IN4320 Machine Learning Exercise: Computational Learning Theory: boosting

Yadong Li(4608283)

March 13, 2017

a. Prove that $e^{-x} \geq (1-x)$

To prove: $e^{-x} \geq (1-x)$ is equal to prove: $e^{-x} - (1-x) \geq 0$.
Let: $f(x) = e^{-x} - (1-x)$ Then $f'(x) = -e^{-x} + 1$
Set the derivative to zero and we get:

$$-e^{-x} + 1 = 0 \quad => \quad x = 0$$

When $x > 0$, $f'(x) > 0$ and when $x > 0$, $f'(x) < 0$. We can tell that when $x = 0$, $f(x)$ has its minimum: $f(0) = 0$.
As the minimum is 0, $f(x) = e^{-x} - (1-x) \geq 0$ is always true. Thus we successfully proved that $e^{-x} \geq (1-x)$. Figure 1 also shows that $e^{-x} \geq (1-x)$.

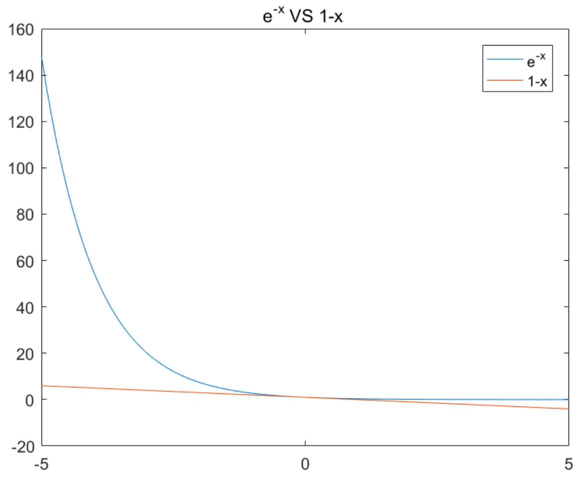

Figure 1: Plot for two expressions.

b. Implement a "weak learner"

To implement this 'weak learner': decision stump, I wrote a function in Matlab, which accepts a dataset with labels as input, and outputs the optimal f^*, θ^* and y^*. The Matlab code is shown below.

```
function [ feat ,theta ,y ] = weakLearner( X,lab)
%weakLearner: A very simple classifier
%  Input: dataset with labels
%  Find the optimal feature f and threshold theta
%  Output: optimal f, theta, and y.

% If a single input(indicating that input is pr_dataset),
% read in the data and label from pr_dataset.
if nargin < 2
    lab = getlab(X);
    X = getdata(X);
end
[n,f] = size(X);
min_score = 10000000;
for i=1:f
    for j = 1:n
        sign = 0; %sign is: "<"
        % calculate predictions
        Theta = ones(n,1)*X(j);
        predict = X(:,i)-Theta<=0;
        predict1 = X(:,i)-Theta>=0;
        % To match the label: 1 and 2 , every prediction adds one.
        predict1 = predict1+1;
        predict = predict+1;
        score =sum(abs(predict-lab));
        score1 = sum(abs(predict1-lab));
        % Check whether should be ">" or "<"
        if score>score1
            score = score1;
            sign =1;
        end
        % Compare and store the minimum score
        if score<min_score
            min_score = score;
            y = sign;
            feat = i;
            theta = X(j);
        end
    end
end
end
```

c. Test the implementation on a simple dataset

To test the implementation in the previous step, a simple dataset was created:two classes from two Gaussian distributions, where the means are $\mu1 = [0, 0]^T$ and $\mu2 = [2, 0]^T$, and the covariance matrices are identify matrices. Figure 2 is the scatterplot of the data. The optimal parameters got from the decision stump is:

$$f^* = 1(indicating\ feature1)\ ,\ \theta^* = 0.6147\ ,\ y^* = 1(x_f < \theta -> class1)$$

If I rescale one of features, the decision stump might change. For example, in the previous case, if I multiply feature 2 by a factor of 10, the optimal parameters stays exactly the same, because the optimal feature is feature 1 and it stays the same. However, if I multiply feature 1 by a factor of 10, parameters stays the same expect that θ^* will be multiplied by 10 as well.

d. Test the implementation on dataset *optdigitsubset*

Use the first 50 objects for each class for training, and the rest for testing. The classification error on the test objects is 0.0615. If I take other random subsets of 50 for training, the testing error doesn't change much. I have run 100 trails, the testing errors are shown in Figure 3. The estimated mean of the error is: 0.0637, and estimated standard deviation is 0.0147. The performance with training on the first 50 object is unexpectedly good for me. I didn't expect such a small number of testing error.

