1 Introduction

This lab explores the use of Adaboost algorithm on a set of Bayes Classifiers for classifying images of lower-case printed characters. Questions of interest include:

1. Is Adaboost approach good enough for this task?
2. What are a good set of features?
3. What is a good number of Bayesian classifiers for this task?
4. How does the number of histogram bins affect the overall classification performance?
5. What is a good way to select training and testing data?

2 Method

Adaboost algorithm is based on the principle that many bad classifiers can work together to come up with a better classifier. For this task, we just use Bayesian classifiers as the individual classifiers. Using these, we explore the character classification task with varying number of classifiers, binning techniques and feature sets. The following section includes the exhaustive list of features explored. In specific, 3 sets of varying feature combinations were explored. Set#1 comprising of higher and medium level features and Set#2 comprising of a combination of medium and low level features. The 3rd set is a range of lower to higher level features like euler number, edge orientation histogram etc. given by Dr. Tom Henderson and are not explained in this document.

2.1 Features

2.1.1 Set#1 - Slope Change Characteristics

This feature encodes the derivate of border pixels. Basically, for each side of the image viz., left, right, top and bottom, the border pixels as marked by the red pixels in the Figure 1, are extracted. For each such border pixel obtained, the non-varying pixel coordinate is stored. For example, when processing the left border, the column coordinates are stored and similarly, for the top border pixels, the row coordinates are stored. From each of the four 1D arrays of values thus obtained, the derivative arrays are computed by replacing an element with difference between the current and previous element. For thus obtained derivative array, the counts of positive, negative and zero values are then stored as the features. A fourth value encodes the sign/neutralness of the first value encountered in the derivative array. As there are 4 vectors, \(4 \times 4 = 16\) feature values are obtained from the whole image under this category. A full explanation of this feature using an example is given in the Verification Section listed later.

2.1.2 Set#1 - Full/Empty Row Characteristic

This characteristic is computed as follows:

\[ \text{feature} = \text{number of fully filled rows} - \text{number of empty rows} \]
Figure 1: Figure showing image of character $a$ with its border pixels marked in red
2.1.3 Set#1 - Full/Empty Column Characteristic
This characteristic is computed as follows:

\[ \text{feature} = \text{number of fully filled columns} - \text{number of empty columns} \]

2.1.4 Set#2 - Euler Number
This attribute is computed using the \texttt{bweuler} function in Matlab as follows:

\[ \text{Euler Number} = \text{number of objects in the image} - \text{total number of holes in the objects} \]

2.1.5 Set#2 - Number of Full Rows
This attribute is computed as follows:

\[ \text{feature} = \text{number of fully filled rows} \]

2.1.6 Set#2 - Number of Empty Rows
This attribute is computed as follows:

\[ \text{feature} = \text{number of empty rows} \]

2.1.7 Set#2 - Number of Full Columns
This attribute is computed as follows:

\[ \text{feature} = \text{number of fully filled columns} \]

2.1.8 Set#2 - Number of Empty Columns
This attribute is computed as follows:

\[ \text{feature} = \text{number of empty columns} \]

2.1.9 Set#2 - Top or Bottom?
This attribute is a Boolean that decides where more number of 1s occur (in a binary image) among the top and bottom halves of the pixels. And is computed as:

\[ \text{feature} = \text{count of 1s in the upper half} > \text{count of 1s in the lower half} \]

2.1.10 Set#2 - Left or Right?
This attribute is a Boolean that decides where more number of 1s occur (in a binary image) among the left and right halves of the pixels. And is computed as:

\[ \text{feature} = \text{count of 1s in the left half} > \text{count of 1s in the right half} \]

2.1.11 Set#2 - Count of protruding terminal pixels
This feature essentially decides how many pixels in the image are terminal. That is, it gives the count of pixels, for which, the number of 1s in its $3 \times 3$ neighborhood is 2 (including itself). Thus, it is computed as follows:

\[ \text{feature} = \text{pixels with } 3 \times 3 \text{ neighborhood counts 2} \]
2.1.12 Set#2 - Count of island pixels

This feature essentially decides how many pixels in the image are isolated. That is, it gives the count of pixels, for which, the number of 1s in its $3 \times 3$ neighborhood is 1 (including itself). Thus, it is computed as follows:

\[
\text{feature} = \text{pixels with } 3 \times 3 \text{ neighborhood counts 1}
\]

3 Pseudo Code

The primary code used in this assignment is listed Appendix A. The code that is used from previous assignments or is given by Prof. Tom Henderson for his course CS 6350/5350 class is not listed in this document. In this section, we just give pseudo code for one particular piece of code listed in the Appendix, \texttt{double X by Gaussian}, which is used to double the size of the input data set to generate a new data set based on the class-wise feature distribution in the original data set.

