Project A2: Using Radial Basis Functions to Classify Letters and Comparison to Multilayer Perceptrons

Scott Almquist

CS6350

September 20, 2012

Introduction

This report looks at using radial basis functions (RBFs) (see Marsland [1] chapter 4), to classify scanned images of characters. The questions I want to answer are:

1.) What features are effective for learning with an RBF?
2.) Should the input vector be all training samples or some representative subset?
3.) How does changing the distance function of the RBF affect the performance?
4.) How does changing the sigma value for the Gaussian of the distance function affect performance?
5.) How does the performance of RBF compare to a MLP?

Method

The code to train the RBF was based on the code posted on the course website [2]. I slightly modified the code so that the training function would be able to use a different distance function and sigma value. The code I used is included in Appendix A for reference.

Careful attention needs to be paid to divide the data into training and testing sets. For each letter, we have 9 examples that can be used for training and testing. I decided, arbitrarily, to use 5 letters for training and 4 for testing for each letter. The code which randomly divides the data into testing and training is listed in Appendix B.

I wrote a function to extract two types of features. One was based on pixel values, and the other low level features. For the pixel values, I selected the outermost pixels for each image, as it was indicated in class that this set of features would work well. For a low level feature, I simply thresholded the image to get a logical map and then used the number of pixels with a value of 1 in each row and column as the features. The code for obtaining features is in Appendix C. The code used to compare the two features sets is listed in Appendix D.

To answer the question about the basis vector, I decided to test the performance of the RBF using all training samples versus using the mode for each training sample. The mode should be representative of the character with the added benefit that any outliers that are producing poor results may be eliminated. The code used to compare using all training data versus just the mode of the training data vectors is listed in Appendix E.

Different distance functions can produce different performance for the RBFs. I decided to test 3 different distance functions: a normalized Gaussian (the code for which was provided on the course website), an unnorrmalized Gaussian:

```
function v = CS5350_Gaussian_D_unnorm(x,w,W,sigma)
%Provide a distance based off unnorrmalized Gaussian
%W and sigma are not used but provided for compatability.

d = norm(x-w);
v = exp(-(d^2)/(2*sigma^2));
```

And a function based off of the inverse of the Euclidian distance squared (with an offset to prevent division by zero):

```
function v = CS5350_Euclidian(x,w,W,sigma)
%Provide a distance based off inverse Euclidian distance squared between x
and w.
%W and sigma are not used but provided for compatability.
%
%   x = vector 1
%   w = vector 2

d = norm(x-w);
v = 1/(1+d^2);
```

Code that compares these methods is listed in Appendix F.

Comparing the difference in performance between multiple sigma values is relatively straight forward. I decide to try a sigma value of .1, 1, and 10 as I felt this would capture a large range of performance. The code to compare the performance of different sigma values is listed in Appendix G.

To compare the RBF to a MLP, I used the code for MLP provided on the course website. Since there are so many possible settings for the RBF and MLPs, I decided the best method of comparison would be to look at the best performing RBF versus the best performing MLP I had for Assignment A1. The code which compares them is listed in Appendix H.

Verification

To test the code to train and run the RBF, I used a logical AND:

```
>> X = [0,0;0,1;1,0;1,1];
>> targets = [0;0;0;1];
>> RBF_and = CS5350_RBF_train(X,targets);
>> CS5350_RBF_recall(RBF_and,[0 0])
ans =
   0.5000
>> CS5350_RBF_recall(RBF_and,[0 1])
ans =
   0.5000
```

```
>> CS5350_RBF_recall(RBF_and,[1 0])
ans =
   0.5000
>> CS5350_RBF_recall(RBF_and,[1 1])
ans =
   0.7311
```

As you can see, the code produces a much higher value for 1, 1 than it does the others, indicating that with proper thresholding the code should work.

For the code to divide the data into test/training data, we can simply use a case when there are 5 classes, 3 samples for each class and 1 sample will be testing data:

```
>> [train, test] = test_and_train(5,3,1)
train =
    2    3    5    6    7    8   11   12   13   15
test =
    1    4    9   10   14
>> [train, test] = test_and_train(5,3,1)
train =
    1    3    5    6    7    8   11   12   13   14
test =
    2    4    9   10   15
```

We can see it produces slightly different results for each run and for every class (1-3, 4-6, etc.) produces 2 training samples and 1 testing.

