1 Support Vector Machine

1. Objective Function:

$$f([\alpha_i]_i) = \begin{bmatrix} 5 & 6 & 0 & -4 \\ 6 & 8 & 0 & -2 \\ 0 & 0 & 0 & 0 \\ -4 & -2 & 0 & 13 \end{bmatrix} \begin{bmatrix} \alpha_1\alpha_1 & \alpha_1\alpha_2 & \dots & \alpha_1\alpha_4 \\ \alpha_2\alpha_1 & \ddots & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_4\alpha_1 & \dots & \dots & \alpha_4\alpha_4 \end{bmatrix}$$

   Constraints:

$$(1)\alpha_0 + (1)\alpha_1 + (-1)\alpha_2 + (-1)\alpha_3 = 0$$

$$0 \leq \alpha_i \leq 100$$

2. ('name', 'error on training set', 'error on validation set') ('smallOverlap', 0.08, 0.08) ('bigOverlap', 0.28, 0.065) ('ls', 0.0, 0.00375) ('nonsep', 0.48, 0.48)

   data_bigOverlap:

   classifier: $0.395x_1 + 0.356x_2 - 0.653 > 0$

   Error rate: 6.5%

   data_smallOverlap:

   classifier: $1.21x_1 + 1.20x_2 - 2.256 > 0$

   Error rate: 8%

   data_ls:

   classifier: $-1.94x_1 + 2.58x_2 - 0.0322$

   Error rate: 0.375%

   data_nonsep:

   classifier: $-0.114x_1 - 0.511x_2 + 0.487$

   Error rate: 48%

3. Results are shown below from testing an SVM with radial basis kernel function on all the datasets given using various values of b (radial basis kernel bandwidth) and C.

   (dataset name, C, b, error on validation set)

   ('smallOverlap', 0.01, 1, 0.115)

   ('smallOverlap', 0.01, 2, 0.105)

   ('smallOverlap', 0.01, 3, 0.1)

   ('smallOverlap', 0.01, 4, 0.075)

   ('smallOverlap', 0.01, 5, 0.07)

   ('smallOverlap', 0.01, 6, 0.06)

('smallOverlap', 0.01, 7, 0.055)
('smallOverlap', 0.01, 8, 0.055)
('smallOverlap', 0.01, 9, 0.055)
('smallOverlap', 0.1, 1, 0.08)
('smallOverlap', 0.1, 2, 0.075)
('smallOverlap', 0.1, 3, 0.075)
('smallOverlap', 0.1, 4, 0.075)
('smallOverlap', 0.1, 5, 0.07)
('smallOverlap', 0.1, 6, 0.065)
('smallOverlap', 0.1, 7, 0.055)
('smallOverlap', 0.1, 8, 0.055)
('smallOverlap', 0.1, 9, 0.055)
('smallOverlap', 1, 1, 0.07)
('smallOverlap', 1, 2, 0.065)
('smallOverlap', 1, 3, 0.065)
('smallOverlap', 1, 4, 0.065)
('smallOverlap', 1, 5, 0.055)
('smallOverlap', 1, 6, 0.055)
('smallOverlap', 1, 7, 0.055)
('smallOverlap', 1, 8, 0.055)
('smallOverlap', 1, 9, 0.055)
('smallOverlap', 10, 1, 0.07)
('smallOverlap', 10, 2, 0.055)
('smallOverlap', 10, 3, 0.055)
('smallOverlap', 10, 4, 0.055)
('smallOverlap', 10, 5, 0.05)
('smallOverlap', 10, 6, 0.045)
('smallOverlap', 10, 7, 0.04)
('smallOverlap', 10, 8, 0.02)
('smallOverlap', 10, 9, 0.015)
('smallOverlap', 100, 1, 0.055)
('smallOverlap', 100, 2, 0.05)
('smallOverlap', 100, 3, 0.03)
('smallOverlap', 100, 4, 0.01)
('smallOverlap', 100, 5, 0.01)

