Hard SVM, Kernels, and Feature Spaces Report

Christian Eid

December 2021

1. Linearly Separable Data

The goal of these machine learning algorithms is to correctly classify data. The data we want to classify consists of a vector with two values

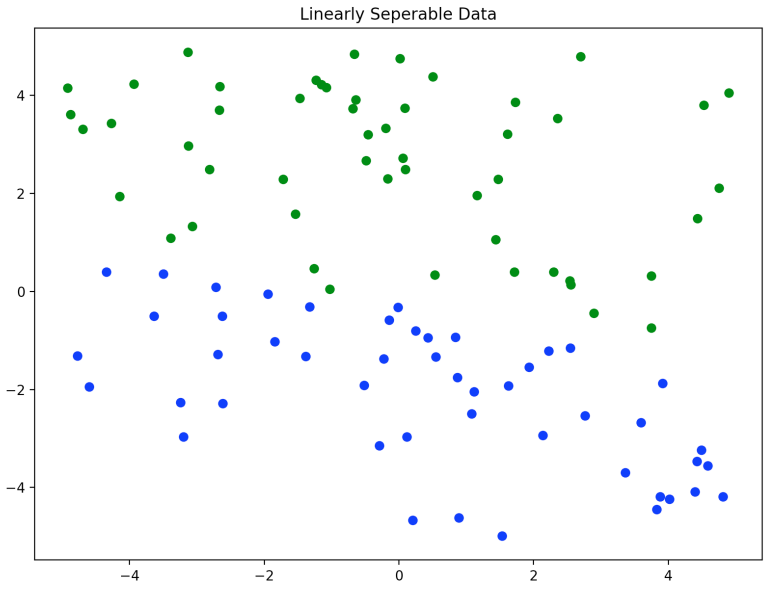
$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We can generate a random weight vector

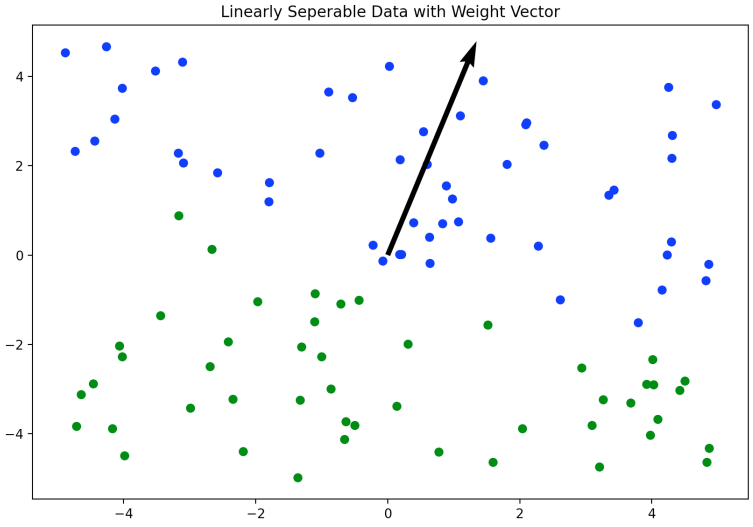
$$w = \begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}$$

By adding a 1 in x such that $x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$, we can do the dot product of x and w. If that dot product is greater than 0, then we classify the data as a 1. If it is less than 0, we classify it as a 0. (Or as -1, depending on the algorithm we use to learn.) The label is y. So, if $x^T w > 0$ then $y = 1$ By doing this for every training data, we have created a set of linearly separable data. The goal of our algorithms is to learn a weight vector w that correctly classifies the data.

If we were to plot the x values using x_1 and x_2 as our axes, and color the point depending on the weight vector classification, we would obtain a graph that looks like the the following.



If we were to plot the weight vector that correctly classifies the data, it would show us that the line perpendicular to that weight vector will classify the data.