To test the feature extraction code, we can replace the first two images with one that is all ones and one that is all zeros, then verify that:

```
>> test = image_features(1);

>> test(1,1,:)

>> test(1,2,:)
```

Produces all zeros and all ones.

Additionally, we can test the low level feature if we know the size of the images (in this case it was 21).

```
test = image_features(1);

test(1,1,:) %produces all 21's

test(1,2,:) %produces all zeros
```

We can also test the distance functions. We will use two tests here, one where the distance between the vectors is zero, and one where it is one.

For our Gaussian, when the distance is zero we expect the distance to be exp(0) = 1. When the distance is 1, (and sigma is sqrt(.5) for ease), we will get exp(-1). For the inverse Euclidian, when the distance is zero we should get 1. When the distance is 1 we should get ½:

```
>> x1 = [0; 0; 0];
>> y1 = [0; 0; 1];
>> x2 = [1; 1; 1];
>> y2 = [1; 1; 1];
>> CS5350_Gaussian_D_unnorm(x1, y1, 1, sqrt(.5))
ans =
  0.367879441171442
>> >> CS5350_Gaussian_D_unnorm(x2, y2, 1, sqrt(.5))
ans =
    1
>> CS5350_Euclidian(x1, y1)
ans =
  0.500000000000000
EDU>> CS5350_Euclidian(x2, y2)
ans =
    1
```

Verification of the MLP code was done in the first assignment.

Data

Figure 1 is a plot of the error of the pixel based features minus the low level projection features for 30 different trials:

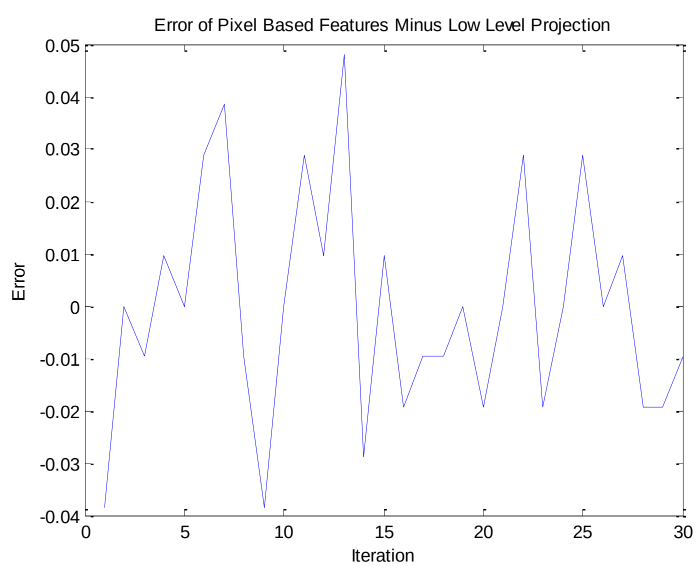

Figure 1

The plot has an average of -3.2051e-004with a standard deviation of 0.0216. The lower 90% confidence interval was -0.0371 and the upper 90% confidence interval was 0.0365.

Figure 2 is a plot of the error of using all testing data minus the error of using just the mode of the testing data

Figure 2

The plot has an average of -0.11122 with a standard deviation of 0.024957 . The lower 90% confidence interval was -0.15364 and the upper 90% confidence interval was -0.068791.

Figure 3 is a plot of the error of using a normalized Gaussian minus the error of using an unnormallized Gaussian for a distance function:

