MCEN 5125: Optimal Design

Classification of Handwritten Digits

Jacob Haimes

1 Introduction

Even now, after the technological breakthroughs of telecommunications and mobile computers, the United States Postal Service alone delivers well over 100 billion units of mail each year [1]. Imagining how many man-hours it would take to process and deliver all of this mail quickly makes it apparent that the automation of any portion of this process would be invaluable. One way to improve the efficiency of this task would be to use a computer to assist in sorting of mail by recipient address.

In this paper, we will formulate a binary classification problem as a minimization problem in §2.1. Minimization problems can be solved in many different ways, but here we will explore two different methods: least squares in §2.2 and linear programming in §2.3. §3 will then present the techniques that were used for pre-processing. Various classifiers are then put to the test in §4, where we explore the differences between the accuracy of the classifiers that we generate. Using our results as a guide, we will introduce the concept of feature engineering and attempt some ourselves. Finally, we will summarize our findings and reflect on our methods in §5.

2 Classifier Problem Formulation

2.1 Classification to Minimization

If you were given a handful of coins and told to sort them, what would your first step be? Perhaps you would sift through them to take out those with unique color, such as pennies. Maybe you would construct a coin sorting contraption, like the one in Fig. (1), and utilize gravity to sort the coins by size. These characteristics (size and color) are examples of *features*, attributes of a set of objects that can be used to sort those aforementioned objects into some set of groups. The features that we use in our problems are explained in detail in §3.

Using this coin sorting example as a guide, we will generalize a way to classify some object $\mathcal{O}$ into either group α or group β. This is a binary classification problem due to the fact that we are placing $\mathcal{O}$ into one of two distinct groups. Importantly, we must know that $\mathcal{O}$ is either a member of α or β before we begin classification. To bring our example in line with this idea, we will say that $\mathcal{O}$ is either a dime (group α) or a quarter (group β). With just a little bit of research, we determine that the average

Figure 1: Image of a hand crank coin sorter [7]. This machine uses gravity and the unique size of the coins to sort each one into its appropriate bin. This is analogous to a classifier using diameter as its sorting feature.

diameter of a dime is 17.91 mm and the average diameter of a quarter is 24.26 mm [4]. At the same time, the color of both dimes and quarters is very similar, so we choose not to take this feature into account for our classifier. This is a particularly convenient decision because it also allows us to ignore the fact that we would need to generate some way of numerically evaluating color, in addition to normalizing both of our features[1]. Utilizing all of this information, we create a procedure that will classify $\mathcal{O}$:

$$z_1 * \textbf{diameter}(\mathcal{O}) + z_2 * \textbf{color}(\mathcal{O}) + z_3 \begin{cases} \geq 0 \implies \mathcal{O} \in \alpha \\ < 0 \implies \mathcal{O} \in \beta \end{cases} \tag{1}$$

where

$$z_1 = 1, \qquad z_2 = 0, \qquad z_3 = \frac{\text{-}(17.91 + 24.26)}{2} = \text{-}20.085.$$

Another way to describe Eq. (1) is to take some *linear combination* of our features - characterized by the vector of coefficients z - and the sign of our result will determine whether $\mathcal{O}$ belongs to α or β. For this simple example it was possible to choose the values for z relatively easily, but then how do we find z in general? Although there are multiple answers, in this paper we will focus on one method, often called *supervised* machine learning. For this kind of solution, we will need some (large) set of N objects whose classification $\mathcal{Y}$ is already known, denoted $\mathcal{O}'_{(N)}$. Because we must define $\mathcal{Y}$ rigidly in order to use it in expressions, we specify that $\mathcal{Y} \in \{1, \text{-}1\}$, where $\mathcal{Y}_{(i)} = 1 \implies \mathcal{O}'_{(i)} \in \alpha$ and $\mathcal{Y}_{(i)} = \text{-}1 \implies \mathcal{O}'_{(i)} \in \beta$. Finally, we denote a more general feature of any object $\mathcal{O}$ to be $f_j(\mathcal{O})$, where $j \in \{1, 2, \ldots, n\}$ and n is the number of features we are examining. Synthesizing all of this with Eq. (1) results in Eq. (2).

$$\underbrace{\begin{bmatrix} f_1(\mathcal{O}'_{(1)}) & f_2(\mathcal{O}'_{(1)}) & \cdots & f_n(\mathcal{O}'_{(1)}) & 1 \\ f_1(\mathcal{O}'_{(2)}) & f_2(\mathcal{O}'_{(2)}) & \cdots & f_n(\mathcal{O}'_{(2)}) & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ f_1(\mathcal{O}'_{(N)}) & f_2(\mathcal{O}'_{(N)}) & \cdots & f_n(\mathcal{O}'_{(N)}) & 1 \end{bmatrix}}_{\text{We define } \mathcal{F}_{(i)} \text{ to be the } i^{\text{th}} \text{ row of this matrix}} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{Y}_1 \\ \mathcal{Y}_2 \\ \vdots \\ \mathcal{Y}_N \end{bmatrix} \tag{2}$$

At this point, our problem is to choose the values of z such that we minimize the error of Eq. (2). The solution to this minimization problem is our binary classifier, z^*, which is comprised of the optimal values of each z_j according to our minimization function. Once we have this classifier, we can use Eq. (3) to classify any object $\mathcal{O}$. The success rate of z^* will be dependent on countless aspects of the problem, including the how we define optimal, the features $\mathcal{F}$, and the training set of objects $\mathcal{O}'$.

$$\mathcal{C}(\mathcal{O}) = \mathcal{F}(\mathcal{O}) * z^* \begin{cases} \geq 0 \implies \mathcal{O} \in \alpha \\ < 0 \implies \mathcal{O} \in \beta \end{cases} \tag{3}$$

2.2 Least Squares Approach

Now that we have our problem in a minimization form, we can explore various methods of developing a solution. We begin with a *least squares* approach. This means that we have specified our objective function to be

$$\textbf{minimize} \left\| \mathcal{Y} - \mathcal{F}(\mathcal{O}') * z \right\|_2. \tag{4}$$

We then use the well known solution to least squares minimization problems to arrive at a formula for our solution,

$$z^*_{\text{LS}} = \mathcal{F}^\dagger(\mathcal{O}') * \mathcal{Y}, \tag{5}$$

[1] Normalization is conducted to ensure that the different features are on a level playing field. For example, perhaps both width and diameter were features that we cared to use. Diameter should not matter more in our decision simply because it is larger in general.

where z^*_{LS} is our optimal set of coefficients for the least squares approach, and $\mathcal{F}^\dagger(\mathcal{O}')$ is the pseudo-inverse of $\mathcal{F}(\mathcal{O}')$.

2.3 Linear Programming Approach

Another technique that we can use to derive our classifier is called *linear programming.* We recall that a linear program consists of an objective function and any number of constraint equalities and inequalities. Upon recognizing that both our minimization framework and linear programming utilize inequalities, it is tempting to jump to the conclusion

$$\begin{aligned} \textbf{minimize} \quad & ??? \\ \textbf{subject to} \quad & \mathcal{F}_{(1)} * z \geq \mathcal{Y}_{(1)} \\ & \mathcal{F}_{(2)} * z \geq \mathcal{Y}_{(2)} \\ & \vdots \\ & \mathcal{F}_{(q)} * z \geq \mathcal{Y}_{(q)} \\ & \mathcal{F}_{(q+1)} * z \leq \mathcal{Y}_{(q+1)} \\ & \vdots \\ & \mathcal{F}_{(N)} * z \leq \mathcal{Y}_{(N)}, \end{aligned} \tag{6}$$

where $\mathcal{Y}_{(1:q)} = 1$ and $\mathcal{Y}_{(q+1:N)} = \text{-}1$. There are two issues with this knee-jerk formulation. One is that we simply don't have an objective function, and the other is that we have mandated that each of our constraint function be true. For simple problems, the rigid constraints may not pose an immediate issue, but as problems get more and more complex, there is an increasing likelihood of outliers in our training set. To solve both of these issues, we introduce two sets of *slack variables*, $u_{(q)}$ and $v_{(r)}$, where $q+r = N$. With their introduction, both of our issues with Eq. (6) can be remedied. We present our actual linear programming formulation in Eq. (7). Note that we have substituted the values of $\mathcal{Y}$ into the inequalities.

$$\begin{aligned} \textbf{minimize} \quad & \sum_{k=1}^{q} u_k + \sum_{k=1}^{r} v_k \\ \textbf{subject to} \quad & \mathcal{F}_{(1)} * z \geq 1 - u_1 \\ & \mathcal{F}_{(2)} * z \geq 1 - u_2 \\ & \vdots \\ & \mathcal{F}_{(q)} * z \geq 1 - u_q \\ & \mathcal{F}_{(q+1)} * z \leq \text{-}(1 - v_1) \\ & \vdots \\ & \mathcal{F}_{(q+r)} * z \leq \text{-}(1 - v_r) \end{aligned} \tag{7}$$

2.3.1 The Margin

Although our current linear program formulation Eq. (7) will result in a reasonable binary classifier, there is actually one more aspect that we would like to consider: the *margin of error*, often shortened to "the margin." A visualization of the margin of error for two different classifiers can be seen in Fig. (2). The margin can be thought of as the shortest distance from a hyperplane to any of the training data points. Consequently, choosing a separating hyperplane that results in the largest possible margin would be beneficial, as this would increase the likelihood that any input would fall on the appropriate side of the classifier. If this is the case, then incorporating the maximization of the margin into our cost function would be an intelligent thing to do.