('smallOverlap', 100, 6, 0.005)
('smallOverlap', 100, 7, 0.0)
('smallOverlap', 100, 8, 0.0)
('smallOverlap', 100, 9, 0.0)
('bigOverlap', 0.01, 1, 0.105)
('bigOverlap', 0.01, 2, 0.105)
('bigOverlap', 0.01, 3, 0.1)
('bigOverlap', 0.01, 4, 0.075)
('bigOverlap', 0.01, 5, 0.07)
('bigOverlap', 0.01, 6, 0.06)
('bigOverlap', 0.01, 7, 0.055)
('bigOverlap', 0.01, 8, 0.055)
('bigOverlap', 0.01, 9, 0.055)
('bigOverlap', 0.1, 1, 0.08)
('bigOverlap', 0.1, 2, 0.075)
('bigOverlap', 0.1, 3, 0.075)
('bigOverlap', 0.1, 4, 0.075)
('bigOverlap', 0.1, 5, 0.07)
('bigOverlap', 0.1, 6, 0.065)
('bigOverlap', 0.1, 7, 0.055)
('bigOverlap', 0.1, 8, 0.055)
('bigOverlap', 0.1, 9, 0.055)
('bigOverlap', 1, 1, 0.07)
('bigOverlap', 1, 2, 0.065)
('bigOverlap', 1, 3, 0.065)
('bigOverlap', 1, 4, 0.065)
('bigOverlap', 1, 5, 0.055)
('bigOverlap', 1, 6, 0.055)
('bigOverlap', 1, 7, 0.055)
('bigOverlap', 1, 8, 0.055)
('bigOverlap', 1, 9, 0.055)
('bigOverlap', 10, 1, 0.07)
('bigOverlap', 10, 2, 0.055)
('bigOverlap', 10, 3, 0.055)
('bigOverlap', 10, 4, 0.055)

('bigOverlap', 10, 5, 0.05)
('bigOverlap', 10, 6, 0.045)
('bigOverlap', 10, 7, 0.04)
('bigOverlap', 10, 8, 0.02)
('bigOverlap', 10, 9, 0.015)
('bigOverlap', 100, 1, 0.055)
('bigOverlap', 100, 2, 0.05)
('bigOverlap', 100, 3, 0.03)
('bigOverlap', 100, 4, 0.01)
('bigOverlap', 100, 5, 0.01)
('bigOverlap', 100, 6, 0.005)
('bigOverlap', 100, 7, 0.0)
('bigOverlap', 100, 8, 0.0)
('bigOverlap', 100, 9, 0.0)
('ls', 0.01, 1, 0.0225)
('ls', 0.01, 2, 0.00875)
('ls', 0.01, 3, 0.00875)
('ls', 0.01, 4, 0.00875)
('ls', 0.01, 5, 0.00875)
('ls', 0.01, 6, 0.00875)
('ls', 0.01, 7, 0.00875)
('ls', 0.01, 8, 0.00875)
('ls', 0.01, 9, 0.00875)
('ls', 0.1, 1, 0.00625)
('ls', 0.1, 2, 0.005)
('ls', 0.1, 3, 0.005)
('ls', 0.1, 4, 0.005)
('ls', 0.1, 5, 0.005)
('ls', 0.1, 6, 0.00375)
('ls', 0.1, 7, 0.00375)
('ls', 0.1, 8, 0.00375)
('ls', 0.1, 9, 0.00375)
('ls', 1, 1, 0.00375)
('ls', 1, 2, 0.00375)
('ls', 1, 3, 0.00375)

('ls', 1, 4, 0.00375)
('ls', 1, 5, 0.00375)
('ls', 1, 6, 0.00375)
('ls', 1, 7, 0.0025)
('ls', 1, 8, 0.0025)
('ls', 1, 9, 0.00125)
('ls', 10, 1, 0.00375)
('ls', 10, 2, 0.00125)
('ls', 10, 3, 0.00125)
('ls', 10, 4, 0.00125)
('ls', 10, 5, 0.00125)
('ls', 10, 6, 0.0)
('ls', 10, 7, 0.0)
('ls', 10, 8, 0.0)
('ls', 10, 9, 0.0)
('ls', 100, 1, 0.00125)
('ls', 100, 2, 0.0)
('ls', 100, 3, 0.0)
('ls', 100, 4, 0.0)
('ls', 100, 5, 0.0)
('ls', 100, 6, 0.0)
('ls', 100, 7, 0.0)
('ls', 100, 8, 0.0)
('ls', 100, 9, 0.0)
('nonsep', 0.01, 1, 0.06)
('nonsep', 0.01, 2, 0.0575)
('nonsep', 0.01, 3, 0.0525)
('nonsep', 0.01, 4, 0.05)
('nonsep', 0.01, 5, 0.0475)
('nonsep', 0.01, 6, 0.0425)
('nonsep', 0.01, 7, 0.04)
('nonsep', 0.01, 8, 0.04)
('nonsep', 0.01, 9, 0.0375)
('nonsep', 0.1, 1, 0.055)
('nonsep', 0.1, 2, 0.05)