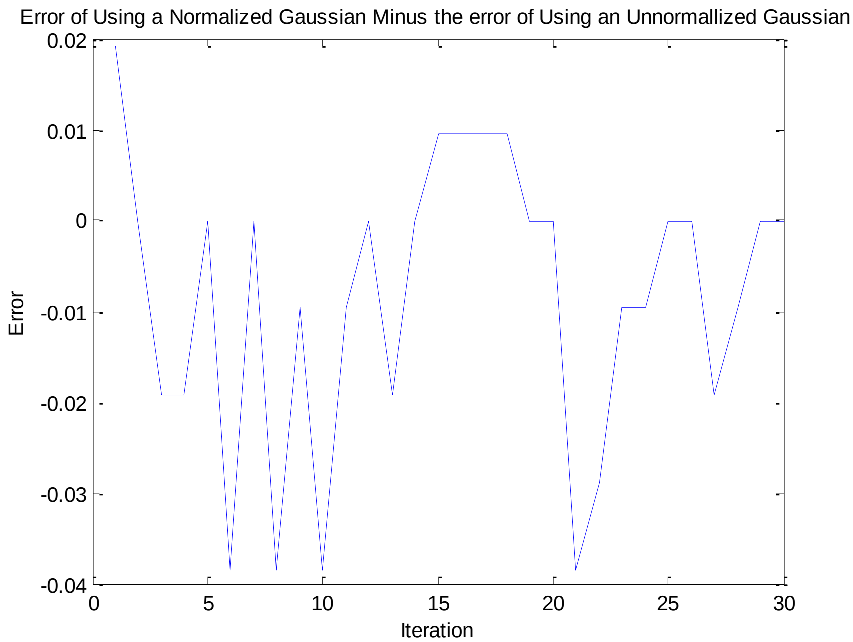

Figure 3

The plot has an average of -0.0083333 with a standard deviation of 0.015917 . The lower 90% confidence interval was -0.035392 and the upper 90% confidence interval was 0.018726.

Figure 4 is a plot of the error of using a normalized Gaussian minus the error of using an inverse Euclidian distance function:

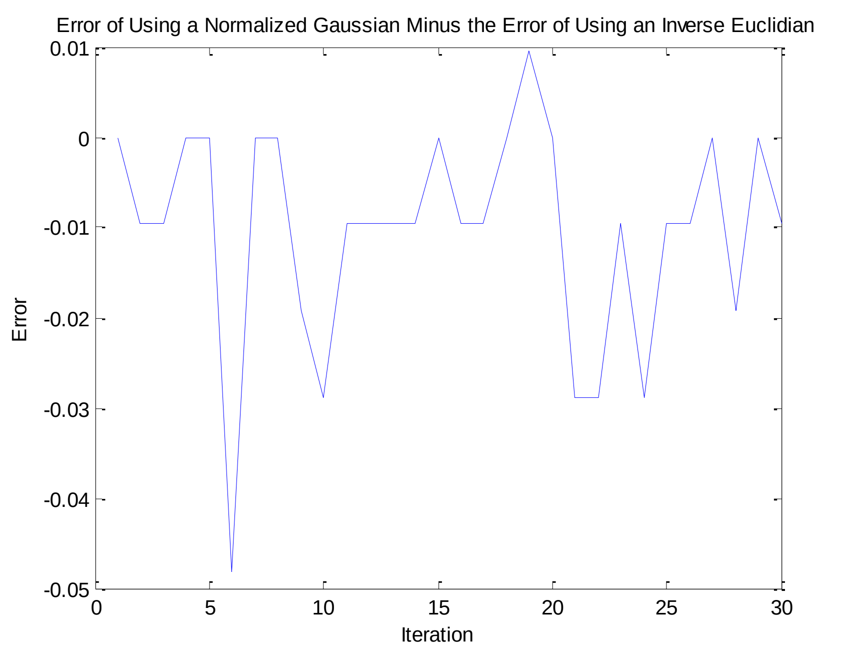

Figure 4

The plot has an average of -0.010256 with a standard deviation of 0.012353 . The lower 90% confidence interval was -0.031257 and the upper 90% confidence interval was 0.010744.