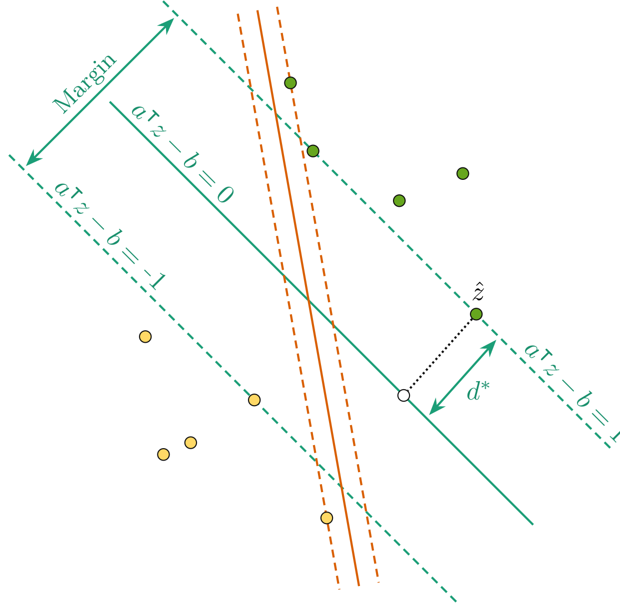

Figure 2: Visualization of two different choices for a dividing line between the two colors of points accompanied by their margins. Both are valid divisions, as both separate the two types of point, but one does so with much more breathing room.

In order to integrate the maximization of the margin into our minimization function, we first define $\hat{z}$ to be the point with the minimum distance to our separating hyperplane $\mathcal{P} = \{z \mid a^\intercal z - b = 0\}$. Note that as a result of this definition, $\hat{z}$ lies on either the hyperplane $a^\intercal z - b = 1$ or $a^\intercal z - b =$ -1. For simplicity, we specify that $\hat{z}$ lies on the hyperplane $a^\intercal z - b = 1$, although the derivation is very similar for the alternate case. The shortest distance between $\hat{z}$ and $\mathcal{P}$ is then the scalar d^* that satisfies the equation

$$d^* \left(\frac{a}{\|a\|_2} \right) = \hat{z} - \mathbf{proj}_{\mathcal{P}}(\hat{z}). \tag{8}$$

By definition,

$$a^\intercal \mathbf{proj}_{\mathcal{P}}(\hat{z}) - b = 0, \tag{9}$$

as $\mathbf{proj}_{\mathcal{P}}(\hat{z})$ lies on the hyperplane $\mathcal{P}$. Noticing that our projection point exists in both Eq. (8) and Eq. (9), we choose to eliminate this variable by conducting a substitution.

$$\implies \quad \mathbf{proj}_{\mathcal{P}}(\hat{z}) = \hat{z} - d^* \left(\frac{a}{\|a\|_2} \right)$$

$$\implies \quad a^\intercal \left(\hat{z} - d^* \left(\frac{a}{\|a\|_2} \right) \right) - b = 0$$

$$\implies \quad d^* \left(\frac{a^\intercal a}{\|a\|_2} \right) = a^\intercal \hat{z} - b$$

$$\implies \quad d^* = \frac{(a^\intercal \hat{z} - b)\|a\|_2}{a^\intercal a}$$

$$\textbf{recall} \quad a^\intercal a = \|a\|_2^2 \qquad \text{and} \qquad a^\intercal \hat{z} - b = 1$$

$$\implies \quad d^* = \frac{1}{\|a\|_2} \tag{10}$$

Now that we have obtained an expression for half the width of the margin, it is clear that if we want to maximize the margin, we will minimize the two-norm of a. We modify our cost function another time, making sure to include a weighting factor γ, as we do not know what linear combination of our two component functions will achieve the best results.

$$\begin{array}{ll} \textbf{minimize} & \|a\|_2 + \gamma \left(\sum_{k=1}^{q} u_k + \sum_{k=1}^{r} v_k \right) \\ \textbf{subject to} & \mathcal{F}_{(1)} * z \geq 1 - u_1 \\ & \mathcal{F}_{(2)} * z \geq 1 - u_2 \\ & \vdots \\ & \mathcal{F}_{(q)} * z \geq 1 - u_q \\ & \mathcal{F}_{(q+1)} * z \leq \text{-}(1 - v_1) \\ & \vdots \\ & \mathcal{F}_{(q+r)} * z \leq \text{-}(1 - v_r) \end{array} \tag{11}$$

The primary reason to utilize the margin is supposedly that the additional term in the cost function makes our formulation more robust in the presence of errors. Instead of taking this at face value, we decide to show that this is true using the following process. Note that Fig. (3) shows an example plot that could be created during an "outlier" iteration.

1. Generate a set $\mathcal{S}$ of N random vectors $X_{i \in 1:N} = \{x \in \mathbb{R}^2 \mid 0 \leq x \leq 1\}$. Each vector in $\mathcal{S}$ can be thought of as coordinates in a 2D plane.

2. Create two subsets of $\mathcal{S}$:
   $\mathcal{S}_{\text{lo}} = \{\mathcal{S} \mid x_1 + x_2 \leq 0.95\}$
   $\mathcal{S}_{\text{up}} = \{\mathcal{S} \mid x_1 + x_2 \geq 1.10\}$

   Note that this creates a zone which must be void of random points. The line that describes the bisection of this region is our *true separating hyperplane* $\mathcal{H}$.

3. Use the linear programs defined in Eq. (7) and Eq. (11) to create one classifier each, denoted $\mathcal{P}_i^{(1)}$ and $\mathcal{P}_i^{(2)}$ respectively.

4. We characterize the error of classifier $\mathcal{P}_i$ as the area between $\mathcal{H}$ and $\mathcal{P}_i$ on the domain $x_1 \in [0, 1]$.

5. Repeat steps 1 - 4 a total of n times, where n is a sufficiently large number.

6. Take the sum of the error for both $\mathcal{P}^{(1)}$ and $\mathcal{P}^{(2)}$.

7. Repeat steps 1 - 6 again. During step 2, add $\mathcal{S}_{\text{outliers}}$ to the set $\mathcal{S}_{\text{lo}}$, where $\mathcal{S}_{\text{outliers}}$ is a set of two vectors that fall in regions $\mathcal{R}_{\text{outliers}}$. Importantly, a majority of each region should be on the opposite side of $\mathcal{H}$ from other vectors in $\mathcal{S}_{\text{lo}}$. See Fig. (3) for a visualization of $\mathcal{R}_{\text{outliers}}$.

The above process was conducted for many different values of γ, using $N = 50$ and $n = 250$. Our results are visualized in Fig. (4). Examining this figure provides insight into multiple factors regarding our linear programs and outliers in training data. We note that regardless of method, the presence of outliers has increases the area of error in general, which makes sense. Most importantly, we see that method 2, characterized by the linear program described in Eq. (11), does indeed have a smaller error value than method 1 for most values of γ when outliers are introduced to the training data. With this demonstration, we can confidently use Eq. (11) as the linear program to define our classifiers, because it is more robust than Eq. (7) in the presence of outliers.

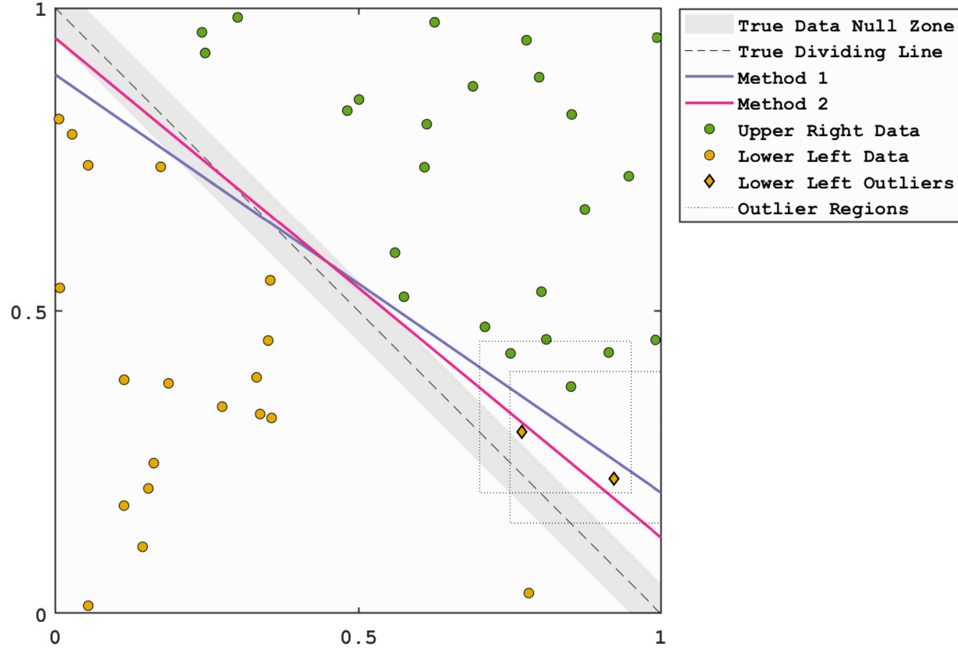

Figure 3: An example iteration for the toy problem used to verify our assumptions about the margin. Method 1 uses the formulation defined in Eq. (7) and method 2 uses the formulation from *Eq.* (11). Note that the outliers, which can be seen in the bottom right of the plot, would not be added to a non-outlier iteration.

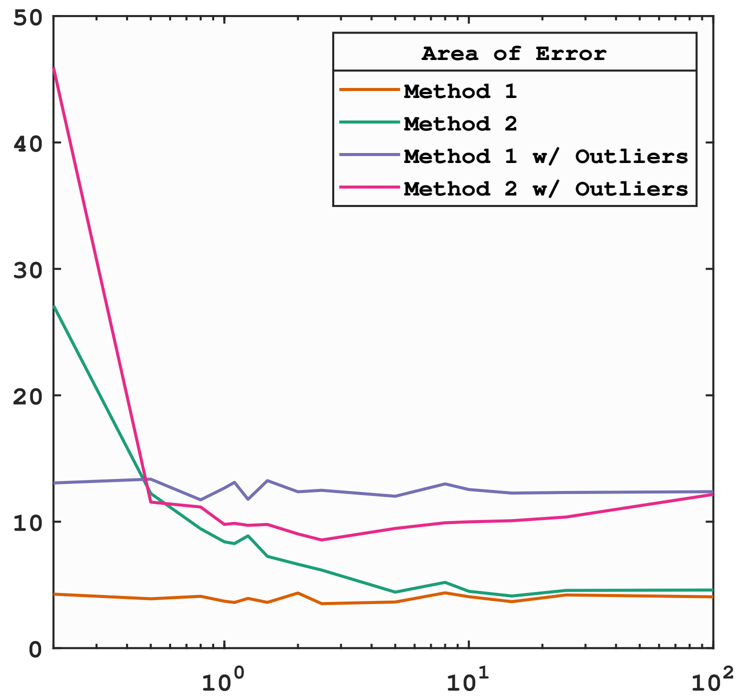

Figure 4: Plot of the area of error for our two methods, with and without outliers, as a function of γ. We see that for the no outlier condition, the error seen in method 2 decreases until it approaches the same value as method 1, while the error for method 2 dips below that of method 1 in the presence of outliers.