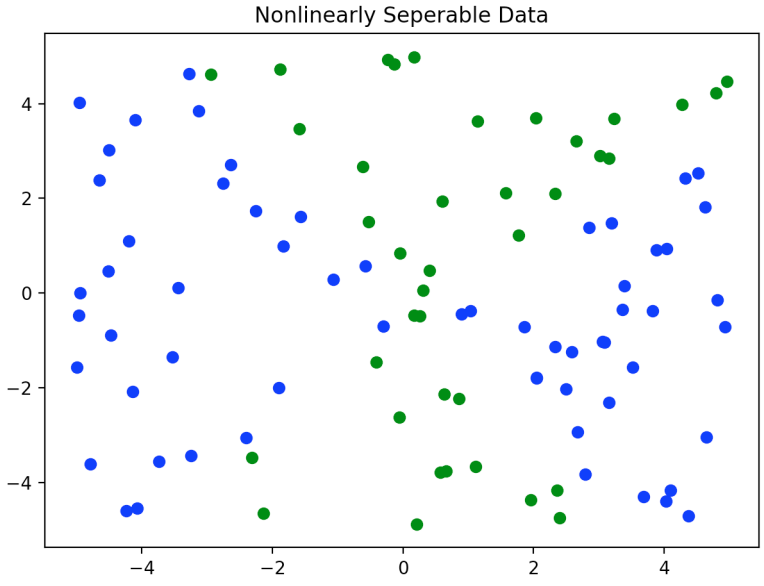
2. Non-linearly Separable Data

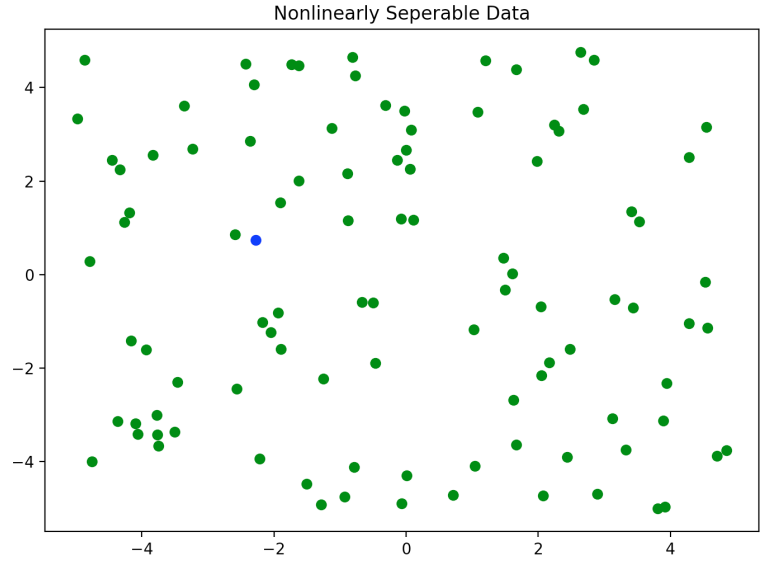
We can map each instance to a feature space of degree 2, and use it to generate a set of non-linearly separable training data. We map each value of x to a feature space such that

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

We can then use a longer weight vector $w = \begin{bmatrix} b \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix}$ to classify the data such that $y = 1$ when $x^T w > 0$ and 0 otherwise. This creates a set of data that is non linearly separable. Shown below are two examples of a non-linearly separable

dataset.





Note that these two could not be separated by a line, but could be separated by a polynomial degree 2 weight vector that describes a hyperbola and ellipse.

The goal of our new algorithms is to find that weight vector. They will map the instances to a feature space, and use that feature space to create a higher dimensional set of data that is linearly separable. We can then use our linear classification techniques on that feature space to generate an appropriate weight vector.

2. Loss Functions

We can measure the success of our algorithm's weight vector by using a loss function, which uses the predicted value of y and the actual value to calculate a loss. The negative gradient of the loss function tells us how to change our weight vector to better classify the data, since the negative gradient points to the lowest loss.

In this program, I use the hinge loss for the Soft-SVM gradient descent algorithm.

3. Algorithms

The Soft-SVM algorithm is a gradient descent algorithm that uses the hinge loss. It generates a random weight vector, and uses the negative gradient of a hinge loss to gradually change the value of the weight vector until the loss is minimized. In addition to gradient descent, I use the perceptron learning algorithm, which uses misclassified data to learn a weight vector until all data is correctly classified. I also use linear programming to learn the data, and quadratic programming to run Hard-SVM on the data. Each of these algorithms actually uses the feature space to classify the data. The Soft-SVM algorithm has a step-size of 10 for all tests in the program. It runs significantly faster with this. It also important to note that the regularization term is very low, as it would not succeed in classifying the data if the regularization was close to 1.