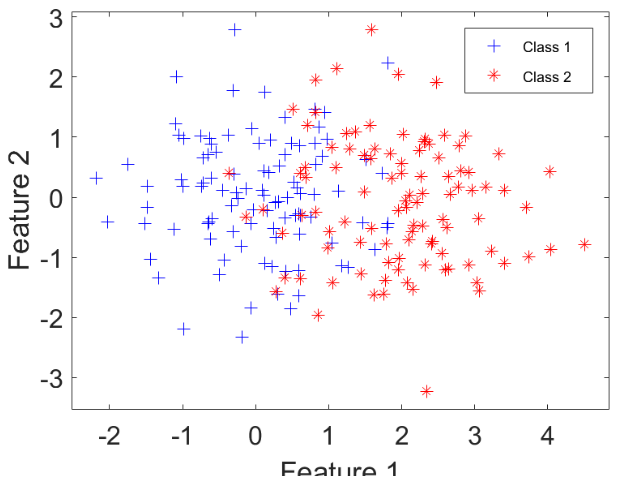

Figure 2: Scatter plot for two classes

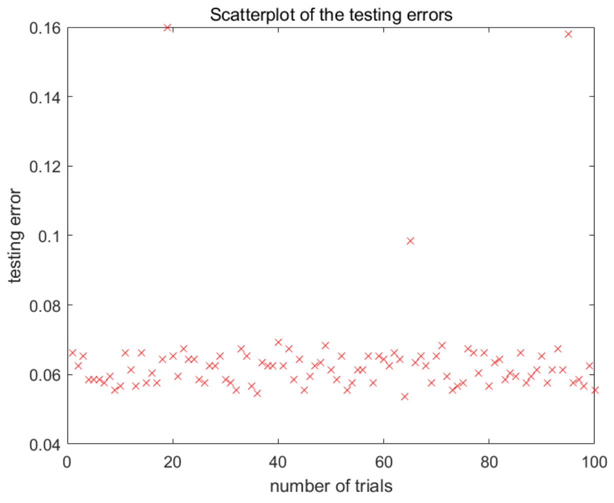

Figure 3: Scatterplot of the testing error in different trails

e. Extend the implementation: Weighted decision stump

To implement a weighted decision stump, I just changed to calculate the weighted sum of error scores, as shown below.

```
score = weight'*abs(predict-lab);
```

I tested this weighted version on a very simple dataset which had just 5 observations each class, as shown in Figure 4.

First, I set all weights for the 10 observations to 1 equally, the optimal result is:

$$f^* = 1(indicating\ feature1)\ ,\ \theta^* = 2.1105\ ,\ y^* = 1(x_f < \theta -> class1)$$

From the scatterplot Figure 4 we could easily find that 2 objects are misclassified, which is the second observation in class $1(x_12 = 2.1525)$ and the third observation in class $2(x_13 = 0.3034)$.

$$class1 : [1.0815\ , 2.1525\ , -0.4176\ , 0.3983\ , -0.9685]$$

$$class1 : [2.1105\ , 2.8007\ , 0.3034\ , 3.2232\ , 4.3710]$$

However, if I assign more weight to the second observations in class 1, the optimal solution would be changed:

$$f^* = 1(indicating\ feature1)\ ,\ \theta^* = 2.1525\ ,\ y^* = 1(x_f < \theta -> class1)$$

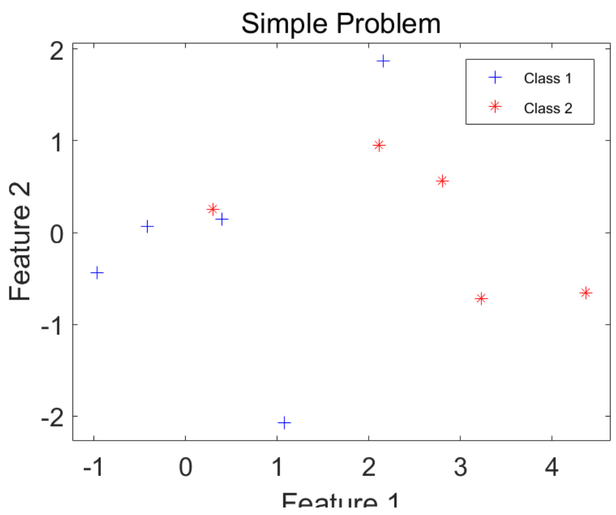

Figure 4: Scatter plot for two classes

Note that θ^* moved towards the second observations in class 1 and it was correctly classified then. I have also tested on other datasets, and these tests convinced me that it has been correctly implemented.

