Classification of Handwritten Digits with Support Vector Machines

Aparajithan Venkateswaran   Perrin Ruth   Derek Wright

James Meiss
APPM 3310, Fall 2017
University of Colorado, Boulder

Abstract

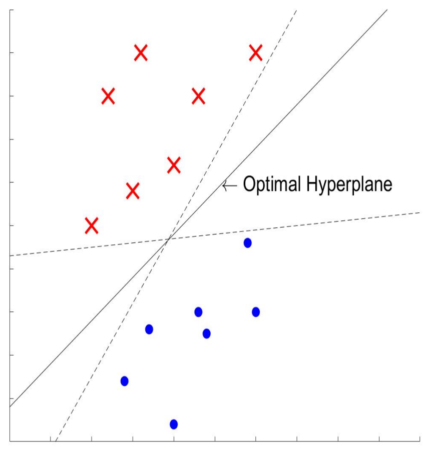
Classifying text in images is an interesting problem. In this report, we study Support Vector Machines (SVM) [1] and how we can use them to classify images of handwritten digits from the MNIST database [2]. SVM classifies data into two different classes by constructing a hyperplane that separates the two classes. For our application, we have 10 *different classes (*10 *digits). So, we constructed* 10 *optimal hyperplanes, one for each digit k, each of which was able to classify the image as either digit k or not.*

In case the data is not linearly separable, it is impossible to construct such an optimal hyperplane. For such data, we need to use a kernel to transform the data such that it becomes linearly separable. We used 4 *different kernels - linear, quadratic, cubic, and sigmoid kernel. The quadratic transformation of the data provided the best results with an accuracy of* 98.08 % *and the sigmoid transformation gave the worst results with an accuracy of* 10.09 %.

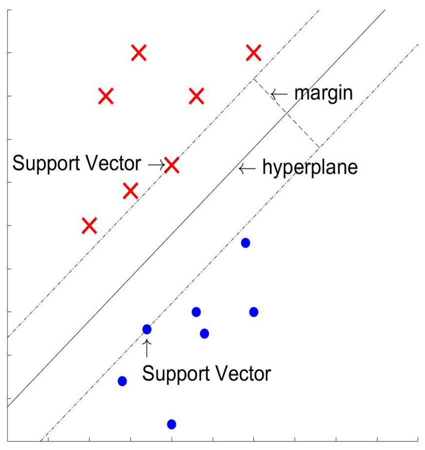
1. Introduction

The problem of using computers to automatically classify images is a very difficult problem, and is, rightly, heavily studied. This is a particularly interesting question because, while computers can efficiently solve large problems that are difficult for humans, they struggle with tasks that humans find easy, such as identifying text in images. Over the past few decades, the broader classification problem has been tackled by mathematicians using varied and novel techniques. One of these techniques, originally discussed by Vladimir Vapnik in 1994, is *Support Vector Machines* [1]. This novel method approached the problem by constructing the *optimal hyperplane* that separates data into two different classes.

For our project, we chose to investigate how we can use Support Vector Machines (SVM) to classify images of handwritten digits i.e., identify the digit. We chose this because recognizing text in images has a wide array of applications from creating digital records of old texts to creating searchable images of directories, menus, etc. Owing to the complexity of the larger problem, we chose a subset - classifying single digits. For our project, we



(a) *Finding the optimal hyperplane*



(b) *The optimal hyperplane*

Figure 1: *Figure 1(a) shows 3 different separating hyperplanes, however only one separates the data most naturally. This is called the optimal hyperplane. Figure 1(b) shows the optimal hyperplane, the margin planes, and the support vectors.*

will be using the MNIST database [2]. The MNIST database, compiled by LeCunn, et. al., consists of $70,000$ grayscale images of handwritten digits. The database was originally compiled for the purpose of building a system that would automatically classify and sort postal mail by zipcodes.

