CAB420 Assignment 1

Aidan Kinzett (n9699210)

April 21, 2019

1 Theory

Given the following equation,

$$L(w) = -\sum_{i=1}^{N} log(\frac{1}{1+e^{y_i(w^T x_i + b)}}) + \lambda ||w||_2^2$$

1.1 Finding the Partial Derivative

Finding the partial derivative

$$\frac{\partial L}{\partial w_j}$$

$$\frac{\partial L}{\partial w_j}(-\sum_{i=1}^{N} log(\frac{1}{1+e^{y_i(w^T x_i + b)}}) + \lambda ||w||_2^2)$$

As the part at the end of the function is a squared L2 norm, it can be derived as below.

$$\frac{\partial L}{\partial w_j}(-\sum_{i=1}^{N} log(\frac{1}{1+e^{y_i(w^T x_i + b)}})) + 2\lambda w_j$$

Simplify,

$$\frac{\partial L}{\partial w_j}(\sum_{i=1}^{N} log(1+e^{y_i(w^T x_i + b)})) + 2\lambda w_j$$

Chain rule,

$$\sum_{i=1}^{N} \frac{\frac{\partial L}{\partial w_j}(1+e^{y_i(w^T x_i + b)})}{1+e^{y_i(w^T x_{i,j} + b)}} + 2\lambda w_j$$

$$\sum_{i=1}^{N} \frac{\frac{\partial L}{\partial w_j}(1) + \frac{\partial L}{\partial w_j}(e^{y_i(w^T x_i + b)})}{1+e^{y_i(w^T x_{i,j} + b)}} + 2\lambda w_j$$

$$\sum_{i=1}^{N} \frac{\frac{\partial L}{\partial w_j}(e^{y_i(w^T x_i + b)})}{1+e^{y_i(w^T x_{i,j} + b)}} + 2\lambda w_j$$

Chain rule,

$$\sum_{i=1}^{N} \frac{y_i x_i e^{y_i(w^T x_i + b)}}{1+e^{y_i(w^T x_{i,j} + b)}} + 2\lambda w_j$$

1.2 Finding the Second Partial Derivative

Finding the second partial derivative

$$\frac{\partial^2 L}{\partial w_j \partial w_k}$$

$$\frac{\partial^2 L}{\partial w_j \partial w_k} = \frac{\partial}{\partial w_k}(\sum_{i=1}^{N} \frac{y_i x_i e^{y_i(w^T x_i + b)}}{1 + e^{y_i(w^T x_{i,j} + b)}} + 2\lambda w_j)$$

$$\sum_{i=1}^{N} \frac{y_i x_i (\frac{\partial}{\partial w_k} e^{y_i(w^T x_i + b)})}{1 + e^{y_i(w^T x_{i,j} + b)}}$$

Using the chain rule,

$$\sum_{i=1}^{N} \frac{e^{y_i(w_k x_i + b)} y_i^2 x_i (\frac{\partial}{\partial w_k}(w^T x_i + b))}{1 + e^{y_i(w^T x_{i,j} + b)}}$$

$$\sum_{i=1}^{N} \frac{e^{y_i(w^T x_i + b)} y_i^2 x_i w_k x_i + b}{1 + e^{y_i(w^T x_{i,j} + b)}}$$

$$\sum_{i=1}^{N} \frac{e^{y_i(w^T x_i + b)} y_i^2 x_i^2 w_k + b}{1 + e^{y_i(w^T x_{i,j} + b)}}$$

1.3 Proving it's a convex function

A function can be defined as convex if the Hessian matrix is positive semi-definite. This is defined as,

$$a^T H a \equiv \sum_{j,k} a_j a_k H_{j,k} \geq 0$$

Where,

$$H_{j,k} = \frac{\partial^2 L}{\partial w_j \partial w_k}$$

This could be shown by contradiction.

2 Practice

2.1 Features, Classes, and Linear Regression (10 Marks)

a) Plot the training data in a scatter plot.

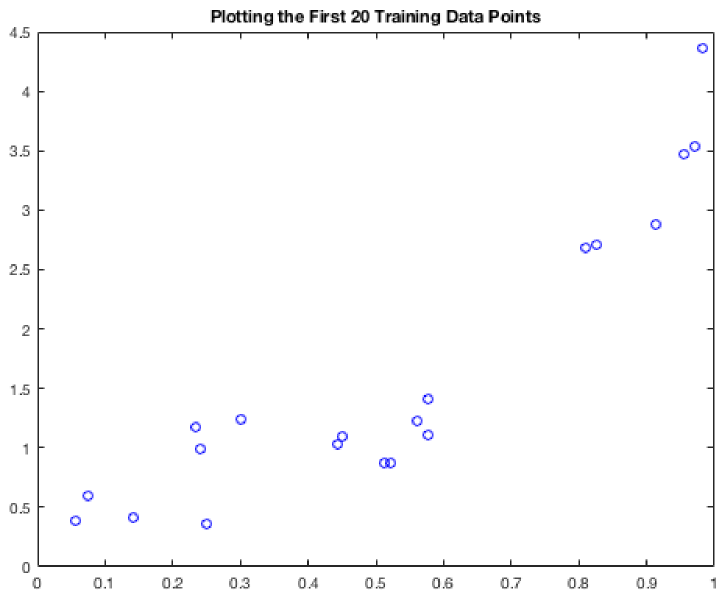

Figure 2.1.1: Training data in a scatter plot.

b) Create a linear regression learner using the above functions. Plot it on the same plot as the training data.

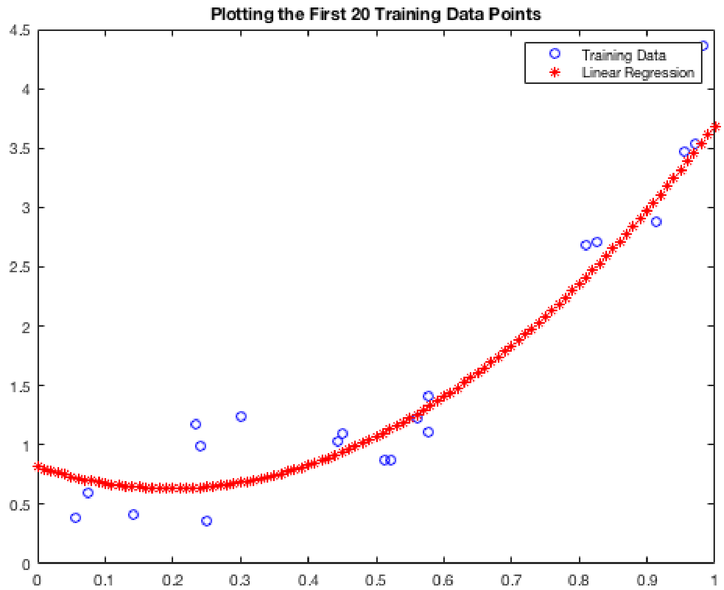

Figure 2.1.2: Training data in a scatter plot with linear regression learner

(c) Create plots with the data and a higher-order polynomial (3, 5, 7, 9, 11, 13).

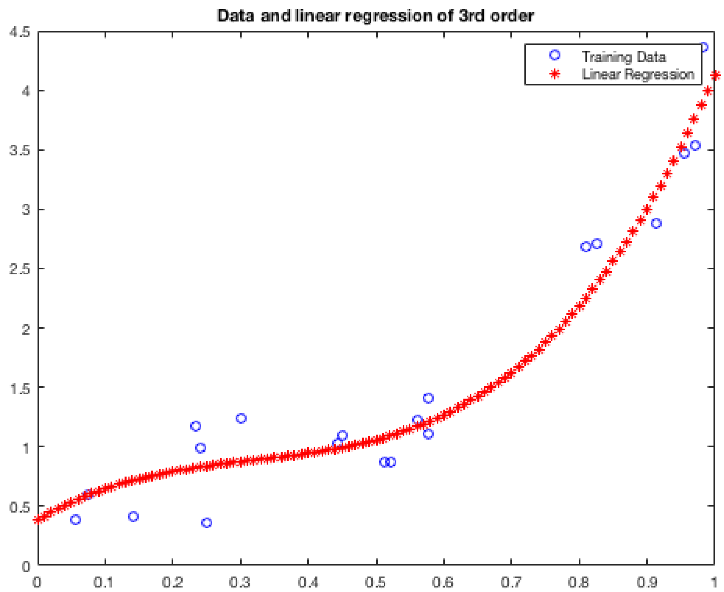

Figure 2.1.3: Training data in a scatter plot with 3rd order linear regression learner

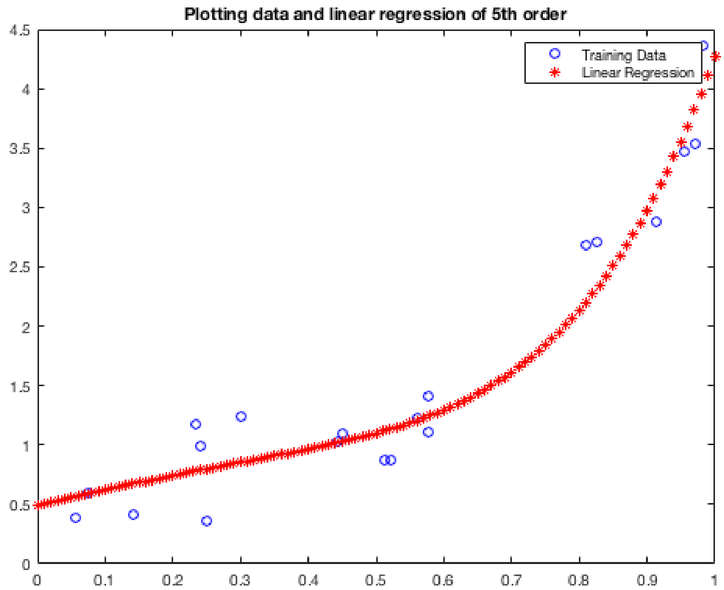

Figure 2.1.4: Training data in a scatter plot with 5th order linear regression learner

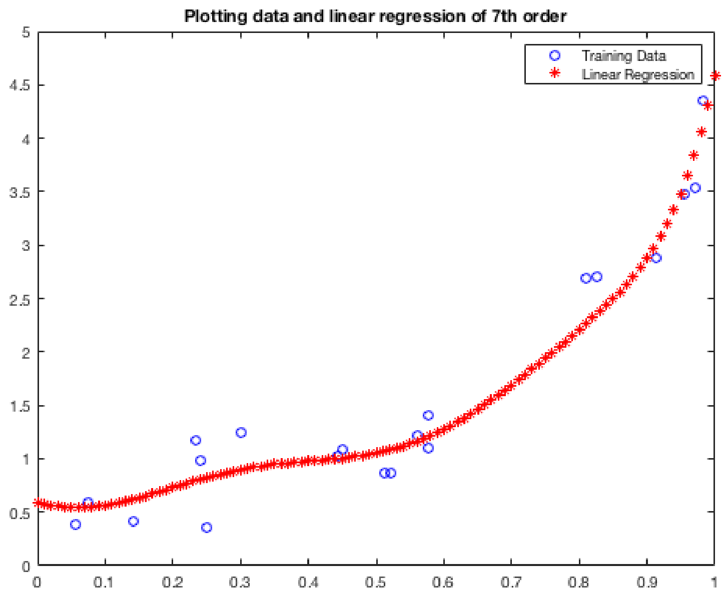

Figure 2.1.5: Training data in a scatter plot with 7th order linear regression learner

