Binary Classification Using Hard-margin Linear Support Vector Machines

Stanley A. Baronett*
Department of Physics and Astronomy,
University of Nevada Las Vegas

(Dated: December 9, 2019)

Using Numpy and a rudimentary, pseudo-quadratic programming solving algorithm, we designed a Support Vector Machine class in Python for linear, hard-margin, binary classification. A simple, linearly-separable, six-sample, two-dimensional dataset was used to train the model for predictions. After optimizing its parameters, we were able to train the model in 0.631±0.025 seconds, with support vectors at worst 7.1% off from their ideal values.

I. INTRODUCTION

Vladimir Vapnik originally developed the algorithm for the Support Vector Machine (SVM) in 1963, but it was not properly noticed or realized by the machine learning (ML) community until he defected to the United States in the 1990s. Since then, this supervised learning model been improved upon and is considered a staple in classification and regression analysis problems.

Among other ML models, SVMs' advantages include: effective in high-dimensional feature-spaces; effective when the number of samples is less than the dimensionality; once trained, is memory efficient in its predictions (decision function); versatile, as various kernel functions can be implemented to improve the model. Applications of SVMs include: outlier detection; handwriting, text and hypertext categorization and recognition; image classification; biological sciences, such as protein and compound classification.

II. THEORETICAL BACKGROUND

Referring to Figure 1, suppose we have a training dataset of objects (or *instances*), each with their own real-valued attributes (or *features*), x_1 and x_2, and classified with either a value of -1 or $+1$, represented as "-" (minus) or "+" (plus) symbols respectively. Intuitively, if the two classes are *linearly separable*, i.e., can be spatially separated by a dividing *line* or *hyperplane* (for higher dimensions), we expect that the *best* hyperplane would fall evenly between the two groups, leaving the greatest *margin* between the nearest instances of either class. In the case of Fig. 1, we can argue that the best hyperplane, or *linear classifier*, to be the dashed orange line, as it corresponds to the *widest possible* set of *parallel* margins, represented as the solid yellow lines.

Formally, we can represent any instance, i, as a vector in "feature space" as

$$\mathbf{x_i} = [x_1, x_2],$$

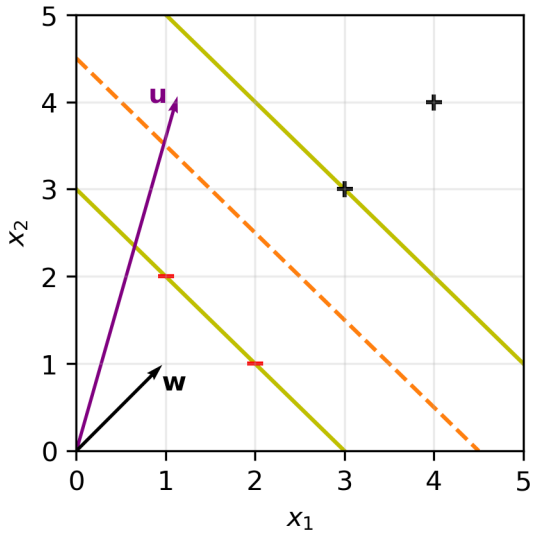

Figure 1: Example of dual-featured training data, belonging to either a positive or negative class. The dashed orange hyperplane defines the best linear classifier that separates the two classes with the widest margin (solid yellow lines). If the projection of some unknown instance **u** (purple) onto the hyperplane's normal **u** is greater than or less than some bias (i.e., to the "right" or "left" of the hyperplane), then we classify it as positive or negative respectively.

and it's corresponding binary, scalar classification as

$$y_i \equiv \begin{cases} -1, & \text{for negative samples,} \\ +1, & \text{for positive samples.} \end{cases} \tag{1}$$

Now, let us define a new vector, **w**, to be *normal* to the hyperplane and its parallel margins, seen as the black vector in Fig. 1. And, let us introduce some *new, unknown* instance (perhaps some test data) that we wish to classify as either a positive or negative sample, represented by the vector **u**, seen in purple. Although we do not yet know the actual direction or length of **w**, as a consequence of its *definition*, we can argue that, if a projection of **u** onto **w** turned out to be *greater* than some fixed value c, then we might classify it as being "posi-