The rest of the report is organized as follows: Section **2** describes the mathematics, significance, and caveats of SVM; Section **3** describes the experiments we performed on the MNIST database using SVM; Section **4** discusses the results we got from our experiments in section **3** and; Section **5** discusses the conclusions of our project, and provides some ideas for future research.

2. Mathematical Formulation

The concept of Support Vector Machines is very simple. The SVM classifies data into two different classes by constructing a hyperplane that separates the two different classes. Figure 1 illustrates a simple example.

Suppose we have training data:

$$(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_m}, y_m)$$
$$\mathbf{x_i} \in \mathbb{R}^n \quad \text{and} \quad y_i \in \{1, -1\}$$

where 1 and -1 represent the two classes. We have chosen $y_i \in \{1, -1\}$ because it makes the math simpler and cleaner, as we will see. Now, suppose that the training data can be separated by the hyperplane[1]:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1}$$

We construct this hyperplane such that:

$$\mathbf{w} \cdot \mathbf{x_i} + b \geq 1 \text{ if } y_i = 1, \tag{2}$$
$$\mathbf{w} \cdot \mathbf{x_i} + b \leq -1 \text{ if } y_i = -1 \tag{3}$$

Once again, choice of 1 and -1 for our margins are arbitrary. Now, notice that we have also constructed two planes that are parallel to our separating plane (1). These planes form the margin for their respective classes.

$$\mathbf{w} \cdot \mathbf{x} + b = 1 \tag{4}$$
$$\mathbf{w} \cdot \mathbf{x} + b = -1 \tag{5}$$

Generally, there are infinitely many hyperplanes that are able to separate the data. To find the *best* hyperplane (the optimal hyperplane) that separates the two classes, we want to maximize the distance between the two margin hyperplanes. This supports our intuition that the *best* separating plane should be as far away from both the classes as possible. See Figure 1(a) to understand why this supports our intuition.

The distance between the margin hyperplanes is given by[2]:

$$d = \frac{2}{\|\mathbf{w}\|} \tag{6}$$

The 2 in the numerator is consequence of our choice that $y_i \in \{1, -1\}$. We want to maximize the distance, and maximizing the distance can be thought of as minimizing $\frac{\|\mathbf{w}\|}{2}$. And minimizing $\frac{\|\mathbf{w}\|}{2}$ is the same as minimizing $\frac{\|\mathbf{w}\|^2}{2}$. So now, we have our optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2} \|w\|^2$$
$$\text{s.t. } \begin{cases} \mathbf{w} \cdot \mathbf{x_i} + b \geq 1 \text{ if } y_i = 1 \\ \mathbf{w} \cdot \mathbf{x_i} + b \leq -1 \text{ if } y_i = 1 \end{cases}$$

[1]An important assumption we make is that the training data are linearly separable. We will later see how we can satisfy this assumption

[2]See Appendix A for proof

This is a Quadratic Programming Problem, and beyond the scope of this report[3]. But, it turns out that our choice of 1 and -1 for the margins in equations (4-5) make this problem optimal i.e., it can be optimized subject to the constraints we have.

One of the constraints of the optimization problem that can be derived is[4]:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i} \tag{7}$$

where α_i are the Lagrange multipliers. From this, we can deduce that the vector orthogonal to the optimal hyperplane is a linear combination of the input vectors i.e.,

$$\mathbf{w} \in \text{span}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}) \tag{8}$$

$$\mathbf{w} \in \text{rng}(X); \quad X = (\mathbf{x_1}\ \mathbf{x_2}\ \ldots\ \mathbf{x_n}) \tag{9}$$

We do not know anything about the independence of $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$. So, we cannot say anything about the basis and dimension of X. Also, with equation (7), we can now rewrite our optimal hyperplane as:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x} + b &= \Big(\sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i}\Big)^{\mathrm{T}} \mathbf{x} + b \\ &= \sum_{i=1}^{m} \alpha_i y_i \langle \mathbf{x_i}, \mathbf{x} \rangle + b \end{aligned} \tag{10}$$