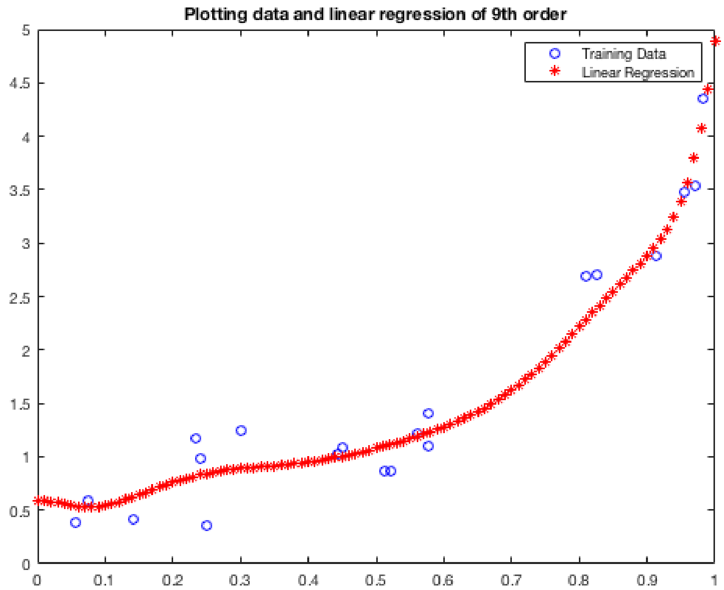

Figure 2.1.6: Training data in a scatter plot with 9th order linear regression learner

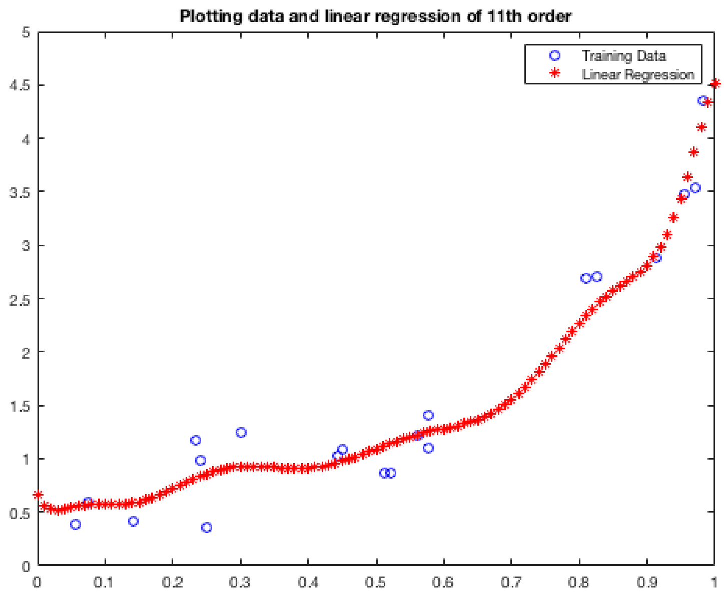

Figure 2.1.7: Training data in a scatter plot with 11th order linear regression learner

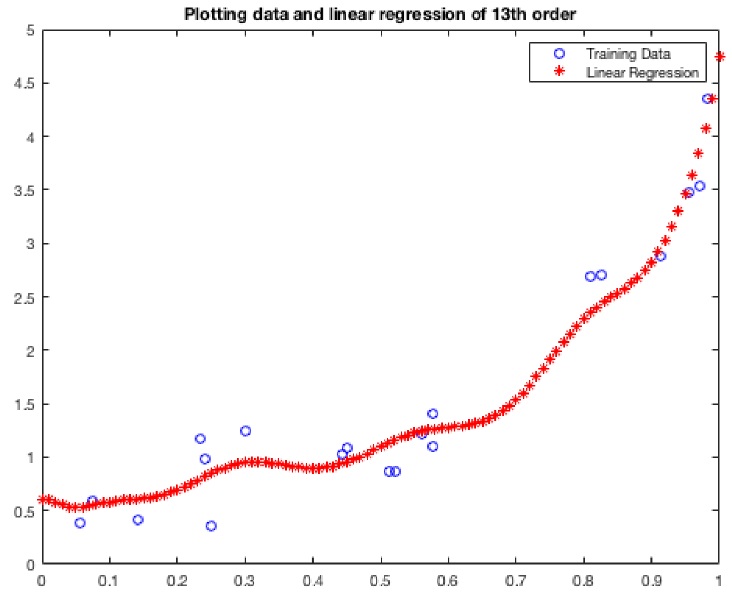

Figure 2.1.8: Training data in a scatter plot with 13th order linear regression learner

(d) Calculate the mean squared error (MSE) associated with each of your learned models on the training data.

```
Training MSE of order 2 regression
tr_err =
0.1092

Training MSE of order 3 regression
tr_err =
0.0828

Training MSE of order 5 regression
tr_err =
0.0813

Training MSE of order 7 regression
tr_err =
0.0783

Training MSE of order 9 regression
tr_err =
0.0771

Training MSE of order 11 regression
```

```
tr_err =
0.0756

Training MSE of order 13 regression
tr_err =
0.0750
```

(e) Calculate the MSE for each model on the test data (in mTestData.txt).

```
Testing MSE of order 2 regression
te_err =
0.0972

Testing MSE of order 3 regression
te_err =
0.0983

Testing MSE of order 5 regression
te_err =
0.0959

Testing MSE of order 7 regression
te_err =
0.1094

Testing MSE of order 9 regression
te_err =
0.1135

Testing MSE of order 11 regression
te_err =
0.1132

Testing MSE of order 13 regression
te_err =
0.1128
```

(f) Calculate the MAE for each model on the test data. Compare the obtained MAE values with the MSE values obtained in above (e).

```
MAE for order 2 regression
te_mae =
0.2599
```

```
MAE for order 3 regression
te_mae =
0.2777

MAE for order 5 regression
te_mae =
0.2745

MAE for order 7 regression
te_mae =
0.2889

MAE for order 9 regression
te_mae =
0.2919

MAE for order 11 regression
te_mae =
0.2940

MAE for order 13 regression
te_mae =
0.2925
```

The MSE on test data and the MAE both increase in the same manner with the increase in order. The MAE values are larger, which is to be expected. Visually, they both seem to be accurate representations of the amount of error, and the differences in values match the differences in the graphs.

(g) Don't forget to label your plots; see help legend. The graphs have appropriate labelling.

2.2 kNN Regression (15 Marks)

(a) Using the knnRegress class, implement (add code to) the predict function to make it functional.

```
   % Test function: predict on Xtest
   function Yte = predict(obj,Xte)
      [Ntr,Mtr] = size(obj.Xtrain);          % get size of training, test data
      [Nte,Mte] = size(Xte);

      classes = unique(obj.Ytrain);          % figure out how many classes & their labels

```

```
    Yte = repmat(obj.Ytrain(1), [Nte,1]); % make Ytest the same data type as Ytrain
    K = min(obj.K, Ntr);                  % can't have more than Ntrain neighbors
    for i=1:Nte,                          % For each test example:
      dist = sum( bsxfun( @minus, obj.Xtrain, Xte(i,:) ).^2 , 2); % compute sum of squared differences
      %dist = sum( (obj.Xtrain - repmat(Xte(i,:),[Ntr,1]) ).^2 , 2); % compute sum of squared differences
      [tmp,idx] = sort(dist);             % find nearest neighbors over Xtrain (dimension 2)
                                          % idx(1) is the index of the nearest point, etc.; see help sort

      Yte(i)=mean(obj.Ytrain(idx(1:K)));    % predict ith test example's value from nearest neighbors

    end;
  end
```

(b) Using the same technique as in Problem 1a, plot the predicted function for several values of k : 1, 2, 3, 5, 10, 50. How does the choice of k relate to the "complexity" of the regression function?

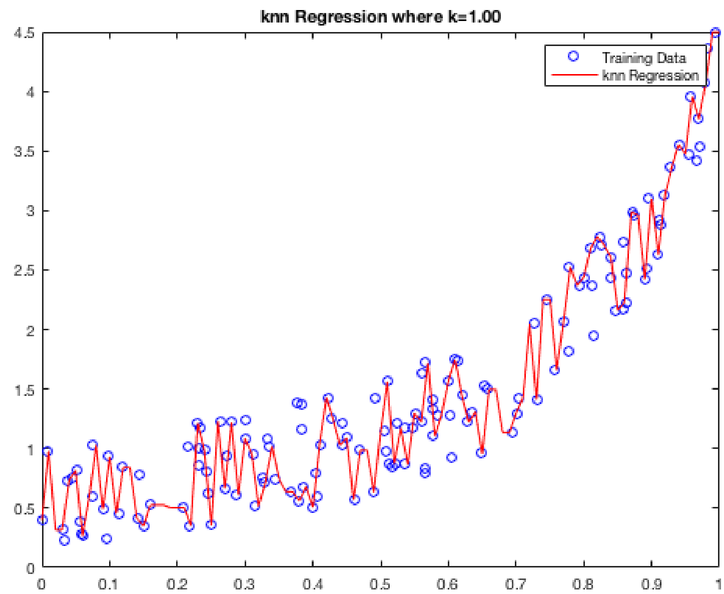

Figure 2.2.1: Predicted function for k=1.

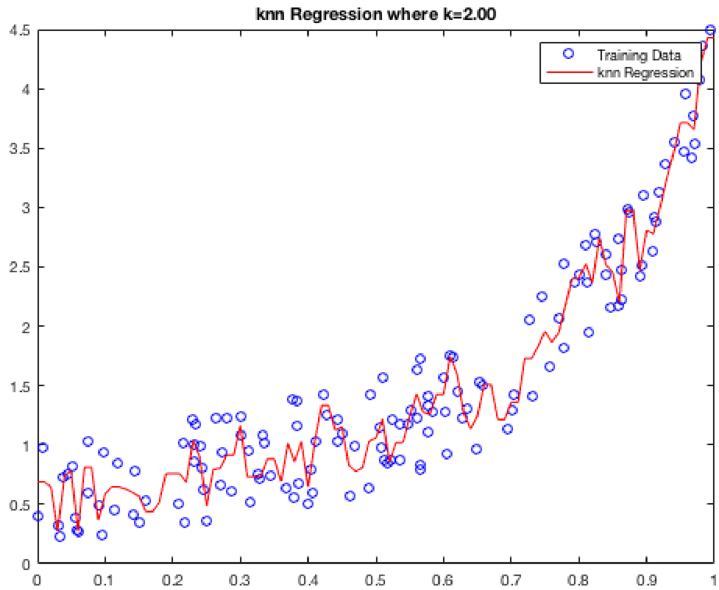

Figure 2.2.2: Predicted function for k=2.

