UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Department of Electrical and Computer Engineering

ECE 544NA Pattern Recognition

Solutions 4

Fall 2013

Assigned: Tuesday, September 24, 2013 Due: Thursday, October 03, 2013

Reading: NNPR Chapter 3

Problem 4.1

The optimal classifier, denoted by $f_B(x)$, is the classifier that minimizes the expected Bayes risk

$$f_B(x) = \arg\min_f E_{Y|X=x}[L(y, f(x))]$$

in this case, $L(y, f(x)) = max(0, 1 - yf(x))$. Furthermore, $y \in \{-1, 1\}$ hence

$$f_B(x) = \arg\min_f [max(0, 1 + f(x))P(y = -1|x) + max(0, 1 - f(x))P(y = 1|x)]$$

The cases $f(x) \geq 1$ and $f(x) \leq -1$ are trivial (it is easy to see that the optimum is always at 1 or -1, respectively). Hence, we consider $-1 \leq f(x) \leq 1$:

$$f_B(x) = \arg\min_f [1 + f(x)(1 - 2P(y = 1|x))]$$

and $f_B(x) = \arg\max_{k \in \{-1,1\}} P(y = k|x)$ is the same as the Bayes optimal for the 0-1 loss. This indicates that not only is the hinge loss an upper bound on the 0-1 loss, but an *unconstrained* minimization of its risk also results in the MAP rule. Of course, in most practical settings, we are interested in *constrained* problems; for example, a linear classifier of the form $f(x) = w^T x + w_0$.

Matlab Exercises

Problem 4.2

Table 1 displays the classification error for all four methods, averaged across 100 random trials. As indicated, a simple linear classifier outperforms all of the other approaches. The advantages and disadvantages of the four algorithms are outlined in Table 2.

(c) In this case, a modest step size of 0.001 was selected; in practice, an adaptive step size that starts off large and gradually decreases also works well. The criterion selected for convergence here is a threshold on the training set classification error (stop learning if w can classify the training set sufficiently well). Alternatively, one could also threshold the change in the weight vector, w. In either case, it is good practice to renormalize the weight vector so that w does not expand into regions in which the gradient is trivial [i.e. regions in which $1 - tanh(.)^2$ is 0].

(d) Again, a step size of 0.001 was used for the perceptron update and the training set classification error was used as the stopping criterion. When the threshold is set to 0 (i.e. stop when classification error on the training set is 0), the algorithm does not converge; hence, the training set is not linearly separable.

(e) See Table 2 for a general overview

Table 1: Average classification error across 100 random trials

classifier	average error	standard deviation
least squares	**0.23**	**0.03**
1-nearest neighbor	0.29	0.03
tanh(.) nonlinearity	0.28	0.04
perceptron	0.35	0.08

