Hand-Written Digit Recognition

Gage Henrich

Abstract

This paper lists and describes the techniques performed to formulate a program for hand-written digit recognition. The program is trained through a series of linear programs which determine support vector classifiers to discriminate between data points. Data points are 728 values ranging from 0 to 256 which indicate the grayscale values of a 28x28 image. A DAG (directed acyclic graph) program is implemented to determine a set of 10,000 images based on the support vector classifiers.

1 Introduction

Handwriting recognition is a commonly studied programming problem, and linear programming is often employed to reach a solution. Today, the large majority of letters sent through the United States Postal Service are read and sorted entirely by computer algorithms [1]. Algorithms are "trained" to recognize letters or values based on a large set of training values, from which it formulates functions to discriminate between data points of grayscale picture values. An example test set is shown in Figure 1.

Figure 1: Sample images from CENPARMI database. [2]

In this paper, we will explore the use of linear programming to effectively discriminate and classify hand-written digits. We will show that this can be achieved by formulating an l_2-*norm* minimization to determine support vector classifiers. We will also introduce and explain the implementation of a large-margin DAG for image classification.

2 LP Formulation

Linear programming is employed to formulate linear functions which discriminate between input values. Given two different sets of data, we seek a hyperplane that effectively separates the sets of two points. This linear discrimination can be achieved with a robust approach, where we seek a function which maximizes the gap in values between the two sets. This is shown in Figure 2.

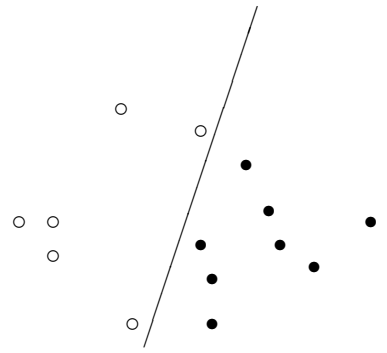

Figure 2: Two data points classified by a linear affine function

In our case, the sets of points cannot be completely separated linearly, so we instead determine linear affine functions that seek to minimize the number of misclassified points. This is a difficult optimization problem to solve. By introducing non-negative slack variables, u and v, we can relax the problem and formulate the problem as an l_2-norm minimization to solve for support vector classifiers. The support vector classifier for the sets of data $\{x_1, \cdots, x_N\}$ and $\{y_1, \cdots, y_M\}$, is the solution of:

$$\begin{array}{rl} \text{minimize} & \|\boldsymbol{a}\|_2 + \gamma(\mathbf{1}^T u + \mathbf{1}^T v) \\ \text{subject to} & a^T x_i - b \geq 1 - u_i \ , \ \ i = 1, \cdots, N \\ & a^T x_i - b \leq -(1 - v_i) \ , \ \ i = 1, \cdots, M \\ & u \geq 0 \ , \ \ v \geq 0 \end{array} \tag{1}$$

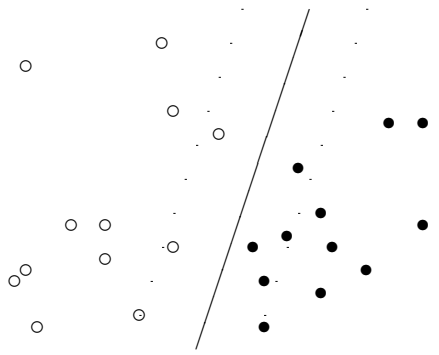

Figure 3: Linear discrimination via support vector classifier

Figure 3 shows linear discrimination of two data sets via support vector classification. These slack variables act as offsets from the support vector which define a "slab" which we seek to maximize. The solid line represents the support vector, given by a and b, and the dotted lines, offset from the support vector by $(1-u)$ and $-(1-v)$, represent the bounds of the "slab." Returning to equation 1, we are trying to minimize the l_2-*norm* of a. This term is proportional to the inverse of the width of our bounds, $-1 \leq a^T - b \leq 1$, which is given by [3]:

$$\frac{2}{\|\boldsymbol{a}\|_2}$$

The second term of the minimization, $\gamma(\mathbf{1}^T u + \mathbf{1}^T v)$, represents the relaxation of the number of misclassified points. The parameter γ is used in weighting the number of misclassified points to the width of our slab. Essentially, we are trying to minimize the number of misclassified points while maximizing our bounds for classification.