For each class C in X

For each Feature f in X

Compute Mean and Variance of f across all the data samples in C

Generate another array of feature samples using the above mean and a very small variance

Concatenate the new feature samples with the original samples to have a double-sized feature sample set for this class and for this feature

End

End

4 Verification of Program

This section presents the verification of the pseudo code listed in the above section.

\[
> \text{[newX, newT]} = \text{double X by Gaussian}\text{(ones(2),ones(2,1),1)}
\]

\[
\text{newX} =
\begin{array}{cc}
1.0000 & 1.0000 \\
1.0000 & 1.0000 \\
0.9869 & 1.0136 \\
0.9853 & 0.9962 \\
\end{array}
\]

\[
\text{newT} =
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
\end{array}
\]

which is expected. Hence the function is verified.
5 Data

This section gives graphs that illustrate the overall performance of Bayesian-classifier based Adaboost classifier and also show the factors affecting its performance.

Figure 2: Figure showing the trend of average percentage accuracy of the Adaboost algorithm across varying number of classifiers (including even number of classifiers). Each point is the average of accuracies from K=9 runs, while the error bars represent corresponding confidence intervals.

Figure 3: Figure showing average percentage accuracies of Adaboost across different feature sets. Each point represents the mean of percentage accuracies from K=9 runs, while the error bars represent corresponding confidence intervals. The number of classifiers used is 1.
Figure 4: Figure showing average percentage accuracies of Adaboost across varying number of histogram bins. Each point represents the mean of percentage accuracies from K=9 runs, while the error bars represent corresponding confidence intervals. The number of classifiers used is 1.

Figure 5: Figure showing the difference in percentage accuracies of Adaboost for the default and random data selection methods across varying number of classifiers. Each point represents the mean of percentage accuracy differences from K=9 runs, while the error bars represent corresponding confidence intervals.

6 Analysis

Figure 2 shows how Adaboost performance varies across different number of Bayesian classifiers used for voting. Even though, the current Adaboost implementation does not fully support even number of classifiers, the even numbers are still included in the graph to get an idea of how the general performance trends
across even and odd number of classifiers. As expected, one can see that for odd number of classifiers, the performance gets generally better with increase in the number of classifiers. Interestingly, even for the even number of classifiers, the general trend remains the same. This might be because, if majority of the Bayesian classifiers vote on the correct class, then even the even number of classifiers works well.

Figure 3 shows the performance of Adaboost across 3 different feature sets mentioned in Section 2. Only one Bayesian classifier is used by the Adaboost in each of these cases, thus giving more importance to features themselves than the number of classifiers. One can see that in general, all the three feature sets perform equally good and though no significant difference is observed, it appears that the feature set using higher level of features tends to have a smaller confidence interval.

Figure 4 shows the performance of Adaboost across different number of histogram bins chosen. Again, only one classifier is used and the input data set is doubled using the gaussain distribution (see Section 3). One can observe a general decreasing trend in the average percentage accuracy with increasing number of bins. This might be due to the fact that size of the data set (even after doubling) is still small and the number of feature samples falling into a single bin decrease with the increase in number of bins, there by increasing the chances of misclassification.

Figure 5 shows the comparison of the Adaboost performances using two different data selection methods. The default one selects first eight samples for the training and the last one for testing, and this is done in a rotating fashion for K = 9 runs. The random one does essentially the same thing, except that it randomly selects a permutation of the samples first, before operating on that. The graph shows the difference in performance between default and random cases, and one can see that no inference can be drawn on which works better from the graph, because the confidence intervals lie on both sides of the X-axis.

