COMP-652 Assignment 2 Solutions

Jordan Frank

03/10/08

1 Gradient of cross-entropy for logistic function.

A) The cross-entropy error function is:

$$J_{CE} = -\sum_{i=1}^{m} y_i \log h(\mathbf{x_i}) + (1 - y_i) \log(1 - h(\mathbf{x_i})),$$

the logistic function is:

$$h(\mathbf{x_i}) = \frac{1}{1 + e^{-\sum_{j=0}^{n} w_j x_{i,j}}},$$

where the weights are implicit, and therefore dropped from the notation. The derivative of the logistic function with respect to the weights is:

$$\frac{\partial}{\partial \mathbf{w}} h(\mathbf{x_i}) = h(\mathbf{x_i})(1 - h(\mathbf{x_i}))\mathbf{x_i}.$$

The derivative of J_{CE} with respect to the weights is:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}} J_{CE} &= -\sum_{i=1}^{m} y_i \frac{\partial}{\partial \mathbf{w}} \log h(\mathbf{x_i}) + (1 - y_i) \frac{\partial}{\partial \mathbf{w}} \log(1 - h(\mathbf{x_i})) \\
&= -\sum_{i=1}^{m} y_i (1 - h(\mathbf{x_i}))\mathbf{x_i} - (1 - y_i) h(\mathbf{x_i})\mathbf{x_i} \\
&= -\sum_{i=1}^{m} (y_i - h(\mathbf{x_i}))\mathbf{x_i}
\end{aligned}$$

B,C) MATLAB code:

```
function q1
    x = load('wpbc_x_normalized.txt');
    y = load('wpbc_yrecur.txt');
    x1 = [x(:,1),ones(size(x,1),1)];
    x2 = [x,ones(size(x,1),1)];
    w = CE_gradient(1E4,x1,y); w'
```

```
    i = linspace(min(x(:,1)),max(x(:,1)))';
    plot(x(:,1),y,'o',i,h(w,[i,ones(size(i),1)]),'-');
    error = CE_error(x1,y,w)
    w = CE_gradient(1E7,x2,y); w'
    error = CE_error(x2,y,w)
end
function w = CE_gradient(steps,x,y)
    eps = 10E-6;
    stepsize = 1E-3;
    w = zeros(size(x,2),1);
    for i=1:steps
        gradient = x' * (h(w,x) - y);
        if (eps > abs(gradient))
            break;
        end
        w = w - stepsize * gradient;
    end
    i
end
function error = CE_error(x,y,w)
    error = 0.0;
    m = size(y,1);
    for i=1:m
        error = error + y(i)*log(h(w,x(i,:))) + ...
            (1 - y(i)) * log(1-h(w,x(i,:)));
    end
    error = -error;
end
function y = h(w,x)
    y = 1./ (1 + exp(-x*w));
end
```

Output, formatted for presentation:

```
% Part B
Number of steps until convergence (gradient < 1E-6) = 501
weights = [0.3844, -1.2061]
cross-entropy = 103.6807
Final logistic function is plotted in Figure 1.

% Part C
Number of steps until convergence (gradient < 1E-6) = 2,135,752
weights = [-21.3498, -1.1707, 14.7987,  6.6317,  1.4421, -0.0690, -1.6893,
            -0.2329,  0.3075, -2.0253, -1.8363, -1.7496,  3.9227, -1.2863,
             0.5893,  2.5556, -2.7584, -1.1908,  1.2994,  0.6497,  9.9913,
             1.4933, -3.9644, -5.2174,  0.2292, -2.8004,  2.7721, -0.2089,
            -1.1007,  0.3993, -0.0583,  0.7000, -1.9590]
cross-entropy = 74.0946
```

Remark: The purpose of the final part of this question is to get you thinking about what exactly makes one classifier "better" than another classifier. At a purely quantitative level, the cross-entropy between the hypothesis and the data set is smaller for the classifier from part B than that from part C. However, the

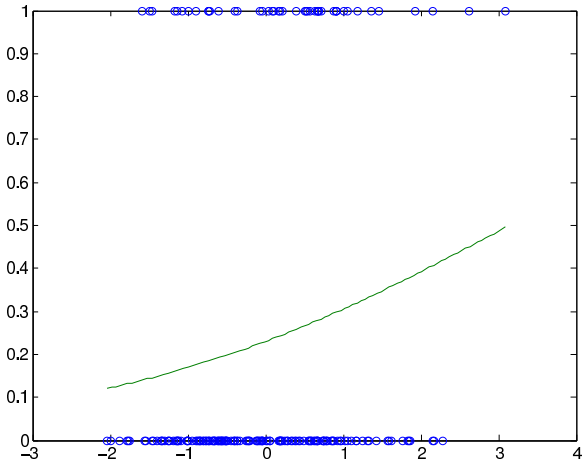

Figure 1: Question 1B: $h(\mathbf{x_i})$ and recurrence data.

number of degrees of freedom in this larger hypothesis class is orders of magnitude greater than the smaller class, yet the error is only reduced by approximately 25%. When a much more complex classifier can only do a little bit better than a much simpler classifier, this should raise suspicions. Maybe the complex classifier is overfitting the data? To test this, one would have to use a technique such as cross validation, which would empirically demonstrate whether the classifier can perform well on data that it was not trained on. The bottom line is that simply comparing the errors is not always an adequate measure of whether one classifier is better than another one.

