Support Vector Machines

Programming Assignment 6

Patrick Humphries (7097-1087-72, pvhumhr@usc.edu)

INF552 Machine Learning for Data Science (32458)

Viterbi School of Engineering

University of Southern California

Spring 2020

Abstract We live in a nonlinear world. In order for Artificial Intelligence (AI) to understand this world, it needs to be able to see and understand images. Support Vector Machines (SVM) do just that. SVM uses extensive kernel libraries and solvers for Quadratic Programming Problems (QPP). This paper documents an exercise of how to implement and use SVM. The requirements are parochial. The QPP solver CVXOPT is not the most modern. However, they are intended to guide the student to an understanding of SVM.

Table of Contents

Part 1a: Implementation

Construction

$$L(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m x_n^T x_m$$

$$\alpha_n \geq 0$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0$$

The world of Support Vector Machines and convex optimization revolves around Lagrangian data points. These data points establish the boundaries for a version of supervised learning that is more accurate than linear regression. The above equation with its two constraints is used for minimizing *loss* to find a hyperplane with maximum margins. A program written by Mathieu Blondel in 2010 was adapted for this exercise.

The terms $x_n^T x_m$ are implemented with this code.

```
# Multiply each features vector by all other feature vectors.
# This satisfies the np.dot(xi, xj) part of the equation.
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i,j] = self.kernel(X[i], X[j])
```

This is a dot product which results in an vector of every set of features multiplied by each other set of features. The looping construct is needed to retain the structure of the data. This is done because the result will be involved in another dot product.

The term $y_n y_m$ is the product of the labels. In the dataset provided, the values were either -1 or +1. It was processed with the following code.

```
# Multiply each label vector by all other label vectors
# Then multiply by the cross product of all features.
# This satisfies the np.dot(yi, yj) part of the equation.
# This results in the quadratic matrix.
P = cvxopt.matrix(np.outer(y,y) * K)
```

The only thing left to do is to use a solver for this Quadratic Programing Problem.

```
# Solve Quadratic Programming Problem.
solution = cvxopt.solvers.qp(P, q, G, h, A, b)

# Lagrangian multipliers.
a = np.ravel(solution['x'])
```

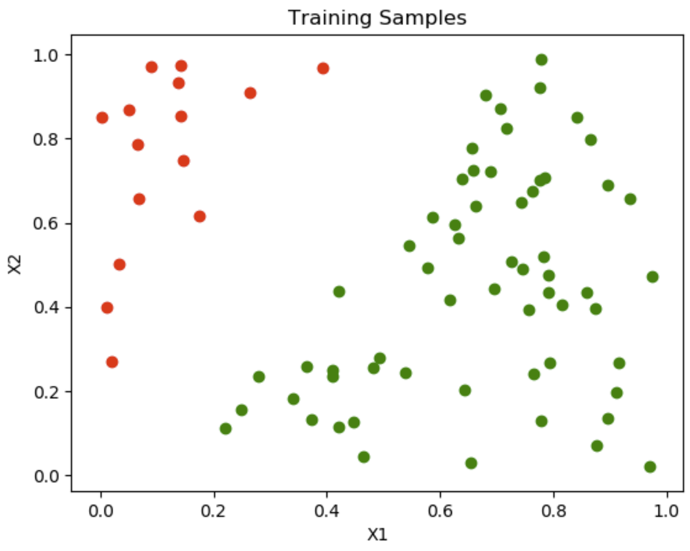
This returns an vector of datapoints. The internal nodes are those with a value of zero. The Lagrangian datapoints are those with a force value of greater than zero. This characteristic is converted into a Boolean vector, used for selecting the corresponding features and labels.