* barons2@unlv.nevada.edu

tive." Specifically,

$$\mathbf{w}\cdot\mathbf{u} \geq c \quad \text{(positive samples)},$$

which implies

$$\boxed{\mathbf{w}\cdot\mathbf{u} + b \geq 0 \quad \textbf{(decision rule)},} \tag{2}$$

where our *bias* $b \equiv -c$. The left-hand side of this equation describes the hyperplane *boundary*, drawn as the dashed orange line in Fig. 1. And, if this inequality holds true, then we classify our new instance $\mathbf{u}$ as "positive;" if false, then "negative." In fact, the general equation of an $(n-1)$-dimensional **hyperplane** is

$$\mathbf{w}\cdot\mathbf{x} + b = 0, \tag{3}$$

where $\mathbf{w}$ and $\mathbf{x}$ are n-dimensional vectors and b is some offset (or "bias") from the origin. If $\mathbf{w}$ and $\mathbf{x}$ are 2-D, the hyperplane is thus 1-D, and Eq. 3 becomes our familiar $mx + b$—the equation of a line.

If we apply Eq. 2 to our existing training data, we find

$$\begin{aligned}\mathbf{w}\cdot\mathbf{x}_+ + b &\geq 1\\ \mathbf{w}\cdot\mathbf{x}_- + b &\geq -1,\end{aligned}$$

where $\mathbf{x}_+$ and $\mathbf{x}_-$ are the known positive and negative samples. If we multiply either side of both equations by our definition for y_i (Eq. 1), we find they both yield

$$\begin{aligned}y_i(\mathbf{w}\cdot\mathbf{x}_i + b) &\geq 1, \text{ or}\\ y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 &\geq 0.\end{aligned}$$

For the instances that lie *exactly on the margins*, which we will refer to as our **support vectors (SV)** (as they "support" our linear, classifying hyperplane), we find our *equation of constraint* on $\mathbf{w}$ and b to be

$$\boxed{y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 = 0 \quad \textbf{(SV constraint)},} \tag{4}$$

Referring to Fig. 2, we can further analyze the *width* of the margins, as this is the value we wish to *maximize*. Let $\mathbf{x}_-$ and $\mathbf{x}_+$ refer to two oppositely-classified SVs, seen in red and black respectively, with their *difference* drawn in blue. Although we do not yet know the *length* of $\mathbf{w}$, we can make it a unit vector by dividing it by its magnitude, shown in green. Thus, the **width** of the margins is the difference of the two SVs projected onto this unit-normal, or

$$\begin{aligned}\text{width} &= (\mathbf{x}_+ - \mathbf{x}_-)\cdot\frac{\mathbf{w}}{\|\mathbf{w}\|}\\ &= \frac{(\mathbf{x}_+\cdot\mathbf{w}) - (\mathbf{x}_-\cdot\mathbf{w})}{\|\mathbf{w}\|}.\end{aligned}$$

But, by definition of y_i (Eq. 1), $x_+ \implies y_i = 1$ and $x_- \implies y_i = -1$. Thus, from our constraint equation (Eq. 4,

$$\begin{aligned}\mathbf{x}_+\cdot\mathbf{w} &= 1 - b,\\ \mathbf{x}_-\cdot\mathbf{w} &= -(1 + b),\end{aligned}$$

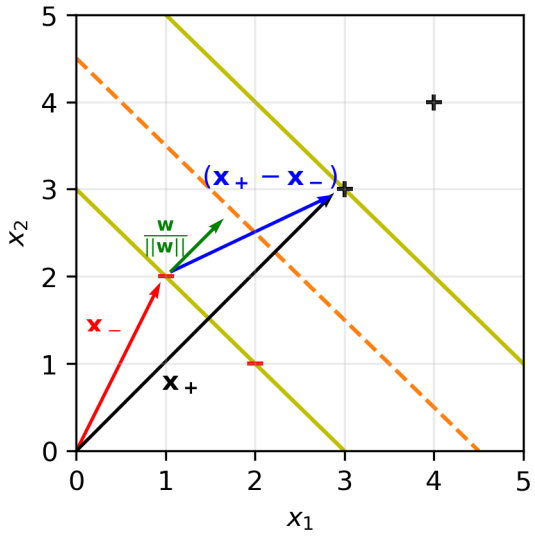

Figure 2: Vector analysis of positive and negative support vectors, the unit-normal vector to the decision boundary hyperplane and its relationship to the width of the margins.