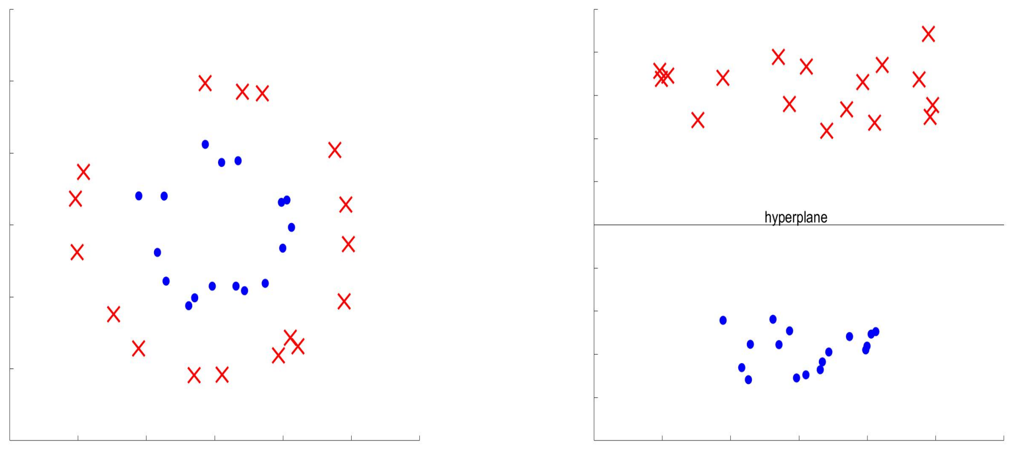
Now, let us turn our attention back to the margin hyperplanes. Notice that these hyperplanes are defined by the vectors that are the closest to the optimal hyperplane. These vectors determine our margin, and thus our optimization problem. Hence, these vectors are called **Support Vectors** and hence this technique is called *Support Vector Machines*. Figure 1(b) shows the support vectors for our simple example.

Going back to our assumption that the data we have are linearly separable, we know that this assumption is not always valid. So, we can use something called a **Kernel**. This technique employs a transformation on the input data to transform the data into a different space, one that is now linearly separable.

Figure 2 illustrates a simple example where we use a kernel transformation to be able to linearly separate our data. A kernel can be thought of as a transformation function applied to the data. A kernel function is usually nonlinear. Using a linear transformation as the kernel would be futile because, the representation matrix of the linear transform, A, would get absorbed by the normal vector of the hyperplane, $\mathbf{w}$, leaving us with the

[3]See Appendix B for a discussion of this optimization problem in detail

[4]See Appendix B for details



(a) *Initial raw data* **(b)** *Data transformed with a kernel*

Figure 2: *Initially, the data, 2(a), is not linearly separable i.e., cannot be separated by a hyperplane. After applying a kernel transform, our transformed data, 2(b), becomes linearly separable. Now, we can use a Support Vector Machine to classify the data.*

raw untransformed data again as seen below:

$$\begin{aligned} \mathbf{x}^{'} &= \mathcal{L}[\mathbf{x}] = A\mathbf{x} \\ \mathbf{w} \cdot \mathbf{x}^{'} + b &= 0 \\ \mathbf{w} \cdot (A\mathbf{x}) + b &= 0 \\ (\mathbf{w}A) \cdot \mathbf{x} + b &= 0 \\ \mathbf{w}^{'} \cdot \mathbf{x} + b &= 0 \end{aligned}$$

We can also think of kernels as replacing the inner products in equation (10) with different feature mappings. For instance, if we have a feature mapping $\phi(\mathbf{x})$, we can replace the inner product with the corresponding Kernel to be:

$$K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{y})$$

For instance, we can define a new polynomial kernel of degree n as:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^{\mathrm{T}}\mathbf{y} + c)^{n}$$

where c is a parameter that controls the relative weighting of $\mathbf{x}$ and $\mathbf{y}$. It turns out that there is a feature mapping ϕ that corresponds to this kernel function i.e., there is a ϕ such that $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{y})$ for all $\mathbf{x}, \mathbf{y}$ [3]. More generally, we can use K as a kernel function only if it corresponds to a feature mapping ϕ.