3 Determining Support Vector Classifiers

We are given a 60,000 x 728 vector representing 60,000 images, each consisting of 728 pixels, which we will call the *Images* matrix. Additionally, we are given a 60,000 x 1 vector indicating the actual value 0-9 of each image, which we will call *Labels*. Thus, if the second element in *Labels* is a 2, we know that the second row of *Images* is a hand-written 2. These matrices will be used to determine support vectors for classification and essentially "train" our program to recognize a digit by its pixels.

To do this, we need to sort our images into separate matrices corresponding to their numerical value. This is achieved by finding all the rows of *Labels* that correspond to a given number, and then lumping those same rows of *Images* into a matrix. Thus the matrix *Zeros* corresponds to all the rows of *Images* representing a 0. Since these matrices are not going to be the same size (there are not the same number of values 0-9), they cannot be organized into one larger matrix. Instead, we will store them in a "cell" called *Values* in Matlab.

$$Values = \{\ [Zeros]\quad [Ones]\quad [Twos]\quad \cdots\quad [Nines]\ \} \tag{2}$$

The matrices are separated and stored in this manner so each individual set of images can be compared to every other. This will allow us to determine classifiers between every combination of values. The number of classifiers, k is given by:

$$k = \frac{N(N-1)}{2} \tag{3}$$

where N is the number of data sets. From equation 1, we have 45 total combinations. A vector *svms* is build corresponding to these 45 combinations.

$$svms = \left\{ \begin{array}{c} 01 \\ 02 \\ \vdots \\ 12 \\ 13 \\ \vdots \\ 23 \\ 24 \\ \vdots \\ 89 \end{array} \right\} = \left(\begin{array}{c} 0vs1 \\ 0vs2 \\ \vdots \\ 1vs2 \\ 1vs3 \\ \vdots \\ 2vs3 \\ 2vs4 \\ \vdots \\ 8vs9 \end{array} \right) \tag{4}$$

We call this matrix *svms* because each element within it represents an SVM that will be performed. This matrix is used to solve the LP to discriminate between every set of images. With our values separated and combinations now defined, we can now solve the optimization problem for our 45 classifiers. In a relatively simple loop in Matlab, we are able to loop through *svms*, and call the appropriate matrices in *Values* to formulate each LP. In Matlab, we utilize the program CVX, which is a modeling system for convex optimization.

For each of the 45 linear discriminations necessary for complete classification, we will perform a convex optimization using CVX. Starting with a γ of 1, we run through all the rows of *svms* and solve for A and b. The 45 A's and b's will be stored in cells in the following method:

$$\left\{ \begin{array}{ccccccccc} [] & 0vs1 & 0vs2 & 0vs3 & 0vs4 & 0vs5 & 0vs6 & 0vs7 & 0vs8 & 0vs9 \\ [] & [] & 1vs2 & 1vs3 & 1vs4 & 1vs5 & 1vs6 & 1vs7 & 1vs8 & 1vs9 \\ [] & [] & [] & 2vs3 & 2vs4 & 2vs5 & 2vs6 & 2vs7 & 2vs8 & 2vs9 \\ [] & [] & [] & [] & 3vs4 & 3vs5 & 3vs6 & 3vs7 & 3vs8 & 3vs9 \\ [] & [] & [] & [] & [] & 4vs5 & 4vs6 & 4vs7 & 4vs8 & 4vs9 \\ [] & [] & [] & [] & [] & [] & 5vs6 & 5vs7 & 5vs8 & 5vs9 \\ [] & [] & [] & [] & [] & [] & [] & 6vs7 & 6vs8 & 6vs9 \\ [] & [] & [] & [] & [] & [] & [] & [] & 7vs8 & 7vs9 \\ [] & [] & [] & [] & [] & [] & [] & [] & [] & 8vs9 \end{array} \right\} \tag{5}$$

Each A matrix is 728 elements long and each b is a single value. They are stored in this method for ease of classification when running the DAG. With our A's and b's defined, we have now effectively 'trained' our program. They will be used within the DAG to classify images according to where they lie relative to each SVM. Note the u and v values are also output from CVX, since they are a part of the optimization. However, they are not necessary for classification, which will be explained in greater detail the next section.