Figure 5 is a plot of the error of using a sigma value of 1 minus the error of using a sigma value of .1

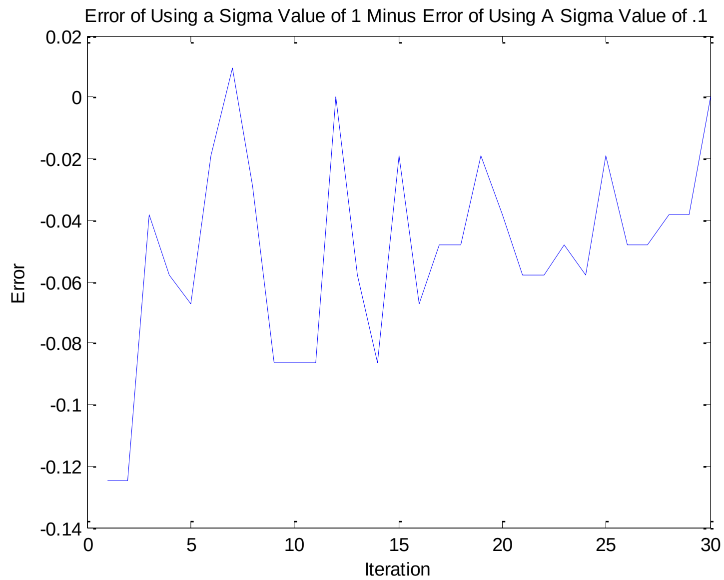

Figure 5

The plot has an average of -0.050321 with a standard deviation of 0.032601 . The lower 90% confidence interval was -0.10574 and the upper 90% confidence interval was 0.0051009.

Figure 6 is a plot of the error of using a sigma value of 1 minus the error of using a sigma value of 10:

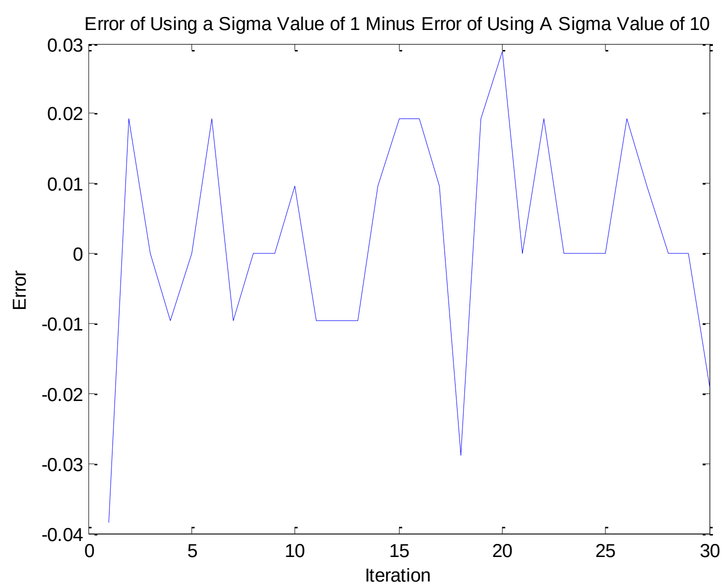

Figure 6

The plot has an average of 0.0022436 with a standard deviation of 0.015294 . The lower 90% confidence interval was -0.023756 and the upper 90% confidence interval was 0.028243.

Finally, Figure 7 is a plot of the error of using an RBF versus the error of using an MLP:

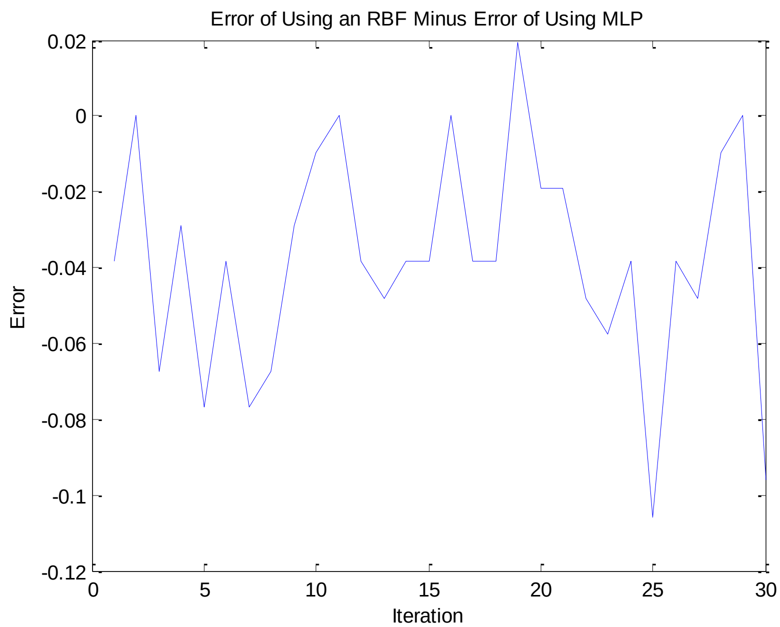

Figure 7

The plot has an average of -0.037821 with a standard deviation of 0.029332 . The lower 90% confidence interval was -0.087685 and the upper 90% confidence interval was 0.012044.