Suppose that K is a valid kernel corresponding to a feature mapping ϕ. Then, consider a set of n vectors $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$. Now, we can define a matrix, called Kernel Matrix K, whose (i, j)-th entry is given by $K_{ij} = K(\mathbf{x_i}, \mathbf{x_j})$. Note that we have overloaded K with both the kernel function and the Kernel matrix. If K is a valid Kernel, then $K_{ji} = K(\mathbf{x_j}, \mathbf{x_i}) = \phi(\mathbf{x_j})^{\mathrm{T}}\phi(\mathbf{x_i}) = \phi(\mathbf{x_i})^{\mathrm{T}}\phi(\mathbf{x_j}) = K(\mathbf{x_i}, \mathbf{x_j}) = K_{ij}$. Thus, K is symmetric. Let $\phi_k(\mathbf{x})$ denote the k-th coordinate of the $\phi(\mathbf{x})$ vector. And, let $\mathbf{z}$ be a vector in $\mathbb{R}^n$.

$$\begin{aligned}
\mathbf{z}K\mathbf{z} &= \sum_i \sum_j z_i K_{ij} z_j \\
&= \sum_i \sum_j z_i K(\phi(\mathbf{x_i}), \phi(\mathbf{x_j})) z_j \\
&= \sum_i \sum_j z_i \sum_k K(\phi_k(\mathbf{x_i}), \phi_k(\mathbf{x_j})) z_j \\
&= \sum_k \sum_i \sum_j z_i K(\phi_k(\mathbf{x_i}), \phi_k(\mathbf{x_j})) z_j \\
&= \sum_k \left(\sum_i z_i \phi_k(\mathbf{x}) \right)^2 \\
&\geq 0 \qquad (11)
\end{aligned}$$

From equation (11), we know that the Kernel matrix is positive semi-definite ($K \geq 0$). Hence, if K is a valid kernel (i.e., corresponds to a feature mapping ϕ), then the corresponding Kernel matrix $K \in \mathbb{R}^{n \times n}$ is symmetric positive definite. More generally, it turns out that this not just a necessary, but also a sufficient, condition for K to be a valid kernel [3].

We must take care when using a kernel transformation. Using the wrong transformation can worsen our results. For instance, if we initially had data that was linearly separable and transformed it with, say, a polar transformation, then our new feature space (transformed data) is no longer linearly separable, decreasing the accuracy of our classifier.

3. Classifying Handwritten Digits

We applied the idea of Support Vector Machines to classifying handwritten digits from the MNIST dataset [2]. The MNIST database consists of 70,000 correctly labeled images of handwritten digits, that are size-normalized and centered. 60,000 of them belong to the training set i.e., these images are used to find the optimal hyperplane. The remaining 10,000 belong to the testing set i.e., we test the accuracy of our hyperplanes with these images. All of them are 28×28 pixel images of a handwritten single digit. Figure 3 shows some sample images from the dataset.