7 Interpretation

The answers to the questions posed are as follows:

1. Is Adaboost approach good enough for this task?
   Yes! One can clearly see from that Figure 2, that the performance is very good even for a small number of classifiers.

2. What are a good set of features?
   For this particular implementation, it seemed that the actual composition of features do not matter much as long as the individual features are acceptable and contributing some information to the classification. Though, it appears that if higher level of features are used, the performance gets more consistent.

3. What is a good number of Bayesian classifiers for this task?
   Any odd number of classifiers above 3 appear good for the current implementation.

4. How does the number of histogram bins affect the overall classification performance?
   More data is required to come up with any significant conclusions regarding this. The current results recite that without more data the general performance decreases because lesser number of feature samples occur in each bin.

5. What is a good way to select training and testing data? Two ways were explored in this implementation, viz., default and random. Both fared similar and neither of the methods appears to be better than the other.

8 Critique

Overall, Bayesian classifier based Adaboost performs well for this task of classifying lower case characters. Surprisingly, the results show that a small number of classifiers were sufficient to get good percentage
accuracies. As expected with the Adaboost, the actual features chosen do not significantly matter, but only the number of classifiers chosen matters. If more of the actual training data was available, it would have been a good experiment to verify how the performance varies with change in the number of histogram bins. Also, surprisingly, the Gaussian replication method appears to be sufficient for the current purpose of data replication. If more time was available, I would have generated larger data sets using this method and again verify the performance across varying numbers of bins. Another experiment of interest would be to see how Adaboost performs on a mixture of MLP, RBF and DT and Bayesian based classifiers instead of just the Bayesian classifiers. Yet another experiment of interest would be to use class-based weights for each classifier based on its previous experience with the classes.

9 Log

Time spent on coding:
- October 20th, 2012 from 2 PM to 5 PM

Time spent on writing the report, generating the results, analyzing the various relationships, coding and conducting experiments:
- October 22nd, 2012 from 2 PM to 4 PM
- October 23rd, 2012 from 10:30 PM to 12 PM
- October 24th, 2012 from 4 PM to 7 PM

A Code

Only the code relevant to this assignment is listed here. Rest of the code not listed here is already listed in the previous reports or is from Prof. Tom Henderson’s code base for CS 6350/5350 class, Fall 2012.

%File Name: Adaboost_Main.m
%Date: October 22, 2012

function [X, targets, data, percent_correct] = Adaboost_Main(num_classifiers, splitMode, MAX_BINS)
% Adaboost_Main - Trains, runs and tests a Adaboost classifier based on
% the available dataset
% On input:
% num_classifiers - the number of Bayesian classifiers used by the Adaboost
% On output:
% X - m*n by p matrix with data items for its rows and features as its columns
% targets - array with m*n elements that represent targets for the
% corresponding data elements
% data - array of structures corresponding to the number of runs.
% Each structure contains the following elements:
% train_X - training set
% test_X - testing set
% avg_train_X - mean of all training elements
% train_targets - targets for the training set
% test_targets - targets for the testing set
% avg_train_targets - target for the average representation
% percent_correct - array of real numbers carrying the percentage of correct
% classifications at each run
% Call:
% [X,targets,data,percent_correct] = Adaboost_Main;
% Author:
% Srivishnu Kaushik Satyavolu
% Fall 2012
% University of Utah

% Create some features on the data
features = cs6350_NN_image_features;

% Get the size of the feature vector
[m, n, p] = size(features);

% Organize features as training data
[X, targets] = CS6350_Adaboost_data_prep(features);

% [X, targets] = CS6350_DT_image_features_mod;
% [X, targets] = CS6350_Adaboost_image_features;

%m = 26;
%n = 9;
%p = size(X,2);

[X, targets] = double_X_by_Gaussian(X, targets, m);
% [X, targets] = double_X_by_Gaussian(X, targets, m);
% [X, targets] = double_X_by_Gaussian(X, targets, m);

% Split the data into training and testing data sets
if(strcmp(splitMode,'default'))
data = split_data(X, targets, m, size(X,1)/m, p);
elseif(strcmp(splitMode,'random'))
data = split_data_random(X, targets, m, size(X,1)/m, p, 800.0/9, 9);
end

n_trials = length(data);

h_waitbar = waitbar(0,'Ensemble Learning');