so that

$$\begin{aligned}\text{width} &= \frac{(1-b)+(1+b)}{\|\mathbf{w}\|}\\ &= \frac{2}{\|\mathbf{w}\|}.\end{aligned}$$

Since our goal is to maximize the width of the margins, this is equivalent to *minimizing* $\|\mathbf{w}\|$. Or, to make the ensuing analysis easier (as we shall see), we wish to minimize

$$\boxed{\frac{1}{2}\|\mathbf{w}\|^2 \quad \textbf{(minimize)}.} \tag{5}$$

The problem of wanting to *optimize* Eq. 5, with *constraint* Eq. 4, lends it self to the use of **Lagrange multipliers**. Therefore, our *objective function* minus our constraints, yields our **Lagrangian**:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_i \alpha_i[y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1], \tag{6}$$

where α_i are the *Lagrange multipliers*. Wishing to minimize $\mathbf{w}$ and maximize b, we take the partial derivatives of the Lagrangian with respect to each variable and set them equal to zero. This yields

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\mathbf{w}} &= \mathbf{w} - \sum_i \alpha_i y_i x_i = 0\\ \mathbf{w} &= \sum_i \alpha_i y_i x_i,\end{aligned} \tag{7}$$

and,

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \alpha_i y_i = 0$$

$$\sum_i \alpha_i y_i = 0. \tag{8}$$

Note: we find in Eq. 7 that $\mathbf{w}$ is simply a *linear combination* of our training vectors.

Finally, substituting Eq. 7 back in for the Lagrangian, we find

$$\mathcal{L} = \frac{1}{2}\left(\sum_i \alpha_i y_i \mathbf{x}_i\right)\cdot\left(\sum_j \alpha_j y_j \mathbf{x}_j\right) - \left(\sum_i \alpha_i y_i \mathbf{x}_i\right)\cdot\left(\sum_j \alpha_j y_j \mathbf{x}_j\right) - \sum_i \alpha_i y_i b + \sum_i \alpha_i,$$

but, from Eq. 8, $\sum_i \alpha_i y_i b = 0$. Rewriting, we arrive at the ultimate objective function we wish to maximize,

$$\boxed{\mathcal{L} = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \ \ (\textbf{maximize})} \tag{9}$$

subject to the linear constraint of Eq. 8.

This is known as the **dual problem**. Since the dual maximization problem is quadratic in α_i, **quadratic programming** (**QP**) algorithms can be used to efficiently solve it. Furthermore, numerical analysis has since shown (although not proven here) that the Lagrangian, or dual problem, of Eq. 9 in fact represents a **convex surface**, meaning there is always one single minimum. This is one advantage over *neural networks* which can suffer from the problem of multiple local minima.

Substituting Eq. 7 back into Eq. 2, we update our decision rule, or **Support Vector Machine** (**SVM**), as

$$\boxed{\sum_i \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{u}) + b \geq 0 \quad (\textbf{SVM}).} \tag{10}$$

It is important to note, looking back at our optimization function (Eq. 9) and constraint equation (4), that the problem of SVMs simply boils down to **inner products** between pairs of the data. Therefore, this simplest version of the SVM can be *extended* in order to better classify more complex data.

For example, suppose a dataset is *not linearly separable* in its current dimensions, i.e., a *linear* hyperplane cannot be constructed in such a way as to completely separate the two classes. However, a **kernel function** can be used to *linearly transform* the data to a *higher-dimensional*, "implicit" feature-space where it may in fact be linearly separable—an approach referred to as a "kernel trick;" in Eq. 9, replace the dot-product $\mathbf{x}_i \cdot \mathbf{x}_j$ with the kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. Since *only* inner products need to be computed for SVMs, there is no need to actually calculate the coordinates of the data in that higher-dimensional space, which means the kernel trick comes at no extra computational cost.

Another technique to extend SVMs is the **soft-margin.** The preceding discussion assumed linear separability, and therefore **hard margins** could be used. However, in the case of data that are not linearly separable, a *hinge loss* function can be introduced, along with a new parameter—typically λ, which modifies $\|\mathbf{w}\|^2$—that determines the trade-off between increasing the margin size and ensuring any $\mathbf{x}_i$ falls on the correct side of the margin. For sufficiently small values of λ, the loss function will behave just like the hard-margin SVM.