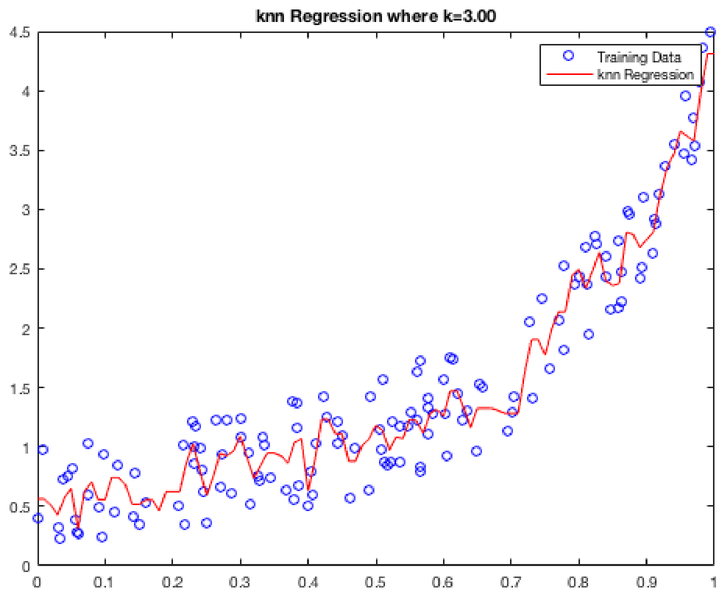

Figure 2.2.3: Predicted function for k=3.

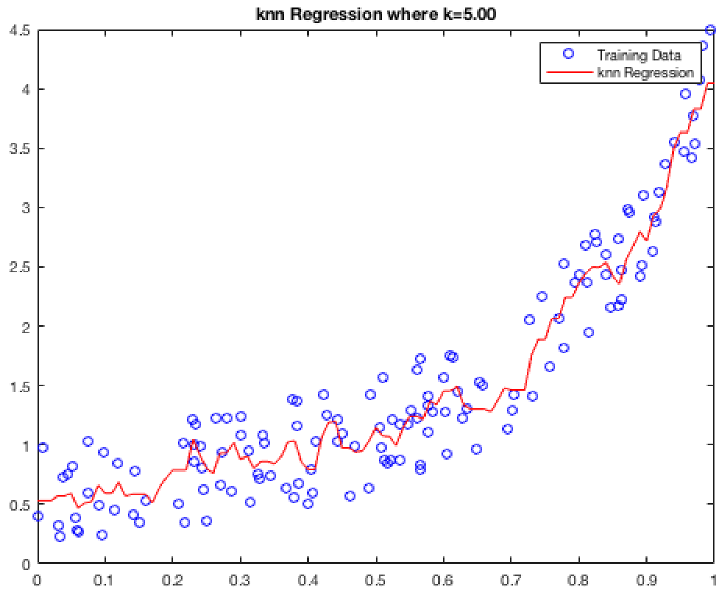

Figure 2.2.4: Predicted function for k=5.

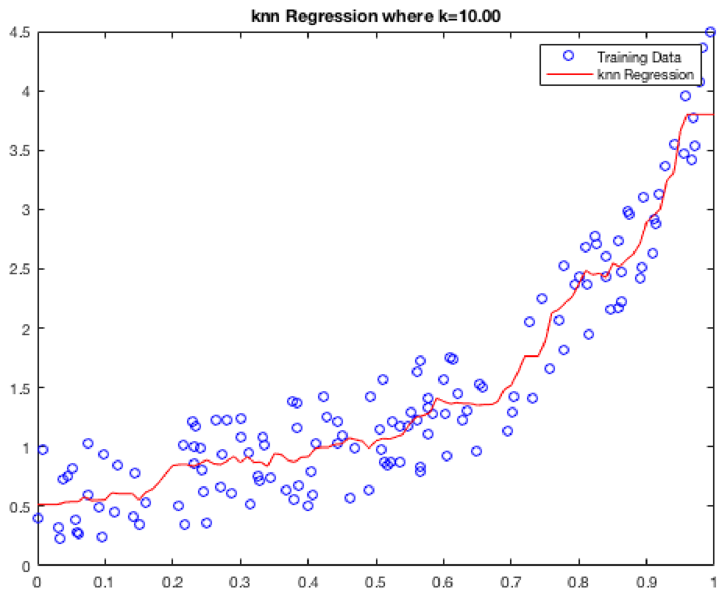

Figure 2.2.5: Predicted function for k=10.

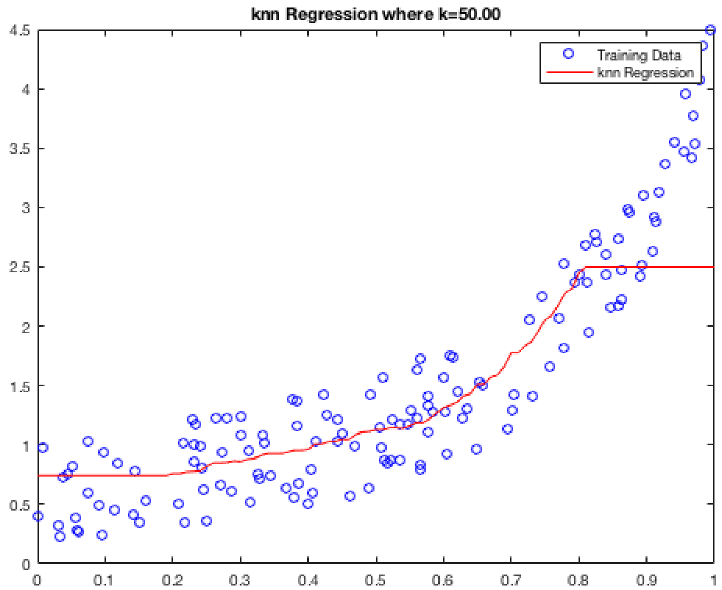

Figure 2.2.6: Predicted function for k=50.

The regression is more complex with lower k values, and less complex with higher k values. When k is low the regression is only looking at one piece of data, so jumps around with the data. When k is high it is looking at a lot of the data, and as a result doesn't follow the data very well.

(c) We discussed in class that the k-nearest-neighbor classifier's decision boundary can be shown to be piecewise linear. What kind of functions can be output by a nearest neighbor regression function? Briefly justify your conclusion. (You do not need to discuss the general case – just the 1-dimensional regression picture such as your plots.)

For most cases, including this one, the output function will be piecewise linear. It is linear as there is only one y value for each x value. Its piecewise because its not able to be plotted as a single function due to how complex it is.

2.3 Hold-out and Cross-validation (15 Marks)

(a) Similarly to Problem 1 and 2, compute the MSE of the test data on a model trained on only the first 20 training data examples for k = 1, 2, 3, . . . , 140. Plot both train and test MSE versus k on a log-log scale (see help loglog). Assign title to your figure (ie. 20 data) and legends to your curves (ie. test, train). Discuss what you observed from the figure.

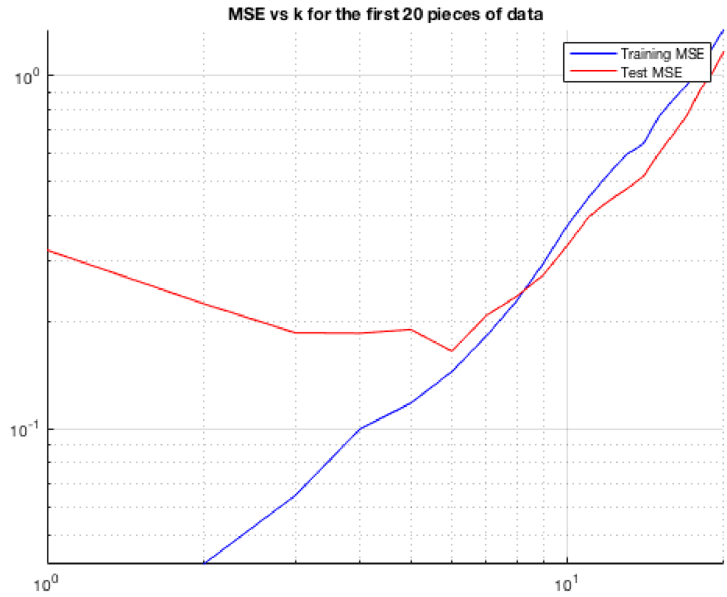

Figure 2.3.1: Train and test MSE versus k on a log-log scale.

The regression is overfitted to the training data with low k values, then as the k value goes up both sets of data because similarly well fitted, then start increasing in error.

(b) Repeat, but use all the training data. What happened? Contrast with your results from Problem 1 (hint: which direction is "complexity" in this picture?).

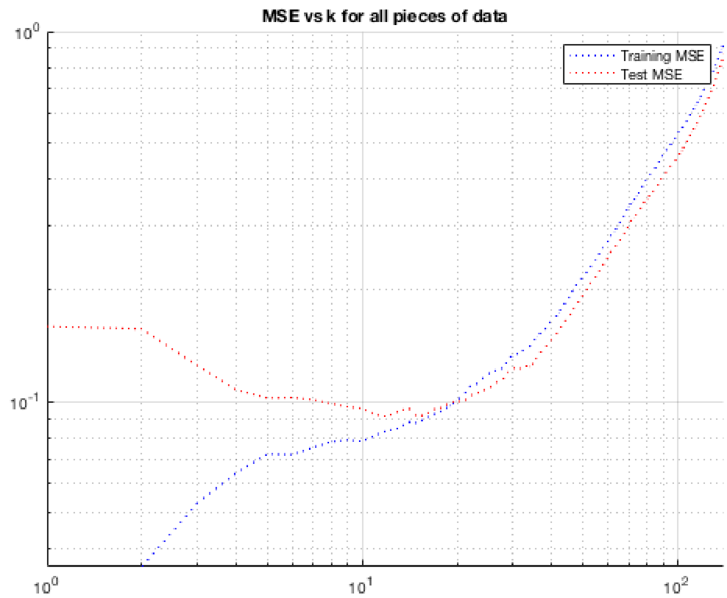

Figure 2.3.2: Train and test MSE versus k on a log-log scale.

The two sets of data start to have a steeper increase in MSE as k increases. The functions keep getting less and less complex, and increase in error.

(c) Using only the training data, estimate the curve using 4-fold cross-validation. Split the training data into two parts, indices 1:20 and 21:140; use the larger of the two as training data and the smaller as testing data, then repeat three more times with different sets of 20 and average the MSE. Plot this together with (a) and (b). Use different colors or marks to differentiate three scenarios. Discus why might we need to use this technique via comparing curves of three scenario?