Data Visualization

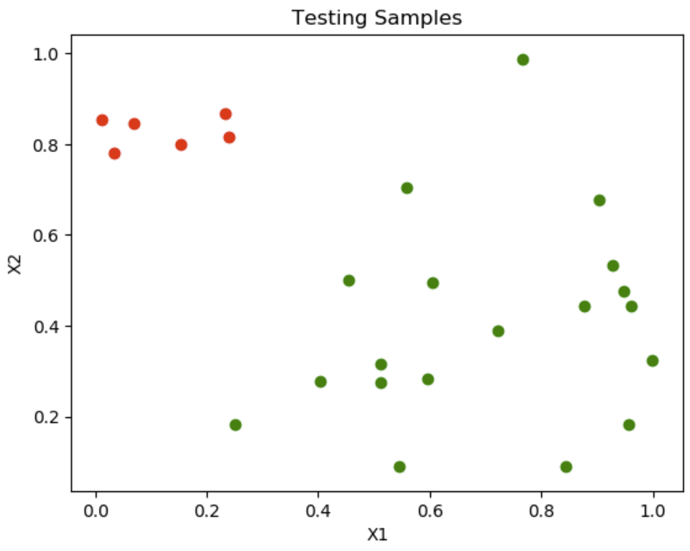
The dataset *linsep.txt* provided the linearly separated data used for training and testing. The data was shuffled and split into two sets of data. One set contains the training features and corresponding training labels. Testing features and labels are in the other set. Seventy-five percent of the data is used for training with the remaining 25 percent for testing.

The red dots are datapoints with a negative one as the label. The green are those with positive labels.

```
Load Data:
X_train.shape: (75, 2) y_train.shape: (75,)
X_test.shape: (25, 2) y_test.shape: (25,)
Plot Data: Training Samples
```



```
Plot Data: Testing Samples
```



Fitting

Fitting, also known as training produced the following output.

```
A:
[ 1.00e+00  1.00e+00  1.00e+00  1.00e+00  1.00e+00  1.00e+00  1.00e+00 ... ]

     pcost       dcost       gap    pres   dres
 0: -1.5792e+01 -3.5898e+01  2e+02  2e+01  2e+00
 1: -1.7363e+01 -3.4375e+01  7e+01  4e+00  5e-01
 2: -2.0470e+01 -3.5464e+01  3e+01  1e+00  1e-01
 3: -2.8030e+01 -3.7239e+01  1e+01  3e-01  3e-02
 4: -3.3429e+01 -3.3654e+01  4e-01  9e-03  1e-03
 5: -3.3595e+01 -3.3598e+01  4e-03  9e-05  1e-05
 6: -3.3597e+01 -3.3597e+01  4e-05  9e-07  1e-07
 7: -3.3597e+01 -3.3597e+01  4e-07  9e-09  1e-09
Optimal solution found.

 X of size 75 contained  3 Lagrangian points.
```

Scoring

```
 accuracy: 100.0 percent.

 Lagrangian Features and Labels:
[0.27872572 0.23552777]          1.0
[0.3917889  0.96675591]          -1.0
[0.02066458 0.27003158]          -1.0
Using negative Lagrangian points.

 hyperplane:  X2 = 1.877334088375481 * X1 + (-0.028248135015668363)
  slope: 1.877334088375481
  intercept: -0.028248135015668363
```

There is 100 percent accuracy. That is not surprising considering how much of the data observes *social distancing*. The support vectors are listed with their features and labels. The equation for the hyperplane is also given. Variable *X2* is described by the vertical axis, and is the dependent variable of *X1*, the independent axis and the independent variable.

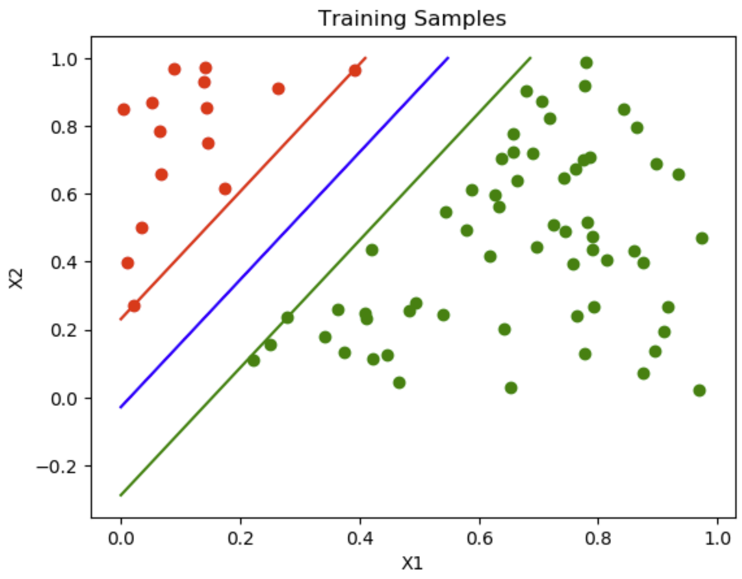
Results Visualization

Now, for the fun part. The first visualization is from the fitting. Note that red line is established by the Lagrangian data points with a non-zero value for negative labels. Positive, non-zero data points are restrained by the green margin.