III. METHODS

Similar to the discussion in the previous section (§II), for this particular study, the feature-space of the datasets will be restricted to two dimensions. Also, the binary-classified training data will be linearly separable to facilitate a hard-margin SVM. Written in Python, our SVM will depend only on Numpy's library for training and prediction and Matplotlib for visualization:

```
import numpy as np
import matplotlib.pyplot as plt
```

Following object-oriented design, we begin building our Support Vector Machine *class*, starting with its *constructor* method:

```
class Support_Vector_Machine:
    def __init__(self, visualization=True):
        self.visualization = visualization
        self.colors = {1:'k', -1:'r'}
        self.markers = {1: '+', -1:'_'}
        if self.visualization:
            self.fig = plt.figure()
            self.ax = self.fig.add_subplot(1,1,1)
```

Next, we define our SVM's relatively simple "predict" method, which will predict the classification for a new, unknown sample. As we saw with our original *decision rule* (§II, Eq. 2), to classify some unknown vector $\mathbf{x}_i$, we simply need to determine the sign of its projection onto $\mathbf{w}$, plus some bias, or

$$\text{sgn}(\mathbf{x}_i \cdot \mathbf{w} + b). \tag{11}$$

Thus,

```
    def predict(self, features):
        # sign(x_i.w + b)
```

```
        classification = np.sign(np.dot(np.array(features),
            self.w) + self.b)
        if classification != 0 and self.visualization:
            self.ax.scatter(features[0], features[1], s=200,
                marker='*', c=self.colors[classification])
        return classification
```

Next, we begin defining our "fit" method which contains all the training logic and optimization algorithms. In addition to passing in our training data, we declare a dictionary to store any values of $\mathbf{w}$ and b, satisfying our constraint Eq. 4, that we come accross, with $\|\mathbf{w}\|$ as their *keys*. We also declare a set of 2-D transformation vectors that we will later use to check the possible "directions" of our test vector $\mathbf{w}$.

```
    def fit(self, data, b_range=5, step_mag=3):
        self.data = data
        # { ||w||: [w,b] }
        opt_dict = {}

        transforms = [[1,1],
                      [-1,1],
                      [-1,-1],
                      [1,-1]]
```

Next, we choose reasonable bounds for our constraint-based optimization problem based on the training data, by identifying the maximum and minimum feature-set values:

```
        all_data = []
        for y_i in self.data:
            for featureset in self.data[y_i]:
                for feature in featureset:
                    all_data.append(feature)
        self.max_feature_value = max(all_data)
        self.min_feature_value = min(all_data)
        all_data = None # free up memory
```

With these bounds in hand, we can define some optimization step sizes:

```
        step_sizes = [self.max_feature_value * 0.1,
                      self.max_feature_value * 0.01,
                      self.max_feature_value * 0.001,
                      self.max_feature_value * 0.0001,]
```

The idea is to descend our *convex* space (Eq. 5 with constraint Eq. 4) to find its single, global minimum, by first taking larger, coarse steps, until we "overstep" the minimum. We then go "back and forth" around the minimum with smaller and smaller steps to determine a more precise and accurate value for the minimum. Of course, fine tuning our best found "minimum," using step sizes of decreasing magnitude, significantly adds to the computational cost.

In addition to finding $\mathbf{w}$, we also need to determine the bias b. As the precision in determining $\mathbf{w}$ will turn out to be more important for the SVM's accuracy than b (as we will later see), we can choose a larger step size:

```
        b_range_multiple = b_range # default is 5
        b_multiple = 5
        latest_optimum = self.max_feature_value*10
```

Although the default value for multiplier for the range of b is 5, it can be specified as a parameter when calling SVM's "fit" method. We can now begin our stepping process:

```
        for step in step_sizes[:step_mag]:
            w = np.array([latest_optimum, latest_optimum])
            optimized = False # for a convex surface
            while not optimized:
```

Again, since the problem-space is a convex surface, we can reset our optimization condition for each subsequent step size, knowing there is only one minimum to be found. The condition will be set true once we complete searching with our smallest step size.

