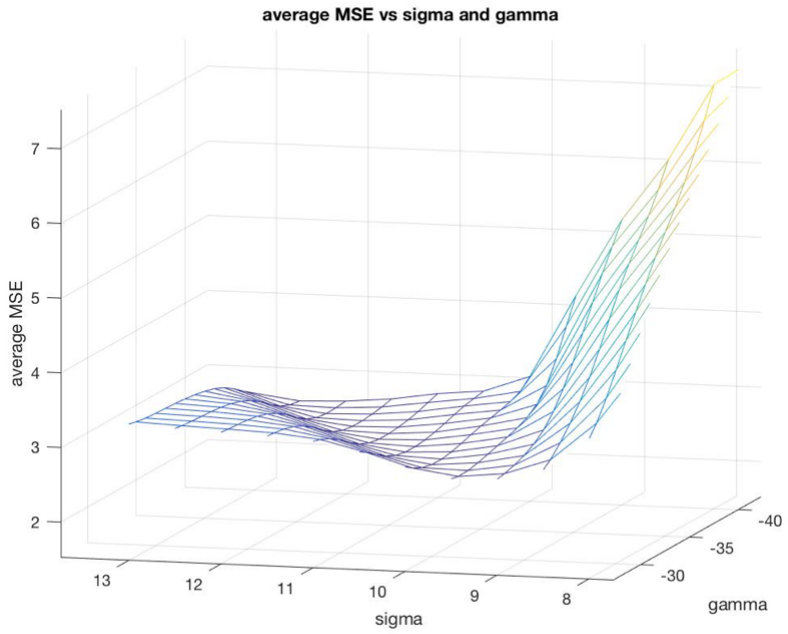

Figure 10: Plot of the "cross-validation error" (mean over folds of validation error) as a function of γ and σ (Orientation 2). Note that the σ and γ axes do not represent the actual value of the parameters; they represent the power of 10 by which they are multiplied.

(d) Over 20 runs, the results obtained for Kernel Ridge regression are the following. The higher training set error is due to suspect overfitting as the validation folds are reused and added to the training set.

	MSE_{train}	MSE_{test}
Kernel Ridge Regression	7.45 ± 1.48	13.76 ± 3.73

Table 7: Recorded training and test MSEs over 20 runs for Kernel Ridge Regression