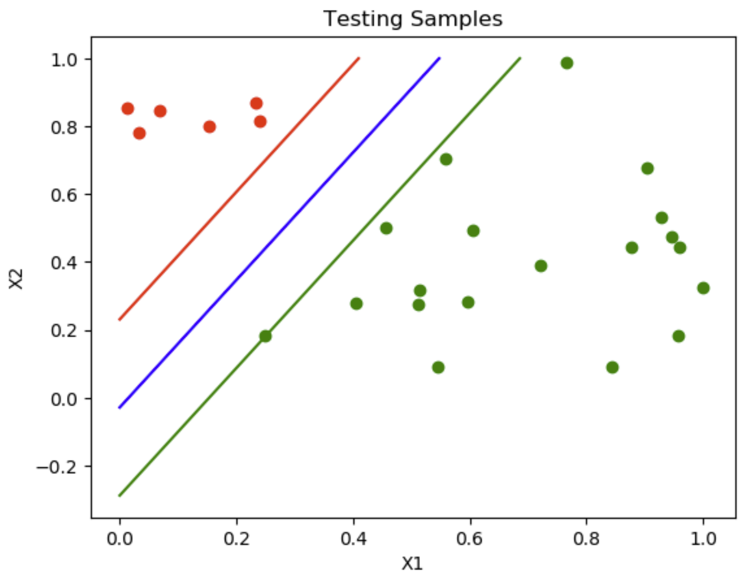
The hyperplane is the blue line.

The second visualization is with the testing data. Given the bifurcated nature of the data, the data points are well behaved.

```
Plot Data: Training Samples
```



```
Plot Data: Testing Samples
```



Part 1b: Implementation

The file *nonlinsep.txt* was provided for this part. The process begins with the visualization of the data.

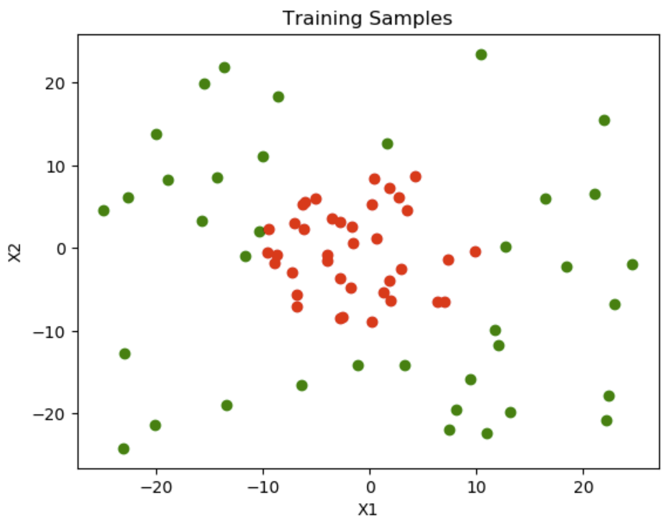
Data Visualization

Oh, my! This is a can of worms.

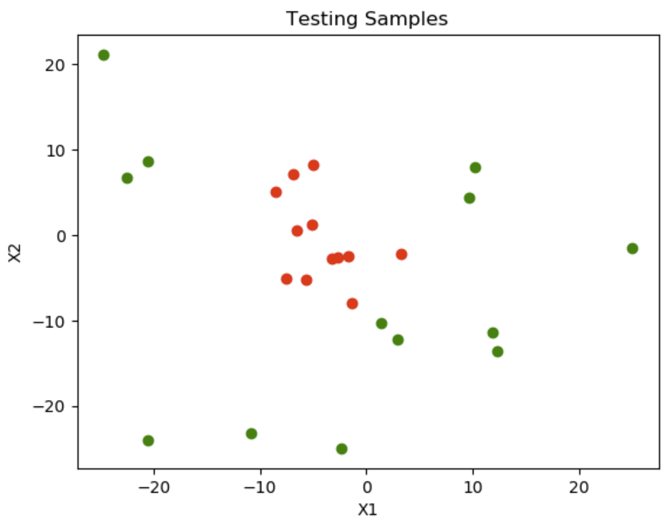
Actually, two kinds of kernels could be used for straightening this data. First, transform the data into radii and angles. The red dots would have relatively short radii. Green longer.