Continuing from the start of the "while loop," we begin iterating through possible values of b. For simplicity, we use a fixed step size, as the accuracy here is less important and will produce similar results.

```
while not optimized:
    for b in np.arange(-1*(self.max_feature_value\
        *b_range_multiple),
        self.max_feature_value*b_range_multiple,
        step*b_multiple):
```

Continuing next with our search for $\mathbf{w}$, nested inside b's for-loop above:

```
            for transformation in transforms:
                w_t = w*transformation
                found_option = True
                # check y_i(x_i.w+b) >= 1
                for y_i in self.data:
                    for x_i in self.data[y_i]:
                        if not y_i*(np.dot(w_t, x_i)+b) >= 1:
                            found_option = False
                            break
                    if not found_option:
                        break
                if found_option:
                    opt_dict[np.linalg.norm(w_t)] = [w_t, b]
```

We check each transformation of our test $\mathbf{w}$ against the constraint equation (Eq. 4). As soon as either test $\mathbf{w}$ or b fails to satisfy the constraint for any one of the training set members we break out of both loops to avoid wasted computation. On the other hand, if a satisfactory $\mathbf{w}$ and b are found, we store them in our dictionary for later.

Finishing our optimization algorithm, outside of the b and nested transformation for-loops:

```
    if w[0] < 0:
        optimized = True
        print('Optimized a step.')
    else:
        w = w - step

norms = sorted([n for n in opt_dict])
#||w|| : [w,b]
opt_choice = opt_dict[norms[0]]
self.w = opt_choice[0]
self.b = opt_choice[1]
latest_optimum = opt_choice[0][0]+step*2
```

If we overstep the minimum, we move on to the next smaller step size; otherwise, we continue to descend the convex plane. After exhausting the possibilities, we sort the dictionary keys by $\|\mathbf{w}\|$, as this is our minimization objective (recall Eq. 5). Taking the first item in the sorted dictionary—the minimum value found for $\|\mathbf{w}\|$—we set our SVM's trained $\mathbf{w}$ and b values and repeat the process if any smaller step sizes remain.

The complete code, including the SVM class' "visualize" method, and along with the proceeding results and analysis, can all be found in a Jupyter Notebook on the public GitHub repository at the following URL: https://github.com/sabaronett/2017-cmp/blob/master/SVM.ipynb.

IV. RESULTS

Using the default parameters (i.e., smallest step size for finding $\mathbf{w}$ down to 0.01% of largest training feature, and a b range multiple of 5), we trained our SVM with the following, 6-instance dataset, stored in a Python dictionary:

```
data_dict = {-1:np.array([[1,8],
                          [2,9],
                          [4,8]]),
              1:np.array([[4,1],
                          [9,2],
                          [7,4]])}
```

A visual result of the training is shown in Fig. 3. Qualitatively, the SVM found the best hyperplane boundary with the widest margins fairly well: the does not appear to be any "better" direction for the hyperplane that would give it a wider margin, and the margins themselves seem to reach their support vectors. In particular, we find there to be a single negative SV, at $[4,8]$, and two positive SVs at $[4,1]$ and $[7,4]$.

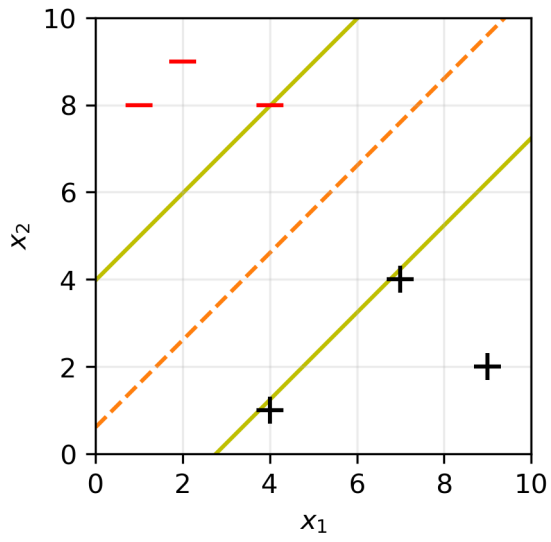

Figure 3: Training results, using a smallest step size of 0.01% and 5× b range multiplier.

We can evaluate each training data point $\mathbf{x}_i$ according to

$$\mathbf{w} \cdot \mathbf{x}_i + b \tag{12}$$

with the best $\mathbf{w}$ and b found from the training, to give us a quantitative assessment of the SVM's training. Recall that, ideally, we expect our SVs to exactly evaluate to unity. The results of these evaluations are shown in Table I. As we can see, given the default optimization parameters chosen, the SVs are reasonably close to 1—at most off by 7%–with the negative SV at $[4,8]$ less than one percent off. We can use the largest error of the SVs as an indicator of the training's success at finding the true minimum for $\|\mathbf{w}\|$ and a maximum for b.