f. AdaBoost Algorithm Implementation

Here is the code for AdaBoost:

```
function [ predLab,beta,para] = adaBoost( X,T,lab)
%adaBoost train several weak classifiers
% Input: X training data, lab label, T number of iterations
% Output: predicted label, beta and parameters for all base classifiers
if nargin < 3
    lab = getlab(X);
    X = getdata(X);
end
%useful info
n = size(X,1);
% Initialize weight, beta and parameters matrix
weight = ones(n,1);
beta = zeros(T,1);
para = zeros(T,3);
% run T times to train T classifiers
for t = 1:T
    p = weight./sum(weight);
    [feat,theta,y] = weightedWeakLearner( X,p,lab);
    para(t,:) = [feat,theta,y];
    [e,pred] = calculateError(feat,theta,y,X,p,lab);
    % if e==0, set e to a really small number to avoid error
    if e==0
        e=0.001;
    end
    beta(t) = e/(1-e);
    weight = weight.*(beta(t).^(1-abs(pred-lab)));
end
%After T iterations, predict labels using T weighted classifiers
predLab = adaPredict(beta,para,X);
end
```

Here is the code for classifying training data or new and unseen data:

```
function [ predLab ] = adaPredict(beta,para,X)
%adaPredict: Predict labels for adaboosting classifiers
% Input: beta(T,1), para(T,3), X(n,f)
% para: [feat,theta,y]
% Output: predLab(n,1)

%Get useful info
N = size(X,1);
T = size(beta,1);

baseScore = 0.5*sum(log(1./beta));
scores = zeros(N,T);
for t=1:T
    feat = para(t,1);
    theta = para(t,2);
    y = para(t,3);
    Theta = ones(N,1)*theta;
    if y==0
        scores(:,t) =X(:,feat)-Theta<=0;
    else
        scores(:,t) =  X(:,feat)-Theta>=0;
    end
    scores(:,t) = scores(:,t)*log(1/beta(t));
end
% Predict labels (0 for class1, 1 for class2)
predLab = sum(scores,2)>=baseScore;
% Tranform labels to 1 for class1 ,2 for class2
predLab = predLab+1;
end
```

g. Test The Implementation on Some Dataset

To test the AdaBoost function, I first created a simple 1-D dataset, as shown in Figure 5. This dataset has two outliers: object 9 in class1 and object 45 in class2. These two objects are, therefore, hard to classify. After train the AdaBoost classifier using this dataset, I visualized the weights for each object at each iteration, as shown in Figure 6. It illustrates that object 9 and 45, as we expected, get high weights in the beginning. Next, I tested on a more difficult dataset: banana

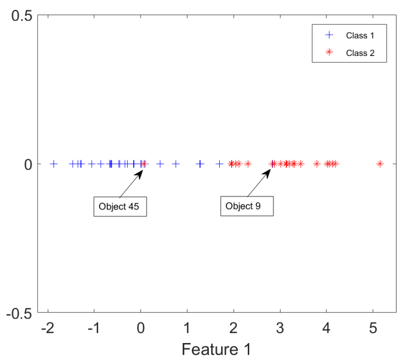

Figure 5: Scatter plot for two classes: 1-D simple case with 50 objects in total.

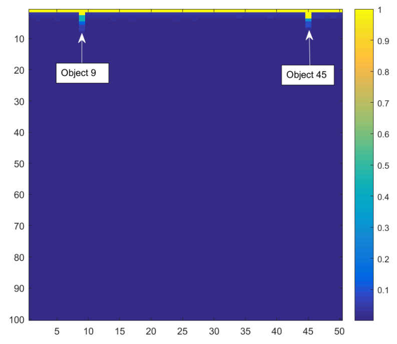

Figure 6: Visualize the weights for each object(X axis) at each iteration(Y axis).

dataset. This data is shown in Figure 7, with 50 observations in all. Still, objects which are hard to classify get higher weights than easier ones, as visualized in Figure 8. For example, objects 11 and 46 get much less weights, due to the fact that they are easy to classify.