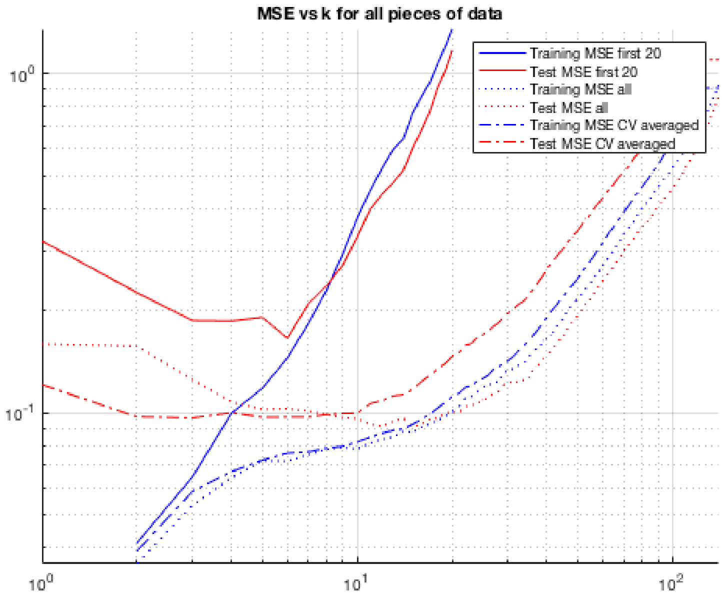

Figure 2.3.3: MSE of different techniques versus k on a log-log scale.

The cross validation allows us to get an estimated accuracy by averaging out the accuracy from multiple models.

2.4 Nearest Neighbor Classifiers (15 Marks)

(a) Plot the data by their feature values, using the class value to select the color.

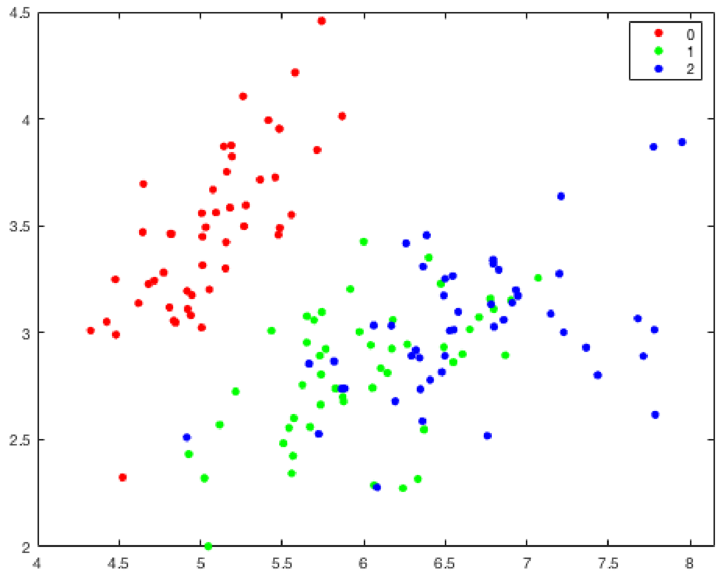

Figure 2.4.1:Plotted data by feature value with coloured classes

(b) Use the provided knnClassify class to learn a 1-nearest-neighbor predictor. Use the function class2DPlot(learner,X,Y) to plot the decision regions and training data together.

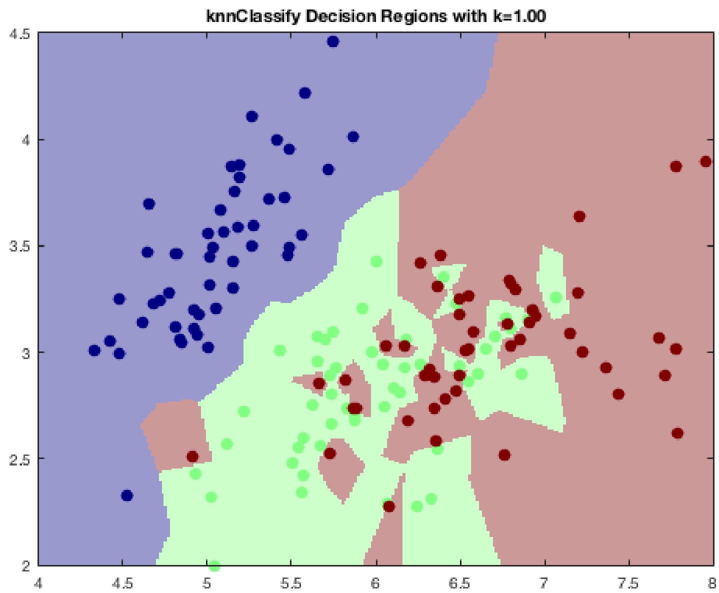

Figure 2.4.2:Plotted data with decision region for k=1

(c) Do the same thing for several values of k (say, [1, 3, 10, 30]) and comment on their appearance.

```
for k=[1, 3, 10, 30]
learner = knnClassify(k, X, Y);
class2DPlot(learner, X, Y);
title(sprintf('knnClassify Decision Regions with k=%.2f',k));
end
```

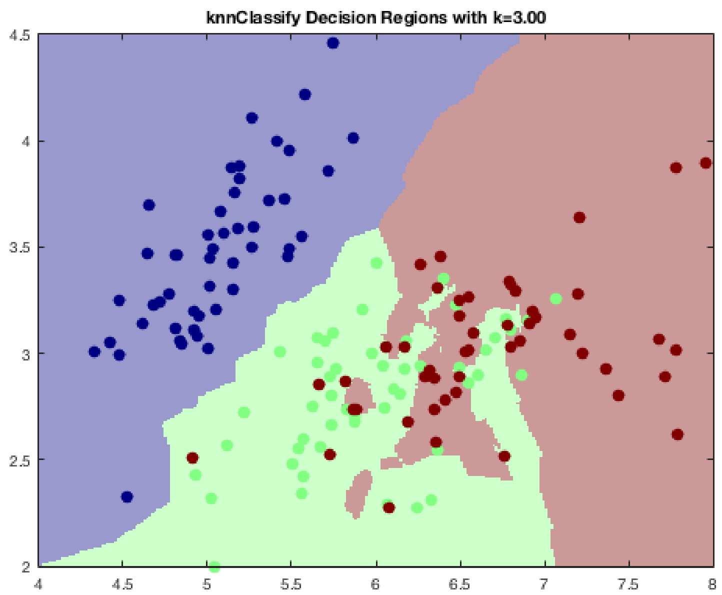

Figure 2.4.3:Plotted data with decision region for k=3

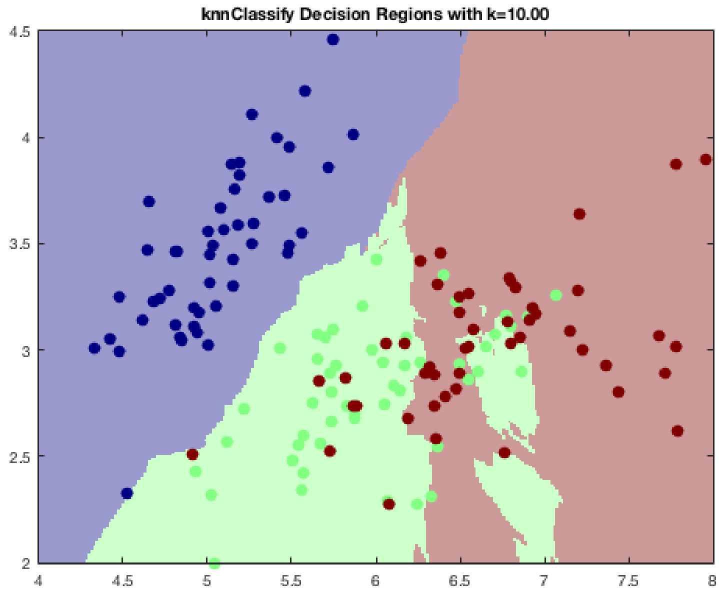

Figure 2.4.4:Plotted data with decision region for k=10

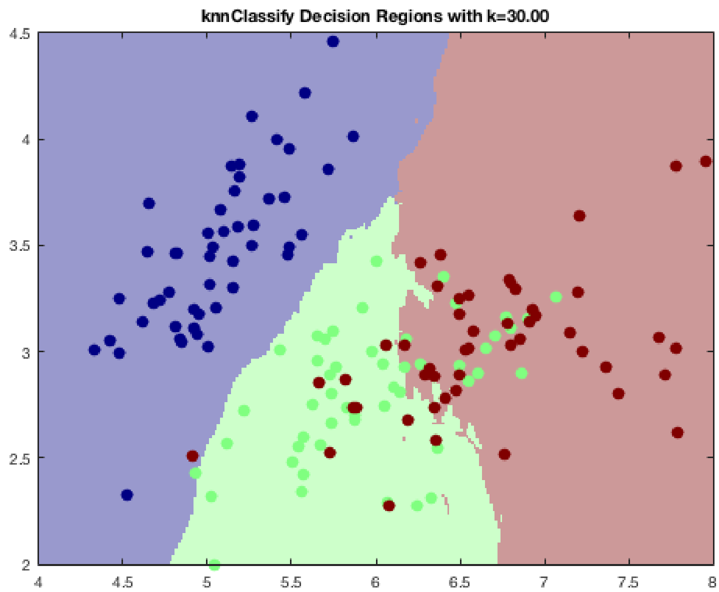

Figure 2.4.5:Plotted data with decision region for k=30

In these graphs small values of k (such as 1) look to be over fitted, with many regions, while larger values (such as 30) are underfitting with a single region. This is for similar reasons as the regression in Section 2. The most appropriate fit appears to be when k=3.

(d) Now split the data into an 80/20 training/validation split. For k = [1, 2, 5, 10, 50, 100, 200], learn a model on the 80% and calculate its performance (# of data classified incorrectly) on the validation data. What value of k appears to generalize best given your training data? Comment on the performance at the two endpoints, in terms of over- or under-fitting.

```
testp = randperm(size(iris,1), ceil(size(iris,1)/5));
trainp = setdiff(1:size(iris,1), testp);

training = iris(testp, :);
testing = iris(trainp, :);

kvalues = [1, 2, 5, 10, 50, 100, 200];
err = zeros(length(kvalues), 1);

for i = 1:length(kvalues)
learner = knnClassify(kvalues(i), training(:, 1:4), training(:, 5));
Yhat = predict(learner, testing(:, 1:4));

err(i) = sum(Yhat(:) ~= testing(:,5));
end

plot(kvalues, err, 'b*-')
title('Perfomance of different k values');
```

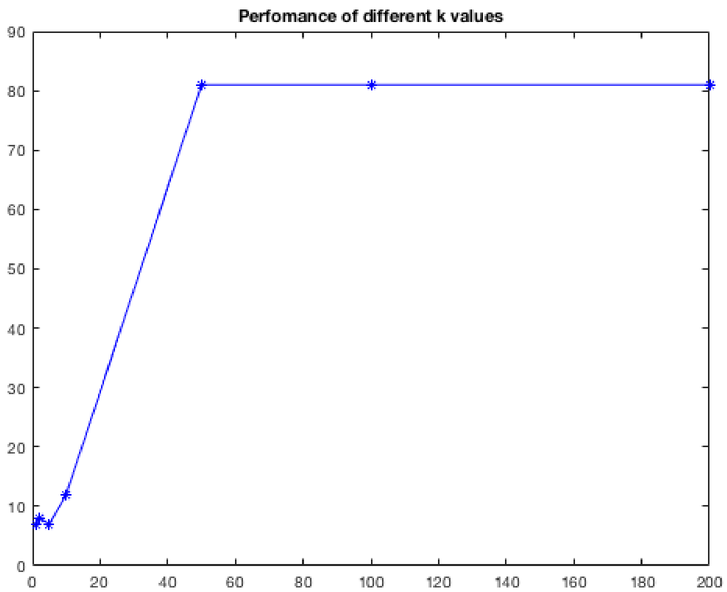

Figure 2.4.6:Performance for different k values

The k value with the least error in this case is k=6. k=1 is slightly overfitting the training data, and causing error when tested against the test data. k=50 up to k=200 are massively underfitting the data, and causing a lot of errors.