2.4 Extension to multi-class Classification

We have now formulated two different methods that we can use to obtain a binary classifier, but our task is to sort the digits, meaning that there are ten categories, not two. One potential way to achieve differentiation between these ten different classifications would be to create a classifier for each digit k such that

$$\mathcal{C}_k(\mathcal{O}) = \mathcal{F}(\mathcal{O}) * z_k^* \begin{cases} \geq 0 \implies \mathcal{O} \in \mathcal{D}_k \\ < 0 \implies \mathcal{O} \notin \mathcal{D}_k, \end{cases} \tag{12}$$

where $\mathcal{D}_k$ is the set of images that our classifier has designated as digit k. This wouldn't work, as there would be a problem if two different classifiers, one for $k = \phi$ and the other for $k = \psi$, both determined that some input $\mathcal{O}$ should be classified as digit ϕ and digit ψ, respectively. We recall, however, that the $\mathcal{F}(\mathcal{O}) * z_k^*$ returns a real valued scalar, not a binary one, and that a larger positive value implies more certainty in a classification. Our multi-classifier $\mathcal{M}$ can then be defined

$$\mathcal{M}_{\text{LS}}(\mathcal{O}) = \mathbf{max}\Big\{\mathcal{C}_0(\mathcal{O}), \mathcal{C}_1(\mathcal{O}), \ldots, \mathcal{C}_9(\mathcal{O})\Big\}, \tag{13}$$

where **max** returns the digit k of the classifier that produced the largest value for $\mathcal{C}_k(\mathcal{O})$. The multi-class classification method described in Eq. (13) will be used in conjunction with our least squares classifiers.

For the linear programming approach, we will instead choose to create k vs. ℓ classifiers. When these classifiers are being trained, we use only the training data labeled as either k or ℓ so that our classifiers will be defined

$$\mathcal{C}_{k\text{v}\ell}(\mathcal{O}) = \mathcal{F}(\mathcal{O}) * z_{k\text{v}\ell}^* \begin{cases} \geq 0 \implies \mathcal{O} \in \mathcal{D}_k \\ < 0 \implies \mathcal{O} \in \mathcal{D}_\ell. \end{cases} \tag{14}$$

This initial definition works well if we know that $\mathcal{O} \in (\mathcal{D}_k \vee \mathcal{D}_\ell)$, but it does not make as much sense when the true classification of $\mathcal{O}$ is unknown. We reexamine $\mathcal{C}_{k\text{v}\ell}$, asking ourselves "what does the output of our classifier actually mean?" Fundamentally, all that we can really say is that *if* the true label of our image $\mathcal{L}(\mathcal{O}) = (k \vee \ell)$ then our classifier has probably designated $\mathcal{O}$ correctly. If $\mathcal{L}(\mathcal{O}) \neq (k \vee \ell)$, our classifier still must place $\mathcal{O}$ in either $\mathcal{D}_k$ or $\mathcal{D}_\ell$. This is crucial, as it allows us to effectively rule out the digit that $\mathcal{C}_{k\text{v}\ell}$ does not pick. Our k vs. ℓ classifiers can then be thought of in the exclusionary manner seen in Eq. (15).

$$\mathcal{C}_{k\text{v}\ell}(\mathcal{O}) = \mathcal{F}(\mathcal{O}) * z_{k\text{v}\ell}^* \begin{cases} \geq 0 \implies \mathcal{O} \notin \mathcal{D}_\ell \\ < 0 \implies \mathcal{O} \notin \mathcal{D}_k \end{cases} \tag{15}$$

With this perspective, we see that we can create a sorting method utilizing a directed acyclic graph (DAG), which is exemplified by the visualization in Fig. (5). This method was used in conjunction with our linear programming approach to form the multi-classifier $\mathcal{M}_{\text{LP}}$. Visualizations of the specific sorting DAG that was used can be seen in Appendix A.

2.5 Weighting Optimization

We now notice that there is still one element of the linear program classifier that we have yet to define. The relative weighting of the maximization of the margin and the minimization of the norms, characterized by the variable γ. From our brief exercise in §2.3.1 we know that changing this weighting will result in classifiers of differing qualities, implying that γ could be optimized. Furthermore, we know that for each of our 45 classifiers, this optimal $\gamma_{k\text{v}\ell}^*$ will be unique.

In an attempt to improve our classifiers, we set up a simple iterative optimization approach. For each classifier, we solve our linear program when $\gamma = .008$, a value that was chosen because it seemed to be generally good in a few initial tests. Once we have our $z_{k\text{v}\ell}^*$ for $\gamma = .008$, we evaluate the classifier using

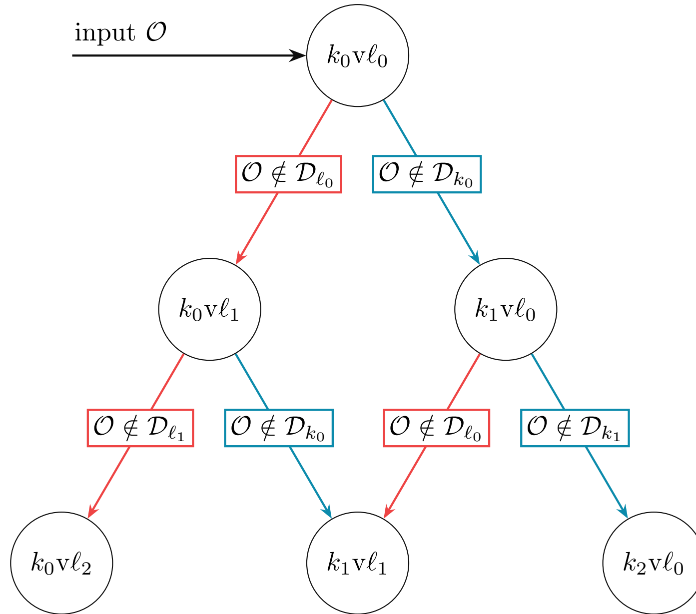

Figure 5: Visualization of a small portion of the DAG used to sort images when using the linear programming formulation. Instead of making a true binary decision tree that is rather deep, we limit the number of nodes and levels that our DAG contains by allowing multiple "parent" nodes to have the same "child" node.

only the testing data labeled either k or ℓ, and save the result. Now Eq. (16) and Eq. (17) are used to determine our next value of γ. In Eq. (16), `rand` is a random real number in $[0, 1]$.

$$\texttt{jump} = \left(\frac{\texttt{rand} + 2}{6} \right) \gamma \tag{16}$$

$$\gamma_{\text{new}} = \gamma + \texttt{jump} \tag{17}$$

The linear program is then solved again, this time using the new value of γ. Similarly to the first iteration, we evaluate our resulting classifier using the appropriate testing data and save our results. At this point, there are three different scenarios, and each is handled differently by our optimization method. If our classifier improved with this change in γ, we save our new γ as the current best, and continue searching for a better γ in the positive direction. If our classifier has the same success rate, we mark that this occurred, but still continue searching for a new gamma in the positive direction. Our search in the positive direction ends when either we have remained at the same success rate for three iterations, or our classifiers success rate decreases. This process is then repeated in the negative direction, with a separately saved best value. Once this side of our search has terminated, the γ that resulted in the best performance was chosen as our optimal γ for that classifier. Our optimized γ values can be seen in Table I.

3 Pre-Processing

Although we have now finalized our problem formulations, there remains one facet of our classification pipeline that we have yet to address: the features. Prior to proposing potential features, we examine the format of the classifier inputs that we will use, as the composition of these objects will dictate the kinds of features that are accessible for this problem. In brief, we will be using a collection of images of handwritten digits called the Modified National Institute of Standards and Technology Database (henceforth referred to as `mnist`) [2].

kvℓ	γe3	kvℓ	γe3	kvℓ	γe3	kvℓ	γe3	kvℓ	γe3
0v1	5.0	1v2	8.0	2v4	4.2	3v7	8.0	5v7	4.7
0v2	8.0	1v3	8.0	2v5	11.4	3v8	16.2	5v8	4.9
0v3	4.1	1v4	30.7	2v6	11.5	3v9	8.0	5v9	11.3
0v4	15.0	1v5	8.0	2v7	15.1	4v5	11.1	6v7	16.1
0v5	8.0	1v6	23.8	2v8	8.0	4v6	14.6	6v8	24.5
0v6	2.7	1v7	15.2	2v9	8.0	4v7	8.0	6v9	4.9
0v7	8.0	1v8	8.0	3v4	4.1	4v8	15.8	7v8	11.0
0v8	14.5	1v9	8.0	3v5	11.3	4v9	11.6	7v9	20.5
0v9	8.0	2v3	10.9	3v6	15.1	5v6	4.3	8v9	4.8

TABLE I: Tabulation of the optimized γ values. Note that these values are all multiplied by 10^{-3}, meaning that even the largest optimized γ is much smaller than the fixed-γ value of 1.

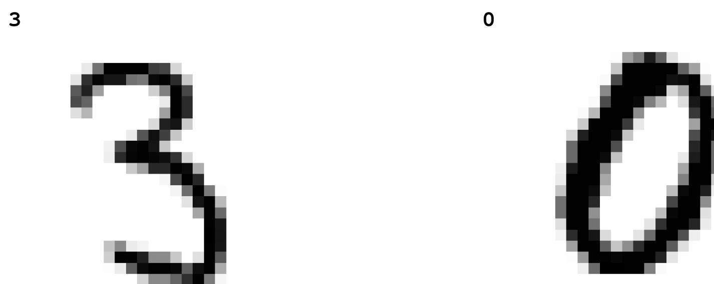

Figure 6: Plots of two images provided in the training data set, created by calling the `vis_dig` function. The label for each picture, as defined by the `mnist` data set, is northwest of both handwritten numbers.