% For each run calculate the train, test and store the classifier performance
for trial_ind = 1:n_trials

waitbar(trial_ind/n_trials);
% Perform training
h = CS6350_Adaboost_train(data(trial_ind).train_X, ...
  data(trial_ind).train_targets, num_classifiers, MAX_EINS);

% Test and correct the percentage of characters correctly classified
percent_correct(trial_ind) = 0;

for test_ind = 1:size(data(trial_ind).test_X,1)
y = CS6350_Adaboost_recall(h, ...
  data(trial_ind).test_X(test_ind, :));
if y == data(trial_ind).test_targets(test_ind)
  percent_correct(trial_ind) ... 
    = percent_correct(trial_ind) + 1;
end
end
percent_correct(trial_ind) = percent_correct(trial_ind) * 100 ... 
   / size(data(trial_ind).test_X,1);

end
close(h_waitbar);
end

% File Name: double_X_by_Gaussian.m
% Date: October 22, 2012

function [newX, newTargets] = double_X_by_Gaussian(X, targets, m)
% double_X_by_Gaussian - doubles the input data set size using
% Gaussian distribution of how each features
% varies in a class
% On input:
% X - m*nSamples by p matrix with data items for its rows and
% features as its columns
% targets - array with m*n elements that represent targets for the
% corresponding data elements
% m - number of classes
% On output:
% newX - 2*m*nSamples by p matrix with data items for its rows and
% features as its columns
% newTargets - array with 2*m*n elements that represent targets for the
% corresponding new data elements
% Call:
% [newX, newT] = double_X_by_Gaussian(X, T, m);
% Author:
% Srivishnu Kaushik Satyavolu
% Fall 2012
% University of Utah

nRows = size(X, 1);
ncols = size(X, 2);
newX = zeros(2*nRows, ncols);
newTargets = zeros(2*nRows, 1);

nSamples = nRows/m;

% for each class look at feature distribution and generate dummy data
% based on the feature distribution
for class = 1:m
  for f = 1:ncols

    % Generate range of indices used in assignment
    destRowRange = 2*(class-1)*nSamples+1 : (2*class-1)*nSamples;
    srcRowRange = (class - 1)*nSamples+1: class*nSamples;

    % The first nSamples values are the same as the original matrix
    newX(destRowRange, f) = X(srcRowRange, f);

    % Compute feature distribution statistics using nSamples
    avgF = mean(X(srcRowRange, f));
    stdF = std(X(srcRowRange, f));

  end
end
% Use a small standard deviation, otherwise the feature set
% sample size is so small that the new data turns to be a
% bad approximation of the original data
newX(destRowRange + nSamples, f) = avgF ...  
  + 0.01.*randn(nSamples, 1);

% Generate new Targets matrix for the data
newTargets(destRowRange, 1) = class;
newTargets(destRowRange + nSamples, 1) = class;
end
end
end

%%File Name: split_data.m
%%Date: October 22, 2012

function data = split_data(X, targets, m, n, p)

    % split_data - splits the given data set into training and testing data sets
    % by choosing 8 samples of each class for training and the
    % remaining 1 for testing
    % On input:
    % X - m*n by p matrix with data items for its rows and features as its columns
    % targets - array with m*n elements that represent targets for the
    % corresponding data elements
    % m - number of different classes
    % n - number of given samples for each class
    % p - size of feature vector
    % On output:
    % data - array of structures corresponding to the number of runs.
    % Each structure contains the following elements:
    % train_X - training set
    % test_X - testing set
    % avg_train_X - mean of all training elements
    % train_targets - targets for the training set
    % test_targets - targets for the testing set
    % avg_train_targets - target for the average representation
    % Call:
    % data = split_data(X, targets, m, n, p);
    % Author:
    % Srivishnu Kaushik Satyavolu
    % Fall 2012
    % University of Utah

    n_total_sets = n;
    n_training_sets = 8 * n_total_sets / 9;
    n_testing_sets = n_total_sets - n_training_sets;

    for current_test_set_num = 1:n_total_sets

        % Calculate the range of training and testing sets
        train_range = (current_test_set_num - 1) * n_training_sets + ...
data(current_test_set_num).avg_train_X = zeros(m, p);