2.5 Perceptrons and Logistic Regression (25 Marks)

(a) Show the two classes in a scatter plot and verify that one is linearly separable while the other is not.

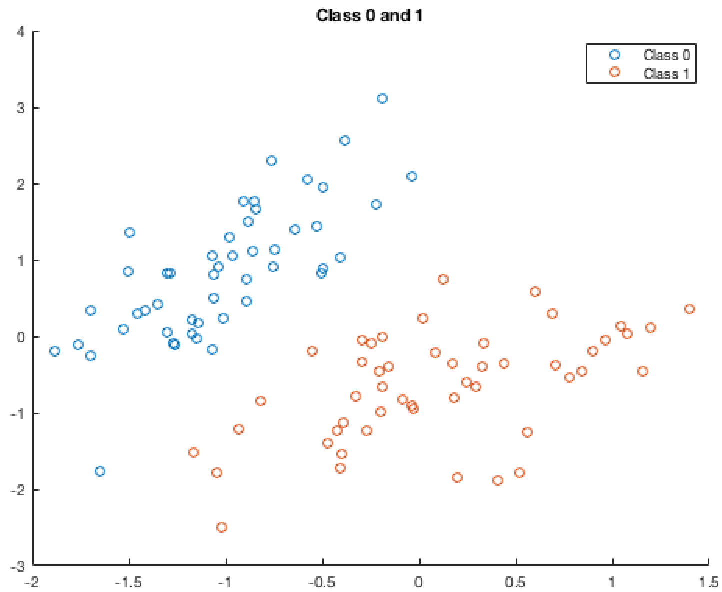

Figure 2.5.1: The data is separable as it is in two distinct groups

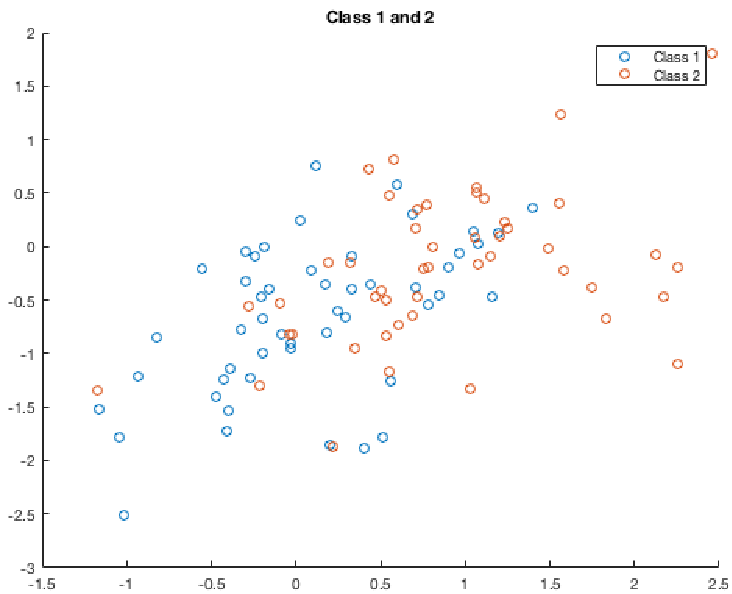

Figure 2.5.2: This data is not separable as it is mixed in with each other, not in well defined groups

(b) Write (fill in) the function @logisticClassify2/plot2DLinear.m so that it plots the two classes of data in different colors, along with the decision boundary (a line). Include the listing of your code in your report.

```
function plot2DLinear(obj, X, Y)
% plot2DLinear(obj, X,Y)
%   plot a linear classifier (data and decision boundary) when features X
    are 2-dim
%   wts are 1x3,  wts(1)+wts(2)*X(1)+wts(3)*X(2)
%
[n,d] = size(X);
if (d~=2)
  error('Sorry -- plot2DLogistic only works on 2D data...');
end

classes = obj.classes;

% plot data
gscatter(X(:,1),X(:,2),Y)
hold on
axis manual

% find boundary line

x = linspace(-2,2);
y = linspace(-2,2);

f = @(x) -(obj.wts(1)+x*obj.wts(2))/obj.wts(3);
y = f(x);

% plot boundary line
plot(x,y,'g--','LineWidth',2,'DisplayName','Boundary')
hold off
```

To demo your function plot the decision boundary corresponding to the classifier $sign(.5 + 1x1 - .25x2)$ along with the A data, and again with the B data.

```
learner = logisticClassify2(); % create "blank" learner
learner = setClasses(learner, unique(YA)); % define class labels using YA
    or YB
wts = [0.5 1 -0.25];
learner = setWeights(learner, wts); % set the learner's parameters
plot2DLinear(learner, XA, YA);

figure;
learner2 = logisticClassify2(); % create "blank" learner
learner2 = setClasses(learner2, unique(YB)); % define class labels using YA
     or YB
wts = [0.5 1 -0.25];
learner2 = setWeights(learner2, wts); % set the learner's parameters
plot2DLinear(learner2, XB, YB);
```

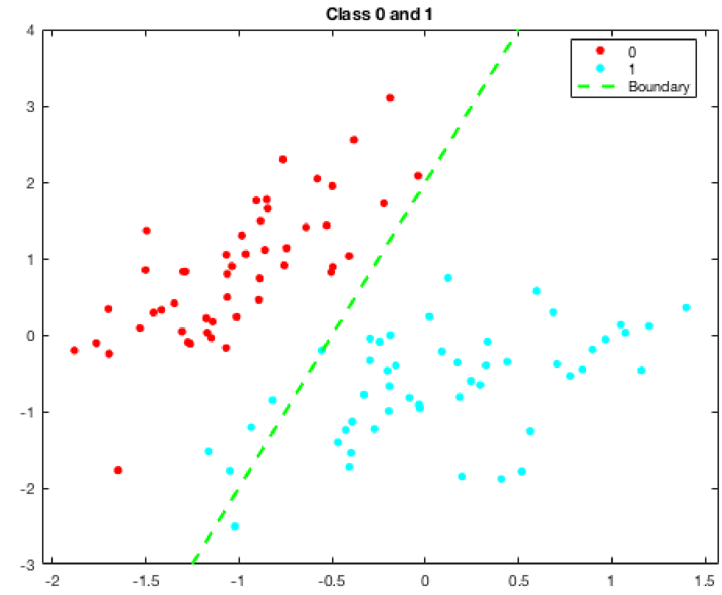

Figure 2.5.3: Classes and decision boundary

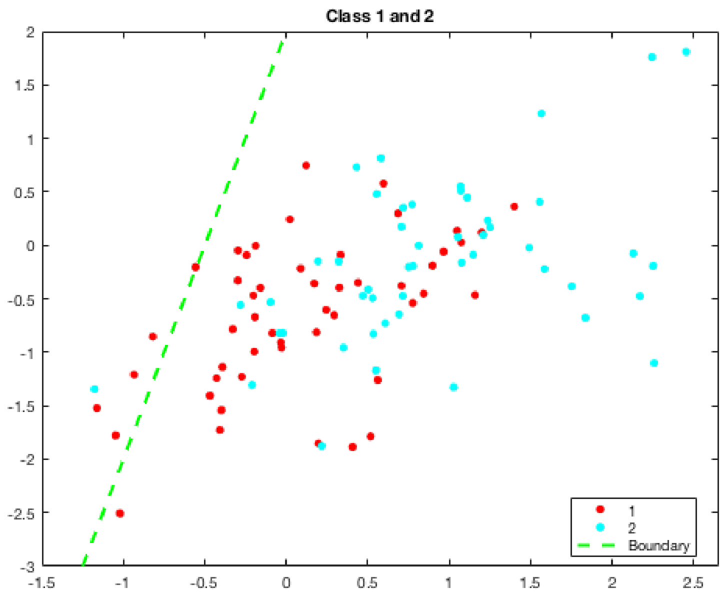

Figure 2.5.4: Classes and decision boundary

(c) Complete the predict.m function to make predictions for your linear classifier.

```
% Test function: predict on Xtest
function Yte = predict(obj,Xte)
```

```
    [Ntr,Mtr] = size(obj.Xtrain);          % get size of training, test data
    [Nte,Mte] = size(Xte);
    classes = unique(obj.Ytrain);          % figure out how many classes & their labels
    Yte = repmat(obj.Ytrain(1), [Nte,1]);  % make Ytest the same data type as Ytrain
    K = min(obj.K, Ntr);                   % can't have more than Ntrain neighbors
    for i=1:Nte,                           % For each test example:
      dist = sum( bsxfun( @minus, obj.Xtrain, Xte(i,:) ).^2 , 2);  % compute sum of squared differences
      %dist = sum( (obj.Xtrain - repmat(Xte(i,:),[Ntr,1]) ).^2 , 2);  % compute sum of squared differences
      [tmp,idx] = sort(dist);              % find nearest neighbors over Xtrain (dimension 2)
      cMax=1; NcMax=0;                     % then find the majority class within that set of neighbors
      for c=1:length(classes),
        Nc = sum(obj.Ytrain(idx(1:K))==classes(c));  % count up how many instances of that class we have
        if (Nc>NcMax), cMax=c; NcMax=Nc; end;        % save the largest count and its class id
      end;
      Yte(i)=classes(cMax);                % save results
    end;
  end
```

Again, verify that your function works by computing & reporting the error rate of the classifier in the previous part on both data sets A and B. (The error rate on data set A should be 0.0505.)

```
errA =
0.0505

errB =
0.4646
```

You can also test this and your previous function by comparing your plot2DLinear output with the generic plotClassify2D function, which shows the decision boundary "manually" by calling predict on a dense grid of locations, rather than analytically as your plot2DLinear function should do.