4. Kernels

Kernels allow us to learn to classify nonlinear data without mapping to a feature space. This allows us to learn complex non-linear weight vectors that would impossible using feature mapping. Feature mapping gets exponentially more costly when the feature space is higher dimensional. Kernels allow us to bypass feature mapping and use the original training set to classify data. Instead of using the feature space instances, it uses a special kernel function and computes the inner product between pairs of data. In this program, two kernel functions are used: the polynomial and Gaussian kernel functions. These functions are used to kernelize the Soft-SVM algorithm. These two algorithms are compared to the Hard-SVM which uses feature-mapping.

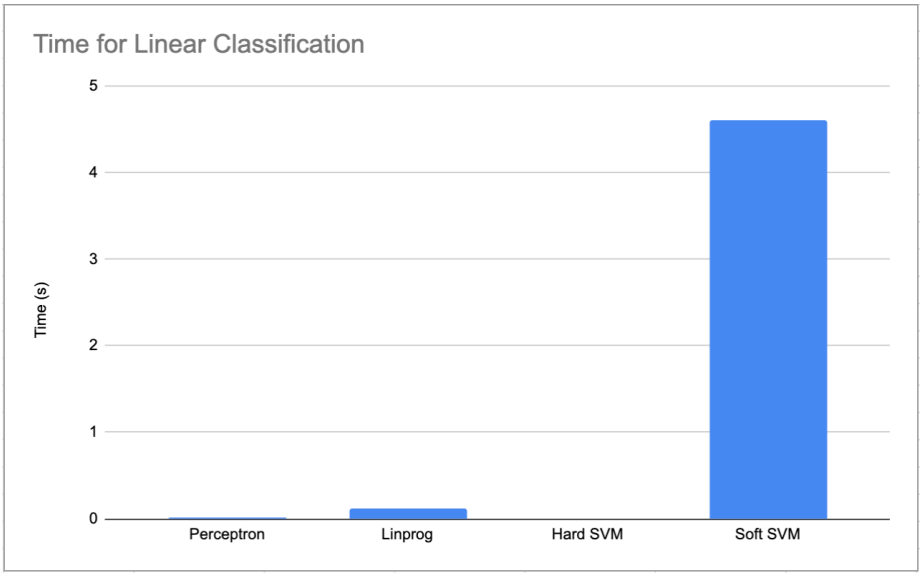
5. Data

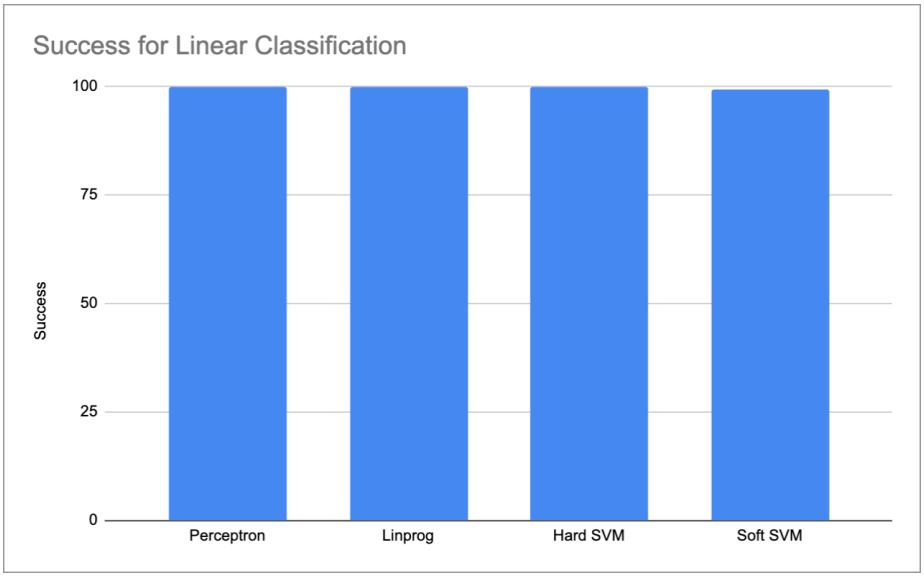
Test 1: Linear Classification

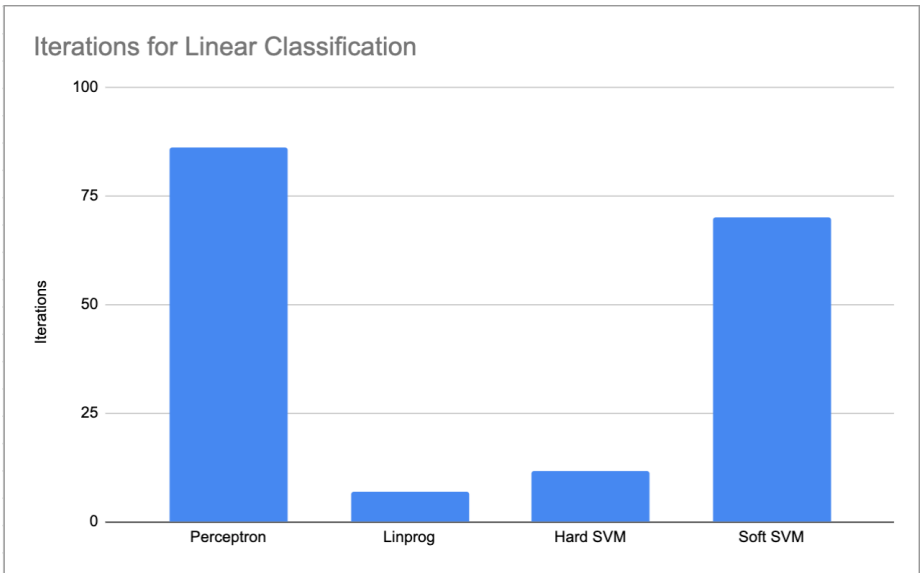
In this test, I ran the perceptron learning algorithm, linear programming, Hard-SVM and Soft-SVM algorithms on a training set of linearly separable data. 100 training samples were generated, along with a random weight vector, which created the label set. After repeating this process 100 times, the average time until completion, success rate, and iteration number were collected.

Algorithm	Time (s)	Success Rate (%)	Iterations
Perceptron	0.02216841385	100	86.15
Linprog	0.1272530477	100	6.8
Hard-SVM	0.00415536785000144	100	11.65
Soft-SVM	4.611070635	99.3	70.15

Note that for the Soft-SVM, the number of iterations represents the average number of weight updates needed to get to a 100% success rate. As the data shows, the perceptron, linprog, and Hard-SVM algorithms take very little time. The Hard-SVM takes the least amount of time. These three algorithms always guarantee a 100% success rate. Soft-SVM, being a gradient descent algorithm, takes a lot of time in calculating the gradient, and must go through many iterations to finally achieve a 100% success rate. It does not guarantee that, however. Shown below is the side by side comparison of time, success rate, and iterations numbers.







Note that the perceptron learning algorithm updates the weights an average of 86.15 times, but goes loops through the training instances an average of 2159.95 times. In this first test, Perceptron, Linprog, and the Hard-SVM seem to be more efficient and accurate than the soft-SVM.