This surface level information is not sufficient to inform feature definition, so we describe our inputs, the images in `mnist`, in increasing detail in hopes of discovering potential features that we can use. Each datum in `mnist` is a row representation of a 28×28 matrix[2]. Each element in these row vectors is prescribed by the `uint8` grayscale value for its corresponding pixel in the square matrix representation. Thus, when reconstructed correctly, each row portrays an image of a handwritten digit, as can be seen in Fig. (6). A simple function that visualizes these pictures along with their intended label, `vis_dig`, can be found in Appendix D.

With this definition we notice that the building blocks of these images are the pixels, and that each building block has an integer value on the range $[0, 255]$. Jointly, we notice that other than image size (which in this case is fixed) any characteristic could be expressed as some linear combination of the pixels. It follows that the smallest set of features that would assuredly encompass all of the possible images would be the `uint8` value of each of the 784 pixels.

Simultaneously, we know that the total number of features that we choose to utilize will have a

[2]Relationships between the linear and two dimensional representations of a matrix are explained in Appendix B.

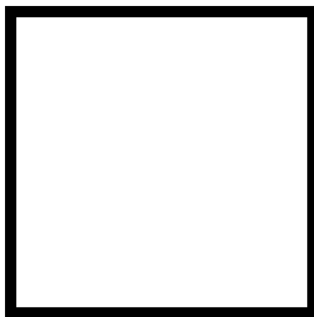

Figure 7: Visualization of the pixels that won't be considered in our solution to this problem. Pixels that will be removed are shaded. The removal of the shaded pixels reduces the number of pixels per image from 784 to 676.

significant effect on computation time, meaning that reducing the number of pixels that we use is beneficial. Fortunately, each image has a one pixel width border of padding in which all values are set to zero. These added pixels are not helpful to us, so we wish to remove them in order to reduce the number of extraneous features in our problem set. A visualization of the pixels that we will be removing can be seen in Fig. (7). To efficiently remove these pixels, we rely on the relation between the one dimensional and two dimensional representations, which is explored in Appendix B.

Before we declare that our data is in a form suitable to use for our classifiers, we will convert the `uint8` values into `double`s, and then divide everything by 255, the maximum value an element can have. This division normalizes our inputs, and converting to the `double` data type prevents loss of information due to rounding. Although not technically necessary here, as all features have the same domain, normalizing is good practice and in this case will result in smaller z^* coefficient values. As our final modification to the data, we append a 1 to the end of each of the image rows. The coefficient associated with this 1 is analogous to the y-direction shift b in the canonical equation for a line in two dimensions $y = mx + b$.

4 Results

With our formulations precisely defined, Matlab implementation of our two classifiers $\mathcal{M}_{\text{LS}}$ and $\mathcal{M}_{\text{LP}}$ is, in general, rather straight forward. A significant contributor to the ease of our implementation is the Matlab-based convex modeling framework: CVX [5], which was used to solve the 45 linear programs that comprise $\mathcal{M}_{\text{LP}}$. CVX has built-in access to four solvers: SDPT3, SeDuMi, Gurobi, and MOSEK. The former two of this set are open source, while the latter two are for commercial use. Although academic licensing for the commercial solvers is easily obtainable, and the commercial solvers are in some cases orders of magnitude faster than their counterparts, we choose instead to use SeDuMi, one of the open source solvers, in support of the open source movement.

Returning to the derivation of our classifiers, we recall that fundamentally each binary classifier is a set of scalars mapped to different pixels on a 28×28 grid. In order to build a more robust understanding of the differences between our binary classifiers, we can visualize them with a *pixel heat map*. The coloring of each pixel on these heat maps is determined by the scalar value associated with that pixel in the binary classifier that is being visualized. To reinforce our interpretation of these heat maps, we will outline what various colors mean for a hypothetical $k\text{v}\ell$ classifier. A dark red pixel corresponds to a large positive value in the binary classifier $\mathcal{C}_{k\text{v}\ell}$. This in turn means that if an input image $\mathcal{O}$ has this pixel filled in, then our classifier is much more likely to classify $\mathcal{O}$ as digit k. In contrast, a dark blue pixel corresponds to a large negative value in $\mathcal{C}_{k\text{v}\ell}$, implying that if $\mathcal{O}$ has this pixel filled in it is more likely to be digit ℓ. A yellow pixel represents a classifier value near zero, which means that a this pixel

is not particularly influential in determining the difference between k and ℓ. Note that when analyzing the least squares classifier visualizations, replace ℓ with *all digits that are not k*.

Visualizations of the binary classifiers that make up $\mathcal{M}_{\text{LS}}$ can be seen in Fig. (8). Before moving on to the binary classifiers in $\mathcal{M}_{\text{LP}}$, we examine these plots on a surface level for general trends. Interestingly, we can see that these heat maps seem to give an impression of their corresponding digit. Personally, I think that the $k = 0$ and the $k = 3$ heat maps create a shape most similar to their digit, while the $k = 9$ heat map is the least related. Another interesting occurrence is that, in general, the more central pixels have a significantly smaller magnitude than those nearest the border padding.

In order to discern the difference that the value of γ plays in our linear program formulation, we decide to evaluate two versions of our formulation. The first, which we will designate as $\mathcal{M}_{\text{LP1}}$, will use a constant $\gamma = 1$ for all classifiers, while the second classifier $\mathcal{M}_{\text{LPopt}}$ implements the γ values shown in Table I. As there are a total of 45 binary classifiers for both $\mathcal{M}_{\text{LPopt}}$ and $\mathcal{M}_{\text{LP1}}$, a small selection of the component binary classifiers are visualized in Fig. (9). When examining the differences between the heat maps, it is clear that the coefficients seen in the $\mathcal{M}_{\text{LP1}}$ visualizations have magnitudes that are significantly larger than those found in $\mathcal{M}_{\text{LPopt}}$. It is also much more difficult to see patterns in the fixed-γ classifier, and the local variance of pixel value is very large.

4.1 Feature Engineering

Classifiers have two primary performance metrics: duration of training time and accuracy of the classifier on novel data. One method of improving either, or both, of these metrics is called *feature engineering*, which is characterized by removing unnecessary features, crafting additional features as some function of already existing features, or some combination of the two. Our removal of border pixels was then a rudimentary implementation of feature engineering.

The most straightforward next step for our problem would be to repeat our border pixel removal again, reducing the size of our evaluation space to the central 24×24 grid. It turns out that this is actually a rather effective way to significantly reduce the total number of features we are using without losing a significant amount of classifier accuracy. In an attempt to mitigate the reduction in success rate that our pixel removal has accrued, we will add some of the information that we have taken away back into the inputs. For each cardinal direction, we add a row or column of pixels that is the average of the missing pixels. In this state, our set of features is comprised of the central 14×14 pixels plus an additional $4 * 14$ features which are the weighted averages of the removed border regions. Based on some of the trends that we see in the $\mathcal{M}_{\text{LPopt}}$ classifier, we also create 5 additional features. All of these features are the average of a region that often exhibited a significant majority of either positive or negative coefficients. The features for this classifier, which we will call $\mathcal{M}_{\text{LPfeat}}$, are outlined on top of the $0v1$ heat map in Fig. (10). Two additional heat map visualizations can be seen in Fig. (11).

4.2 Analysis

Although there are many different ways to measure accuracy for a multi-classifier, we will use two of the more general types of metric. The first, *overall error rate*, denoted E_{oa}, represents the likelihood that our classifier makes a mistake. In other words this is the total number of errors, divided by the sample size. The other measurements that we will use to evaluate our classifier's accuracy are the *true label k rates*, denoted $L_{(k)}$. These metrics can be thought of as the fraction of images labeled as digit k that were correctly classified as digit k.

The values of these metrics for each of our classifiers operating on the testing data can be viewed in Table II. We note that across the board, the digits 8 and 9 are particularly difficult to identify, while 0 and 1 are rather easy to pick out. This makes sense as there are many variations of the handwriting patters for 8 and 9 compared to 0 and 1. We also want to determine whether or not our classifiers are *overfit*, a descriptor given to models that have been trained on too much data. When this happens,