Analysis

Looking at the data for the feature extraction, it looks like both sets of features performed approximately equally. The average was incredibly close to zero. We can determine from this that both our pixel value features and projection features worked equally well. Since the projection features required fewer nodes, however, in a real application they would be preferable due to less computational space and time requirements.

When presented with the option of using all the testing data vs. just using the mode of the testing data, it is clear that using all of the testing data is vastly superior to using only the mode. It seems that the mode does not accurately represent the entire vector and allows for many errors to get through.

The tests for different distance measurements were somewhat inconclusive. Although the normalized Gaussian performed better on average, the confidence interval show that we cannot be very certain that it is better. It is likely that the normalized Gaussian is better, but I have not proved that it is.

For sigma values, it we seem to have the same issue. .1 on average performed worse than 1, however, not to within 90% confidence. It is likely that 1 is better, but these tests are not conclusive. For 1 versus 10, the performance was almost equal.

For MLP versus RBF, the results say that they are close to being equal. It is likely that RBF is slightly more accurate, but even if it is not, the computational requirements for RBF (several seconds) versus MLP (several minutes) make it a much more desirable method of classification for this problem.

Interpretation

1.) What features are effective for learning with an RBF?

We have shown that both pixel level features (of all of the outer pixels) and low level features (adding all of the threshold values for each row and column) are equally effective for classifying. The low level projections, however, are slightly better computationally.

2.) Should the input vector be all training samples or some representative subset?

It appears as if the mode of the training samples is not representative of training data and does not provide for accurate classification.

3.) How does changing the distance function of the RBF affect the performance?

A normalized Gaussian distance function is probably better than an unnormalized or inverse Euclidian distance function, but the results are somewhat inconclusive.

4.) How does changing the sigma value for the Gaussian of the distance function affect performance?

For very low values of sigma performance is slightly degraded, but for higher values performance is approximately equal.

5.) How does the performance of RBF compare to a MLP?

The RBF may be slightly better in terms of accuracy, however it is by far superior in terms of computational requirements.

Critique

Many of the tests we performed here turned out to be somewhat inconclusive. We can start to make guesses as to what the correct answer is, but our confidence intervals covered a very wide range of answers. Given more time it would be good to run a lot more experiments on more data to get a better idea of performance.

It seems as if the mode was not very representative of the training data. It would be good to investigate alternative methods for determining representative subsets of the data.

Log

Monday 2:00-4:00 PM

Tuesday 5:30:8:00 PM

Wednesday 3:30-10 PM

References

[1] S. Marsland. Machine Learning: An Algorithmic Perspective. CRC Press, Boca Raton, FL, 2009.

[2] T. Henderson. *Machine Learning CS5350/CS6350*. University of Utah.
http://www.cs.utah.edu/~tch/CS5350/