4 DAG Classification

4.1 Overview

We implement Decision Directed Acyclic Graphing with our support vectors for image classification. A decision DAG is a set of boolean expressions that test an input to determine its classification. The DAG for classes '0vs2' is shown in Figure 4.1.

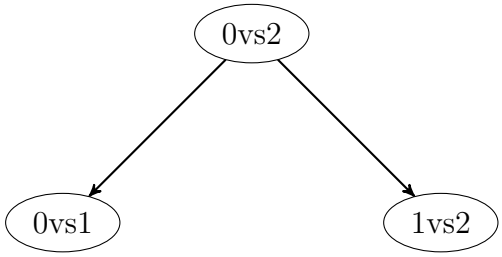

Figure 4: Decision DAG for finding the best out of three classes

Given a data set, the DAG in Figure 4.1 utilizes the corresponding support vectors to determine classification between classes 0 and 2. Then it continues on to the '0vs1' support vector if it determines it is not a 2; conversely, if the DAG determines it is not a 0, it continues to the '1vs2' case. It is here that we use the 45 support vectors we determined in the previous section. Using our A and b cells that are shown in (5), we are able to easily implement a DAG to determine hand-written digits.

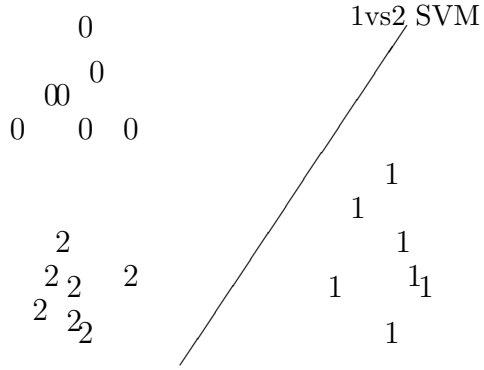

Figure 5: 1vs2 SVM shown in the space of three-class data set

Figure 5 illustrates the 1vs2 support vector in the space corresponding to the DAG in Figure 4.1. The hyperplane to the left of this line represents the 'not 1' and the hyperplane to the right indicates a 'not 2'. Thus, from the flow of the DAG, if the image being tested is in the left hyperplane, the image is be classified as a 2, since the 0vs2 SVM was already tested. Together, Figures 4 and 5 provide an accurate representation of how the DAG and support vectors are used in classification.

4.2 Implementation

We are given a 10,000x728 matrix corresponding to 10,000 test images. We essentially want to run through all of the rows of this matrix and test each with our 45 SVMs. This is where we will implement a DAG as a method of image classificaton. The DAG we will implement is similar to the one shown in Figure 4, except that it is much larger, since there are ten classes rather than three.

The pass/fail criteria for each node of our DAG depend upon the corresponding SVM, which is based on the A and b. Here, we denote the cells which contain the A matrices and b values as $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{b}}$.

$$\hat{\boldsymbol{A}}, \hat{\boldsymbol{b}} = \left\{ \begin{array}{ccccccccc} [] & 0vs1 & 0vs2 & 0vs3 & 0vs4 & 0vs5 & 0vs6 & 0vs7 & 0vs8 & 0vs9 \\ [] & [] & 1vs2 & 1vs3 & 1vs4 & 1vs5 & 1vs6 & 1vs7 & 1vs8 & 1vs9 \\ [] & [] & [] & 2vs3 & 2vs4 & 2vs5 & 2vs6 & 2vs7 & 2vs8 & 2vs9 \\ [] & [] & [] & [] & 3vs4 & 3vs5 & 3vs6 & 3vs7 & 3vs8 & 3vs9 \\ [] & [] & [] & [] & [] & 4vs5 & 4vs6 & 4vs7 & 4vs8 & 4vs9 \\ [] & [] & [] & [] & [] & [] & 5vs6 & 5vs7 & 5vs8 & 5vs9 \\ [] & [] & [] & [] & [] & [] & [] & 6vs7 & 6vs8 & 6vs9 \\ [] & [] & [] & [] & [] & [] & [] & [] & 7vs8 & 7vs9 \\ [] & [] & [] & [] & [] & [] & [] & [] & [] & 8vs9 \end{array} \right\} \tag{6}$$