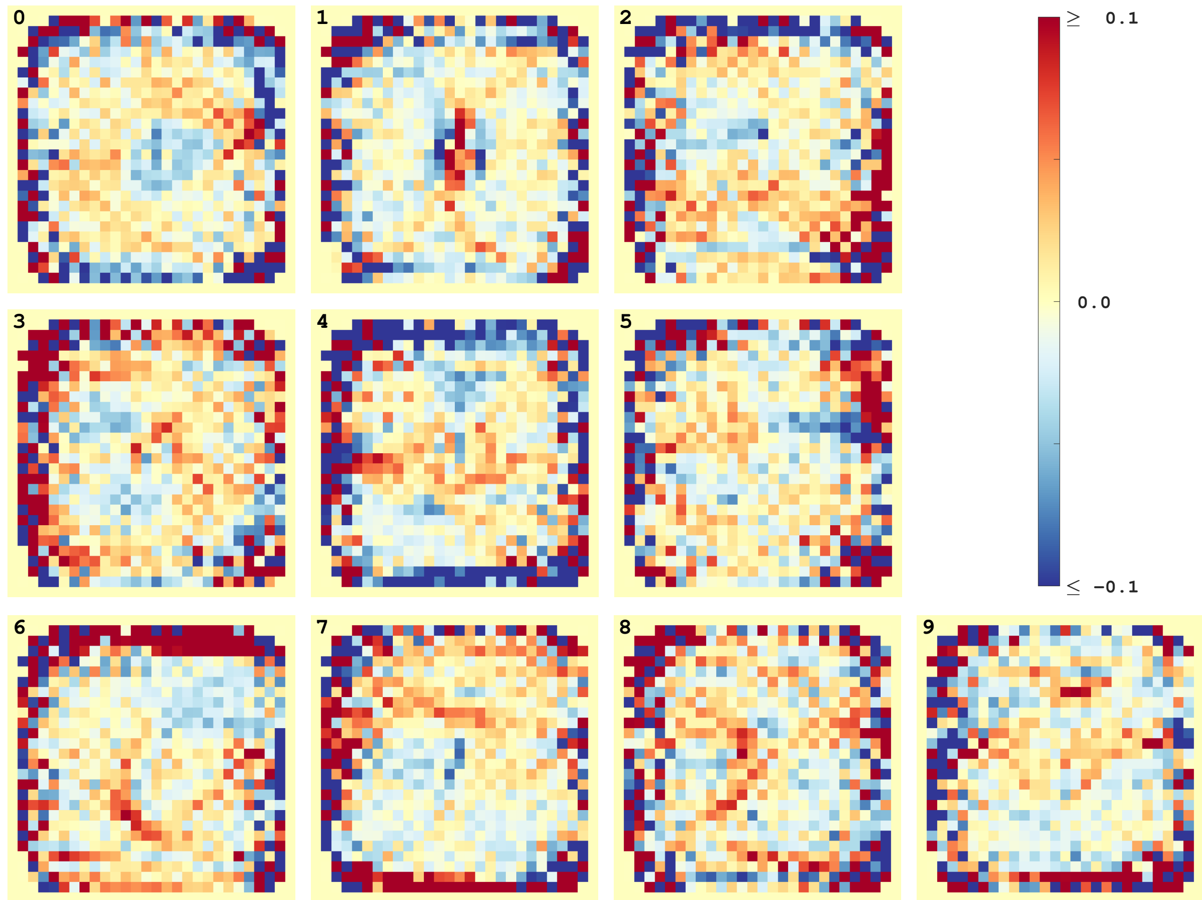

Figure 8: In this figure, the weighting of each pixel in the coefficient matrix z_k for each digit $k \in \{0, 1, \ldots, 9\}$ is mapped to a color (northeast corner). The digit k can be seen in the northwest corner of its corresponding image. Note that we *have* included the one pixel width border that was removed during pre-processing, and that the value of each of these pixels has been set to 0. The `colormap` used in this image originates from the ColorBrewer package, and its bounds were informed by a Fig. (14.3) in Boyd and Vandenberghe [3].

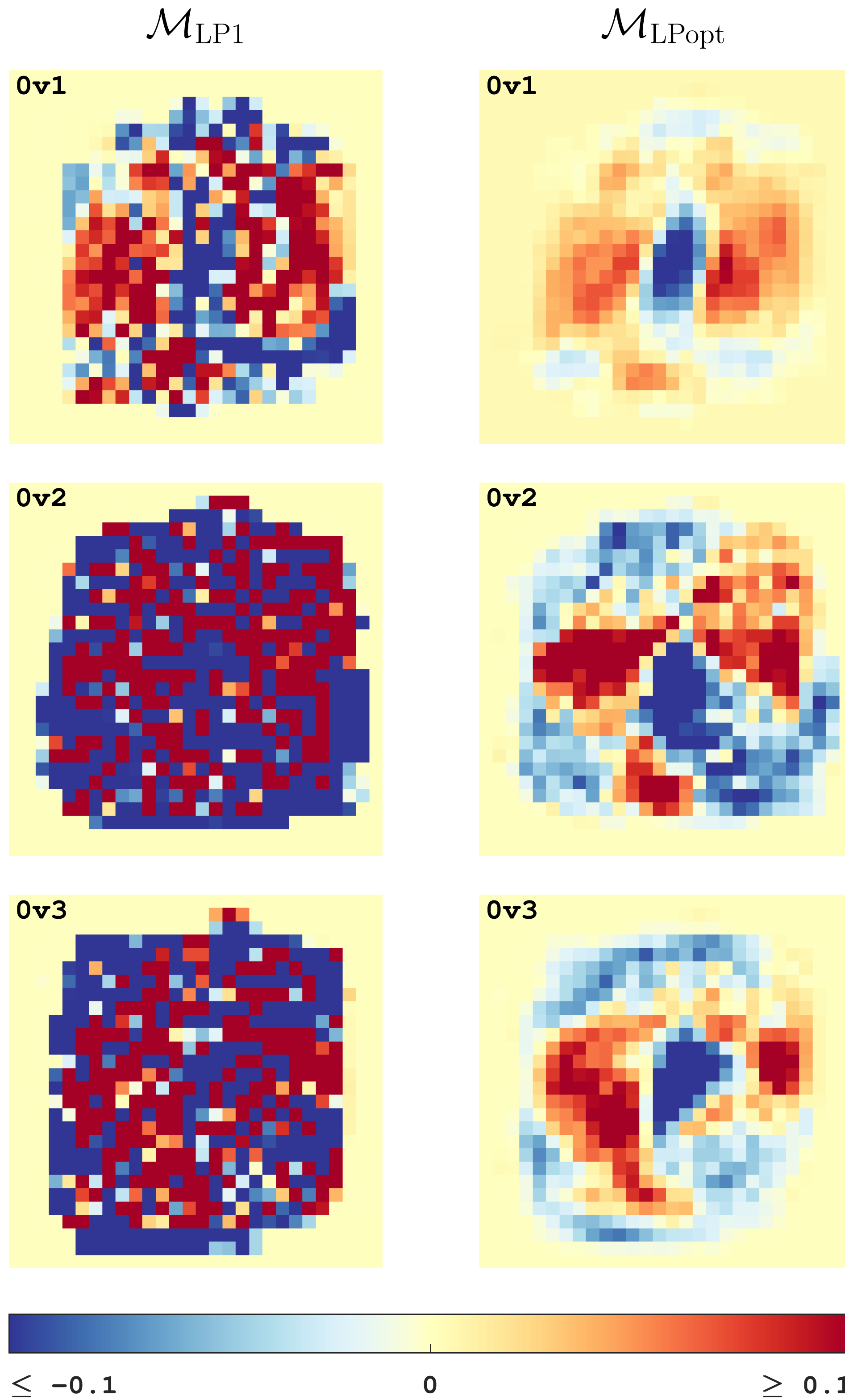

Figure 9: Coefficient visualizations for three different classifiers (0v1, 0v2, and 0v3). Similarly to all other coefficient visualizations in this report, we have included the initial 1 pixel border and used the same `colormap` and `colormap` limits. This uniformity assists in our ability to compare these classifiers in future sections.

≤ -0.1   0   ≥ 0.1

0v1

1 1 1 1 2 5

+ +

Figure 10: Visualization of the feature coefficients for the 0v1 binary classifier in $\mathcal{M}_{\text{LPfeat}}$. Outlined in black are the regions that were averaged to create our engineered features. Note that the features in the topmost image are a weighted average of the 6 pixels they are comprised of. The pixels closest to the center of the image are weighted significantly greater than those near the edge.

(a) $\mathcal{M}_{\text{LPfeat}}$ 0v2 binary classifier

(b) $\mathcal{M}_{\text{LPfeat}}$ 0v3 binary classifier

Figure 11: Visualization of the feature coefficients for the 0v2 and 0v3 binary classifier in $\mathcal{M}_{\text{LPfeat}}$. Outlined in black are the regions that were averaged to create our engineered features. We see that there are general trends along the edges that are much more pronounced than the edges coefficients had been in previous classifiers. Additionally, we note that the slightly off center region in the images furthest to the right are both dork blue, suggesting that an input image of a 0 rarely has those pixels filled in.

Metric		Value (%)			
		$\mathcal{M}_{\text{LS}}$	$\mathcal{M}_{\text{LP1}}$	$\mathcal{M}_{\text{LPopt}}$	$\mathcal{M}_{\text{LPfeat}}$
Overall Error Rate (E_{oa})		13.99	7.92	5.20	6.28
True Label Rates ($L_{(k)}$)	0	96.33	95.61	98.47	98.16
	1	97.53	96.92	98.77	98.41
	2	78.68	89.53	93.41	92.44
	3	86.93	92.67	94.65	93.56
	4	89.71	94.30	95.72	95.32
	5	73.88	88.90	91.03	89.46
	6	91.23	93.01	96.76	95.30
	7	86.09	92.41	94.65	93.87
	8	77.93	88.09	91.58	89.73
	9	79.48	88.50	92.17	89.99
ΔE_{oa}		-0.26	6.18	0.32	0.06

Table II: Comparison of the accuracy metrics of our least squares classifier $\mathcal{M}_{\text{LS}}$ and our three linear program formulated classifiers, $\mathcal{M}_{\text{LP1}}$, $\mathcal{M}_{\text{LPopt}}$, and $\mathcal{M}_{\text{LPfeat}}$ operating on the testing data. The last two entries show the difference between our classifiers acting on the testing versus the training data, which is used to investigate whether or not a classifier has been overtrained. The equation for ΔE_{oa} is defined in Eq. (18).

our classifier becomes really good at differentiating between inputs that are similar to the ones it was trained on, but will typically perform poorly when operating on inputs that are unique. If a classifier is overfit, we would expect to see overall error rates that were significantly worse than those for the training data. Most of our classifiers seem to be within an acceptable margin of error, but the error difference between testing and training data for $\mathcal{M}_{\text{LP1}}$ is an order of magnitude larger than the others. Based on this quantitative comparison and our visualizations, we conclude that $\mathcal{M}_{\text{LS}}$, $\mathcal{M}_{\text{LPopt}}$, and $\mathcal{M}_{\text{LPfeat}}$ are not overfit, while $\mathcal{M}_{\text{LP1}}$ most likely is. The definition of ΔE_{oa} can be seen in Eq. (18).

$$\Delta E_{\text{oa}} = E_{\text{oa}}(\texttt{test}) - E_{\text{oa}}(\texttt{train}) \tag{18}$$

Another method of representing the performance of a classifier is a *confusion matrix*. For each type of label, these matrices display the number of inputs that were classified as each different potential outcome. These are often an effective way to identify where a classifier is having difficulties. The confusion matrices for $\mathcal{M}_{\text{LS}}$, $\mathcal{M}_{\text{LPopt}}$, and $\mathcal{M}_{\text{LPfeat}}$ can be found in Appendix C.