Appendix A: RBF Training Code

```
function nodes = CS5350_RBF_train(X,targets, gauss, sigma)
% CS5350_RBF_train - create radial basis network
% (see Machine Learning, Marsland, chapter 4)
% On input:
%     X (num_samps by x_dim array): training input samples
%     targets (num_samps by y_dim array): training output samples
%     gauss indicates the distance function (default) 1 = normalize gaussian
%           2 indicates unnormalized guassian, 0 indicates 1/(1+euclidian
%           disatnce)
%      sigma only for gaussian distances, indicates the standard deviation
%           of the distance.
% On output:
%     nodes (neural net data structure):
%        (i).layer (int): layer of node in network
%           .to (int vector): nodes connected to in next layer
%           .from (int vector): nodes coming from previous layer
%           .h (string): name of h function
%           .g (string): name of g function
%           .w (vector): weights to next layer nodes
%           .a (float): activation value (a = g(h(w,x)))
%           .inj (float): input value (inj = h(w,x))
%           .del (float): backprop error
% Call:
%     X = [0,0;0,1;1,0;1,1];
%     targets = [0;1;1;1];
%     RBF_or = CS5350_RBF_train(X,targets);
% Author:
%     T. Henderson
%     UU
%     Fall 2012
%

if nargin == 2
    gauss = 1;
    sigma = 1;
elseif nargin == 3;
    sigma = 1;
end

[num_samps,x_dim] = size(X);
[num_targets,y_dim] = size(targets);
layers = [x_dim,num_samps,y_dim];

num_layers = length(layers);
num_nodes = sum(layers) + 1;
for n = 1:num_nodes  % Create nodes
    if n<=x_dim
        nodes(n).layer = 1;

        if gauss == 1
            nodes(n).h = 'CS5350_Gaussian_D_norm';
        elseif gauss == 2
            nodes(n).h = 'CS5350_Gaussian_D_unnorm';
        else
```

```
            nodes(n).h = 'CS5350_Euclidian';
        end

        nodes(n).g = 'CS5350_AI_g_fun_ident';
    elseif ((x_dim<n)&&(n<=x_dim+num_samps))||(n==num_nodes)
        nodes(n).layer = 2;

        if gauss == 1
            nodes(n).h = 'CS5350_Gaussian_D_norm';
        elseif gauss == 2
            nodes(n).h = 'CS5350_Gaussian_D_unnorm';
        else
            nodes(n).h = 'CS5350_Euclidian';
        end

        nodes(n).g = 'CS5350_AI_g_fun_ident';
    else
        nodes(n).layer = 3;
        nodes(n).h = 'CS5350_AI_h_dot';
        nodes(n).g = 'CS5350_AI_g_fun_logit';
    end
    nodes(n).to = [];
    nodes(n).from = [];
    nodes(n).a = 0;
    nodes(n).inj = 0;
    nodes(n).w = [];
    nodes(n).del = 0;
    nodes(n).sigma = sigma;
end

% Create bias node
n = num_nodes;
nodes(n).layer = 2;
nodes(n).to = [x_dim+num_samps+1:num_nodes-1];
nodes(n).from = [];
nodes(n).h = 'CS5350_AI_h_dot';
nodes(n).g = 'CS5350_AI_g_fun_ident';
nodes(n).a = -1;
nodes(n).inj = -1;
nodes(n).w = zeros(layers(3),1);
nodes(n).del = 0;

layer_nodes(1).list = [1:x_dim];
layer_nodes(2).list = [x_dim+1:x_dim+num_samps,num_nodes];
layer_nodes(3).list = [x_dim+num_samps+1:num_nodes-1];

for n = 1:x_dim  % Set layer 1 nodes
    nodes(n).to = layer_nodes(2).list(1:end-1);
    nodes(n).w = zeros(layers(2),1);
end
for n = x_dim+1:x_dim+num_samps  % Set hidden layer nodes
    nodes(n).to = [x_dim+num_samps+1:num_nodes-1];
    nodes(n).w = zeros(layers(3),1);
    nodes(n).from = [1:x_dim];
end
for n = x_dim+num_samps+1:num_nodes-1  % Set output layer
```

```
    nodes(n).from = [x_dim+1:x_dim+num_samps,num_nodes];
end

% Assign input values as weights
for s = 1:num_samps
    for n = 1:x_dim
        nodes(n).w(s) = X(s,n);
    end
end

% Compute activations for each input
G = zeros(num_samps,layers(2)-1);
for s = 1:num_samps
    [yp,nn] = CS5350_RBF_recall(nodes,X(s,:));
    for n = x_dim+1:x_dim+num_samps
        G(s,n-x_dim) = nn(n).a;
    end
end

% Solve for hidden layer weights
for n = x_dim+num_samps+1:num_nodes-1
    yn = targets(:,n-(x_dim+num_samps));
%     wn = G\yn;    % Not robust
    wn = pinv(G)*yn;
    for nw = x_dim+1:x_dim+num_samps
        nodes(nw).w(n-(x_dim+num_samps)) = wn(nw-x_dim);
    end
end

tch = 0;
```