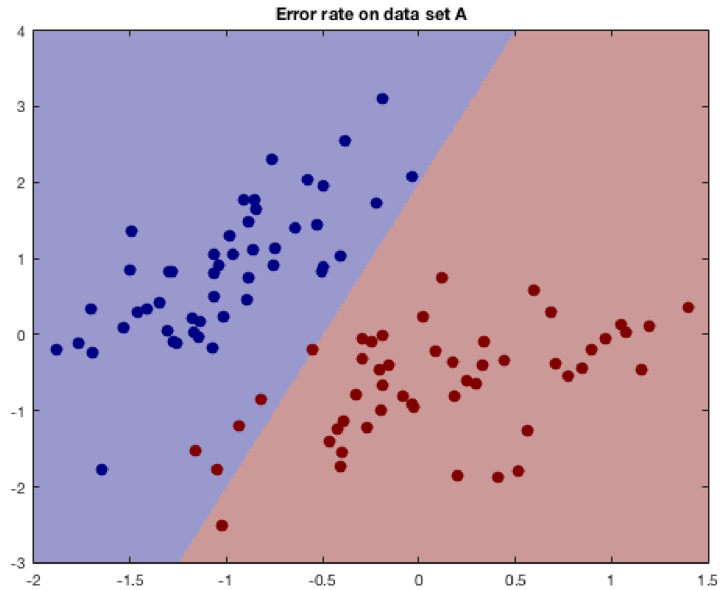

Figure 2.5.5: Classes and decision boundary plotted using plotClassify2D

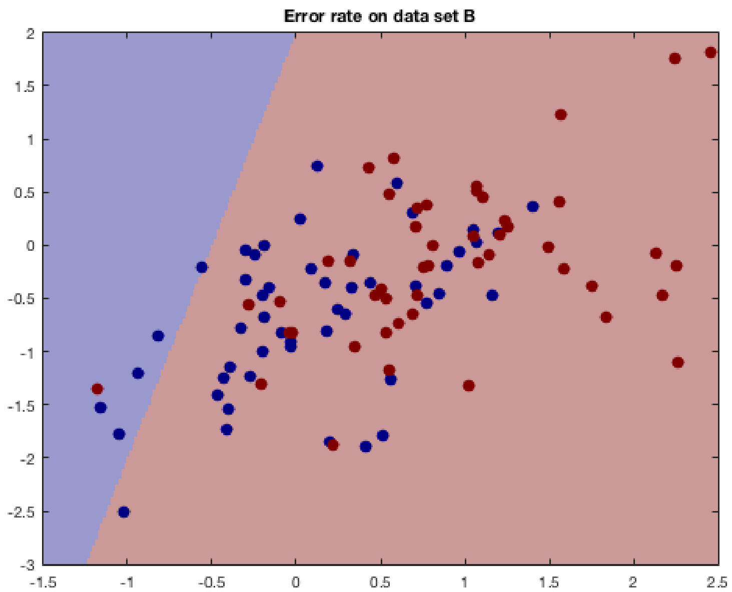

Figure 2.5.6: Classes and decision boundary plotted using plotClassify2D

(d) In my provided code, I first transform the classes in the data Y into "class 0" (negative) and "class 1" (positive). In our notation, let $z = \theta x^{(i)}$ is the linear response of the perceptron, and σ is the standard logistic function.

$$\sigma(z) = (1 + exp(-z))^{-1}$$

The (regularized) logistic negative log likelihood loss for a single data point j is then

$$J_j(\theta) = -y^{(j)} log\sigma(\theta x^{(j)T}) - (1-y^{(j)}) log(1-\sigma(\theta x^{(j)T})) + \alpha \sum_i \theta_i^2$$

where $y(j)$ is either 0 or 1. Derive the gradient of the regularized negative log likelihood J_j for logistic regression, and give it in your report.

The derivative of the gradient can be found by computing $\frac{\partial J_j(\theta)}{\partial \theta_i}$ over the loss for point j, as given by $x^{(j)}$, $y^{(i)}$. The derivative then:

$$\frac{\partial J_j(\theta)}{\partial \theta_i} = x^j((1+exp(z))^{-1} - y^{(j)}) + 2.\alpha.\theta_i$$

(e) Complete your train.m function to perform stochastic gradient descent on the logistic loss function. This will require that you fill in:

(1) computing the surrogate loss function at each iteration ($J = 1/m \sum J_j$)

```
while (~done)
step = stepsize/iter;                    % update step-size and evaluate current
    loss values
Jsur(iter) = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic
    (obj, X)) + reg * sum((obj.wts * obj.wts')')); %%% TODO: compute
    surrogate (neg log likelihood) loss
J01(iter) = err(obj, X, Yin);
```

(2) computing the prediction and gradient associated with each data point $x^{(i)}, j^{(i)}$

```
% Compute linear responses and activation for data point j
y = logistic(obj,X(j,:));
```

(3) a gradient step on the parameters θ

```
% Compute gradient:
grad = X1(j,:) * (y - Y(j)) + 2 * reg * obj.wts;
```

(4) a stopping criterion (usually either stopIter iterations or that J has not changed by more than stopTol since the last iteration through all the data).

```
  %%% Check for stopping conditions
changeinJ = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(
    obj, X)) + reg * obj.wts * obj.wts');
if (iter == stopIter || abs(changeinJ - Jsur(iter)) < stopTol)
done = true;
end;
```

(f) Run your logistic regression classifier on both data sets (A and B); for this problem, use no regularization ($\alpha = 0$). Describe your parameter choices (stepsize, etc.) and show a plot of both the convergence of the surrogate loss and error rate, and a plot of the final converged classifier with the data (using e.g. plotClassify2D). In your report, please also include the functions that you wrote (at minimum, train.m, but possibly a few small helper functions as well).

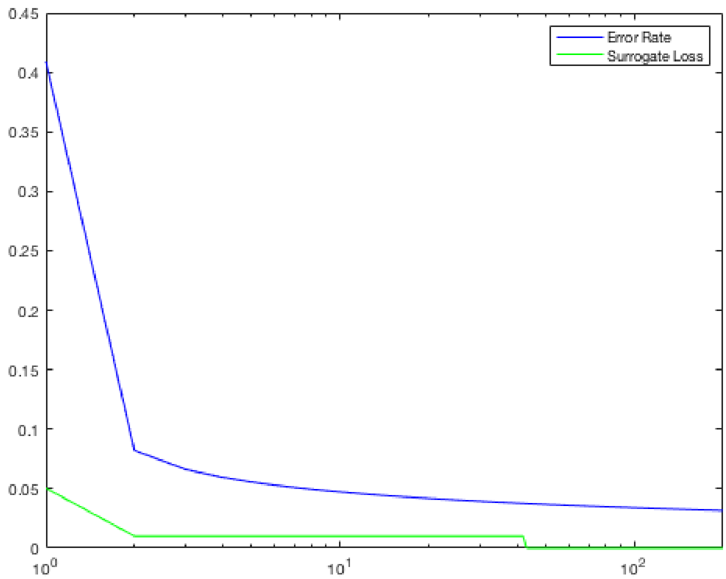

Figure 2.5.7: Surrogate loss and error rate for data set A

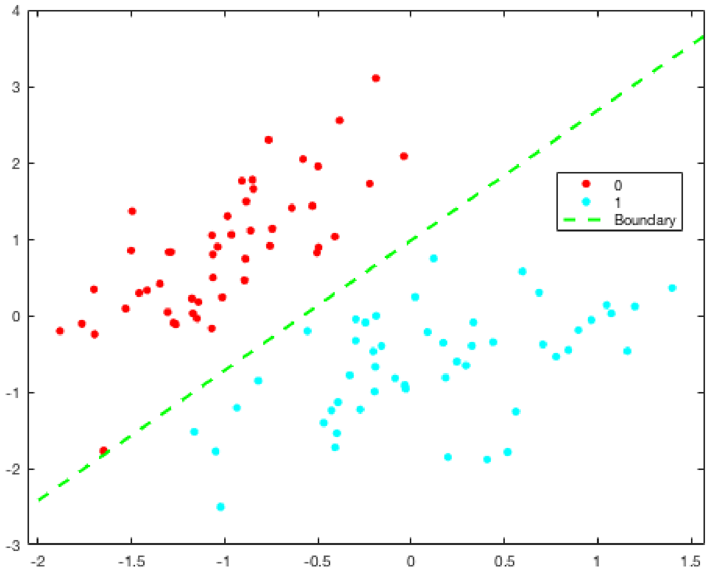

Figure 2.5.8: Final converged classifier for data set A

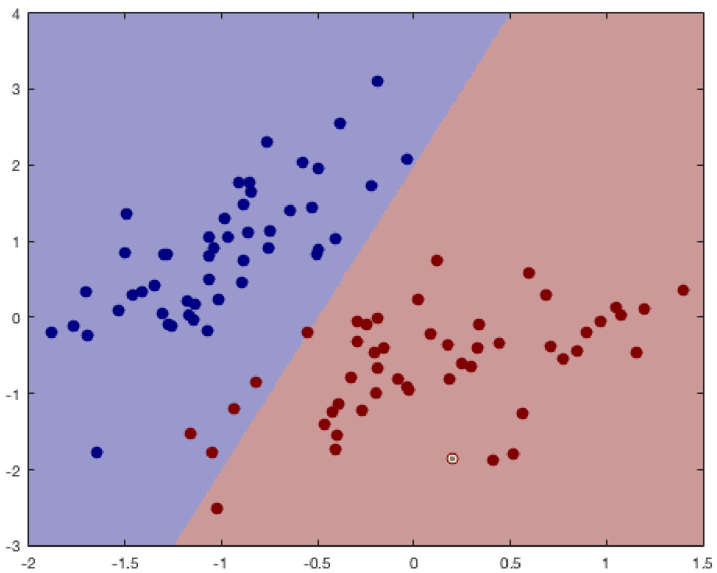

Figure 2.5.9: Classes and decision boundary plotted using plotClassify2D for data set A

