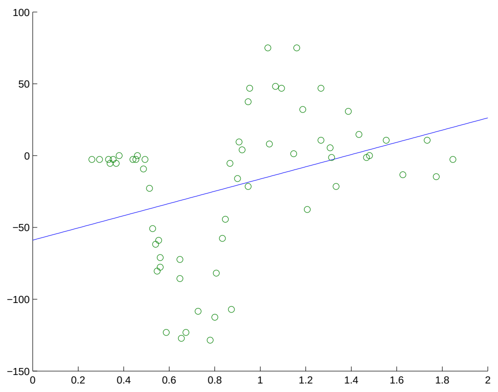
Kyle Benson

CS 273A - Machine Learning: Fall 2013

Homework 2

Problem 1: Linear Regression

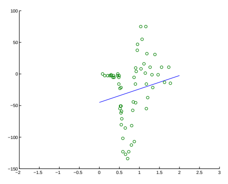
(a) Done



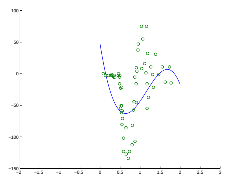
(b)

Training MSE: 2235.8
Test MSE: 2414.7

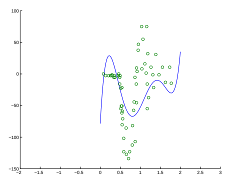
(c) Polynomial training errors:



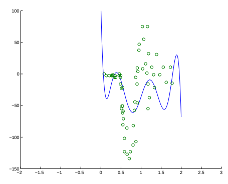
1



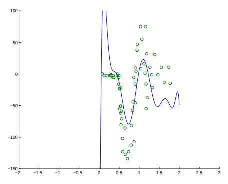
3



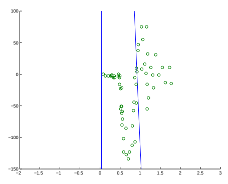
5



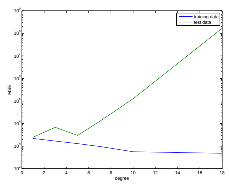
7



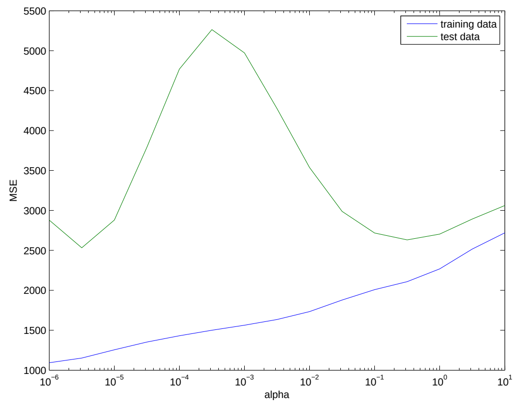
10



18



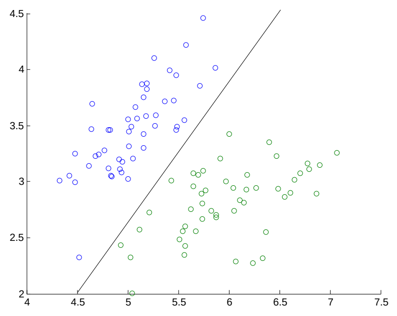
MSE



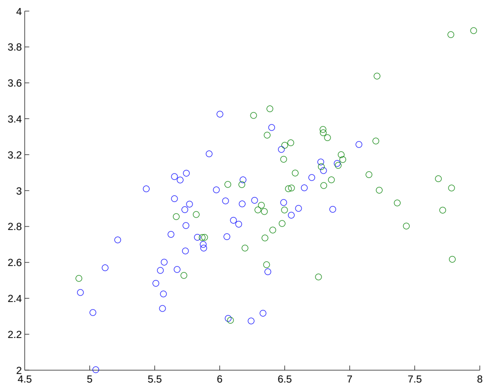
(d)

Problem 2: Perceptron Classifiers

(a) classes 0 vs. 1 (linearly separable as shown in plot)



classes 1 vs. 2 (not linearly separable)