Although we know that there are significantly faster algorithms for this problem, it is still relevant to examine the total amount of time that it took to train each of these classifiers. Our least squares approach was the fastest at 2.12 minutes and .21 minutes per classifier. Training for $\mathcal{M}_{\text{LP1}}$ took 110.64 minutes with an average binary classifier time of 2.46 minutes, while training $\mathcal{M}_{\text{LPopt}}$ took 101.77 minutes with an average classifier time of 2.26 minutes. The classifier created with our engineered features, $\mathcal{M}_{\text{LPfeat}}$, was much faster than the other linear programming classifiers with a total time of 53.35 minutes and a mean per binary classifier time of 1.19 minutes. Using $\mathcal{M}_{\text{LPopt}}$ and $\mathcal{M}_{\text{LPfeat}}$ for our measurement, we can say that by decreasing total number of features by 62% (676 $\rightarrow$ 257) we were able to train our classifier 1.9 times as fast (101.77 $\rightarrow$ 53.35) at the cost of increasing our overall error rate by about 1% (5.20% $\rightarrow$ 6.28%).

5 Conclusion

We recall that the goal of this report was to create a multi-classifier for images of handwritten digits using both a least squares and linear programming approaches. While formulating our problem, we discussed how classification can be thought of as a minimization problem through the aid of our real world coin sorting example, and then formulated both the least squares and linear programming versions of our classification problem. While building intuition for the linear programming approach, we verified that including a term for the maximization of the margin is beneficial to our resulting classifiers through the use of a smaller toy problem.

With our formulations fully defined the entire process was implemented in MATLAB, and classifiers for each method were found using built in MATLAB functions for the least squares classifier, and the SeDuMi solver via CVX for the linear programming classifiers. Our results were then analyzed individually and comparatively, which led to the conclusion that the fixed-γ solver $\mathcal{M}_{\mathrm{LP1}}$ seems to be overtrained, while the other three classifiers do not. Additionally, we found that our linear program classifiers did perform significantly better than our least squares solver, but at the cost of a significant increase in training time.

One potentially interesting direction to go with the project would be to compare the least squares and linear programming methods independently from the methods of multi-classification (maximum of "one vs. other" and DAG using "one vs. one"). Another idea worth exploring would be to implement an infinity norm linear programming approach, and examine the differences between it and our other classifiers.

References

[1] United States Postal Service, “A decade of facts and figures,” Postal Facts - U.S. Postal Service, 01-Apr-2021. [Online]. Available: https://facts.usps.com/table-facts/.

[2] Y. LeCun, C. Cortes, and C. Burges, “The MNIST Database,” *MNIST handwritten digit database*. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[3] S. Boyd and L. Vandenberghe, “Chapters 13 and 14,” in *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*, Cambridge: Cambridge University Press, 2018, pp. 245-278 and 285–304.

[4] “Coin Specifications,” United States Mint. [Online]. Available: https://www.usmint.gov/learn/coin-and-medal-programs/coin-specifications.

[5] Michael Grant and Stephen Boyd. *CVX: Matlab software for disciplined convex programming*, version 3.0 beta. http://cvxr.com/cvx, December 2017.

[6] C. Brewer. (2013). [Online]. Available: https://colorbrewer2.org

[7] "Nadex Anti-Jam Hand Crank Coin Sorter and Wrapper," Ubuy. [Online]. Available: https://www.ubuy.com.tr/en/product/21NU1T00-nadex-anti-jam-hand-crank-coin-sorter-and-wrapper-sort-up-to-350-coins-per-minute-into-bins-sorting.

A Directed Acyclic Graph Visualizations

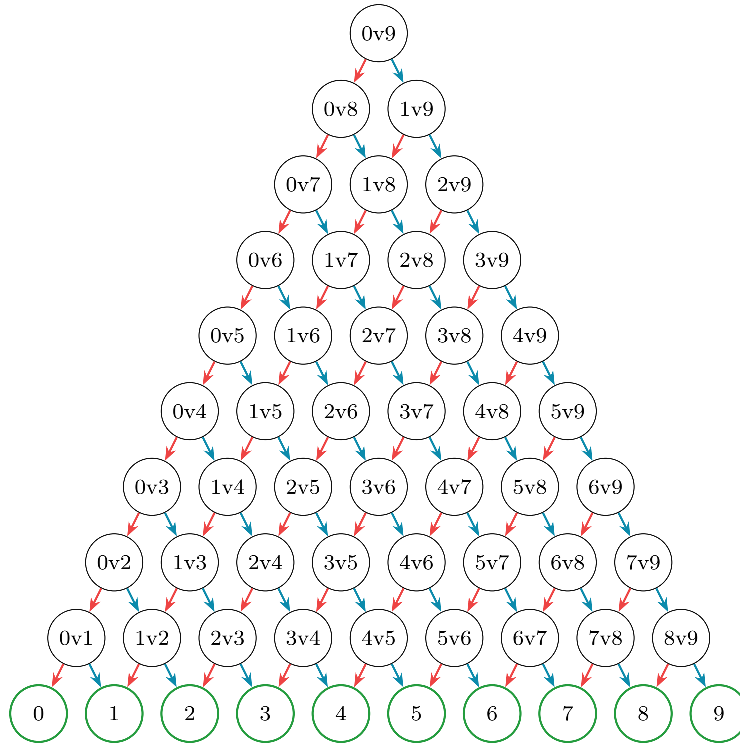

Figure 12: Visualization of the directed acyclic graph (DAG) used to determine the classification of each input for the linear programming approach. If a classifier determines a positive value for an input, flow continues along the red arrow, while obtaining a negative value results in the input being passed along the blue arrow. Once an input reaches a green circle, the classifier has designated that input as the value inside the circle.

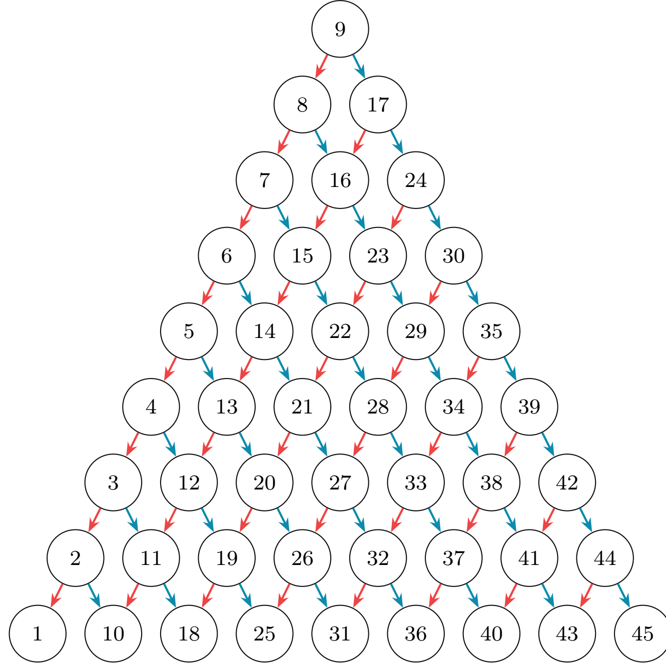

Figure 13: An additional visualization of the DAG used for the linear programming classifiers. The key difference between this figure and Fig. (12) is that each binary classifier is given an index an index in a one dimensional container. This representation is not necessary when reading the report, but is used in the code implementation.

B 1D - 2D Index Relations

When converting from a one dimensional array, P_{n^2}, to a two dimensional array, $Q_{n\times n}$, Eq. (19) can be used to determine corresponding indices[3]. Similarly, Eq. (20) can be used to convert from two dimensional indices to one dimensional indices. A visual representation of the pixel locations (v and w in an $n\times n$ matrix) their corresponding index in the provided form (u in a $1\times n^2$ array) is given in Eq. (21).

$$g : P_u \rightarrow Q_{v,w}$$

$$\text{where} \;\; g(u) = (v, w) = \left((u-1 \pmod n) + 1, \quad \left\lfloor \frac{u-1}{n} \right\rfloor + 1 \right) \tag{19}$$

$$h : Q_{v,w} \rightarrow P_u$$

$$\text{where} \;\; h(v, w) = u = n(w-1) + v \tag{20}$$

$$\begin{bmatrix} 1 & n+1 & 2n+1 & \cdots & n(n-1)+1 \\ 2 & n+2 & 2n+2 & \cdots & n(n-1)+2 \\ 3 & n+3 & 2n+3 & \cdots & n(n-1)+3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & 2n & 3n & \cdots & n^2 \end{bmatrix} \tag{21}$$

[3]Eq. (19) and Eq. (20) are only true when converting between arrays and matrices whose indexing begins at 1.

C Confusion Matrices

	Prediction $\mathcal{M}_{\mathrm{LS}}(\texttt{test})$										
Digit k	0	1	2	3	4	5	6	7	8	9	**Total**
0	944	0	18	4	0	23	18	5	14	16	1042
1	0	1107	54	18	22	18	10	40	46	11	1326
2	1	2	812	23	6	3	10	16	11	2	886
3	2	2	25	878	1	72	0	6	30	17	1033
4	2	3	14	5	881	24	22	26	27	80	1084
5	7	1	0	17	5	659	17	0	39	1	746
6	14	5	41	9	10	24	874	1	15	1	994
7	2	1	23	25	2	14	0	885	13	75	1040
8	7	14	39	21	11	38	7	0	759	4	900
9	1	0	6	10	44	17	0	49	20	802	949
All	980	1135	1032	1010	982	892	958	1028	974	1009	10000

(a) $\mathcal{M}_{\mathrm{LS}}(\texttt{test})$

	Prediction $\mathcal{M}_{\mathrm{LPopt}}(\texttt{test})$										
Digit k	0	1	2	3	4	5	6	7	8	9	**Total**
0	965	0	2	2	2	6	2	1	0	0	980
1	0	1121	2	3	0	2	2	1	4	0	1135
2	4	1	964	9	14	6	10	9	13	2	1032
3	0	0	9	956	1	16	1	10	14	3	1010
4	1	0	4	1	940	1	8	2	3	22	982
5	7	1	3	32	3	812	12	2	17	3	892
6	9	2	5	1	3	10	927	0	1	0	958
7	0	4	20	5	8	2	1	973	2	13	1028
8	4	2	8	16	5	30	8	7	892	2	974
9	4	5	0	4	26	10	2	18	10	930	1009
All	994	1136	1017	1029	1002	895	973	1023	956	975	10000

(b) $\mathcal{M}_{\mathrm{LPopt}}(\texttt{test})$

	Prediction $\mathcal{M}_{\text{LPfeat}}(\texttt{test})$										
Digit k	0	1	2	3	4	5	6	7	8	9	**Total**
0	962	0	2	2	1	9	2	1	1	0	980
1	0	1117	4	3	0	2	2	1	6	0	1135
2	4	1	954	10	14	10	14	14	10	1	1032
3	2	0	11	945	0	26	1	10	9	6	1010
4	1	0	3	4	936	1	7	3	2	25	982
5	8	1	4	42	4	798	13	6	13	3	892
6	12	2	7	2	7	14	913	0	1	0	958
7	1	7	21	3	7	3	0	965	8	13	1028
8	3	6	4	18	7	32	9	8	874	13	974
9	6	6	1	7	35	13	1	21	11	908	1009
All	999	1140	1011	1036	1011	908	962	1029	935	969	10000

(c) $\mathcal{M}_{\text{LPfeat}}(\texttt{test})$

TABLE III: Confusion matrices for the classification of testing images using the least squares classifier, $\mathcal{M}_{\text{LS}}$ (a) and the linear programming classifiers, $\mathcal{M}_{\text{LPopt}}$ (b) and $\mathcal{M}_{\text{LPfeat}}$ (c). All diagonal entries, which represent the number of images correctly classified for each digit, are blue, while the off diagonal entries are orange.