The code implemented for Exercise 10 is as follows (`exec10.m`):

```
% Loading Boston dataset

clear all;
close all;
clc;

data = load('boston.mat');
sample = data.boston;


% Initialsing the variables gamma and sigma as instructed from the question

a=[-40:-26];
length_gamma=length(a);
gamma=(2*ones(1,(length_gamma))).^a;


b=[7:0.5:13];
length_sigma=length(b);
sigma=(2*ones(1,(length_sigma))).^b;
```

```
21
22 % The total number of the rows in the data base is needed for the splitting
23 % of train,valdation and test sets. The training set is 2/3 of the total
24 % dataset.
25
26 [rows,~]=size(sample);
27 train_set=floor(2/3*rows);
28
29 % Initialising number of K foldds and iterations.
30 k_folds=5;
31 iteration=20;
32 fold_subset=round(train_set/k_folds);
33
34 % These zero arrays elemsnts will get filled after each iterations.
35
36 min_mse_val=zeros(1,iteration);
37 mse_train_final=zeros(1,iteration);
38
39
40
41 for count=1:iteration
42     count
43
44   % The data set is split into to main training and test set.
45     sample_data = sample(randperm(rows),:);
46
47     set_x =  sample_data(:,1:13);
48     set_y =  sample_data(:,14);
49
50     x_train = set_x(1:train_set,:);
51     y_train = set_y(1:train_set,:);
52
53     x_test = set_x(train_set + 1:rows,:);
54     y_test = set_y(train_set + 1:rows,:);
55
56    % The following for loop attempts to separate 5 different layers within
57    % the main training set. At each iteration one of five the subsets
58    % will become the validation set and the remaining four sets of the main
59    % training set will become the sub training set. The validation layer will
60    % change and shuffle each time with the help of function circshift.
61
62        for k = 1:k_folds
63              x_val_fold = x_train(1:fold_subset,:);
64              x_train_fold = x_train(fold_subset+1:end,:);
65              y_val_fold = y_train(1:fold_subset);
66              y_train_fold = y_train(fold_subset+1:end);
67              x_train = circshift(x_train,[fold_subset, 0]);
68              y_train = circshift(y_train,[fold_subset, 0]);
69
70 %              xtrain = x_train_fold(:,:,k);
71
72
73 %       The aim if the following for loop is to predict the value of y on
74 %       the validation set.
75
76         for i=1:length_gamma
77
78            for j=1:length_sigma
79
```

```
80                kernel_matrix= Kernel_m(x_train_fold,x_val_fold,sigma(1,j));
81
82                k_matrix_train=kernel_matrix(1:size(...
83                    x_train_fold,1),1:size(x_train_fold,1));
84
85                end_kernel=size(kernel_matrix,1);
86
87                k_matrix_val=kernel_matrix(...
88                    size(x_train_fold,1)+1:end_kernel,1:size(x_train_fold,1));
89
90               alpha_dual=kridgereg(k_matrix_train,y_train_fold,gamma(1,i));
91               mse_y_val(i,j,k)=dualcost(k_matrix_val,y_val_fold,alpha_dual);
92
93
94            end
95          end
96        end
97
98
99        % Here different validation sets are added up together which gives
100       % the matrix a third dimention with the size of the k fold.
101        mse_validation_set=sum(mse_y_val,3);
102
103       % To calculate the avarage mse validation, it needs to be divided by
104       % the number of k-folds. This will make the matrix 2 dimensional
105       % again.
106
107        average_mse_val=mse_validation_set/k_folds;
108
109 %
110 %          if count==20 %only plot for one iterration
111 %              figure;
112 %              mesh(log2(sigma),log2(gamma),average_mse_val);
113 %              xlabel('Sigma');
114 %              ylabel('Gamma');
115 %              zlabel('Average MSE');
116 %
117 %          end
118       %  Finding the optimal sigma and gamma values from the validation set
119       %  and reusing that on the main test set.
120
121       [row_gamma, coloumn_sigma]=find(average_mse_val==min(min(average_mse_val
              )));
122        opt_gamma=gamma(1,row_gamma);
123        opt_sig=sigma(1,coloumn_sigma);
124
125        min_mse_val(count)=min(min(average_mse_val));
126
127      % re-calculating the K-matrix in order to perdict y on the test set.
128
129      k_mat_train=Kernel_m(x_train,[],opt_sig);
130      new_alpha=kridgereg(k_mat_train,y_train,opt_gamma);
131
132      mse_train_final(count)=dualcost(k_mat_train,y_train,new_alpha);
133
134
135      k_mat_test=Kernel_m(x_train,x_test,opt_sig);
136
137      end_kernel_2=size(k_mat_test,1);
```

```
138
139       new_alpha_2=(kridgereg((k_mat_train),y_train,opt_gamma));
140
141       k_test=k_mat_test(train_set+1:end_kernel_2,1:size(x_train,1));
142
143       mse_test_final(count)=dualcost(k_test,y_test,(new_alpha_2))
144
145
146
147   end
148
149   % Part D)
150
151   mean_min_mse_train = mean(mse_train_final);
152   sd_min_mse_train = std(mse_train_final)
153
154   mean_min_mse_test = mean(mse_test_final)
155   ds_min_mse_test = std(mse_test_final)
```

Additionally, the following function, `Kernel_m.m` is used to generate the Gaussian kernel:

```
1  function fcn = Kernel_m(x1,x2,v)
2  % Gaussian Ridge Regression Kernel
3  % function calculates gaussian kernel using Eq.11 from assignmnet handout
4  % Input:
5  % x1 training data (x1 and x2 should have same no of columns)
6  % x2 validation data
7  % sigma is the variance parameter of gaussian kernel
8  % Output:
9  % Kernel matrix (can b inspected by imagesc(K))
10
11 x = [x1; x2];
12 %xt=x';
13
14 [m, ~] = size(x);
15
16 fcn = zeros(m,m);
17
18 for i=1:m
19     for j=1:m
20         fcn(i,j) = exp(-(norm(x(i,:)-x(j,:),2).^2)/(2*v.^2));
21     end
22 end
```

Below is a table of summary results for Exercises 9 and 10.

Part II

Question 1

To train a classifier for two-class data, consider performing linear regression with the Gaussian kernel,

$$K_\beta(\mathbf{x},\mathbf{t}) = \exp(-\beta||\mathbf{x}-\mathbf{t}||_2^2) = \exp\Big(-\frac{||\mathbf{x}-\mathbf{t}||_2^2}{2\sigma^2}\Big),$$

where $\sigma > 0$ is the kernel 'width'. We can see that that the role of β (or σ) is to amplify the distance between $\mathbf{x}$ and $\mathbf{t}$. $K_\beta(\mathbf{x},\mathbf{t})$ tends to zero if the distance between $\mathbf{x}$ and $\mathbf{t}$ is much greater than σ; if we select β to be very large (and σ very small), only the points within a certain distance of $\mathbf{x}$ will affect a prediction. A large value of β, (small σ), results in a larger variance - averaging over less points - and smaller bias - using closer instances - tending to overfit with a stricter decision boundary.