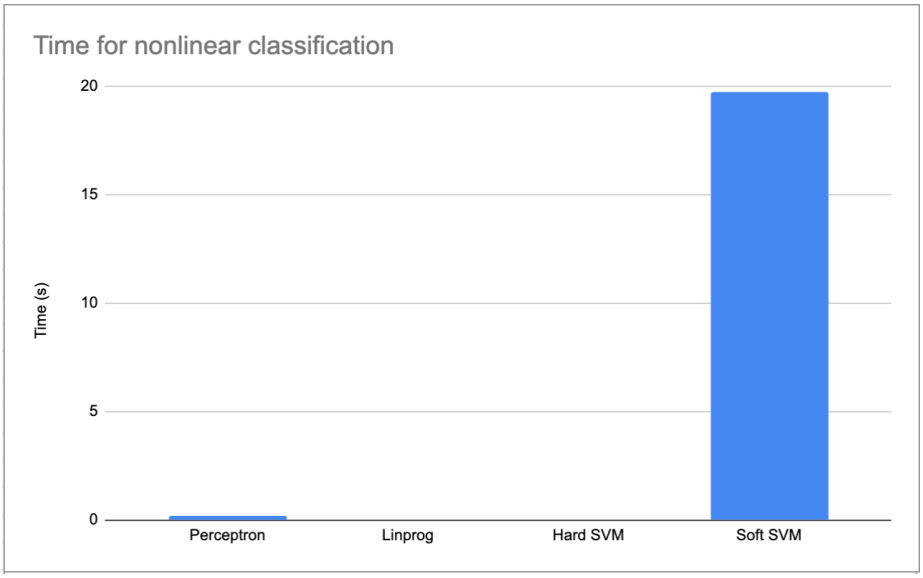
Test 2: Nonlinear Classification

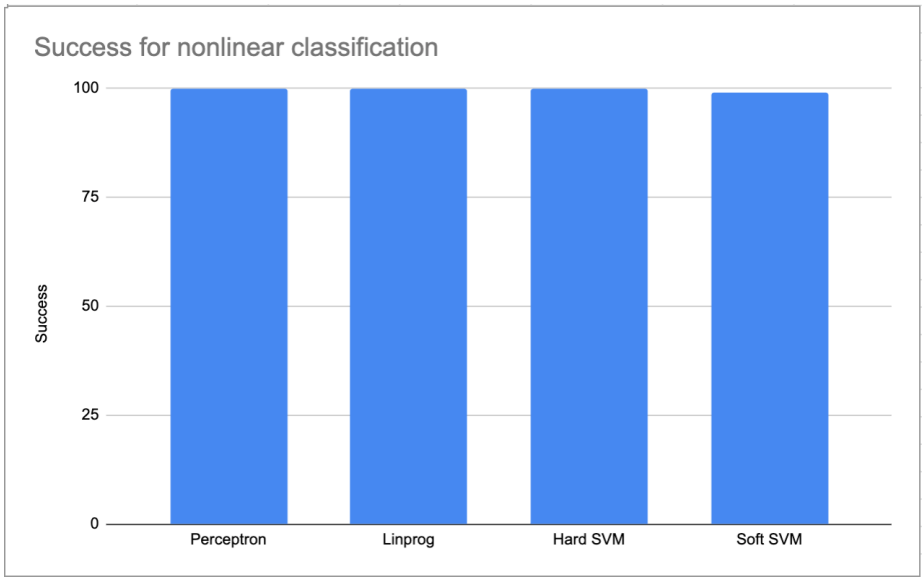
In this test, the 100 instances were mapped to feature spaces. A random weight vector that fits the feature space was created, and used to generate non-linearly

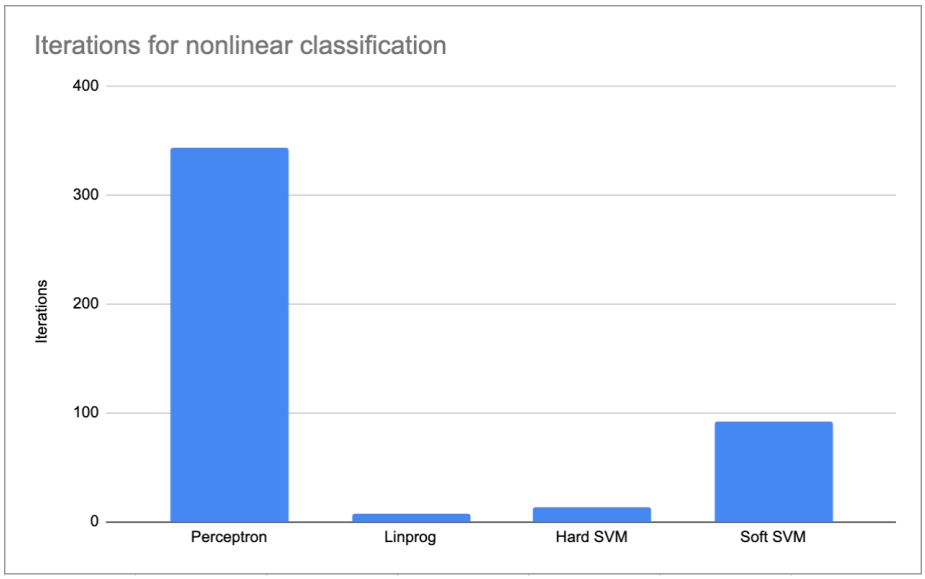
separable labels to the training data. The perceptron, linprog, Hard-SVM and Soft-SVM were given the feature mapping and had to learn this longer weight vector. This process was repeated 100 times. The data can be seen below.