Appendix B: Code to Divide Data Into Training/Testing Sets

```
function [ train, test ] = test_and_train( classes, samples, num_test  )
%TEST_AND_TRAIN Provides indexes for testing and training data.
%
%   classes = number of classes available
%   sample = number of samples for each class
%   num_test = number of samples that should be in the test set
%   test = indexes of the test set (assumes all data is in one column)
%   train = indexes of training set

test = [];
train = [];

for ii = 1:classes
    total = randperm(samples) + samples*(ii-1);
    test = [test total(1:num_test)];
    train = [train total(num_test+1:end)];
end

test = sort(test);
train = sort(train);


end
```

Appendix C: Code for Feature Extraction

```
function [ features ] = image_features( pixel )
%FEATURES_ROW_VALUES Returns the features of the images. If pixel == 1 then
%the features are pixel level, otherwise it will return low level features.

letters = ['a','b','c','d','e','f','g','h','i','j','k','l','m',...
    'n','o','p','q','r','s','t','u','v','w','x','y','z'];
digits = ['1','2','3','4','5','6','7','8','9'];


if pixel
    features = zeros(26,9,80);
else
    features = zeros(26,9,42);
end

for l = 1:26
    let = letters(l);
    for d = 1:9
        dig = digits(d);
        filename = strcat('A1',let,dig,'.jpg');
        im_l_d = imread(filename);

        width = size(im_l_d, 2);
        height = size(im_l_d, 1);
        %all images the same size
        if width < 21 || height < 21
            im_l_d(21, 21) = 0;
        end
        im_l_d_bin = im_l_d>100;
        if pixel
            features(l,d,1:21) = im_l_d_bin(1,:);
            features(l,d,22:42) = im_l_d_bin(height,:);
            features(l,d,43:61) = im_l_d_bin(2:end-1, 1);
            features(l,d,62:end) = im_l_d_bin(2:end-1, width);
        else
            %use projections for non-pixel
            features(l,d,1:21) = sum(im_l_d_bin, 1);
            features(l,d,22:end) = sum(im_l_d_bin, 2)';
        end
    end
end

end
```

Appendix D: Comparing Pixel Level Features and Low Level Projections

```
clear;
close all;

features_pixel = image_features(1);
features_low = image_features(0);

%Organize features as training data
[X_pixel,targets_pixel] = CS5350_MLP_data_prep(features_pixel);
[X_low,targets_low] = CS5350_MLP_data_prep(features_low);

for ii = 1:30
    ii
    %Split data into training and testing sets
    [train_idx, test_idx] = test_and_train(26,9,4);

    X_train_pixel = X_pixel(train_idx,:);
    X_test_pixel = X_pixel(test_idx,:);
    targets_train_pixel = targets_pixel(train_idx,:);
    targets_test_pixel = targets_pixel(test_idx,:);

    X_train_low = X_low(train_idx,:);
    X_test_low = X_low(test_idx,:);
    targets_train_low = targets_low(train_idx,:);
    targets_test_low = targets_low(test_idx,:);

    RBF_nodes_low = CS5350_RBF_train(X_train_low, targets_train_low);
    RBF_nodes_pixel = CS5350_RBF_train(X_train_pixel, targets_train_pixel);


    err_RBF_low(ii) = CS5350_error_RBF(RBF_nodes_low, X_test_low,
targets_test_low);
    err_RBF_pixel(ii) = CS5350_error_RBF(RBF_nodes_pixel, X_test_pixel,
targets_test_pixel);
end
```