	MSE_{train}	MSE_{test}
Naive Regression	82.7 ± 6.0	88.1 ± 12.1
Attribute 1 (LR)	71.7 ± 5.5	72.5 ± 10.8
Attribute 2 (LR)	72.9 ± 5.4	77.2 ± 9.8
Attribute 3 (LR)	61.9 ± 5.0	67.8 ± 11.2
Attribute 4 (LR)	80.3 ± 5.3	84.6 ± 11.4
Attribute 5 (LR)	67.7 ± 5.3	71.5 ± 10.6
Attribute 6 (LR)	42.4 ± 3.7	42.1 ± 7.5
Attribute 7 (LR)	72.8 ± 5.2	74.0 ± 12.5
Attribute 8 (LR)	77.7 ± 6.1	82.1 ± 12.2
Attribute 9 (LR)	70.9 ± 5.3	73.5 ± 11.8
Attribute 10 (LR)	64.4 ± 5.5	68.4 ± 10.2
Attribute 11 (LR)	63.4 ± 4.7	64.2 ± 9.3
Attribute 12 (LR)	74.5 ± 5.8	77.3 ± 11.2
Attribute 13 (LR)	36.8 ± 2.7	39.9 ± 5.3
Kernel Ridge Regression	7.45 ± 1.48	13.76 ± 3.73

Table 8: Final summary table of MSEs for Exercises 9 and 10

In brief, in Gaussian kernel regression, nearby points have much greater contributions to predictions. β (or σ) determine how rapidly neighbouring influences fall off with distance. **Setting $\beta \to \infty$ ($\sigma \to 0$), leads to a more local prediction (less neighbours weigh in) tending to 1-NN - locally constant, piecewise constant labelling**. Conversely, $\beta \to 0$, $\sigma \to \infty$ leads to a global majority prediction with all neighbours having the same weight (small variance, large bias, less overfitting but over-smoothed decision boundary).

Question 2

(a) We may wish to have a non-zero threshold for our classifier. Introducing a bias value allows us to classify positively if $f_{\mathbf{w},\mathbf{b}}(\mathbf{x}) > \theta$ for a given θ. The perceptron training procedure now operates as follows:

Again, one starts by initialising a weight vector $\mathbf{w}_0$, usually to $\mathbf{w}_0 = \mathbf{0}$. This time, one must also initialise a bias $b_0 = 0$. Assuming the training samples $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)\}$ are linearly separable, we iterate 'online', processing one example at a time over each epoch. For each example, one checks if y_i has been misclassified (i.e. $y_i(\langle \mathbf{w}_k \cdot \mathbf{x}_i + b_k \rangle \leq 0$, where index k is the number of mistakes made). If misclassified, besides updating our weights (directly in the primal form that is $\mathbf{w}_{k+1} = \mathbf{w}_k + \eta y_i \mathbf{x}_i$, where $\eta \in \mathbb{R}^+$ is a specified learning rate), we must also update our bias value. In the update step, $b_{k+1} = b_k + \eta y_i R^2$, where R denotes the maximum norm of an input vector (we can set $\eta = 1$ and have $R^2 = 1$ when computing). We keep track of the mistakes made ($k = k + 1$ if wrong classification) and keep iterating, updating $\mathbf{w}_{k+1}$ and b_{k+1} until there are no mistakes in an epoch. This modified process returns a parameter for the trained bias b_k in addition to the weights $\mathbf{w}_k$. Geometrically, the bias b_k shifts away the decision boundary in the direction of $\mathbf{w}_k$.

(b) The bound on the number of mistakes increases.

Let R denote the maximum norm of an input vector $\mathbf{x}$, $R = \max_{1 \leq i \leq l} ||\mathbf{x}_i||$, where l is the number of training examples in a linearly separable training set S. The hard margin weight vector is denoted $\mathbf{w}_{opt}$, such that $||\mathbf{w}_{opt}|| = 1$, with:

$$y_i(\langle \mathbf{w}_{opt} \cdot \mathbf{x}_i \rangle + b_{opt}) \geq \gamma,$$

for $1 \leq i \leq l$, where γ is the geometric margin. When Novikoff's Theorem is given for zero bias, $b_{opt} = 0$ the number of mistakes/updates is bounded by,

$$\frac{R^2}{\gamma^2}.$$

This result arises from 2 inequalities - upper and lower bound on $||\mathbf{w}_t||$. Consider these (they can be seen in slides 35-37 of Lecture 3. Online Learning). Now, consider incorporating the bias term and it being updated in the algorithm (standard update - no R^2 factor). The number of iterations will depend on the margin of the weight training set (that is including bias - augmented). This new margin $\gamma_{new} = \gamma$ if the optimal bias is zero. In any case, γ_{new} is always lesser than or equal to γ, potentially being smaller. Hence, the bound

of the augmented margin can be worse (specifically four times worse) than that specified by Novikoff for $b_opt = 0$. Accounting for the bias, the number of mistakes is bounded by $\left(\frac{2R}{\gamma}\right)^2$.