Algorithm	Time (s)	Success Rate (%)	Iterations
Perceptron	0.1584556503	100	342.92
Linprog	0.02712849176	100	7.2
Hard-SVM	0.00492045244	100	13.2
Soft-SVM	19.71722374	98.92	92.08

In similar fashion to the linear classification test, the Perceptron, Linprog, and Hard-SVM take very little time compared to the Soft-SVM. They all get to 100% success rate, whereas Soft-SVM averages to around 99%, which is still very high. In comparison to linear classification, the number of weight updates is higher. The perceptron increases its weight update by almost 400%, and loops through the training set 2159.95 times. The Soft-SVM does not increase iteration count as significantly, but still does increase a bit. The linprog and Hard-SVM barely increase the iteration count. Surprisingly, the linprog tests for the nonlinear classification took, on average, less time than the linear classification. Shown below are the side by side comparisons.







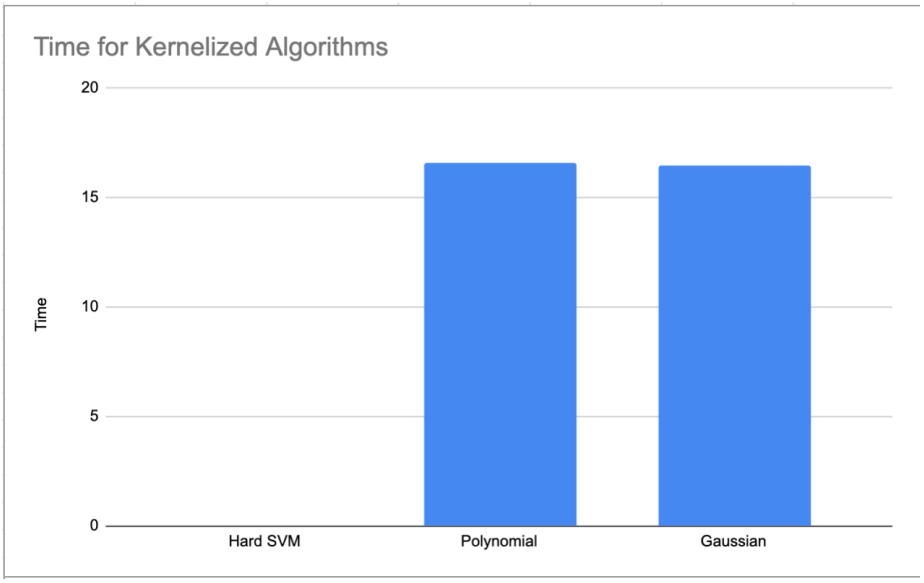
Test 3: Kernelized Algorithms

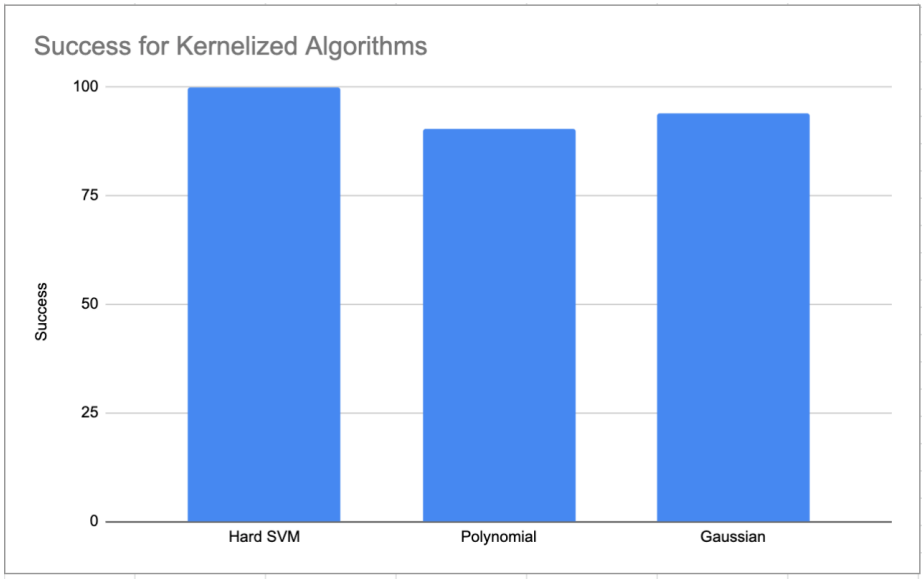
The last test uses the Kernelized Soft-SVM gradient descent algorithm. The gradient descent algorithm is kernelized through the loss function, which sums kernel functions on instance pairs. The polynomial and gaussian kernel are used in Soft-SVM, and compared to the Hard-SVM. The Kernel algorithms do not use the feature mapping, and only rely on the original training instances. However, the produce an alpha vector of same length as the number of training