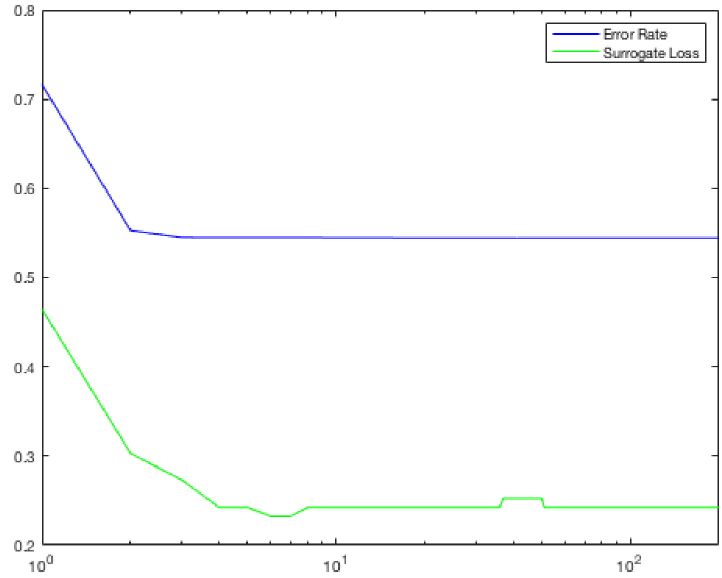

Figure 2.5.10: Surrogate loss and error rate for data set B

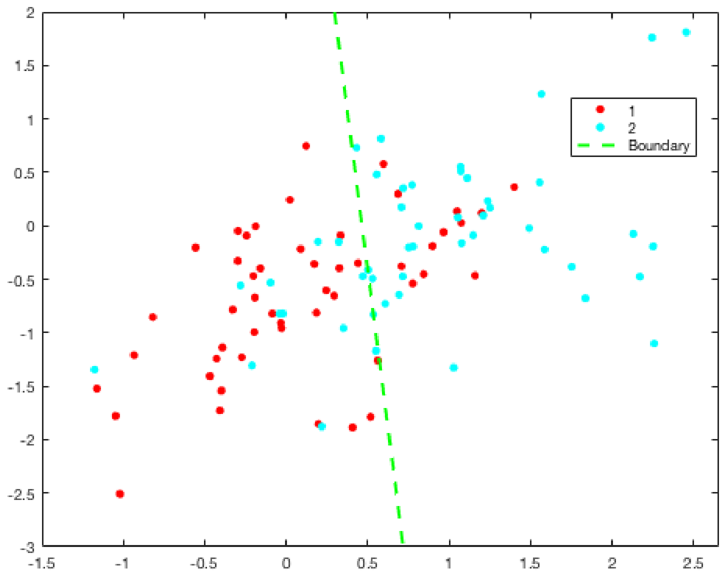

Figure 2.5.11: Final converged classifier for data set B

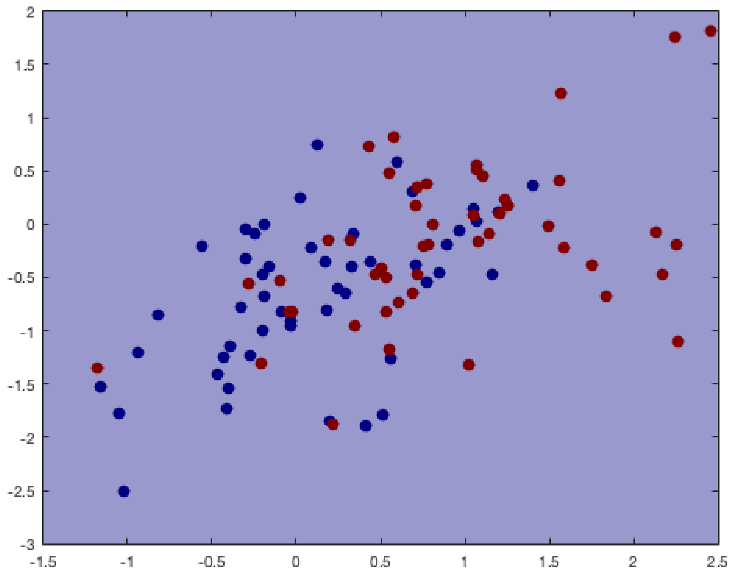

Figure 2.5.12: Classes and decision boundary plotted using plotClassify2D for data set B

```
function obj = train(obj, X, Y, varargin)
% obj = train(obj, Xtrain, Ytrain [, option,val, ...])  : train logistic classifier
%     Xtrain = [n x d] training data features (constant feature not included)
%     Ytrain = [n x 1] training data classes
%     'stepsize', val  => step size for gradient descent [default 1]
%     'stopTol',  val  => tolerance for stopping criterion [0.0]
%     'stopIter', val  => maximum number of iterations through data before stopping [1000]
%     'reg', val       => L2 regularization value [0.0]
%     'init', method   => 0: init to all zeros;  1: init to random weights;
% Output:
%   obj.wts = [1 x d+1] vector of weights; wts(1) + wts(2)*X(:,1) + wts(3)*X(:,2) + ...


  [n,d] = size(X);            % d = dimension of data; n = number of training data

 % default options:
  plotFlag = true;
  init     = [];
  stopIter = 100; % Lowered to improve performance. Initially 1000.
  stopTol  = -1;
  reg      = 0.0;
  stepsize = 1;

```

```
  i=1;                                              % parse through various options
  while (i<=length(varargin)),
    switch(lower(varargin{i}))
    case 'plot',       plotFlag = varargin{i+1}; i=i+1;   % plots on (true/false)
    case 'init',       init     = varargin{i+1}; i=i+1;   % init method
    case 'stopiter',   stopIter = varargin{i+1}; i=i+1;   % max # of iterations
    case 'stoptol',    stopTol  = varargin{i+1}; i=i+1;   % stopping tolerance on surrogate loss
    case 'reg',        reg      = varargin{i+1}; i=i+1;   % L2 regularization
    case 'stepsize',   stepsize = varargin{i+1}; i=i+1;   % initial stepsize
    end;
    i=i+1;
  end;

  X1    = [ones(n,1), X];      % make a version of training data with the constant feature

  Yin = Y;                              % save original Y in case needed later
  obj.classes = unique(Yin);
  if (length(obj.classes) ~= 2) error('This logistic classifier requires a binary classification problem.'); end;
  Y(Yin==obj.classes(1)) = 0;
  Y(Yin==obj.classes(2)) = 1;           % convert to classic binary labels (0/1)

  if (~isempty(init) || isempty(obj.wts))   % initialize weights and check for correct size
    obj.wts = randn(1,d+1);
  end;
  if (any( size(obj.wts) ~= [1 d+1]) ) error('Weights are not sized correctly for these data'); end;
  wtsold = 0*obj.wts+inf;

% Training loop (SGD):
iter=1; Jsur=zeros(1,stopIter); J01=zeros(1,stopIter); done=0;
while (~done)
  step = stepsize/iter;                 % update step-size and evaluate current loss values
  Jsur(iter) = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(obj, X)) + reg * sum((obj.wts * obj.wts')'));   %%% TODO: compute surrogate (neg log likelihood) loss
  J01(iter) = err(obj, X, Yin);

  if (plotFlag), switch d,              % Plots to help with visualization
    case 1, fig(2); plot1DLinear(obj,X,Yin);  %  for 1D data we can display the data and the function
    case 2, fig(2); plot2DLinear(obj,X,Yin);  %  for 2D data, just the data and decision boundary
    otherwise, % no plot for higher dimensions... %  higher dimensions
```

```
    visualization is hard
  end; end;
  fig(1); semilogx(1:iter, Jsur(1:iter),'b-',1:iter,J01(1:iter),'g-'); drawnow;

  for j=1:n,
    % Compute linear responses and activation for data point j
    y = logistic(obj,X(j,:));

    % Compute gradient:
    grad = X1(j,:) * (y - Y(j)) + 2 * reg * obj.wts;

    obj.wts = obj.wts - step * grad;      % take a step down the gradient
  end;


 %   done = false;
 %%% Check for stopping conditions
  changeinJ = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(obj, X)) + reg * obj.wts * obj.wts');
  if (iter == stopIter || abs(changeinJ - Jsur(iter)) < stopTol)
    done = true;
  end;

  iter = iter + 1;
end;
```

(g) To implement the mini batch gradient descent on the logistic function complete your train_in_batches.m function. This will require that you :

(1) fill in create_mini_batches.m function, which generates the mini batches of data. shuffle your data inside this function and set the batch size to 11;

```
function mini_batches = create_mini_batches(obj, X,y, batch_size )

%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here

data_values = [X,y];

data_values = data_values(randperm(size(data_values, 1)), :); % shuffle your data
n_mini_batches = ceil(size(data_values, 1)/batch_size)  - 1; %  based on your data and the batch size compute the number of batches
mini_batches = zeros(batch_size,3,n_mini_batches);

for i = 1:n_mini_batches
    disp(i)
   %TODO extract the minibatch values
   mini_batches(:,:,i) = data_values(i+((i-1)*batch_size):batch_size+((i-1)*(batch_size+1)), :);
end

end
```

(2) update the training iterations in train_in_batches.m, in contrast to the training in section (e), in this function training will be performed on each of these data batches;

```
function obj = train_in_batches(obj, X, Y, batch_size, varargin)
% obj = train(obj, Xtrain, Ytrain [, option,val, ...])  : train logistic classifier
%     Xtrain = [n x d] training data features (constant feature not included)
%     Ytrain = [n x 1] training data classes
%     'stepsize', val  => step size for gradient descent [default 1]
%     'stopTol',  val  => tolerance for stopping criterion [0.0]
%     'stopIter', val  => maximum number of iterations through data before stopping [1000]
%     'reg', val       => L2 regularization value [0.0]
%     'init', method   => 0: init to all zeros;  1: init to random weights;
% Output:
%   obj.wts = [1 x d+1] vector of weights; wts(1) + wts(2)*X(:,1) + wts(3)*X(:,2) + ...


  [n,d] = size(X);            % d = dimension of data; n = number of training data

  % default options:
  plotFlag = true;
  init     = [];
  stopIter = 1000;
  stopTol  = -1;
  reg      = 0;
  stepsize = 1;

  i=1;                                       % parse through various options
  while (i<=length(varargin)),
    switch(lower(varargin{i}))
    case 'plot',      plotFlag = varargin{i+1}; i=i+1;   % plots on (true/false)
    case 'init',      init     = varargin{i+1}; i=i+1;   % init method
    case 'stopiter',  stopIter = varargin{i+1}; i=i+1;   % max # of iterations
    case 'stoptol',   stopTol  = varargin{i+1}; i=i+1;   % stopping tolerance on surrogate loss
    case 'reg',       reg      = varargin{i+1}; i=i+1;   % L2 regularization
    case 'stepsize',  stepsize = varargin{i+1}; i=i+1;   % initial stepsize
    end;
    i=i+1;
  end;

  X1    = [ones(n,1), X];     % make a version of training data with the constant feature


```

```
  Yin = Y;                                % save original Y in case needed later
  obj.classes = unique(Yin);
  if (length(obj.classes) ~= 2) error('This logistic classifier requires a binary classification problem.'); end;
  Y(Yin==obj.classes(1)) = 0;
  Y(Yin==obj.classes(2)) = 1;             % convert to classic binary labels (0/1)

  if (~isempty(init) || isempty(obj.wts))   % initialize weights and check for correct size
    obj.wts = randn(1,d+1);
  end;
  if (any( size(obj.wts) ~= [1 d+1]) ) error('Weights are not sized correctly for these data'); end;
  wtsold = 0*obj.wts+inf;

% Training loop (SGD):
iter=1; Jsur=zeros(1,stopIter); J01=zeros(1,stopIter); done=0;
while (~done)
  step = stepsize/iter;                  % update step-size and evaluate current loss values
  wtsTrans = (obj.wts * obj.wts')';
%   disp((-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(obj, X))))
  Jsur(iter) = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(obj, X)) + reg * sum(wtsTrans));
  J01(iter) = err(obj,X,Yin);

  if (plotFlag), switch d,               % Plots to help with visualization
    case 1, fig(2); plot1DLinear(obj,X,Yin);  %  for 1D data we can display the data and the function
    case 2, fig(2); plot2DLinear(obj,X,Yin);  %  for 2D data, just the data and decision boundary
    otherwise, % no plot for higher dimensions... %  higher dimensions visualization is hard
  end; end;
  fig(1); semilogx(1:iter, Jsur(1:iter),'b-',1:iter,J01(1:iter),'g-'); drawnow;
  mini_batches = create_mini_batches(obj, X, Y, batch_size); %%% call your create_mini_batches function with batchsize = 11
  number_of_batches = size(mini_batches,3);
  for j=1:number_of_batches,
    % Compute linear responses and activation for minibatch j

    batch_X = mini_batches(:,1:2,j);
    batch_Y = mini_batches(:,3,j);
    batch_X1 = [ones(size(mini_batches,1),1), batch_X];

    grad = zeros(size(mini_batches, 1), size(obj.wts, 2))
    for a=1:size(mini_batches,1)
        datapoint_h = logistic(obj,X(a,:));

        % Compute gradient:
```

```
        %%% TODO ^^^
        grad(a, :) = batch_X1(a,:) * (datapoint_h - batch_Y(a));
%           grad(a, :) = batch_X1(a,:) * (datapoint_h - batch_Y(a)) + 2 * reg
    * obj.wts

    end
    % Compute gradient:
    %%% TODO ^^^
    grad = mean(grad) + 2 * reg * obj.wts

    obj.wts = obj.wts - step * grad;       % take a step down the gradient
  end;

%   done = false;
  %%% TODO: Check for stopping conditions
changeinJ = mean(-Y .* log(logistic(obj, X)) - (1 - Y) .* log(1 - logistic(
    obj, X)) + reg * obj.wts * obj.wts');
  if (iter == stopIter || abs(changeinJ - Jsur(iter)) < stopTol)
    done = true;
  end;
  wtsold = obj.wts;
  iter = iter + 1;
end;
```