Question 3

(a) Consider we have: $K_c(\mathbf{x}, \mathbf{z}) = c + \sum_{i=1}^{n} x_i z_i = c + \mathbf{x}^T z$. $\mathbf{x}^T z = \sum_{i=1}^{n} x_i z_i$ is positive semidefinite. Kernel $K_c(\mathbf{x}, \mathbf{z})$ is positive semi-definite if every value is positive semi-definite and it is symmetric. Since a linear combination of positive semi-definite values is also positive semi-definite, $c >= 0$ i.e. c must be positive semi-definite. Alternatively, note that $K_c(\mathbf{x}, \mathbf{z})$ is positive semidefinite for $c \geq 0$ since we are dealing with a reduction of $\mathbf{a}^T K_c(\mathbf{x}, \mathbf{z})\mathbf{a} = \sum_{i,j} a_i a_j K_c(\mathbf{x}, \mathbf{z})$.

(b)) Suppose we use $K_c(\mathbf{x}, \mathbf{z})$ as a kernel function with linear regression. We will have the following dual solution to such:

$$y_{pred} = \sum_{i=1}^{n} \alpha_i k_c(x, x_j),$$

where $\alpha = (K_c + \lambda \mathbf{I})^{-1}$. c influences the solution by shifting away the decision boundary in the direction of $\mathbf{w}$. A bias shifts the prediction y_{pred}.

Question 4

(a)

Least Squares

The implementation of the least squares regression classifier is presented below as two functions. These are `least_squares_w.m` and `least_squares_pred.m`

`least_squares_w.m` learns the weights from the training examples for least squares regression classification. Note we can, in this underdetermined case, use `weights = pinv(x_train)*y_train` to calculate the weights of minimal norm consistent with the data. Efficiency of implementation is not the focus. Inputs to the function are the training examples `X_train` and their labels `y_train`. The output is the learned weights `weights`.

`least_squares_pred.m` utilises the weights learned by the previous function to predict the labels of a test set for least squares regression classification. It intakes the test examples `x_test` and the learned weights `w_learned.m`. It outputs the predicted labels `y_pred` of the test examples. The regression vector $\mathbf{w}$ (`w_learned`) defines the classifier $f_{\mathbf{w}} = sign(\mathbf{x}^T\mathbf{w})$ (`y_pred = sign(x_test*w_learned)`. If this expression is false, a predicted label is assigned the value -1.

`least_squares_w.m`

```
function weights=least_squares_w(x_train,y_train)
% least_squares_w.m computes the weights for LS regression
% classification. inputs: x_train - training examples,
% y_train - training labels
    % may use pinv to compute w of consistent minimal norm
    % efficiency of implementation not focus
    weights = pinv(x_train)*y_train;
end
```

`least_squares_pred.m`

```
function y_pred = least_squares_pred(x_test,w_learned)
% least_squares_pred.m - predicts labels for LS regression
% classification using learned weights output of least_squares_w.m
% inputs: x_test -> test examples, w_learned-> weights learned from
% the training set. output: y_predict -> predicted labels for test
%set.
    % regression vector w defines classifier. f_w = sign(x^T*w)
    y_pred = sign(x_test*w_learned);
    if y_pred == 0;
```

```
        y_pred = -1;
    end
end
```

Perceptron

The implementation of the perceptron algorithm training is presented below as `perceptron_w.m`. This function intakes the training examples `x_train`, a $m \times n$ matrix, and their respective true labels `y_train`, a $m \times 1$ vector, as inputs. It additionally intakes a vector of weights, intialised to ones in the main code. The algorithm operates 'on-line', operating over one example at a time. An example is misclassified if $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b_k) \leq 0$ (`y_train(j)*y_pred <= 0` with `y_train(j)*y_pred <= 0`) . In this case the bias is not incorporated into the equation and is assumed to be zero. The weights are updated only if a training example is mislabelled by the current weights; here these are updated directly (primal form). In that case, we compute $\mathbf{w}_{k+1} = \mathbf{w}_k + \eta y_i \mathbf{x}_i$ (`weights = weights + y_train(j)*x_train(j,:)`), where η is a learning rate (here set to 1) and k indexes the number of mistakes made. This procedure is only iterated over one epoch; due to how we assign ore labels (first column of the training examples) the system is quickly trained.

```
function  weights = perceptron_w(x_train,y_train,weights)
% perceptron_w.m computes the training weights of the perceptron
% algorithm. inputs: x_train - training examples,
% y_train - training labels. weights - initialised to ones.
    for j = 1:size(x_train,1); % online algorithm. one eg at time
        % only one epoch needed - trains quickly
        y_pred = sign(x_train(j,:)*weights'); % y_pred
        if y_train(j)*y_pred <= 0; % if misclassification, update
            weights = weights + y_train(j)*x_train(j,:); % weight update
        end
    end
    return
end
```