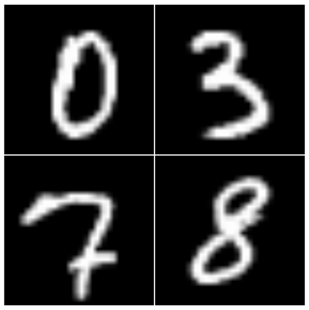


Figure 3: *Sample images from MNIST dataset*

Our classification problem is that of learning to recognize the correct digit. With classifying digits, we have 10 different classes i.e., $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Since this is a multi-class classification problem (there are more than 2 classes), we employed the one-versus-rest classification technique. In this technique, we construct 10 different hyperplanes - each hyperplane distinguishes the input image between one class and the rest. For example, one such hyperplane would classify the input image as either 0, or not. Another hyperplane would classify the input image as either 3, or not. Thus, we constructed 10 different hyperplanes. Note that we need 10, and not 9, hyperplanes to account for the possibility that the image we are classifying might not be any of the 10 digits. Given an image, we then looped through each of 10 hyperplanes and checked if the image belongs to that class. A drawback with this method is that we also run into the possibility of the same image being assigned to two different classes. When that happened, we chose the hyperplane that gave a higher output value for the image.

Before actually finding the optimal hyperplanes, let us introduce some old notation to our new data. Let the number of training images be m. Let $X \in \mathbb{R}^{m \times n}$, be our training set. Let us denote each row of X as $\mathbf{x_i^T}$, which is, in turn, a single image. Usually images are matrices with pixel values as entries for the matrix. Here, we unrolled the 2-D matrix into a one dimensional vector. Let the length of the unrolled vector be n. Therefore, $\mathbf{x_i} \in \mathbb{R}^n$. And thus, the pixel values of the image become features in our input feature space. Since we are working with gray-scale images, we do not have to worry about RGB channels. In our case, $m = 60,000$, and $n = 28 \times 28 = 784$.

Our y_i associated with each of the $\mathbf{x_i}$ depends on the hyperplane we are constructing. Assume we are constructing the k^{th} hyperplane i.e., the hyperplane that classifies the image as either digit k, or not digit k. Then $y_i = 1$ if image $\mathbf{x_i}$ represents digit k, and $y_i = -1$ otherwise. These values follow from our mathematical formulation in Section **2**. Let us define the normal to this hyperplane as $\mathbf{w}^{(k)}$. Since our y_i depends on the value k, let us re-define our notation as $y_i^{(k)}$, where

$$y_i^{(k)} = \begin{cases} 1, \text{ if } x_i \text{ represents } k \\ -1, \text{ otherwise} \end{cases}$$

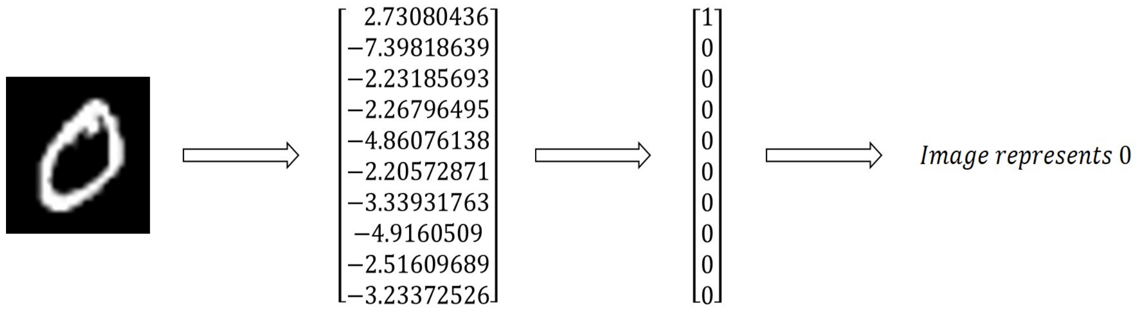


Figure 4: *With the given image (pixel values), we compute the output with respect to each of the 10 hyperplanes. The output is vectorized with i-th element of the vector representing the output with the hyperplane that distinguishes between digit $i-1$ and the rest. We then take all the values that yielded a positive output to determine which class the image belongs to.*

With this modified notation, we trained the support vector machine and obtain our 10 hyperplanes. As discussed previously, for predicting the digit on an image, we can use equation (12) that uses our new notation.

$$f^{(k)}(x_i) = \mathbf{w}^{(k)} \cdot \mathbf{x_i} + b \tag{12}$$

If $f^{(k)}(x_i) \geq 0$, then $\mathbf{x_i}$ represents k, and it does not represent k otherwise. If, using this method, we find that $\mathbf{x_i}$ represents multiple k_j, then we choose a k_a such that

$$f^{(k_a)}(x_i) = \max_j f^{(k_j)}(x_i)$$

Finally, to make computation simpler, we set the intercept as the origin i.e., $b = 0$. Since the optimization problem is beyond the scope of this project, we used `sklearn` [4] library to help us with the optimization process[5].