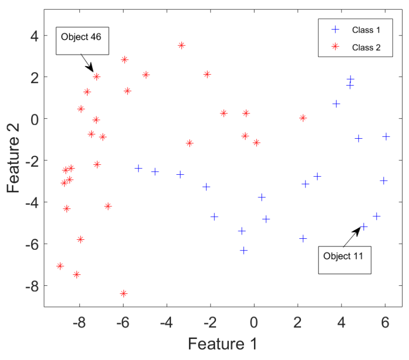

Figure 7: Scatter plot for two classes: Banana dataset with 50 observations in all

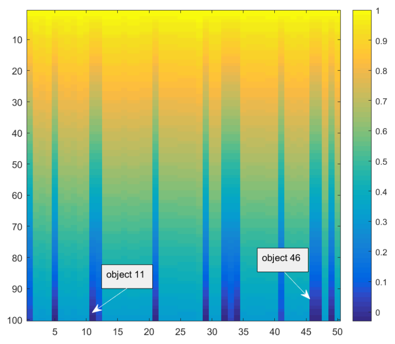

Figure 8: Visualize the weights for each object(X axis) at each iteration(Y axis). The weights are transformed in order to get a more uniformed distribution.

h. Test the implementation on dataset *optdigitsubset*

In this section, I tested the implementation on dataset *optdigitsubset*. I Used the first 50 objects for each class for training, and the rest for testing. The result is shown in Figure 9. The minimal testing error 0.007805 was obtained when iteration number was 17. Figure 9 shows that as iterations get bigger, the training error drops dramatically and keeps to zero. The testing error, however, as iterations increase, drops at first, then variates around a certain range.

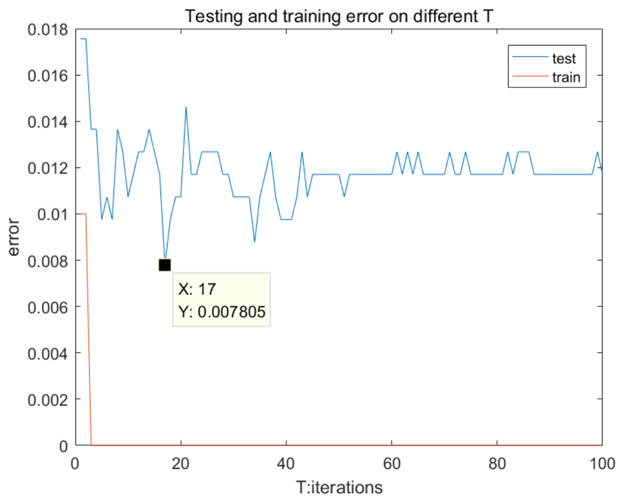

Figure 9: Testing and training error on different iterations.

After I found the optimal T: 17, I trained the AdaBoost classifier again and got all the weights for each object during each iterations. I visualized it in Figure 10. From this figure, I noticed that object 18 always got a high weight during all iterations. And object 66 and object 86 sometimes got high weights during the iterations. In order to figure out the reasons for their high weights, I visualized these images, as shown in Figure 11, 12,13. These images are somewhat not looking "good", as they all have some distortions or strange shapes. This evidence may justify them to be difficult classifying objects and thus having higher weights during iterations.

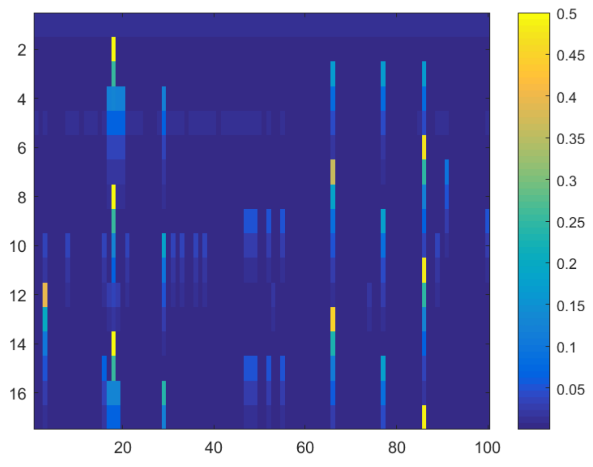

Figure 10: Visualize the weights for each object(X axis) during each iteration(Y axis).

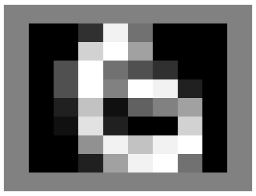

Figure 11: Object 18

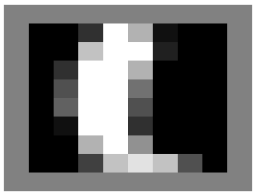

Figure 12: Object 66

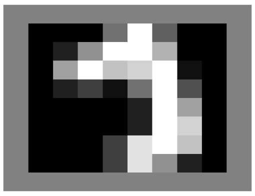

Figure 13: Object 86