The function `perceptron_pred.m` utilises the weights learned by the previous function to predict the labels of a test set for the perceptron algorithm. It intakes the test examples `x_test` and the learned weights `w_learned.m`. It outputs the predicted labels `y_pred` of the test examples. The weight vector $\mathbf{w}$ (`w_learned`) defines the classifier $f_{\mathbf{w}} = sign(\mathbf{x}^T\mathbf{w})$ (`y_pred = sign(x_test*w_learned`). If this expression is false, a predicted label is assigned the value -1.

```
function y_pred = perceptron_pred(x_test,w_learned)
% perceptron_pred.m - predicts labels for perceptron
% classification using learned weights output of least_perceptron_w.m
% inputs: x_test -> test examples, w_learned-> weights learned from
% the training set. output: y_predict -> predicted labels for test
% set.
y_pred = sign(x_test*w_learned');
if y_pred == 0;
    y_pred = -1;
end
```

Winnow

The implementation of the winnow algorithm training is presented below as `winnow_w.m`. This function intakes the training examples `x_train`, a $m \times n$ matrix, and their respective true labels `y_train`, a $m \times 1$ vector, as inputs. Note the algorithm has been adapted for use with $\{-1, 1\}^n$ for the patterns and $\{-1, 1\}$ for the labels. α, the promotion/demotion parameter has been set to one. The winnow algorithm is very similar to the perceptron algorithm but uses a multiplicative instead of an additive scheme.

In this algorithm, weights are initialised as a vector of zeros outside of the code. The `threshold`, Θ is initialised with a value of n (number of training examples) - this gives good bounds for linear separability. $y_{pred} = 0$ if $\mathbf{w}_t \cdot \mathbf{x}_t < \Theta = n$ and $y_{pred} = 0$ otherwise. This is reflected in the code by the condition `y_pred = sign(x_train(i,:)*weights'-threshold)`. If the label is misclassified (`y_train(i) != y_pred`), the weights are updated under a multiplicative scheme $\mathbf{w}_{t+1,i} = \mathbf{w}_{t,i} 2^{(y_t - y_{pred})^2 x_{t,i}}$. This procedure is only

iterated over one epoch; due to how we assign ore labels (first column of the training examples) the system is quickly trained.

```
function weights = winnow_w(x_train,y_train,weights)
threshold = size(x_train,1);
% winnow_w.m computes the training weights of the winnow
% algorithm. inputs: x_train - training examples,
% y_train - training labels. weights - initialised to zeros.
% in main code
    % trains quickly - only one epoch needed
    for i = 1:size(x_train,1) % online algorithm one eg at a time
        y_pred = sign(x_train(i,:)*weights'-threshold);
        if y_train(i) ~= y_pred; % if misclassification, update
            % multiplicative scheme
            weights = weights .* 2.^((y_train(i)-(y_pred>=0)) ...
                                        .*x_train(i,:));
        end
    end
    return
end
```

The function `winnow_pred.m` utilises the weights learned by the previous function to predict the labels of a test set for the winnow algorithm. It intakes the test examples `x_test` and the learned weights `w_learned.m`. It outputs the predicted labels `y_pred` of the test examples. The weight vector $\mathbf{w}$ (`w_learned`) defines the classifier $f_{\mathbf{w}} = sign(\mathbf{x}^T\mathbf{w} - \Theta)$ (`y_pred = sign(x_test*w_learned - n)`. If this expression is false, a predicted label is assigned the value -1.

```
function y_pred = winnow_pred(x_test,w_learned)
% winnow_pred.m - predicts labels for winnow
% classification using learned weights output of winnow_w.m
% inputs: x_test -> test examples, w_learned-> weights learned from
% the training set. output: y_pred-> predicted labels for test
% set.
    % regression vector w defines classifier. f_w = sign(x^T*w)
    threshold = size(x_test,1);
    y_pred = sign(x_test*w_learned'-threshold);
    if y_pred == 0;
        y_pred = -1;
    end
end
```

1-NN

For 1-nearest-neighbour classification, the built-in MATLAB function `fitcknn.m` is used with `NumNeighbors` set to 1. This function returns 1-nearest neighbor classification coefficients based on the predictor data `X_train` and response `Y_train`. The function is called in the main program as follows:

```
Mdl = fitcknn(X_train,Y_train,'NumNeighbors',1,'Standardize',1);
y_pred = predict(Mdl,Xtest);
```