4. Results

With the techniques, and assumptions described in the previous section, we used Python and `sklearn` to build a Support Vector Machine classifier to classify handwritten digits from the MNIST dataset.

We first fit our hyperlanes using 60, 000 training images without any kernel transform i.e., we used the linear kernel. Then, we tested the accuracy of our hyperplanes by using it to classify 10, 000 testing images. With the linear kernel, we got an accuracy of 86.49%. Figure 4 shows how we processed the output from each of the 10 hyperplanes and then deciphered to which class the image belonged to. Some of the important code snippets are discussed in Appendix C.

[5]See Appendix B for details.

Accuracy of our classifiers		
Kernel	Training Accuracy	Testing Accuracy
Linear	87.27 %	86.49 %
Polynomial (deg=2)	100 %	98.08 %
Polynomial (deg=3)	99.99 %	97.85 %
Sigmoid	9.92 %	10.09 %

Table 1: *Accuracy of different classifiers*

We then transformed our data with the polynomial kernel. We first used a quadratic transformation, and then a cubic transformation. We also tried using the sigmoid kernel[6]. Our results for all of these are summarized in Table 1.

As we can see from Table 1, our data (the images) is not completely linearly separable. In fact, we get the best accuracy on our testing images when we used a polynomial of degree 2 i.e., a quadratic transformation. It makes sense to think that transforming the data into the wrong feature space will make it harder, and sometimes impossible, to find a decent hyperplane that separates the data. Our intuition is supported by the sigmoid kernel where our accuracy drops by a factor of 10 compared to the linear kernel.

5. Conclusion

Our results show that it is possible to classify images using the idea of Support Vector Machines i.e., by transforming the pixel values and constructing the optimal separating hyperplane. We saw that the Kernel matrix associated with these transformation can be constructed by any positive semidefinite matrix i.e., a valid transformation is obtained by relaxing the positive definiteness property associated with an inner product. By making a linear transformation (using raw pixel values), our support vector machine has an accuracy of 86.49 %. With polynomial transformations of degree 2 and 3, we got an accuracy of 98.08 % and 97.85 % respectively. As expected, making a *wrong* transformation can make it impossible to construct a hyperplane. Our results with the sigmoid transformation support our belief with a reduced accuracy of just 10.09 %. A key idea we learned from this project was that it is possible to reduce a nonlinear problem into a linear version that can be solved using existing, or simpler techniques.

We experimented with 4 different transformations, each with constructing 10 hyperplanes. We can improve upon this design by using different kernels for each of the 10 different hyperplanes i.e., one number that we are classifying, say 3, could be in a different space than another number, say 5. So our model would perform better by constructing

[6]Sigmoid kernel uses the transformation $K(\mathbf{x}, \mathbf{y}) = \tanh(\gamma \mathbf{x}^{\mathrm{T}}\mathbf{y} + r)$ where γ and r are the kernel parameters. We used $\gamma = \frac{1}{m} = \frac{1}{60{,}000}$ and $r = 0$.

a hyperplane in the linear transformation for classifying 3, and a hyperplane in the polynomial transformation for classifying 5.

This technique is very much relevant today and has many applications. Moving forward, we can also build classifiers for the alphabets. Then, we can use the sliding window technique (scanning each $m \times m$ pixel box in an image) on an image with text to classify and read the text present in the image. Support Vector Machines can also be used in other classification problems such as predicting if a patient has a tumor, or not; music recommendation and; cataloging images of different animal species. The applications are countless.