D Matlab Code

D.1 Linear Program Training Script

```
%% Create Fixed Gammas Classifier
[A1,b1,t1] = ...
    LP_classifier('fixed_gammas','fixed_gammas.mat','prep_basic');

%% Create Optimized Gammas Classifier
[Aopt,bout,topt] = ...
    LP_classifier('opt_gammas','opt_gammas.mat','prep_basic');

%% Create Classifier with Feature Engineering
[Afeat,bfeat,tfeat] = ...
    LP_classifier('feat','opt_gammas.mat','prep_MF',6,[1 1 1 1 2 5]);
```

D.2 Linear Program Testing Script

```
%% Call evaluation function for fixed gamma classifier
[Mconf_fixed,oer_fixed,tlk_fixed] = ...
    eval_LPMC('te','HP_fixed_gammas.mat','prep_basic');

%% Call evaluation function for optimized gamma classifier
[Mconf_opt,oer_opt,tlkopt] = ...
    eval_LPMC('te','HP_opt_gammas.mat','prep_basic');

%% Call evaluation function for feature engineering classifier
[Mconf_feat,oer_feat,tlk_feat] = ...
    eval_LPMC('te','HP_feat.mat','prep_MF',6,[1 1 1 1 2 5]);
```

D.3 Feature Engineering Script Script

```
%% Load images
load('mnist.mat','images_test');

%% Call prep function appropriately
images_test_play = prep_MF(images_test,6,[1 1 1 1 2 5]);

%% Save
save('images_test_play.mat','images_test_play')
```

D.4 Linear Program Classifier Function

```
function [A,b,time] = LP_classifier(name,gamma_mat,prep_func,varargin)
%LP_CLASSIFIER

%% Create strings for prep function, loading data, and saving data
hyper_mat = sprintf('HP_%s.mat',name);
xdata_mat = sprintf('data_%s',name);

%% Load in the data
% Training image data
load('mnist.mat','images','labels');

% Pre-process training images
S = feval(prep_func,images,varargin{:});

% Remember size of S matrix
%   N_img = number of rows = number of images input
%   N_feat = number of columns = number of features
[N_img, N_feat] = size(S);

% Create and save logical array containing relevant label data
%   True implies image at the corresponding index is labeled as the current
    digit.
digitKey = labels == 0:9;

% Create list of all digit pairings
N_pairs = nchoosek(10,2);
pairs = nchoosek(0:9,2);

% Sort rows by corresponding label
im_sort = arrayfun(@(d) S(digitKey(:,d),:),1:10,'UniformOutput',false);

%  For each digit, save the number of images whose labels are that digit
N_digimg = arrayfun(@(d) size(im_sort{d},1),1:10);

% Initialize containers for multiple variables
x_star = zeros(N_feat,N_pairs);
f_tilde = zeros(N_img,N_pairs);
time = zeros(1,N_pairs);

% Load in gammas to use for these classifiers
load(gamma_mat,'gammas');
```

⋮

⋮

```
% Solve each least squares problem with CVX
for p = 1:N_pairs
    N1 = N_digimg(pairs(p,1)+1);
    N2 = N_digimg(pairs(p,2)+1);
    cvx_begin
        variable x(N_feat)
        variable u(N1) nonnegative
        variable v(N2) nonnegative
        minimize ( norm( x ) + gammas(p)*( sum(u) + sum(v) ) )
        subject to
            im_sort{1,pairs(p,1)+1}*x + eye(N1)*u >= 1
            im_sort{1,pairs(p,2)+1}*x - eye(N2)*v <= -1
    cvx_end

    x_star(:,p) = x;
    f_tilde(:,p) = S*x;
    time(p) = cvx_cputime;
    fprintf(['\n\t\t\t**********************************************\n' ...
             '\t\t\t    %i vs %i Classifier calculated\n\n' ...
             '\t\t\t    Duration of previous LP: %.2f minutes\n' ...
             '\t\t\t    Total CPU time:          %.2f minutes\n\n' ...
             '\t\t\t    %i classifiers remaining\n' ...
             '\t\t\t**********************************************\n\n'], ...
             pairs(p,1),pairs(p,2),time(p)/60,sum(time)/60,N_pairs-p);
  end

%% Save properly formatted A and b that define each hyperplane for Shalom
A = x_star(1:end-1,:);
b = x_star(end,:);
save(hyper_mat,'A','b');
save(xdata_mat,'time','N_digimg','varargin');
```

D.5 Basic Image Prep Function

```
function A = prep_basic(imgs)
%PRE_PROC_BASIC pre-processing images
%   Function to get input images into the desired form for the least
%   squares machine learning problem. This involves changing the data type
%   to double, dividing by 255, and removing the "border" pixels of each
%   image.

% Check and save size of input
%   Input should have each image as a single row in a matrix. That matrix
%   should be X by Y, where X is the number of input images, and Y is the
%   total number of pixels in an image. Because the images are square, the
%   sqaure root of Y will be a natural number.
n = sqrt(size(imgs,2));
if n ~= floor(n)
    error('Input images should be a row representation of a square matrix.');
end
N = size(imgs,1);
```

⋮

⋮

```
% Create mask for input matrix
%   We want to remove all "border" pixels from this data. We have already
%   determined the indices of the elements to remove, so we create a mask
%   in which ones will be removed, and zeros will remain.
mask = zeros(1,n^2);                          % initialize
mask( 1:n ) = 1;                              % left
mask( (n+1):n:(n*(n-1)+1) ) = 1;              % top
mask( n:n:(n*n) ) = 1;                        % bottom
mask( (n*(n-1)+1):(n^2) ) = 1;                % right
mask = repmat(mask,size(imgs,1),1);           % stack rows X times

% Convert image values to double and normalize
%   Because the input has values from [0,255], we divide the whole thing by
%   255 in order to obtain only values between [0,1]. We also transpose the
%   input, so that each column is a different image, because of the way
%   linear indices are defined.
imgs = double(imgs)' / 255;

% Use the mask and reshape output
%   Logical indexing allows us to remove all undesired indices from the
%   input matrix in one operation, but the output is a one dimensional
%   array. Using the reshape function and knowledge of the output geometry
%   we reconstruct our masked images. Finally, we take the transpose again,
%   so that each row is a different image.
imgs = reshape(imgs(~mask'),(n-2)^2,N)';

% Add column of ones to end of imgs to get A matrix for basic problem
%   train_s_stack is a column vector containing the pixel values of each
%   image in the training data set concatenated with a [1], which
%   will correspond to the constant shift of our hyperplane.
A = [ imgs, ones(N,1) ];

end
```

D.6 Feature Engineering Prep Function

```
function A = prep_MF(imgs,m,w)
%PRE_MF pre-processing images
%   Function to get input images into the desired form for the least
%   squares machine learning problem. This involves changing the data type
%   to double, dividing by 255, and removing the "border" pixels of each
%   image.

% Check and save size of input
%   Input should have each image as a single row in a matrix. That matrix
%   should be X by Y, where X is the number of input images, and Y is the
%   total number of pixels in an image. Because the images are square, the
%   sqaure root of Y will be a natural number.
n = sqrt(size(imgs,2));
if n ~= floor(n)
    error('Input images should be a row representation of a square matrix.');
end
N = size(imgs,1);

% Convert image values to double and normalize
%   Because the input has values from [0,255], we divide the whole thing by
%   255 in order to obtain only values between [0,1].
imgs = double(imgs) / 255;

% Indices of the elements in the leftmost and rightmost columns.
% Add some value k*n to the entire vector to shift k columns to the right,
% subtract k*n to shift k columns to the left.
left = 1:n;
right = (n*(n-1)+1):(n^2);

% Indices of the elements in the top and bottom rows. Add some value k to
% the entire vector to shift k rows down, subtract by k to shift k rows up.
top = 1:n:(n*(n-1)+1);
bottom = n:n:(n*n);

% Save number of pixels that will be removed from each row/column
m0 = m+2;
m1 = n-1-m;

% Take the weighted average of the border pixels that we are going to
% remove, excluding the outermost border pixels, as these will always be
% zero.
ave_left_mpix   = imgs(:, left(m0:m1)+n   ) * w(1);
ave_right_mpix  = imgs(:, right(m0:m1)-n  ) * w(1);
ave_top_mpix    = imgs(:, top(m0:m1)+1    ) * w(1);
ave_bottom_mpix = imgs(:, bottom(m0:m1)-1 ) * w(1);
for i = 2:m
    ave_left_mpix   = ave_left_mpix   + imgs(:, left(m0:m1)+i*n  ) * w(i);
    ave_right_mpix  = ave_right_mpix  + imgs(:, right(m0:m1)-i*n ) * w(i);
    ave_top_mpix    = ave_top_mpix    + imgs(:, top(m0:m1)+i     ) * w(i);
    ave_bottom_mpix = ave_bottom_mpix + imgs(:, bottom(m0:m1)-i  ) * w(i);
end
```