Generalisation Error

The function `accuracy.m` is implemented to calculate the generalisation error between y_{train} and y_{pred} for a given classifier. The generalisation error is given as a value between 0 and 1, corresponding to the mean number of mistakes. A mistake counts 1, a correct label counts 0. The generalisation error will reach 10% at a mean value of 0.1 mistakes.

```
function generalisation_err = accuracy(x_train,y_pred)
% accuracy.m computes the generalisation error between
% y_train and y_pred for a given classifier.
% inputs -> x_train: the training examples, recall that their
```

```
% true labels are given by x_train(:,1) - first col.
% y_pred -> predicted labels for examples
% output -> generalisation_err for a given classifier.
    generalisation_err = mean(x_train(:,1)~=y_pred);
end
```

Plots

Figure 11 shows a plot of sample complexity for least squares regression. The vertical axis represents m, the minimum number of examples to incur no more than 10% generalisation error. The horizontal axis shows the number of dimensions, in this case denoted n.

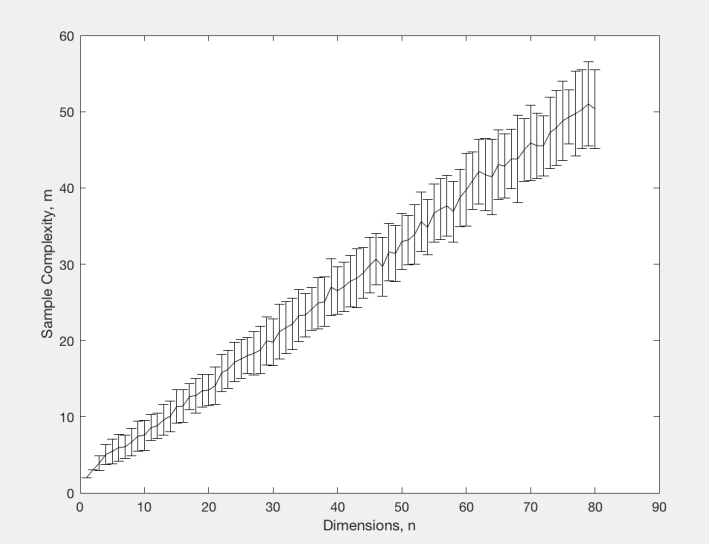

Figure 11: Plot of sample complexity for LSR with error bars. 80 dimensions total, 50 trials.

Figure 12 shows a plot of sample complexity for the perceptron algorithm. The vertical axis represents m, the minimum number of examples to incur no more than 10% generalisation error. The horizontal axis shows the number of dimensions, in this case denoted n.

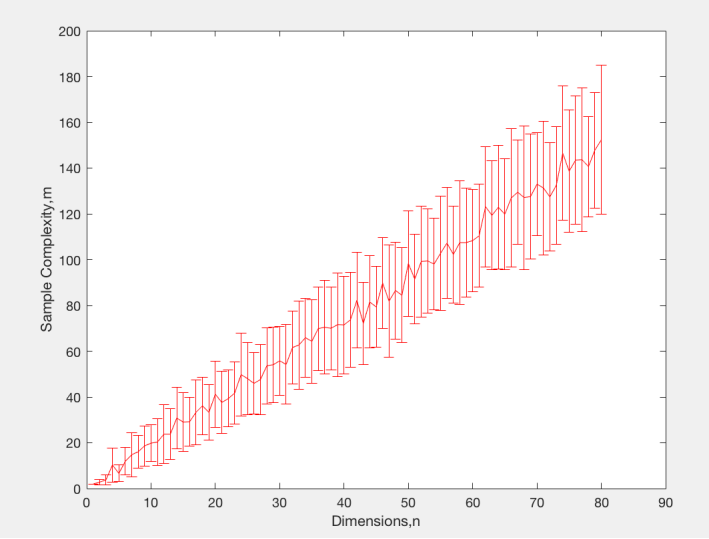

Figure 12: Sample complexity for the perceptron algorithm with error bars (SD). 80 dimensions total, 50 trials.

Figure 13 shows a plot of sample complexity for the winnow algorithm. The vertical axis represents m, the minimum number of examples to incur no more than 10% generalisation error. The horizontal axis shows the number of dimensions, in this case denoted n.

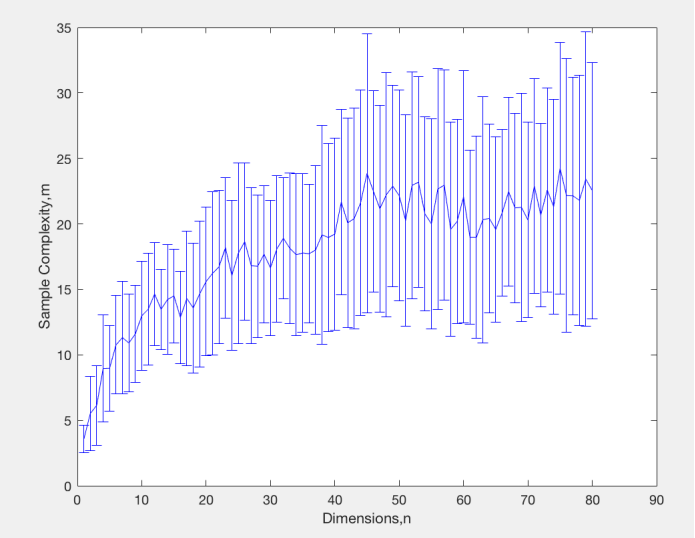

Figure 13: Sample complexity for the winnow algorithm with error bars (SD). 80 dimensions total, 50 trials.