Table I: Evaluations of Eq. 12 with training data and found values for $\mathbf{w}$ and b.

$[x_1\, x_2]$	$\mathbf{w} \cdot \mathbf{x}_i + b$
[1 8]	1.899
[2 9]	1.899
[4 8]	1.008
[4 1]	1.071
[9 2]	2.259
[7 4]	1.071

Next, we tested our SVM's prediction capabilities, asking it to classify the following data points:

```
predict = [[0.5,1],
```

```
        [1,3],
        [3,4],
        [3,5],
        [4.5,5],
        [5,6],
        [6,5],
        [5,8]]
```

The results of our SVM's predictions are shown in Fig. 4

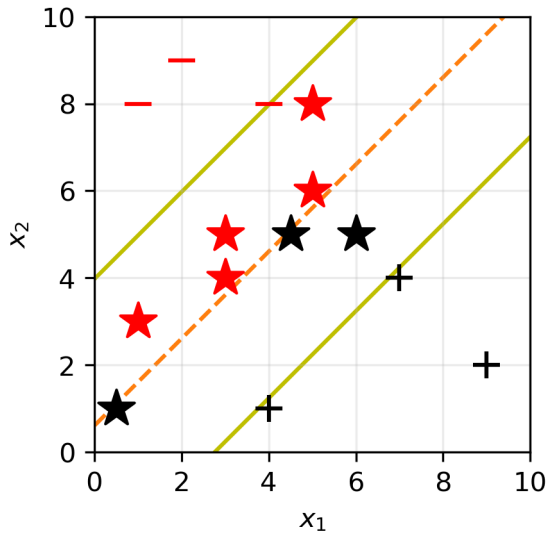

Figure 4: Results of the SVM's predictions. Newly classified points are marked with a star and colored accordingly.

As expected, the SVM is able to classify any new points based on the SVM's training model. Since the SVM's "predict" method simply calculates the sign of the evaluation of Eq. 12 (see Eq. 11), we found the computational time to predict any new point to take *less than a millisecond* (644 ± 11.6 µs). This shows that, once trained, an SVM is very efficient in its predictions.

V. ANALYSIS

A. Optimizing b Range

As we saw in developing the SVM code (§III), there is some room for optimization—in terms of both performance and accuracy of the SVM—by adjusting some of the parameters. Using the results of the previous section as a reference, we can first make adjustments to the range multiplier for b values we search and test, and see how this affects the performance of training the SVM. Keeping the smallest step size for finding $\mathbf{w}$ fixed to the default 0.01%, we tried values for the b-range multiplier from 1 to 5 (the latter being the default). Using Python's "%timeit" function, the average results and standard deviations for seven runs each are shown in Table II.

Table II: Time performance for various b-range multipliers.

b-Range Multiplier	Avg. Time (s)
1	0.58 ± 0.0093
2	1.18 ± 0.035
3	1.79 ± 0.0555
4	2.37 ± 0.0415
5	3.03 ± 0.0503

As expected, larger values for the b-range multiplier increased computation time, as there was a larger range of b values to be tested, with a multiplier of 5 incurring, on average, six times the cost of that of 1. Surprisingly, however, we found that these different values tested yielded *no observable differences* in the accuracy of the trained SVM: the qualitative graphs of the decision boundary hyperplane and margins were identical, and the values and errors of the support vector evaluations were the same as in §IV (at least to three significant figures).

This analysis suggests—as we also expected earlier—that optimizing b is of lesser importance than finding a good minimum for $\|\mathbf{w}\|$ in the SVM. Moving forward, we can continue to use a range multiplier of 1 and expect comparable results with increased performance.

B. Optimizing $\|\mathbf{w}\|$ Step Size

With our value for the b-range multiplier optimized, we can proceed in analyzing how adjusting the step size with which we search for a minimum in $\|\mathbf{w}\|$ affects our SVM. Recall from §III, that our algorithm descends the convex space defined by the dual problem to find its single minimum—initially with coarse steps, and later with finer ones. If we overstep the minimum, we reverse course and more slowly descend once again.