(3) change the training method in logisticClassify2 (set it into "train_in_batches") and run your mini batch logistic regression classifier. In your report please include the functions that you wrote create_mini_batches.m and train_in_batches.m)

```
% Train Set A
train_in_batches(learner, XA, YA,11, 'stopIter', 100, 'stepsize', 0.1);
legend('Error Rate', 'Surrogate Loss');
% Plot final converged classifier decision boundaries.
figure();
plotClassify2D(learner, XA, YA);

%% Train Set B
figure
train_in_batches(learner, XB, YB,11, 'stopIter', 100, 'stepsize', 0.1);
legend('Error Rate', 'Surrogate Loss');
% Plot final converged classifier decision boundaries.
figure();
plotClassify2D(learner, XB, YB);
```

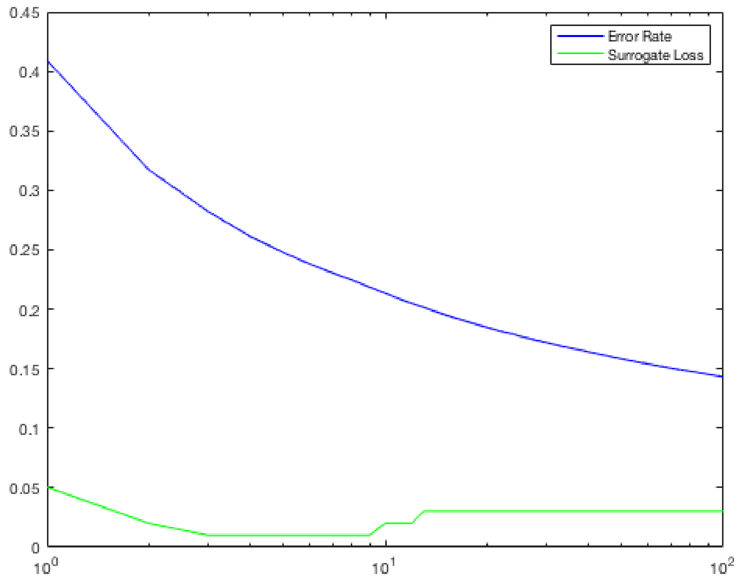

Figure 2.5.13: Surrogate loss and error rate for data set A trained in minibatches

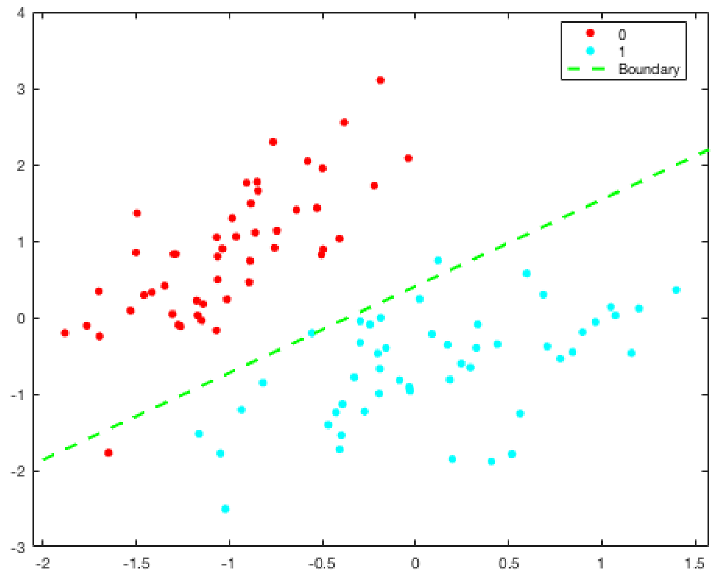

Figure 2.5.14: Final converged classifier for data set A trained in minibatches

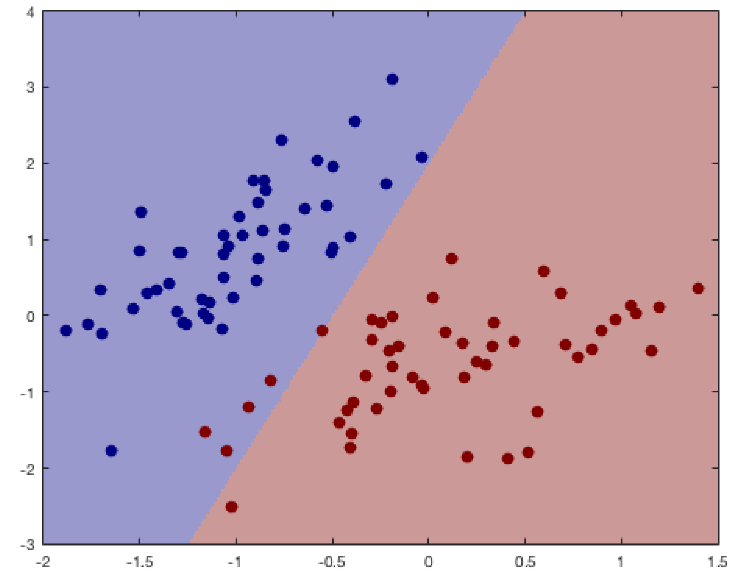

Figure 2.5.15: Classes and decision boundary plotted using plotClassify2D for data set A trained in minibatches

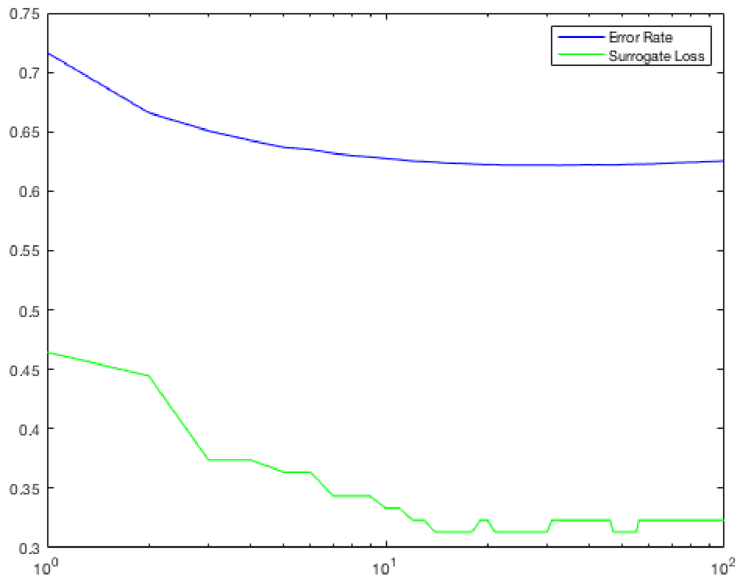

Figure 2.5.16: Surrogate loss and error rate for data set B trained in minibatches

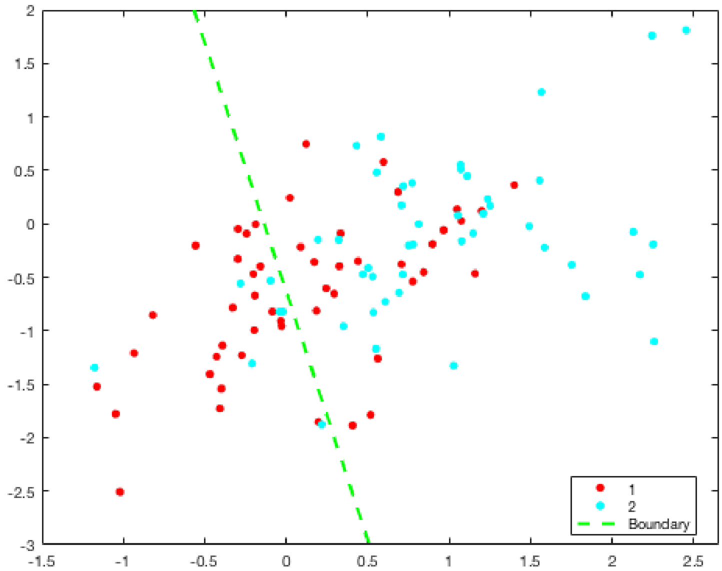

Figure 2.5.17: Final converged classifier for data set B trained in minibatches

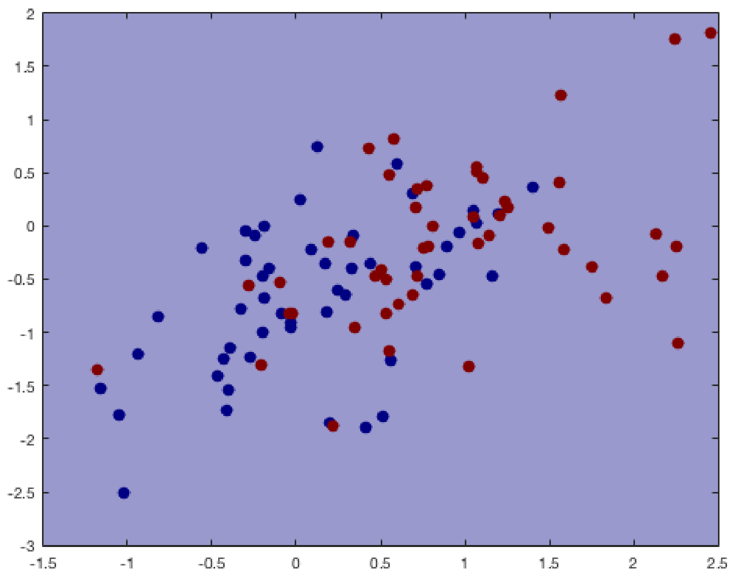

Figure 2.5.18: Classes and decision boundary plotted using plotClassify2D for data set B trained in minibatches