Currently, the multi-class classification problem requires constructing k different hyperplanes. Future research in this area could be studying if it is possible to construct a single set of $m < k$ hyperplanes that form a box that encloses and separates the different classes, and if that method will be a more efficient way for approaching the multi-class classification problem than the one-versus-rest technique.

References

[1] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Springer Science+Business Media, 1 ed., 1995.

[2] Y. LeCun, C. Cortes, and C. Burges, "MNIST Handwritten Digit Database." `http://yann.lecun.com/exdb/mnist/`, 1999. Accessed 2017-12-03.

[3] A. Ng, "CS229 Lecture Notes: Support Vector Machines." `http://cs229.stanford.edu/notes/cs229-notes3.pdf`, September 2017. Accessed 2017-12-06.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, pp. 273–297, Sep 1995.

[6] A. Kowalczyk, "SVM: Understanding Math - The Optimal Hyperplane." `https://www.svm-tutorial.com/2015/06/svm-understanding-math-part-3/`, June 2015. Accessed 2017-12-06.

[7] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," tech. rep., April 1998.

A. Distance Between Two Parallel Hyperplanes

We have our optimal hyperplane (13) and the two margin hyperplanes (14), (15) that are equidistant from hyperplane (13).

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{13}$$
$$\mathbf{w} \cdot \mathbf{x} + b = \delta \tag{14}$$
$$\mathbf{w} \cdot \mathbf{x} + b = -\delta \tag{15}$$

Let the distance between the two planes be d. We know that all these hyperplanes are parallel. Therefore, we can write the vector from hyperplane (14) to hyperplane (15) as:

$$\mathbf{k} = d\frac{\mathbf{w}}{\|\mathbf{w}\|} \tag{16}$$

Now, let $\mathbf{x_0}$ be a point on the hyperplane (15) and $\mathbf{x_1}$ be a point on the hyperplane (14) that is in the direction of the normal vector $\mathbf{w}$ i.e., $\mathbf{x_1} = \mathbf{x_0} + \mathbf{k}$.

$$\begin{aligned}
\mathbf{w} \cdot \mathbf{x_1} + b &= \delta \\
\mathbf{w} \cdot (\mathbf{x_0} + \mathbf{k}) + b &= \delta && (\mathbf{x_1} = \mathbf{x_0} + \mathbf{k}) \\
\mathbf{w} \cdot \mathbf{x_0} + b + \mathbf{w} \cdot \mathbf{k} &= \delta \\
\mathbf{w} \cdot \mathbf{k} - \delta &= \delta && (\mathbf{w} \cdot \mathbf{x_0} + b = -\delta) \\
d\frac{\mathbf{w} \cdot \mathbf{w}}{\|\mathbf{w}\|} &= 2\delta \\
d &= \frac{2\delta}{\|\mathbf{w}\|} && (17)
\end{aligned}$$

Now, if we set $\delta = 1$ to make our math simpler, we can rewrite (17) as:

$$d = \frac{2}{\|\mathbf{w}\|}$$

B. Finding the Optimal Hyperplane

Enough justice to the optimization problem cannot be done without going beyond the scope of linear algebra. However, we will still try to explain the problem without delving too much into the nonlinearities.

We are presented with the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2} \|w\|^2$$
$$\text{s.t. } \begin{cases} \mathbf{w} \cdot \mathbf{x_i} + b \geq 1 \text{ if } y_i = 1 \\ \mathbf{w} \cdot \mathbf{x_i} + b \leq -1 \text{ if } y_i = 1 \end{cases}$$

Let us define a new function $g_i(\mathbf{w}, b)$ as:

$$g_i(\mathbf{w}, b) = -y_i(\mathbf{w} \cdot \mathbf{x_i} + b) + 1 \tag{18}$$

Now, if there are m training examples in our dataset, we can rewrite the problem in a familiar manner, and let's call it the **primal** optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2} \|w\|^2$$
$$\text{s.t. } g_i(\mathbf{w}, b) \leq 0 \,, \quad i = 1, 2, \ldots, m$$