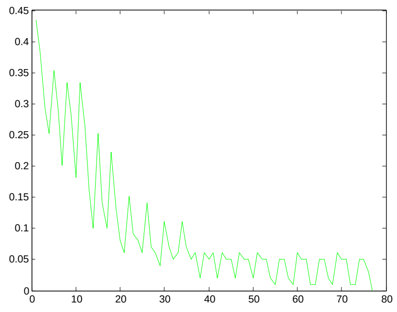


(b)
```
iris=load('~/repos/cs273a/hw3/src/data/iris.txt');
X = iris(:,1:2); Y=iris(:,end);
XA = X(Y<2,:); YA=Y(Y<2); % 0 vs 1
XB = X(Y>0,:); YB=Y(Y>0); % 1 vs 2

%%%% PART A %%%%

%hold on;
%scatter(X(Y==0,1), X(Y==0,2));
%scatter(X(Y==1,1), X(Y==1,2));
%hold off;
%%saveas(gcf, '../figs/prob2a_0v1', 'pdf');
%close
%
%hold on;
%scatter(X(Y==1,1), X(Y==1,2));
%scatter(X(Y==2,1), X(Y==2,2));
%hold off;
%%saveas(gcf, '../figs/prob2a_1v2', 'pdf');
%close

%%%% PART C %%%%
step = 0.01;
nIter = 1e3;

%pc = perceptClassify(XB,YB, step,nIter);
```

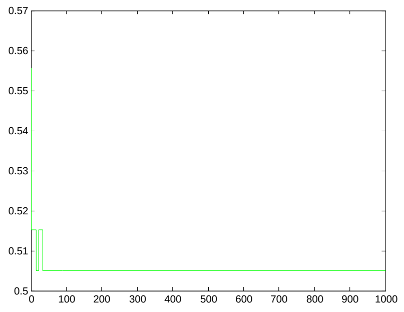
```
%pc = perceptClassify(XA,YA, step, nIter);

%%%% PART D %%%%
xs = XA;
%xs = XB;
ys = YA.*2-1;
%ys = (YB-1).*2-1;

lc = linearRegress(xs, ys);

pc = perceptClassify(xs, ys);
weights = getWeights(lc);
pc = setWeights(pc, weights);
plot2DLinear(pc, xs, ys);

error = err(pc, xs, ys)
```

(c) The **XA,YA** data converged much faster than the **XB,YB** data. The former also reached a pretty good error rate, due to it being linearly separable, as opposed to the latter that never got much better than 0.5, which is the same as a random guess.

0 vs. 1

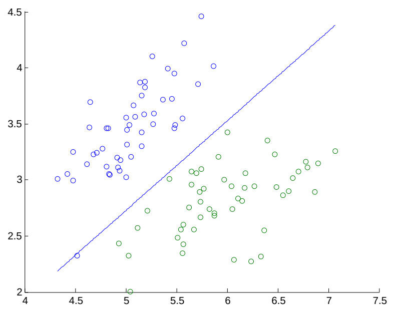


1 vs. 2

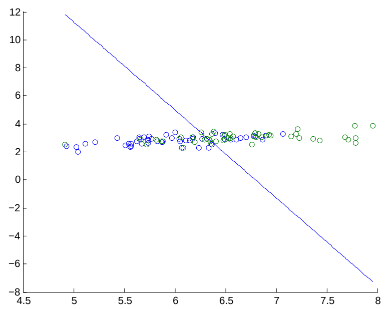


(d) The **XA,YA** data actually resulted in a higher error rate (0.0101) when using weights from a linear regression. This perceptron algorithm is able to take a step so that the decision boundary gets the blue point in the bottom left of the plots correctly, whereas the linear regression will

0 vs. 1



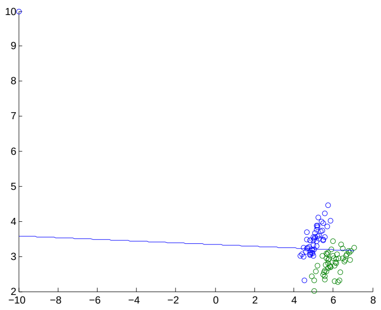
1 vs. 2



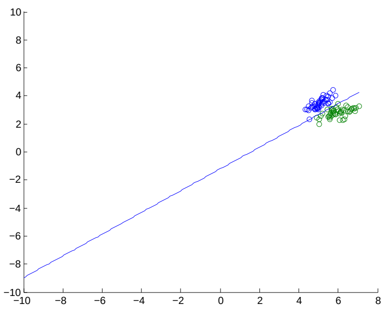
simply try to find a rough center of mass between the data, which might miss some of the points near the ideal boundary.