Figure 14 shows the sample complexity for the one-nearest neighbour algorithm. The vertical axis represents m, the minimum number of examples to incur no more than 10% generalisation error. The horizontal axis shows the number of dimensions, in this case denoted n. In this case, only one trial has been performed and we have iterated over only 25 dimensions.

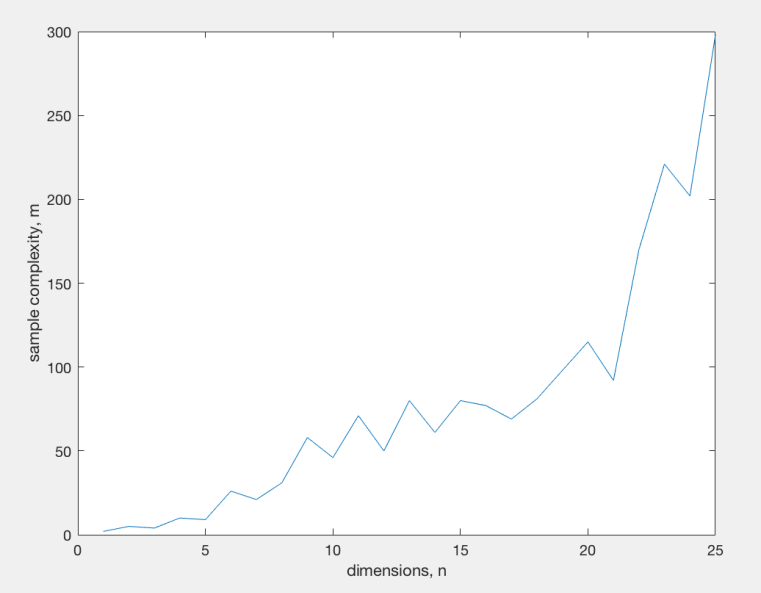

Figure 14: Sample complexity for the 1-NN algorithm. 25 dimensions total, 1 trial.