Now, we can construct the Lagrangian for our optimization problem:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x_i} + b) - 1] \tag{19}$$
$$\text{s.t. } \alpha_i \geq 0 \,, \quad i = 1, 2, \ldots, m$$

where α_i are our Lagrange multipliers. To construct the dual form of our problem, set the derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and b to zero. We have:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i} = 0$$
$$\implies \mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i} \tag{20}$$
$$\text{and } \frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \alpha) = \sum_{i=1}^{m} \alpha_i y_i = 0 \tag{21}$$

Simplifying (19) with (20) and (21), we have

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle \mathbf{x_i}, \mathbf{x_j} \rangle \tag{22}$$

Using (22), our constraints $\alpha_i \geq 0$ and (21), we obtain the following **dual** optimization problem:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle \mathbf{x_i}, \mathbf{x_j} \rangle \tag{23}$$

$$\text{s.t.} \quad \alpha_i \geq 0\,, \quad i = 1, 2, \ldots, m \tag{24}$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{25}$$

We can easily verify that the solutions to the **primal** correspond to the solutions of the **dual**. It also happens that the Karush-Kuhn-Tucker (KKT) conditions hold true for our optimization problem. For any nonlinear programming to be optimal, KKT are necessary conditions. This guarantees that our optimization problem can be optimized.

Constraints (24-25) make it impossible for us to modify a single α_i without changing any other α_j because of (25). Therefore, we must update at least two α_i's simultaneously to keep satisfying constraints. This idea is the motivation behind the Sequential Minimal Optimization (SMO) algorithm which we will use to optimize the **dual** problem. Interested readers can refer [7] for the original discussion of the SMO algorithm. A complete description of the mathematics of this algorithm is beyond the scope of this report. So, we will end our discussion by providing a very simple psuedocode for the algorithm that is adapted from [3]:

```
Repeat until convergence {
        1. Select some pair $\alpha_i$ and $\alpha_j$ to update next (using a heuristic that
           tries to pick the two that will allow us to make the biggest
           progress towards the global maximum)
        2. Reoptimize $W(\alpha)$ with respect to $\alpha_i$ and $\alpha_j$, while holding all the
           other $\alpha_k$'s $(k \neq i, j)$ fixed
}
```

C. Code

Here are some code snippets from the actual program that we used to construct hyperplanes and classify images. Code Snippet **(1)** shows some important libraries that we used.

Code Snippet 1: *Importing important libraries*

```
import numpy as np
from sklearn import svm # Provides the optimizer
```

Code Snippet **(2)** shows how we used `sklearn` to find the optimal hyperplane. Here, `X_train` is a `numpy` array of the training images that have been unrolled and vectorized. `y_train` is a `numpy` array of the correct label for each of the corresponding images in `X_train`.

Code Snippet 2: *Creating and optmizing the support vector machine*

```
# Sets up a simple support vector machine
lin_clf = svm.LinearSVC()
# Finds the optimal hyperplane by solving the optimization problem
lin_clf.fit(X_train, y_train)
```

Code Snippet **(3)** shows how we can then retrieve the information about the hyperplanes, and the calculate the prediction from the optimized model.

Code Snippet 3: *Predicting the class*

```
# Finds the output for each of the 10 optimal hyperplanes
output = lin_clf.decision_function([X_train[1]])
>>> output = [[ 2.73080436 -7.39818639 -2.23185693 -2.26796495 -4.86076138
    -2.20572871 -3.33931763 -4.9160509 -2.51609689 -3.23372526 ]]

# Uses the output to predict the class
prediction = lin_clf.predict([X_train[1]])
>>> prediction = [0]
```

You can find the entire code we used at our GitHub repository - `https://github.com/AparaV/multi-digit-classifier`.