% Initialize the indices
index = 0;
train_index = 1;
test_index = 1;

% Split into K=9 disjoint sets, 8 into training and 1 into testing
for class_num = 1:m
for sample_num = 1:n
index = index + 1;
if( sample_num == current_test_set_num )
data(current_test_set_num).test_X(...
test_index, 1:p) = X(index,:);
data(current_test_set_num).test_targets(...
test_index, 1) = targets(index, 1);
test_index = test_index + 1;
else
data(current_test_set_num).train_X( ...;
train_index, 1:p) = X(index,:);
data(current_test_set_num).train_targets( ...;
train_index, 1) = targets(index, 1);
train_index = train_index + 1;
end
end
end

%Compute average representation
for class = 1:m
for ind =i:n_training_sets
data(current_test_set_num).avg_train_X(class, 1:p)...
    = data(current_test_set_num).avg_train_X( ...
        class, 1:p) + data(current_test_set_num) ...
        .train_X((class-1)*n_training_sets+ind, 1:p);
end
data(current_test_set_num).avg_train_X(class, 1:p) = ...;
data(current_test_set_num).avg_train_X(class, 1:p) .../
/n_training_sets;
data(current_test_set_num).avg_train_targets(class, 1) = class;
end
end
end

%File Name: split_data_random.m
%Date: October 22, 2012

function data = split_data_random(X, targets, m, n, p, percent_training ...,
    , nRandomTrials)
% split_data - splits the given data set into training and testing data sets
% by choosing a random permutation of percent_training of samples
% of each class for training and the rest for testing
% On input:
% X - m*n by p matrix with data items for its rows and features as its columns
% targets - array with m*n elements that represent targets for the
%   corresponding data elements
% m - number of different classes
% n - number of given samples for each class
% p - size of feature vector
% percent_training - gives the percentage of training samples to be chosen
%   for each class
% nRandomTrials - number of required random data sets
% On output:
% data - array of structures corresponding to the number of runs.
% Each structure contains the following elements:
% train_X - training set
% test_X - testing set
% avg_train_X - mean of all training elements
% train_targets - targets for the training set
% test_targets - targets for the testing set
% avg_train_targets - target for the average representation
% Call:
% data = split_data_random(X, targets, m, n, p, 90, 30);
% Author:
% Srivishnu Kaushik Satyavolu
% Fall 2012
% University of Utah

n_training = floor(percent_training*n/100.0);
N_testing = n - n_training;

for current_trial_num = 1:nRandomTrials
    data(current_trial_num).avg_train_X = zeros(m, p);
    for class = 1:m
        % Get a random permutation of the sample indices
        % of the current class
        indexes = randperm(n);

        % Take the first percent_training samples from each class into
        % training and the rest into testing data sets
        data(current_trial_num).train_X((class-1)*n_training ... + 1:1:n_training, :) = X((class-1)*n_training ... + indexes(1:n_training), :);
        data(current_trial_num).train_targets((class-1)*n_training ... + 1:1:n_training, :) = targets((class-1) ... * n_training + indexes(1:n_training), 1);
        data(current_trial_num).test_X((class-1)*n_testing ... + 1:1:n_testing, :) = X((class-1)*n_training ... + indexes(n_training+1:end), :);
        data(current_trial_num).test_targets((class-1)*n_testing ... + 1:1:n_testing, :) = targets((class-1) ... * n_training + indexes(n_training+1:end), 1);
end

% Compute average representations for each class
for class = 1:m
    for ind = 1:n_training
        data(current_trial_num).avg_train_X(class, p) = ...
        data(current_trial_num).avg_train_X(class, p) ...
        + data(current_trial_num).train_X((class-1) ... * n_training + ind, p);
    end

    data(current_trial_num).avg_train_X(class, p) = ...
    data(current_trial_num).avg_train_X(class, p) ...
    / n_training;
    data(current_trial_num).avg_train_targets(class, 1) = class;
end
end

References