On the other hand, the classifier from part B is certainly a bit naive. Without a better understanding of the dataset, it could be a mistake to arbitrarily choose one feature and make predictions based on that limited view of the data.

2 Alternative error function for logistic regression.

A) Yes, minimizing J_{SSQ2} corresponds to maximizing the likelihood of the data under the noise model given by:

$$y_i = \sigma(\mathbf{w}^\top \mathbf{x_i} + e_i),$$

where the e_i are iid., mean 0, Gaussian random variables representing the noise. This can be seen if we let $y_i' = \sigma^{-1}(y_i)$, and then rewrite J_{SSQ2} as:

$$J_{SSQ2} = \sum_{i=1}^{m} (y_i' - \mathbf{w}^\top \mathbf{x_i})^2.$$

Now we can just follow the exact arguments from slides 50-52 from Lecture 2, replacing y_i with y_i' and $h(\mathbf{x_i})$ with $\mathbf{w}^\top \mathbf{x_i}$. There are no tricks, so we will omit the details.

B) We can simply perform linear regression. Since we have that:

$$\sigma^{-1}(y_i) = log \frac{y}{1-y},$$

we simply transform each of the y_i to create alternate data set $D' = \{(\mathbf{x_i}, y_i')\}$, where $y_i' = \sigma^{-1}(y_i)$, and then

minimize the following familiar function:

$$\sum_{i=1}^{m}(y_i' - \mathbf{w}^\top \mathbf{x_i})^2,$$

which we all know how to do using linear regression.

Sure, we could perform gradient descent, but when the data isn't so large that computing a matrix inversion is infeasible, linear regression is preferable.

3 Backprop.

A) The following MATLAB code implements an ANN learning algorithm.

```
function [w1,w2] = backprop(x,y,hiddensize,steps)
    alpha = 1.0;
    [m,inputsize] = size(x);
    % Initialize the weights to be uniform between [-0.1,0.1]
    w1 = (rand(inputsize+1,hiddensize) - 0.5) ./ 5; % input-hidden layer weights
    w2 = (rand(hiddensize+1,1) - 0.5) ./ 5; % hidden-output layer weights
    d = zeros(hiddensize+1,1);
    idxs = randint(steps,1,[1,m]);
    for i = 1:steps
        % randomly pick a training example
        act = forwardpass(x(idxs(i),:),w1,w2);
        out = act(hiddensize+1);
        % calculate delta for hidden-output layer
        d(hiddensize+1) = out * (1 - out) * (y(idxs(i)) - out);
        d(1:hiddensize) = act(1:hiddensize) .* (1-act(1:hiddensize)) ...
            .* w2(2:hiddensize+1) * d(hiddensize+1);
        for j=1:hiddensize
            w1(:,j) = w1(:,j) + alpha * d(j) * [1, x(idxs(i),:)]';
        end
        w2(:) = w2 + alpha * d(hiddensize+1) * [1, act(1:hiddensize)']';
        if mod(i,1000) == 999
            % every 1000 steps, evaluate on all inputs and break
            % if SSQ < 1E-4
            err = 0.0;
            for j=1:m
                act = forwardpass(x(j,:), w1, w2);
                err = err + (y(j) - act(hiddensize+1))^2;
            end
            if err < 1E-4
                break;
            end
        end
    end
    i
    w1
    w2
end
function act = forwardpass(input,w1,w2)
```

```
    % computes the node activations for a given input and weights
    hiddensize = size(w1,2);
    act = zeros(hiddensize+1,1);
    act(1:hiddensize) = sigmoid(w1,[1, input]);
    act(hiddensize+1) = sigmoid(w2,[1, act(1:hiddensize)']);
end
function y = sigmoid(w,x)
    y = 1 ./ (1 + exp(-x * w));
end
```

After 201000 steps, with a learning rate of 1.0, the weights obtained for the XOR function were:

```
w1 = [ 3.0230,  8.3089;
      -7.1127, -5.5628;
      -7.1172, -5.5673]
w2 = [ -5.8243; -12.2221; 12.1054]
```

and the outputs of the network for each input value were:

0	0	0.0046
0	1	0.9953
1	0	0.9953
1	1	0.0058

for an SSQ value of 9.9574×10^{-5}.

B,C) We won't burden this solution guide with a bunch more numbers, but suffice it to say that for the 3-input parity function, one can attain an SSQ of less than 10^{-4} with 2 hidden nodes and a learning rate of 1.0 in 4674000 steps, and an SSQ of less than 10^{-4} for the 4-input parity function with 3 hidden nodes, a learning rate of 1.0, and 19501000 steps.