The **XB,YB** data seemed to actually perform better than the perceptron algorithm with an error rate of 0.2626. This is likely due to the linear regression finding a reasonable center, whereas the perceptron algorithm frequently gets stuck in very unsuitable situations due to the mingling of the data with respect to their classes.

(e) The error actually increased in the linear regression case (1st image below) due to the new point being so far away from the rest and skewing the center that the regression algorithm was looking for. It didn't affect the perceptron algorithm because it was in the correct direction as the rest of the points, and so was quickly classified correctly.



10
9
8
7
6
5
4
3
2
−10
−8
−6
−4
−2
0
2
4
6
8



10
8
6
4
2
0
−2
−4
−6
−8
−10
−10
−8
−6
−4
−2
0
2
4
6
8

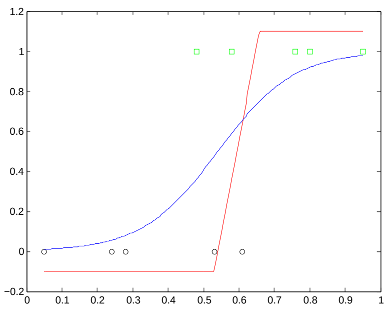
Problem 3: Logistic Regression

(a) Done

(b) See (c)

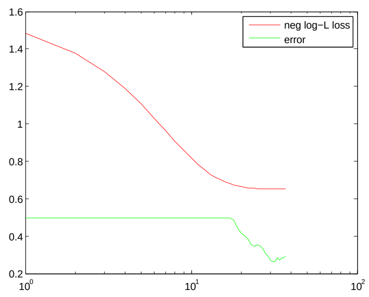
(c) Using Mathematica, I derived the following: $\frac{dJ(\theta)}{d\theta} = \frac{1}{m}[-y^{(j)}\log((1+\exp -x^{(j)T}\theta')^{-1}) - (1-y^{(j)})\log(1-(1+\exp(-x^{(j)T}\theta'))^{-1}) + \alpha\sum_i \theta_i^2)]$

```
function obj = train(obj, Xtrain, Ytrain, stepsize, maxSteps)
% obj = train(obj, Xtrain, Ytrain, stepsize, maxSteps)
%     Xtrain = [n x d] training data features (constant feature not included)
%     Ytrain = [n x 1] training data classes (binary, e.g., +1 or -1)
%     stepsize = step size for gradient descent ("learning rate")
%     maxSteps = maximum number of steps before stopping
%
alpha = 0;
losstol = 0.0000000001;

if (nargin < 5) maxSteps = 5000;  end;  % max number of iterations
if (nargin < 4) stepsize = .01;   end;  % gradient descent step size

plotFlag = 0;                     % with plotting

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

Xtrain1= [ones(n,1), Xtrain];    % make a version of training data with the constant feature

% Get class id values and replace with values 1..C
[Ytrain, obj.classes] = toIndex(Ytrain);
if (length(obj.classes)~=2) error('Y values must be binary!'); end;  % check correct binary labeling
Ytrain = Ytrain-max(Ytrain) + 1;            % convert Y to 0/1 for ease

obj.theta = randn(1,d+1);          % initialize weights randomly
obj.theta = obj.theta * 0;

% Outer loop of stochastic gradient descent:
iter=1;                           % iteration #
done=0;                           % end of loop flag
err=zeros(1,maxSteps);            % misclassification rate values
nlll=zeros(1,maxSteps);            % misclassification rate values
prev_loss=Inf;                            % keep track of previous
                                        % error and quit when it
                                        % goes down by a tiny amount
while (~done)
  % Step size evolution
  %stepi = stepsize/iter;              % logistic method:
  %decreasing harmonically
  %stepi = stepsize/iter;              % logistic method: decreasing harmonically
  stepi = stepsize;

  % Stochastic gradient update (one pass)
  for i=1:n,  % for each data example,
    % compute gradient of regularized logistic negative log
    % likelihood loss function
    resp = Xtrain1(i,:)*obj.theta';
    logit_term = (-1/(exp(resp)+1));
    grad = Xtrain1(i,:) * (logit_term - Ytrain(i)+1) / n;

    % was trying to do this in full vector form, but was causing issues...
    %logit_term = (1+exp(-Xtrain1*obj.theta')).^(-1);
    %sum_term = (logit_term - Ytrain)' * Xtrain1;
    %grad = alpha + sum_term/n;
    obj.theta = obj.theta - stepi * grad;                   % Take a step down the gradient
  end;

  % Compute current error values (missclassification rate)
  err(iter)  = mean( (Ytrain~=round((1+exp(-Xtrain1*obj.theta')).^(-1))));

  % Compute regularized logistic negative log likelihood loss
  nlll_first_term = -Ytrain'*log((1+exp(-Xtrain1*obj.theta')).^(-1));
  nlll_second_term = -(1-Ytrain')*log(1-(1+exp(-Xtrain1*obj.theta')).^(-1));
  alpha_term = alpha*sum(obj.theta.^2);
  nlll(iter) = n^(-1)*(nlll_first_term + nlll_second_term) + alpha_term;

  % Make plots, if desired
  if (plotFlag),
  fig(1);
  semilogx(1:iter, nlll(1:iter), 'r-', 1:iter, err(1:iter),'g-'); %plot regularized logistic negativ
  legend('neg log-L loss', 'error');
  fig(2); switch d,                                % Plots to help with visualization
      case 1, plot1DLinear(obj,Xtrain,Ytrain);      %  for 1D data we can display the data and the f
      case 2, plot2DLinear(obj,Xtrain,Ytrain);      %  for 2D data, just the data and decision bound
      otherwise, % no plot for higher dimensions... %  higher dimensions visualization is hard
    end;
  drawnow;
  end;

  %stop when no errors or out of time or changes by small amount
  done = (iter >= maxSteps || (prev_loss - nlll(iter)) < losstol);
  prev_loss = nlll(iter);
  iter = iter + 1;
end;

err(end)
nlll(end)
```