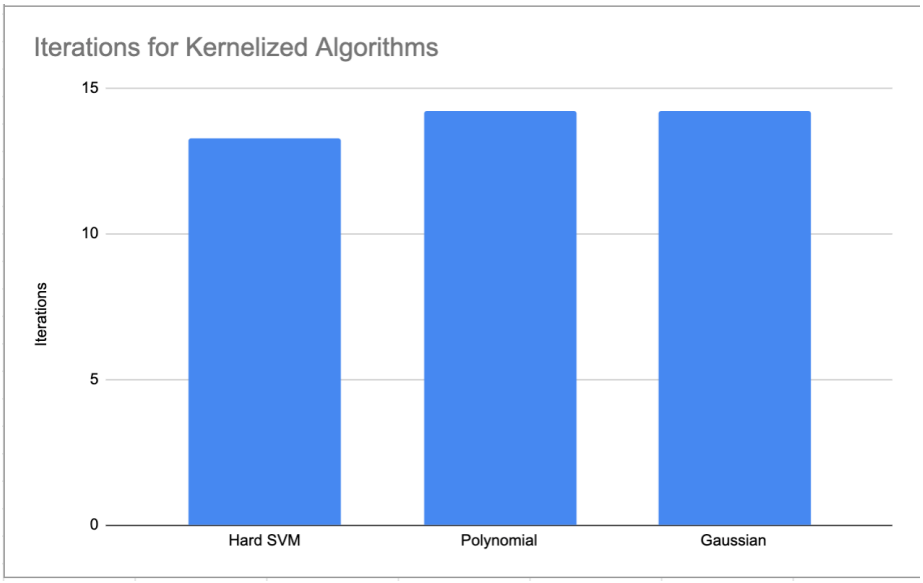
instances. To classify a new instance, we must use all the instances that were used to create the alpha weight vector. The time, success rate, and iteration count were taken for the algorithms. The gradient descents ran for 15 iterations or until at 100% accuracy, at which point they stopped. They also ran with a step-size of 10, which really sped up the algorithms. So, the data shows the average time and iterations taken to reach the average success rate.

Algorithm	Time (s)	Success Rate (%)	Iterations
Hard-SVM	0.00336976676	100	13.28
Polynomial Soft-SVM	16.57493335	90.36	14.24
Gaussian Soft-SVM	16.46826186	93.76	14.24

The data shows that the Hard-SVM, despite using the feature mapping, takes significantly less time than the kernelized algorithms, and results in a higher success rate. It also finishes in an average of fewer instances. The Gaussian kernelized Soft-SVM takes slightly less time than the Polynomial, and has a higher success rate. They shared the exact average iteration count.







However, compared to Soft-SVM that uses feature mapping, the Kernelized SVM algorithms take less time, and go through fewer iterations to result in a slightly lower success rate.

6. Kernelized Perceptron Learning Algorithm

The Kernel trick lets us never go into the feature space by using the Kernel function on the dot product of an instance pair. The normal perceptron learning

algorithm looks as follows:
1. $w \leftarrow 0$
2. while $y_i(x_i^T w) <= 0 \ \forall i$
    3. $w \leftarrow w + y_i x_i$

We can replace the weight vector with alpha vector, and use the Kernel and instance pairs to determine misclassified instances. Multiplying y_i by the classified instance tells us if its misclassified, after which we can update our alpha vector. We can then replace the update rule for preceptron with the classified instance. In the normal perceptron, we update the weight with the negative of the gradient, $y_i x_i$. We can kernelize the perceptron loss to get $L_s(\alpha) = 1/P \sum -y_i(\sum_k \alpha_k K(x_k, x_i))$. If we take the negative gradient, we get the kernelized perceptron algorithm.

1. $\alpha \leftarrow 0$
2. while $y_i(\sum_k \alpha_k K(x_k, x_i)) <= 0$
    3. $\alpha_i \leftarrow \alpha_i + \sum_k K(x_k, x_i)$