('nonsep', 0.1, 3, 0.05)
('nonsep', 0.1, 4, 0.05)
('nonsep', 0.1, 5, 0.05)
('nonsep', 0.1, 6, 0.0475)
('nonsep', 0.1, 7, 0.0425)
('nonsep', 0.1, 8, 0.04)
('nonsep', 0.1, 9, 0.04)
('nonsep', 1, 1, 0.045)
('nonsep', 1, 2, 0.045)
('nonsep', 1, 3, 0.04)
('nonsep', 1, 4, 0.035)
('nonsep', 1, 5, 0.035)
('nonsep', 1, 6, 0.035)
('nonsep', 1, 7, 0.025)
('nonsep', 1, 8, 0.025)
('nonsep', 1, 9, 0.025)
('nonsep', 10, 1, 0.04)
('nonsep', 10, 2, 0.03)
('nonsep', 10, 3, 0.025)
('nonsep', 10, 4, 0.0225)
('nonsep', 10, 5, 0.02)
('nonsep', 10, 6, 0.0175)
('nonsep', 10, 7, 0.0175)
('nonsep', 10, 8, 0.01)
('nonsep', 10, 9, 0.01)
('nonsep', 100, 1, 0.0325)
('nonsep', 100, 2, 0.0175)
('nonsep', 100, 3, 0.015)
('nonsep', 100, 4, 0.0075)
('nonsep', 100, 5, 0.0025)
('nonsep', 100, 6, 0.0025)
('nonsep', 100, 7, 0.0025)
('nonsep', 100, 8, 0.0025)
('nonsep', 100, 9, 0.0025)

Increasing both C and b (the bandwidth) decreases error rate on the validation set, with the optimal performance being $C = 100$ and $b = 7$, 8 or 9. Using the maximum values of C and b minimized the validation error for all datasets.

Data from the smallOverlap dataset is shown below and referenced in answers to parts (a) and (b).

(C, geometric margin: $1/||w||$, error rate, number of support vectors)

(0.01, 2.1818056369776855, 0.08, 114)

(0.1, 0.63132037056137913, 0.085, 66)

(1, 0.34291076910827101, 0.08, 50)

(10, 0.31684915985167916, 0.08, 48)

(100, 0.31066728459715598, 0.08, 48)

(a) As C increases, the geometric margin decreases. This will always happen as C is increased, because lowering C allows the SVM to misclassify some points, allowing the separator to be further from the points that it correctly classifies.

(b) As C increases, the number of support vectors decreases.

(c) If we optimize C based on training data, C will tend to become very small as this is what maximizes the geometric margin since lowering C allows more slack. But letting C get small and making a bigger margin on the training data is in a sense just being ignorant about how the data really behave. This would essentially be pretending that the data are easily separable and have a wide separation, when really, they aren't.

Selecting C to maximize the geometric margin on the validation set would be a better approach. Here, we get some external guidance as to how much slack we should allow on the training data, based on how well it will perform on a new, unseen data set. This won't just make C tend to 0; if C dropped to 0, the performance on the validation set would likely turn out very poor! Instead, C will be guided towards some reasonable optimum that will likely turn out well on additional test data.

In general, this is what we usually do when picking parameters: optimize parameters based on their performance with validation data rather than training data. Another example would be for choosing the degree of a polynomial regression. Optimizing this strictly on the training data would lead to severe overfitting. Instead we choose the degree which will optimize the trained model's performance on validation data.