Table 2: Advantages and disadvantages of the four classifiers

classifier	**advantages**	**disadvantages**
least squares	Extremely simple to train no additional parameters to tune	Cost function can penalize *correct* cases
k-nearest neighbor	Simple to implement	Classifier is a function of the training set value of k needs to be tuned
tanh(.)	Arbitrarily close approximation to the 0-1 loss	gradient descent (sensitive to initialization)
perceptron	Upper bound on the 0-1 loss simple update rule useful test for linear separability	Poor performance (when data are not linearly separable)

```
function results =  runexps(numtrials)
%runexps(numtrials) runs numtrials independent experiments for this
%problem set
%results contains results for 1) KNN, 2) least squares, 3) gradient descent
%with tanh(.) nonlinearity, and 4) the perceptron algorithm
results = zeros(4, numtrials);
load diabetes_normalized.mat
diab_labels = diabetes_normalized(:,1);
diab_features = diabetes_normalized(:,2:9);
numsamples = length(diab_labels);
numtrain = ceil(numsamples*0.8);
numtest = numsamples - numtrain;

for i = 1:numtrials
    randinds = randperm(numsamples); %random partitioning of the entire dataset
    traininds = randinds(1:numtrain); %first 80% is training data
    testinds = randinds(numtrain + 1: numsamples); %next 20% is test data

    mytraindata = diab_features(traininds,:);
    mytraindatabias = [mytraindata ones(numtrain, 1)]; %include the bias term

    mytestdata = diab_features(testinds, :);
    mytestdatabias = [mytestdata ones(numtest, 1)];

    y_train = diab_labels(traininds);
    y_test = diab_labels(testinds);
    %Linear classifier
    %learn the classifier
    w_linear = ((mytraindatabias'*mytraindatabias)) \ (mytraindatabias'*y_train);
    %classify test points
```

```
    y_linear = 2.*(mytestdatabias*w_linear >= 0) - 1;
    %compare with true labels
    results(1,i) = sum(y_linear ~= y_test)/length(y_test); %compute and record the error

    %Knn classifier with k = 1
    y_1nn = knn_classify(1, mytestdata, mytraindata, y_train);
    results(2,i) = sum(y_1nn ~= y_test)/length(y_test);

    %gradient descent with tanh(.) activation function
    w_tanh = graddesc(mytraindatabias, y_train, 0.001, 0.28);
    y_tanh = 2.*(mytestdatabias*w_tanh >= 0) - 1;
    %compare with true labels
    results(3,i) = sum(y_tanh ~= y_test)/length(y_test); %compute and record the error

    %the perceptron algorithm (hinge loss)
    w_perc = perceptron(mytraindatabias, y_train, 0.001, 0.28);
    y_perc = 2.*(mytestdatabias*w_perc >= 0) - 1;
    %compare with true labels
    results(4,i) = sum(y_perc ~= y_test)/length(y_test); %compute and record the error

end
```

k-nn classifier:

```
function knn_labels = knn_classify(k, mytestdata, mytraindata, y_train)
%this function classifies labels based on the knn classificaton rule
numtest = size(mytestdata, 1);
knn_labels = zeros(numtest, 1);

for i = 1:numtest %for each test point
    testpt = mytestdata(i,:);
    numtrain = size(mytraindata, 1);
    thedists = zeros(numtrain, 1);
    for j = 1:numtrain
        thedists(j) = norm(testpt - mytraindata(j,:));
    end

    [~,closestpts] = sort(thedists, 'ascend');
    if sum(y_train(closestpts(1:k))) >= 0
        knn_labels(i) = 1;
    else
        knn_labels(i) = -1;
    end

end
```

Gradient descent with $tanh(.)$ activation function:

```
function w = graddesc(traindata, trainlabels, eta, threshold)
%given the training data and labels, graddesc computes the optimal weight
%vector for a linear classifier y = w'x + b with a tanh(.) non-linearity
```

```
%using gradient descent
[~, numfeats] = size(traindata);
w = randn(numfeats, 1); %initialize to something small
w = w./norm(w);
while 1
    %gradient update
    w = w + eta*traindata'*((trainlabels - tanh(traindata*w)) .* (1-tanh(traindata*w)).^2);
    w = w./norm(w); %make sure w is not too large

    y = 2.*(traindata*w >= 0) - 1; %classify training set
    if sum(y ~= trainlabels)/length(trainlabels) < threshold
        break;
    end
end
```

The perceptron algorithm:

```
function w = perceptron(traindata, trainlabels, eta, threshold)
%Performs gradient descent using the hinge loss (perceptron algorithm)

[~, numfeats] = size(traindata);
w = randn(numfeats, 1); %initialize to something small
%note: w can also be initialized based on the training labels
w = w./norm(w);

while 1 %keep updating
    y = 2.*(traindata*w >= 0) - 1; %classify training set
    errorlocs = (y ~= trainlabels); %where do we make an error?

    if (sum(errorlocs) > 0) % if we make at least one error
        w = w + eta.*traindata(errorlocs,:)'*trainlabels(errorlocs);
    else
        disp('linearly separable');
        break;
    end

    if sum(errorlocs)/length(errorlocs) < threshold %convergence criterion
        break;
    end
end
```