Appendix E: Comparing Using All Training Data Versus the Mode of Training Data

```
clear;
close all;

features_full = image_features(0);
%features_mode = mode(features_full, 2);

%Organize features as training data
[X, targets] = CS5350_MLP_data_prep(features_full);

for ii = 1:30
    ii
    %Split data into training and testing sets
    [train_idx, test_idx] =
test_and_train(size(features_full,1),size(features_full, 2), 4);
    X_train = X(train_idx,:);
    X_test = X(test_idx,:);
    targets_train = targets(train_idx,:);
    targets_test = targets(test_idx,:);

    train_idx = reshape(train_idx, 5, 26)';
    [~, dummy] = meshgrid(1:5, 0:25);
    train_idx = train_idx-9*dummy;
    clear features_mode
    for jj = 1:26
        features_mode(jj,:,:) = features_full(jj, train_idx(jj,:), :);
    end
    features_mode = mode(features_mode, 2);

    [X_mode, targets_mode] = CS5350_MLP_data_prep(features_mode);

    RBF_nodes_full = CS5350_RBF_train(X_train, targets_train);
    RBF_nodes_mode = CS5350_RBF_train(X_mode, targets_mode);

    err_full(ii) = CS5350_error_RBF(RBF_nodes_full, X_test, targets_test);
    err_mode(ii) = CS5350_error_RBF(RBF_nodes_mode, X_test, targets_test);

end
```

Appendix F: Code Comparing Different Distance Functions

```
clear;
close all;

features = image_features(0);

%Organize features as training data
[X,targets] = CS5350_MLP_data_prep(features);

for ii = 1:30
    %Split data into training and testing sets
    [train_idx, test_idx] = test_and_train(26,9,4);

    X_train = X(train_idx,:);
    X_test = X(test_idx,:);
    targets_train = targets(train_idx,:);
    targets_test = targets(test_idx,:);


    RBF_nodes_ngauss = CS5350_RBF_train(X_train, targets_train, 1, 1);
    RBF_nodes_ugauss = CS5350_RBF_train(X_train, targets_train, 2, 1);
    RBF_nodes_euclid = CS5350_RBF_train(X_train, targets_train, 0, 1);

    err_RBF_ngauss(ii) = CS5350_error_RBF(RBF_nodes_ngauss, X_test,
targets_test);
    err_RBF_ugauss(ii) = CS5350_error_RBF(RBF_nodes_ugauss, X_test,
targets_test);
    err_RBF_euclid(ii) = CS5350_error_RBF(RBF_nodes_euclid, X_test,
targets_test);

end
```

Appendix G: Code Comparing Different Sigma Values

```
clear;
close all;

features = image_features(0);

%Organize features as training data
[X,targets] = CS5350_MLP_data_prep(features);

for ii = 1:30
    ii
    %Split data into training and testing sets
    [train_idx, test_idx] = test_and_train(26,9,4);

    X_train = X(train_idx,:);
    X_test = X(test_idx,:);
    targets_train = targets(train_idx,:);
    targets_test = targets(test_idx,:);


    RBF_nodes_tenth = CS5350_RBF_train(X_train, targets_train, 1, .1);
    RBF_nodes_one = CS5350_RBF_train(X_train, targets_train, 1, 1);
    RBF_nodes_ten = CS5350_RBF_train(X_train, targets_train, 1, 10);

    err_RBF_tenth(ii) = CS5350_error_RBF(RBF_nodes_tenth, X_test,
targets_test);
    err_RBF_one(ii) = CS5350_error_RBF(RBF_nodes_one, X_test, targets_test);
    err_RBF_ten(ii) = CS5350_error_RBF(RBF_nodes_ten, X_test, targets_test);
end
```

Appendix H: Code to Compare RBFs and MLPs

```
clear;
close all;

features = image_features(0);

%Organize features as training data
[X,targets] = CS5350_MLP_data_prep(features);

for ii = 1:30
    ii
    %Split data into training and testing sets
    [train_idx, test_idx] = test_and_train(26,9,4);
    X_train = X(train_idx,:);
    X_test = X(test_idx,:);
    targets_train = targets(train_idx,:);
    targets_test = targets(test_idx,:);

    %MLP nodes
    nodes = [size(features, 3), 50, 26];
    num_trials = 100;

    [MLP_nodes,trace] = CS5350_AI_backprop(X_train,targets_train, nodes,
num_trials);

    err_MLP(ii) = CS5350_error_MLP(MLP_nodes, X_test, targets_test);

    RBF_nodes = CS5350_RBF_train(X_train, targets_train);

    err_RBF(ii) = CS5350_error_RBF(RBF_nodes, X_test, targets_test);

end
```