The layout of the cell shown above is essentially a DAG. Starting at the top right element of 0vs9, if an image is not a 9, it continues down the DAG to 0vs8 and so on. Thus our rationale for storing A and b this way becomes apparent. We can simply run through the rows of $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{b}}$ and test each SVM. For a given test image, *Image*, the pass/fail criteria for each node of each SVM depend on the following expression:

$$Image * A^T - b \tag{7}$$

For the 0vs1 case, we test $Image * \hat{\boldsymbol{A}}(1,2)^T - \hat{\boldsymbol{b}}(1,2)$. If the expression is greater than 0, we know it is not a 1. Conversely, if it not greater than 0, we know the image is not a 0. Following this expression, we can formulate a relatively simple method of DAG implementation. Starting at$\hat{\boldsymbol{A}}(1,2)$ and $\hat{\boldsymbol{b}}(1,2)$, we run across all the columns of cells $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{b}}$ until we reach $\hat{\boldsymbol{A}}(1,9)$ and $\hat{\boldsymbol{b}}(1,9)$. If the boolean expression:

$$Image * A^T - b > 0 \tag{8}$$

is true for all of the columns of the first row (the sum off all 9 tests is 9) we know the image is a 0. If an image is less than zero for one of the tests and the sum is subsequenly less than 9, we move down to the second rows of $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{b}}$. If the boolean given by equation 8 is true for each of the tests in the second row, we know the image is a 1. We follow this all the way down to the last rows of $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{b}}$.

We know have a simple and effective method of characterizing our 10,000 test images based on our SVMs. The code of this DAG implementation is shown in Appendix A.

5 Results

We were asked to determine the values of 10,000 hand-written digits. Additionally, we were given a vector corresponding to the actual values of these digits to test the accuracy. Given the extensive amount of calculations necessary to perform optimizations for all 45 SVMs, the code takes a considerable amount of time to run (around three hours).

Starting with $\gamma = 1$, the accuracy was found to be approximately 90%, meaning the program misclassified 1,000 images. The γ value directly influences the accuracy of the classification since it applies a weight for the SVMs. Thus, the gamma was adjusted to decrease the accuracy. Since the run-time of the program is so long, gamma values were adjusted and the code was run overnight. It was determined that a smaller γ resulted in a smaller classification error. Table 1 lists the tested gamma values and corresponding accuracies.

γ	**Accuracy (%)**
1	89.96
0.01	90.67
0.0001	92.59
0.00001	93.94
0.00005	94.57
0.000075	94.60

Table 1: Tested gamma values

As indicated by Table 1, for the tested gamma values, the optimal was 0.000075, resulting in an error of only 5.4%. Thus, for the final optimization performed, only 540 were misclassified. This error could be further reduced with the addition of preliminary image processing. Image number 869 of the 10,000 test images is shown in Figure 6.

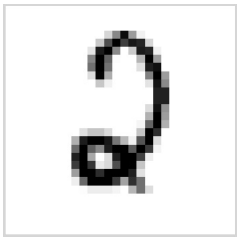

Figure 6: One of the hand-written test images

The border arround the 728 pixels that make up the image suggests the majority of the image is attributed to white space. The number of pixels that make up the images could have been reduced to omit the areas where all images had white space. This would have involved converting the 1x728 image vectors to 28x28 matrices and determining where the grayscale value changed from zero from the left, right top and bottom of every image. This would certainly reduce the run-time of the optimizations performed, and may possibly reduce the classification error.