However, the red dots seemed to be grouped. So a Gaussian kernel is used.

```
Load Data:
X_train.shape: (75, 2) y_train.shape: (75,)
X_test.shape: (25, 2) y_test.shape: (25,)
Plot Data: Training Samples
```



```
Plot Data: Testing Samples
```



Fitting

```
A:
[ 1.00e+00 -1.00e+00 -1.00e+00  1.00e+00  1.00e+00  1.00e+00 -1.00e+00 ... ]

     pcost       dcost       gap    pres   dres
 0: -1.1672e+01 -3.8354e+01  2e+02  9e+00  2e+00
 1: -1.7965e+01 -4.9179e+01  8e+01  4e+00  9e-01
 2: -5.9681e+01 -9.3667e+01  8e+01  3e+00  7e-01
 3: -9.0320e+01 -1.1670e+02  4e+01  1e+00  3e-01
 4: -9.1721e+01 -9.7109e+01  7e+00  1e-01  3e-02
 5: -9.2573e+01 -9.3161e+01  6e-01  3e-03  8e-04
 6: -9.2811e+01 -9.2884e+01  8e-02  3e-04  8e-05
 7: -9.2851e+01 -9.2859e+01  9e-03  1e-14  5e-15
 8: -9.2856e+01 -9.2856e+01  3e-04  5e-14  6e-15
 9: -9.2856e+01 -9.2856e+01  8e-06  1e-14  7e-15
Optimal solution found.

 X of size 75 contained  27 Lagrangian points.
```

It is likely the relatively high number of Lagrangian data points is due to the boundary between the red and green dots. The reds are surrounded by greens.

```
def gaussian_kernel(x, y, sigma=5.0):
    return np.exp(-linalg.norm(x-y)**2 / (2 * (sigma**2)))
```

The Gaussian kernel makes use of numpy functions vector normalization and exponentiation. The *sigma* determines step sizes.

Scoring

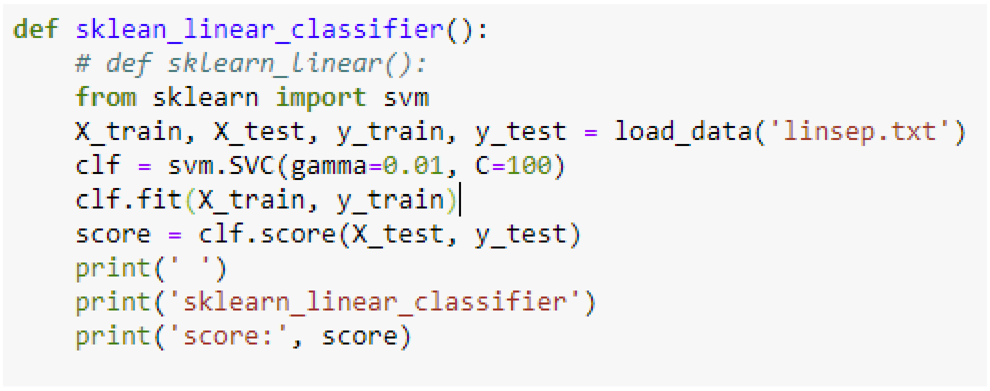
```
 accuracy: 88.0 percent.

 Lagrangian Features and Labels:
[-10.02833317  11.09354511]      1.0
[-9.46760885  2.36139525]        -1.0
[12.74780931  0.19913032]        1.0
[-19.9566285   13.84906795]      1.0
[-20.08588052 -21.4203383 ]      1.0
[-10.260969     2.07391791]      1.0
[ 10.99442024 -22.39709701]      1.0
[-23.03961851 -24.17915876]      1.0
[21.9265387  15.53137584]        1.0
[  3.28969027 -14.15854536]      1.0
[-15.50532883  19.94336178]      1.0
[-13.58465635  21.94036504]      1.0
[24.55963843 -1.9636474 ]        1.0
[ 1.66404809 12.68562818]        1.0
[ 9.90143538 -0.31483149]        -1.0
[ 22.19384568 -20.81599498]      1.0
[11.75880948 -9.85890377]        1.0
[-13.37889747 -18.96276407]      1.0
[ 22.42614347 -17.79484871]      1.0
[4.27289989 8.67079427]          -1.0
[-22.96901354 -12.67010547]      1.0
[-9.53754332 -0.51895777]        -1.0
[ 0.20162846 -8.81260121]        -1.0
[ -1.08933763 -14.10562483]      1.0
[ 6.99249663 -6.41143087]        -1.0
[10.42163247 23.45279171]        1.0
[22.88511672 -6.75299078]        1.0
Using positive Lagrangian points.

 hyperplane:  X2 = 1.4234958066672327 * X1 + (12.228069166977425)
  slope: 1.4234958066672327
  intercept: 12.228069166977425
```