⋮

⋮

```
ave_left_mpix   = ave_left_mpix   / sum(w);
ave_right_mpix  = ave_right_mpix  / sum(w);
ave_top_mpix    = ave_top_mpix    / sum(w);
ave_bottom_mpix = ave_bottom_mpix / sum(w);

ave_border_mpix = [ ave_left_mpix ave_right_mpix ave_top_mpix ave_bottom_mpix
    ];

%%
% Average of center [shifted by (1,1)] 7x4 pixels
feat_mask = padarray(true(7,4),[10 13],'pre');
feat_mask = padarray(feat_mask,[11 11],'post');
feat_mask = reshape(feat_mask,1,[]);
more_feats = mean(imgs(:,feat_mask),2);

% Average of top left corner of innermost 20x20 square
feat_mask = padarray(true(10),[4 4],'pre');
feat_mask = padarray(feat_mask,[14 14],'post');
feat_mask = reshape(feat_mask,1,[]);
more_feats = [ more_feats mean(imgs(:,feat_mask),2) ];

% Average of bottom left corner of innermost 20x20 square
feat_mask = padarray(true(10),[14 4],'pre');
feat_mask = padarray(feat_mask,[4 14],'post');
feat_mask = reshape(feat_mask,1,[]);
more_feats = [ more_feats mean(imgs(:,feat_mask),2) ];

% Average of top right corner of innermost 20x20 square
feat_mask = padarray(true(10),[4 14],'pre');
feat_mask = padarray(feat_mask,[14 4],'post');
feat_mask = reshape(feat_mask,1,[]);
more_feats = [ more_feats mean(imgs(:,feat_mask),2) ];

% Average of top right corner of innermost 20x20 square
feat_mask = padarray(true(10),[14 14],'pre');
feat_mask = padarray(feat_mask,[4 4],'post');
feat_mask = reshape(feat_mask,1,[]);
more_feats = [ more_feats mean(imgs(:,feat_mask),2) ];

% Create mask for input matrix
%   We want to remove all "border" pixels from this data. We have already
%   determined the indices of the elements to remove, so we create a mask
%   in which ones will be removed, and zeros will remain.
[top_inds,bottom_inds] = deal(zeros(1,n*m));
for i = 1:m+1
    top_inds(n*(i-1)+1:n*(i-1)+n) = top+(i-1);
    bottom_inds(n*(i-1)+1:n*(i-1)+n) = bottom-(i-1);
end
top_inds = sort(top_inds);
bottom_inds = sort(bottom_inds);
```

⋮

⋮

```
mask = zeros(1,n^2);                  % initialize
mask( 1:(m+1)*n ) = 1;                % left
mask( n*(n-(m+1))+1:n^2 ) = 1;        % right
mask( top_inds ) = 1;                 % top
mask( bottom_inds ) = 1;              % bottom
mask = repmat(mask,size(imgs,1),1); % stack rows X times

% Transpose images so that we can use linear indexing
imgs = imgs';

% Use the mask and reshape output
%   Logical indexing allows us to remove all undesired indices from the
%   input matrix in one operation, but the output is a one dimensional
%   array. Using the reshape function and knowledge of the output geometry
%   we reconstruct our masked images. Finally, we take the transpose again,
%   so that each row is a different image.
imgs = reshape(imgs(~mask'),(n-2*(m+1))^2,N)';

% Add the averaged value of the 3 border pixels (after removing the
% outermost pixels) as features.
imgs = [ imgs ave_border_mpix more_feats ];

% Add column of ones to end of imgs to get A matrix for basic problem
%   train_s_stack is a column vector containing the pixel values of each
%   image in the training data set concatenated with a [1], which
%   will correspond to the constant shift of our hyperplane.
A = [ imgs, ones(N,1) ];

end
```

D.7 Script to Explore Weighting Values

```
%% Initialize variables
N_pairs = nchoosek(10,2);
pairs = nchoosek(0:9,2);
[x,t,best_u,best_d] = deal(cell(N_pairs,1));

%% Find "optimal" gamma for each classifier
for i = 1:N_pairs
    [x{i},t{i},best_u{i},best_d{i}] = ...
        gamma_opt(.008,pairs(i,1),pairs(i,2),'train_setup.mat','test_setup.mat'
            );
end

% Sort gamma output
gammas = zeros(N_pairs,1);

for i = 1:N_pairs
    % Save best up and down gamma of current classifier for clarity
    U = best_u{i};
    D = best_d{i};

    % Based on number of false positives and negatives, choose which gamma
    % is the better of the two. If both gammas resulted in the same number
    % of false positives and negatives, choose the smaller gamma. If the
    % number of false positives is greater in one while the number of false
    % negatives is greater in the other, choose the one that results in the
    % least number of errors total.
    if U(1) == D(1) && U(2) == D(2)
        gammas(i) = D(3);
    elseif U(1) < D(1)
        if U(2) <= D(2)
            gammas(i) = U(3);
        else
            dif = [D(1) - U(1),U(2) - D(2)];
            if dif(1) > dif(2)
                gammas(i) = U(3);
            else
                gammas(i) = D(3);
            end
        end
    else
        if U(2) >= D(2)
            gammas(i) = D(3);
        else
            dif = [U(1) - D(1),D(2) - U(2)];
            if dif(1) > dif(2)
                gammas(i) = D(3);
            else
                gammas(i) = U(3);
            end
        end
    end
end
```

D.8 Function to Create a Single Binary Classifier with Weighting Input

```
function [x,f,t] = create_classifier(D1,D2,gamma,setup)
%CREATE_CLASSIFIER
%   Function to create a classifier between unique digits D1 and D2 that
%   uses a weighting of gamma on the slack variables

% Verify that first two inputs are acceptable
%   Note that if the latter of the two digits is smaller than the first we
%   switch their order.
if mod(D1,1) || mod(D2,1) || D1 == D2 || D1 < 0 || D2 < 0 || D1 > 9 || D2 > 9
    error('First two inputs must be two unique digits.')
elseif D1 > D2
    temp = D2;
    D2 = D1;
    D1 = temp;
    clear('temp');
end

% Load in all additional necessary data
load(setup); %#ok<LOAD>

% Determine index of input pairing
[~,p] = ismember([D1 D2],pairs,'rows');

% Save the number of images whose label is D1 or D2
N1 = N_digimg(pairs(p,1)+1);
N2 = N_digimg(pairs(p,2)+1);

% CVX Problem
cvx_begin
    variable x(N_feat)
    variable u(N1) nonnegative
    variable v(N2) nonnegative
    minimize ( norm( x ) + gamma*( sum(u) + sum(v) ) )
    subject to
        im_sort{1,pairs(p,1)+1}*x + eye(N1)*u >= 1
        im_sort{1,pairs(p,2)+1}*x - eye(N2)*v <= -1
cvx_end

% Classifier values for each image
f = S*x;

% Time to create this solver
t = cvx_cputime;

end
```

D.9 Function to Test Single Binary Classifier

```
function [Fneg,Fpos] = test_classifier(x,D1,D2,setup)
%TEST_CLASSIFIER
%   Function to test a single classifier, requires input of the two digits
%   that are being chosen between (D1,D2).

% Load necessary data
load(setup); %#ok<LOAD>

% Use classifier on testing data
f = S*x;

% Take the subset of outputs whose label was supposed to be either D1 or D2
d1 = f(labels_test == D1);
d2 = f(labels_test == D2);

% Save the index of the false negatives (incorrectly labeled as D1) and
% false positives (incorrectly labeled as D2).
Fneg = find(d1 <  0);
Fpos = find(d2 >= 0);

end
```

D.10 Function Used for Margin Verification

```
function area = margin_test(outlier_flag,w)
%MARGIN_TEST

if ~exist('outlier_flag','var')
    outlier_flag = 0;
    w = 1;
end

S = 50;
if outlier_flag
    N_out = 2;
else
    N_out = 0;
end
x = rand(S-N_out,1);
y = rand(S-N_out,1);

lr = .95;
ur = 1.1;

x1 = x((x + y) <= lr);
y1 = y((x + y) <= lr);

if outlier_flag
    linex = .1*rand(1,N_out) + [.7 .75];
    liney = 1-linex;
    x1 = [ x1; linex'+.15*rand(N_out,1) ];
    y1 = [ y1; liney'+.15*rand(N_out,1) ];
end

x2 = x((x + y) >= ur);
y2 = y((x + y) >= ur);

N1 = size(x1,1);
N2 = size(x2,1);

A1 = [x1 -y1 ones(N1,1)];
A2 = [x2 -y2 ones(N2,1)];

cvx_begin quiet
    variable z_a(3)
    variable u_a(N1) nonnegative
    variable v_a(N2) nonnegative
    minimize ( sum(u_a) + sum(v_a) )
    subject to
        A1 * z_a + eye(N1)*u_a >= 1
        A2 * z_a - eye(N2)*v_a <= -1
cvx_end
```

⋮

⋮

```
cvx_begin quiet
    variable z_b(3)
    variable u_b(N1) nonnegative
    variable v_b(N2) nonnegative
    minimize ( norm(z_b) + w*(sum(u_b) + sum(v_b)) )
    subject to
        A1 * z_b + eye(N1)*u_b >= 1
        A2 * z_b - eye(N2)*v_b <= -1
cvx_end

p1_a = [ 0,           z_a(3) /z_a(2)  ];
p2_a = [ 1, (z_a(1) + z_a(3))/z_a(2) ];
p1_b = [ 0,           z_b(3) /z_b(2)  ];
p2_b = [ 1, (z_b(1) + z_b(3))/z_b(2) ];

area(1) = polyarea([mean([lr,ur]),mean([lr-1,ur-1]),p1_a(1),p2_a(1),mean([lr,ur
   ])],[0,1,p1_a(2),p2_a(2),0]);
area(2) = polyarea([mean([lr,ur]),mean([lr-1,ur-1]),p1_b(1),p2_b(1),mean([lr,ur
   ])],[0,1,p1_b(2),p2_b(2),0]);

end
```