When calling our fit function, the "step-mag" parameter sets how small of a step size our SVM is willing to go in finding a minimum $\|\mathbf{w}\|$, as a percentage of the largest feature found in the training set; a parameter set to 1 stops at 0.1, and 4 stops at 0.0001. We are most interested in assessing the computational performance, and how well they find a minimum for $\|\mathbf{w}\|$, when adjusting this parameter. First, we will qualitatively compare the differences with graphs of each determined decision boundary hyperplane and against the same prediction dataset shown in Fig. 5.

As expected, stopping sooner and at the coarser step sizes yields poorer results, in terms of minimizing $\|\mathbf{w}\|$ which corresponds to maximizing the width of the margins; graph (a) has the narrowest margins, while (d) has the widest. We can also see in (a) that the margins are nowhere close to the positive and negative "SVs." In (b), the positive SVs fair better, but the negative margin is off. And, looking closely at the remaining two, we see

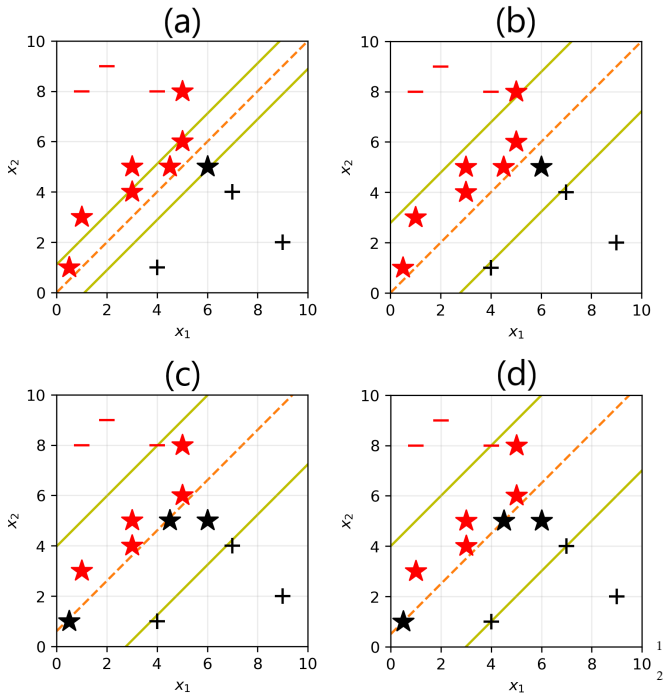

Figure 5: Results of the SVM's predictions for different training models where the smallest step size used to minimize $\|\mathbf{w}\|$ was varied. (a), (b), (c), and (d) correspond to stopping at 0.1, 0.01, 0.001, and 0.0001 of the largest training feature respectively.

the margin lines passing closer through the center of the SV markers for (d) than (c).

Most importantly, notice how the position of the separating boundary hyperplane shifts and affects the predictions. In (c) and (d), the two points at [4.5, 5] and [0.5, 1] are properly classified as positive samples, just as we saw in the results in §IV. However, because of the rougher minimization, these same two points are instead classified as negative samples in (a) and (b). Quantitatively, we present the average time performance results, as well as evaluations of Eq. 12 for both positive and negative SVs, in Table III.

Table III: Time performances and ±SV evaluations (Eq. 12) for various step sizes in minimizing $\|\mathbf{w}\|$.

Smallest Step	Graph	Avg. Time (s)	-SV	+SV
0.1	(a)	0.013 ± 0.0001	3.600	2.700
0.01	(b)	0.051 ± 0.001	1.440	1.080
0.001	(c)	0.631 ± 0.025	1.008	1.071
0.0001	(d)	33.3 ± 0.751	1.001	1.003

As we assessed with their respective graphs, stopping at a step size of 0.1 or 0.01, although computationally efficient, produces inaccurate boundary results that produce wrong predictions. Meanwhile, stopping at either 0.001 or 0.0001 gives more consistent results, with the SVs deviating less than 10% from their ideal values (less than 1% for the 0.0001 case). However, going all the way down to a step size of 0.0001 takes *over 50 times longer* than stopping at 0.001, even though they both fair the same with our prediction set. Given their relative cost-benefit, we believe stopping at 0.001 is sufficient and gives the best mix of accuracy and performance when training our SVM.