The accuracy is 88 percent. Additional executions returned accuracies in the low 90's. This is expected from the visualization of the data and that there is a definite boundary. A linear hyperplane was calculated, but that is not useful with non-linear data.

Part 2: Software Familiarization

As expected, *scikit-learn* is used for software familiarization. It is full-featured with a large supporting community. An obvious tool for the data scientist.

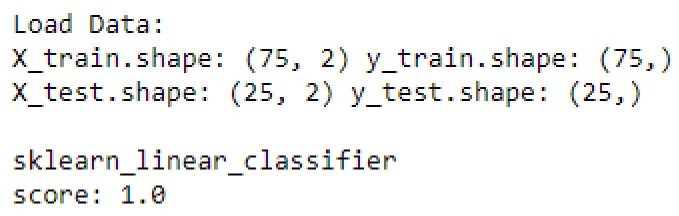
Code



```
def sklean_linear_classifier():
    # def sklearn_linear():
    from sklearn import svm
    X_train, X_test, y_train, y_test = load_data('linsep.txt')
    clf = svm.SVC(gamma=0.01, C=100)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    print(' ')
    print('sklearn_linear_classifier')
    print('score:', score)
```

Since I lack a math and algorithm background, the assignment took about eighty hours to complete. This part only took five minutes. To be fully versed in this tool, many hours of experimentation and development would be required.

Scoring



```
Load Data:
X_train.shape: (75, 2) y_train.shape: (75,)
X_test.shape: (25, 2) y_test.shape: (25,)

sklearn_linear_classifier
score: 1.0
```

As expected with the neatly separated data, the accuracy is 100 percent.

Part 3: Applications

There are two apparent strengths of SVM. One is that there are a variety of kernels that will transform non-linear data into linear data. Second, the convex solution bounded by Lagrangian datapoints can be solved as a Quadratic Programming Problem.

Reality is nonlinear. Artifacts include faces, shapes, and patterns. When these artifacts can be transposed into a grid of numerical data, then SVM can predict what the artifact is after sufficient training.

Autonomous Vehicles Tesla is emphasizing the use of visual recognition. Shapes of road signage, car shapes, and the human form are digitized and predicted using SVM [https://www.tesla.com/support/autopilot].

Facial Recognition In the Peoples Republic of China, anyone applying for mobile or Internet service must subject to a facial scan [https://qz.com/1721321/chinas-new-weapon-of-choice-is-facial-recognition-technology/]. This enables the Chinese Communist Party to control its population. With these digitized facial images, SVM is used to identify the person. This capability will soon be coming to an authoritarian despot near you.

Text Categorization *Watson* by IBM is an incredible application of Artificial Intelligence (AI) [https://www.ibm.com/blogs/watson/2017/06/why-it-matters-that-ai-is-better-than-humans-at-their-own-games/]. One demonstration application is its victory in *Jeopardy.* This AI had to infer references from millions of pages of text. In order to read these pages, text categorization was required. Once digitized, the AI could determine characters using SVM. From characters to words to works, the AI learns the references.

Handwriting Interpretation A person's handwriting is relatively unique. Using SVM to determine the author of handwritten text is used in fraud detection [https://teresadeberry.com/ai-and-machine-learning-revolutionize-forensic-handwriting-recognition-analysis/].