(b) The main program implemented for Question 4 is presented below: `secondpart.m`

```
clear all;
max_dimensions = 80;
n_trials = 50; % number of iterations for averaging
low_dim_limit = 30;
mid_dim_limit = 50;
high_dim_limit = 65;
low_multiplier = 60;
mid_multiplier = 40;
high_multiplier = 30;
higher_multiplier = 20;
rng(45) % set random seed
```

```
12
13 for dimension = 1:max_dimensions
14     if dimension <= low_dim_limit;
15         n_examples = dimension*low_multiplier;
16     elseif dimension > low_dim_limit && dimension <=mid_dim_limit
17         n_examples = dimension*mid_multiplier;
18     elseif dimension > mid_dim_limit && dimension <=high_dim_limit
19         n_examples = dimension*high_multiplier;
20     else
21         n_examples = dimension*higher_multiplier;
22     end
23     x = zeros(n_examples,dimension,n_trials);
24     for trial = 1:n_trials     % iterations for averaging
25         % matrix generation -1s and 1s
26         x(:,:,trial) = ones(n_examples,dimension) ...
27                             - floor(rand(n_examples,dimension)*2)*2;
28         % PERCEPTRON
29         error = 10^8;   % initialisation
30         example = 1; % initialisation
31         while error>0.10
32             weights_PERC = ones(1,dimension); % weights init
33             x_train = x(1:example,:,trial);
34             y_train = x_train(:,1);
35             x_test = x(example+1:end,:,trial);
36             weights_PERC = perceptron_w(x_train,y_train,weights_PERC);
37             y_pred_PERC = perceptron_pred(x_test,weights_PERC);
38             error = accuracy(x_test,y_pred_PERC);
39             example = example+1;
40         end
41         % sample complexity
42         error_PERC(dimension,trial) = error;
43         example_PERC(dimension,trial) = example;
44         % winnow
45         error = 10^8; % initialisation
46         example = 1; % initialisation
47         while error>0.1
48             weights_WINNOW= zeros(1,dimension);  % initialisation
49             weights_WINNOW(1,:) = ones(1,dimension); % initialisation
50             x_train = x(1:example,:,trial);
51             y_train = x_train(:,1);
52             x_test = x(example+1:end,:,trial);
53             weights_WINNOW_new = winnow_w(x_train,y_train,weights_WINNOW);
54             y_pred_WINNOW = winnow_pred(x_test,weights_WINNOW_new);
55             error = accuracy(x_test,y_pred_WINNOW);
56             example = example+1;
57         end
58         % SAMPLE COMPLEXITY
59         error_winnow(dimension,trial) = error;
60         example_WINNOW(dimension,trial) = example;
61         % LEAST SQUARES
62         error = 10^8; % initialisation
63         example = 1; % initialisation
64         while error>0.1
65             x_train = x(1:example,:,trial);
66             y_train = x_train(:,1);
67             x_test = x(example+1:end,:,trial);
68             weights_LS = least_squares_w(x_train,y_train);
69             y_pred_LS = least_squares_pred(x_test,weights_LS);
70             error = accuracy(x_test,y_pred_LS);
```

```
71              example = example+1;
72          end
73          %sample complexity
74          error_LS(dimension,trial) = error;
75          example_LS(dimension,trial) = example;
76      end
77      if mod(dimension,5) == 0
78              disp('Generalisation error calculated for # dimensions: ');
                    dimension
79      end
80  end
81
82  dimensions = 1:max_dimensions;
83
84  % plots
85  figure
86  errorbar(dimensions,mean(example_PERC,2),std(example_PERC'),'-r');
87  xlabel('Dimensions,n');
88  ylabel('Sample Complexity,m');
89  hold on;
90
91  figure;
92  errorbar(dimensions,mean(example_WINNOW,2),std(example_WINNOW'), '-b');
93  xlabel('Dimensions,n');
94  ylabel('Sample Complexity,m');
95  hold on;
96
97  figure;
98  errorbar(dimensions,mean(example_LS,2),std(example_LS'), '-k');
99  xlabel('Dimensions, n');
100 ylabel('Sample Complexity, m');
101
102 % K-NN
103 % ntrials = 1;
104 % max_dimensions = 25
105 % for dimension = 1:max_dimensions
106 %   if dimension <= low_dim_limit;
107 %         n_examples = dimension*low_multiplier;
108 %   elseif dimension > low_dim_limit && dimension <=mid_dim_limit
109 %         n_examples = dimension*mid_multiplier;
110 %   elseif dimension > mid_dim_limit && dimension <=high_dim_limit
111 %         n_examples = dimension*high_multiplier;
112 %   else
113 %         n_examples = dimension*higher_multiplier;
114 %   end
115 %   x = zeros(n_examples,dimension,n_trials);
116 %   for trial = 1:n_trials    % iterations for averaging
117         % matrix generation -1s and 1s
118 %        x(:,:,trial) = ones(n_examples,dimension) ...
119 %                             - floor(rand(n_examples,dimension)*2)*2;
120
121 %        error = 10^8;   % initialisation
122 %        example = 1; % initialisation
123 %        while error>0.1
124 %        x_train = x(1:example,:,trial);
125 %        y_train = x_train(:,1);
126 %        x_test = x(example+1:end,:,trial);
127 %        Mdl = fitcknn(x_train,y_train,'NumNeighbors',1,'Standardize',1);
128 %        y_pred = predict(Mdl,x_test);
```

```
%
%           error = accuracy(x_test,y_pred);
%           example= m+1;
%       end
%       example_NN(dimension,trial) = example;
%       error_NN(dimension,trial) = error;

% end

% dimensions = 1:25
% figure;
% (dimensions,mean(example_NN,2));
% xlabel('dimensions',n);
% ylabel('Sample Complexity, m');
```

(b) Effectively, computing the sample complexity "exactly" by summation would be extremely expensive computationally. In fact, it is already quite expensive for higher dimensions (the code takes around 2-3 minutes to run). We must use a fixed number of test data samples; noting that in this manner sampling biases will be introduced.

We use several mechanisms to acheive an appropriate trade-off between accuracy and computation time. Firstly we set the number of trials `trials` to 50, and the number of `dimensions` to 80. Averaging over many trials allows us to average over sampling biases and obtain a closer representation of the 'true' complexity. We introduce several tiers of `multipliers`. These are the values by which the number of dimensions are multiplied to obtain of the number training examples used. In order to make the program less computationally expensive we set lower values for the multiplier as we iterate over higher dimensions. The code is heavily optimised to run quickly, splitting up weight learning and label prediction into different steps. This explanation is complemented by the methods explanation in **(a)**.

(c) Based on Figures 10-13, it appears that for the winnow algorithm $m = \Theta(n \log n)$, for least squares regression $m = \Theta(n)$, for the perceptron algorithm $m = \Theta(n)$ and for 1-NN , $m = \Theta(2^n)$.

(d)-

(e)-