2 Logistic Regression (LR)

1.
$$NLL(w) = \sum_i \log(1 + \exp(-y^{(i)}(x^{(i)}w + w_0))$$
$$NLL(w) = \sum_i \log(1 + \exp(-y^{(i)}(x^{(i)}X^T\alpha + w_0))$$
$$NLL(w) = \sum_i \log(1 + \exp(-y^{(i)}(f(x^{(i)}) + w_0))$$

, where:
$$f(x) = xX^T\alpha$$
$$f(x) = \sum_j K(x, x^{(j)})\alpha_j$$

Here, K is a kernel function which gives the dot product of two vectors x_a and x_b, possibly in a transformed space, i.e. $\phi(x_a) * \phi(x_b)$.

2. I implemented logistic regression with smoothed L1 regularization, using a penalty of $\sqrt{||\alpha||_1^2 + \epsilon}$, where $\epsilon = 0.0000001$.

3. Increasing λ caused the α vector to be both lower in magnitude and more sparse.

   Here are some values for λ and the corresponding α vectors that are computed on a subset of the "ls" data set:

   $\lambda = 0.0001$, $\alpha =$

```
[  2.41847028e+00   2.88931450e-04  -1.91484206e-01  -1.70179087e-03
   3.25816493e+00   3.61459686e-01   1.10543081e+00   9.99311758e-01
   1.54273792e+00   2.31627345e+00   7.45881937e-02  -1.29548824e+00
   1.27376250e+00   5.48178489e-01  -1.48065236e+00  -1.37602189e+00
  -1.80351903e+00  -6.40105831e-01  -1.73712235e+00  -1.33100161e+00
   1.33789765e+00  -5.82192993e-01  -1.95357256e-03   1.08856718e+00
  -2.36242413e+00  -3.51349396e+00]
```

   $\lambda = 0.01$, $\alpha =$

```
[ 2.0027146   0.09358828 -0.17248321  0.00372521  2.67724932  0.34311617
  0.91894831  0.87582331  1.29255722  1.91202766  0.14759878 -1.04707678
  1.0607957   0.45428338 -1.23041484 -1.17592927 -1.51506491 -0.57038873
 -1.45508168 -1.13882162  1.07293089 -0.50759425 -0.00550705  0.91457435
 -1.96604573 -2.9402225 ]
```

   $\lambda = 1$, $\alpha =$

```
[ 0.1687428   0.51599742  0.30213681  0.29481348  0.11909471  0.25919003
  0.12088451  0.30941376  0.19161711  0.13759177  0.42730321  0.42040996
  0.13959925 -0.34902078 -0.14749712 -0.28157096 -0.23782224 -0.26288725
 -0.21293495 -0.28014943 -0.46379548 -0.19588088 -0.26305971 -0.26758934
 -0.21346784  0.06947596]
```

$\lambda = 10$, $\alpha =$

```
[ 0.01000634  0.04802424  0.0246105   0.02380873  0.0045708   0.01990863
  0.00476675  0.02540719  0.01251065  0.00659589  0.03831388  0.0375592
  0.00681567 -0.0466791  -0.02461602 -0.03929461 -0.03450494 -0.03724909
 -0.03178024 -0.03913898 -0.05924479 -0.02991314 -0.03726797 -0.03776388
 -0.03183859 -0.00086152]
```

4. Using a gaussian kernel, here is the performance of my LR implementation as a function of λ and the kernel bandwidth.

   (λ, bandwidth, error)

   (0.1, 1, 0.0875)

   (0.1, 10, 0.09625)

   (0.1, 100, 0.0925)

   (1, 1, 0.09125)

   (1, 10, 0.08875)

   (1, 100, 0.1175)

   (10, 1, 0.1175)

   (10, 10, 0.1175)

   (10, 100, 0.1175)

5. The SVM solutions are more sparse, while LR tends to give high weight to more of the data points.

3 Multi-Label Classification

I created a multi-class SVM using the one-versus-many technique, where classification is made based on which class is scored with highest probability from a series of classifiers, each classifier being trained on data where elements from the class of interest are labeled +1 and elements from all other classifiers are labeled -1. Using a Gaussian kernel with bandwidth 10, and a training set consisting of 4000 data points selected at random from the training data, the classifier classifies my test set with an error rate of 0.46.

Using a Logistic Regression classifier, again using one-versus-many to separate multiple classes, my error rate was 0.86.