References

[1] Mauk, Ben. *Life's Little Mysteries*. 20 November 2012. "How Do Post Office Machines Read Addresses?" http://www.lifeslittlemysteries.com/198-how-do-post-office-machines-read-addresses.html

[2] Center for Pattern Recognition and Machine Intelligence. 2010. http://www.cenparmi.concordia.ca/

[3] Boyd and Vanderberghe. *Convex Optimization*. Cambridge University Press. 2004

A Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%   Handwriting Recognition Program   %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%       MCEN 4228  Optimal Design     %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%            Gage Henrich             %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%% prepare and categorize data ----------------------------------
%%% load Training and 'images' Data
load mnist;

%%% categorize training values
zero = find(labels == 0);
one = find(labels == 1);
two = find(labels == 2);
three = find(labels == 3);
four = find(labels == 4);
five = find(labels == 5);
six = find(labels == 6);
seven = find(labels == 7);
eight = find(labels == 8);
nine = find(labels == 9);

%%% lump like training values into separate matrices
vals{1} = images(zero,:);
vals{2} = images(one,:);
vals{3} = images(two,:);
vals{4} = images(three,:);
vals{5} = images(four,:);
vals{6} = images(five,:);
vals{7} = images(six,:);
vals{8} = images(seven,:);
vals{9} = images(eight,:);
vals{10} = images(nine,:);


%%% build 'svms' vector for all combinations ----------------------
%%% 0 vs 1, 0 vs 2, ... ,1 vs 2,1 vs 3, ... , 2 vs 3, ... , 8 vs 9
left = [];
for i = 1:10
    left = [left; (i-1) * ones(10-i,1)];
end
length = (0:9)';
```

```
place = [tril(ones(10))];
place(:,1) = [];
right = [];
for i = 1:9
    right = [right; length.*place(:,i)];
    right(right == 0) = [];
end

svms = [left right];


%%% classification through optimization ---------------------------
%%% run cvx through all combinations given by 'svms' matrix constructed
%%% above to compare all values of 'images' and determine determine
%%% support vector classifiers
g = 0.000075;
A = [];
B = [];
U = [];
V = [];

tic;
for i = 1:45
    clear Set1
    clear Set2
    for j = 1:10
        if svms(i,1) == j-1
            Set1 = vals{j};
        end
        if svms(i,2) == j-1
            Set2 = vals{j};
        end
    end
R = max(size(Set1));
Q = max(size(Set2));
cvx_begin
cvx_precision low
variables a(784) b(1) u(R) v(Q)
minimize (norm(a) + g*(ones(1,R)*u + ones(1,Q)*v))
double(Set1)*a - b >= 1 - u;
double(Set2)*a - b <= -(1 - v);
u >= 0;
v >= 0;
cvx_end
A = [A; a'];
```

```
B = [B; b];
U = [U; max(u)];
V = [V; max(v)];
end
time = toc;

%%% reshape A to be an upper-triangular matrix of the form:
%%% [] 0v1 0v2 ... 0v9
%%% [] []  1v2 ... 1v9
%%%  .  .   .   .   .
%%%  .  .   .   .   .
%%%  .  .   .   .   .
%%% [] []   [] ... 8v9
count1 = 1;
count = 2;
for i = 1:9
    for j = count:10
    Astar{i,j} = A(count1,:);
    Bstar{i,j} = B(count1);
    count1 = count1 + 1;
    end
    count = count+1;
end

save Henrich_Dag Astar Bstar

%%% dag ------------------------------------------------------------
%%% directed acyclic graphing employed for image determination
%%% loops through images_test and compares rows using support vectors
%%% for classification

images_test = double(images_test);
images_pred = 10*ones(10000,1);
for i = 1:10000
    count=2;
    count2 = 9;
    value = 0;
    for j = 1:9
        True = 0;
        for jj = count:10
           if images_test(i,:)*Astar{j,jj}' - Bstar{j,jj} > 0
            True = True + 1;
           end
        end
        if True == count2;
```

```
            images_pred(i) = value;
            break
        end
        count=count+1;
        count2=count2-1;
        value=value+1;
    end
end
ten = find(images_pred == 10);
images_pred(ten) = 9;


%%% calculate accuracy of prediction based on labels
accuracy = sum(images_pred == labels_test)/10000


%%% plot a handwritten digit
nHold = 1253;
IMAGE = reshape(images_test(nHold,:),28,28)';
labels_test(nHold)
figure,imagesc(IMAGE)
colormap(flipud(gray(256)))
axis equal
set(gca, 'YTick', []);
set(gca, 'XTick', []);
axis off
```