C. Scikit-learn Comparison

To compare the performance of our algorithm against a real-world, production-level SVM, we implemented Scikit-learn's C-Support Vector Classification, using a linear kernel as follows:

```
from sklearn import svm
clf = svm.SVC(kernel='linear')
```

Feeding it the same test data, we found their SVM to take, on average, 214 ± 4.36 µs (7 runs of 1000 loops each) to train the data; this is *60 times* faster than even our fastest training time—corresponding to the least accurate model.

With Scikit-learn's linear SVM trained, we also invoked it to predict the same data as before. The results were the same as our two most "accurate" SMVs, of smallest step sizes 0.001, graph (c), and 0.0001, graph (d), with the exception of the point at [3, 4]. Based on Scikit-learn's training, the model classified [3, 4] as belonging to the *positive class*, instead of negative, with all else being the same. Therefore, it appears its corresponding decision boundary hyperplane is slightly skewed from our most precise SVM.

We also found its "support vector" attribute returned only *two* SVs: the negative sample at [4, 8] and positive at [7, 4]. Unlike ours, their trained model did not consider the positive sample at [4, 1] to be classified as a support vector. If this is the case, their resulting decision boundary may have a somewhat shallower "slope"—pivoted around the two SVs—than ours, which would account for predicting [3, 4] instead as a positive sample (see graphs (c) or (d) in Fig. 5).

VI. INTERPRETATION AND CONCLUSIONS

We readily acknowledge the limitation of our SVM as to the kinda of data it can train from and predict, as well as its computational performance. In its current form, it can—at best—train from a *linearly separable* dataset, with a 2-dimensional feature-space, but can reasonably classify new and unknown samples.

Several steps could be taken to expand the code in terms of versatility and performance. For starters, because of the way we attempt to rudimentarily solve the QP problem by transforming and testing the four possible transformations of **w**, the code is currently restricted to 2-dimensional feature-spaces. As a 3-dimensional feature space would require $2^3 = 8$ exhaustive transformations, and so on for higher dimensions, we should first rewrite this portion and verify it can train with any n-dimensional feature-space data, so long as it is linearly separable.

The next important improvement would be to implement a *soft-margin* calculation by introducing a *hinge loss* function. As mentioned in §II, this would allow the SVM the flexibility to handle data that is *not* linearly separable (which is often the case with real datasets), by introducing a trade-off between margin-width and proper boundary separation (hyperplane).

Finally, the ultimate goal would be to incorporate *kernel functions*. A "kernel trick" would substantially extend our SVMs applicability to a variety of datasets—even if it is non-linearly separable in its proper n-dimensional feature-space—by *transforming* the data to higher feature-spaces where linear separability may arise. Properly doing so would also add very little computational cost to the model.

However, our QP problem solver, in its current and basic form, would likely not be extendable to support and and reasonably solve the QP problems involving soft-margins and kernel functions. At that point, the most logical next step would be to invoke a third-party, QP library to help solve these problems. One example is **CVXOP**, which can be used to solve convex quantum programming or Lagrange dual problems. Another popular and robust library is **LIBSVM**, which is in fact what Scikit-learn uses.

General limitations of SVM machines can include a large memory footprint for extremely large datasets. *Sequential minimal optimization* (SMO), developed in 1998 at Microsoft Research, attempts to address this by iteratively reducing the QP problem into smaller subproblems based on sub-sets of the training data; SMO has been implemented in LIBSVM. Another disadvantage to SVMs is that they do not directly provide probability estimates, and doing so requires an expensive five-fold cross-validation. Finally, SVMs suffer from the "curse of dimensionality," in that, by resorting to higher dimensions with the kernel functions, even seemingly *dense* data can become relatively *sparse*, making training more difficult.

VII. ACKNOWLEDGEMENTS

Much of the theory's progression in §II follows Patrick H. Winston's *MIT OpenCourseWare* "Lecture 16: Learning: Support Vector Machines," as part of his series 6.034 on Artificial Intelligence (2010).[1]. As for the methodology, §III, most of the techniques, structures and algorithms originate from *Python Programming Tutorial*, "Beginning SVM from Scratch in Python."[2]

[1] Https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-16-learning-support-vector-machines/.

[2] Https://pythonprogramming.net/svm-in-python-machine-learning-tutorial/?completed=/svm-constraint-optimization-machine-learning-tutorial/.