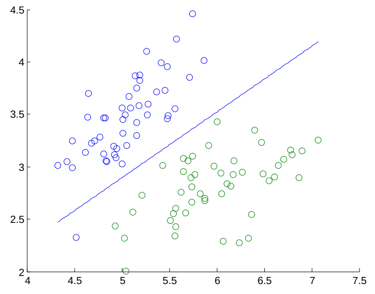


(d)

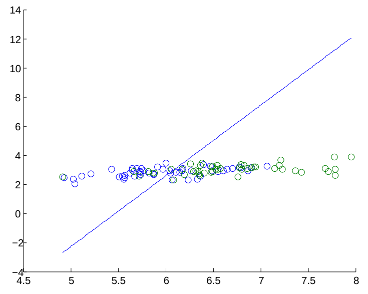
(e) On 2-d data:



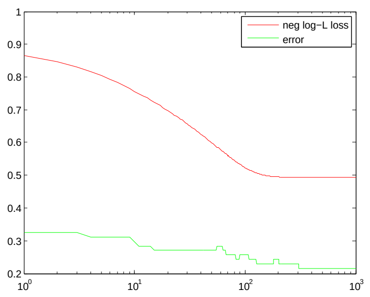
(f)



0 vs. 1

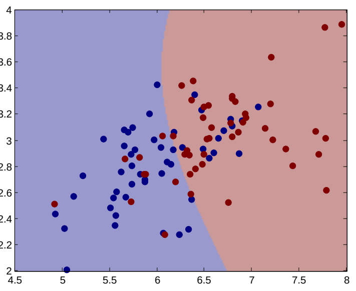


1 vs. 2



(g)

training error: 0.2162
test error: 0.3200



(h)

Problem 4: High Dimensional Data

(a) For settings, I used a step size of 0.1 and a maximum of 1000 iterations. I chose these because the step size still resulted in good convergence, but would do so faster than a smaller step size. The error rate also seemed to change very little after around 1000 iterations, almost insignificantly so.

best error: 0 best loss: 0.0016 training error: 0 test error: 0

(b) Interestingly, the classifiers seem to perform quite well without all the additional features. The error rate is slightly higher, but still seems acceptable considering how much less complex the decision algorithm will be with so many fewer features. Furthermore, there is no difference between 2 and 10 features, so it likely takes a lot more additional features to make any significant difference in error rate.

training error 2 features: 0
test error with 2 features: 0.0100
training error 10 features: 0 test error 10 features: 0.0100