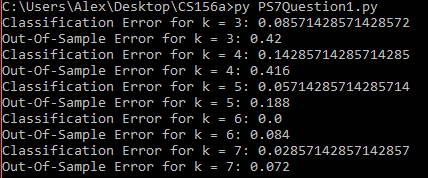
CS156a Problem Set #7

1. Work is in my code:

1. **import** numpy
3. file\_out = open('ps6out.txt', 'r')
5. x2 = []
6. y2 = []
7. **for** line2 **in** file\_out:
8. fields2 = line2.strip().split()
9. x2.append([float(fields2[0]), float(fields2[1])])
10. y2.append(float(fields2[2]))
12. file\_out.close()
14. xtest = []
15. **for** i **in** range(len(x2)):
16. xtest.append([1, x2[i][0], x2[i][1], x2[i][0]\*\*2, x2[i][1]\*\*2, x2[i][0] \* x2[i][1], \
17. abs(x2[i][0] - x2[i][1]), abs(x2[i][0] + x2[i][1])])
19. file\_in = open('ps6in.txt', 'r')
21. xstart = []
22. y = []
23. **for** line **in** file\_in:
24. fields = line.strip().split()
25. xstart.append([float(fields[0]), float(fields[1])])
26. y.append(float(fields[2]))
28. file\_in.close()
30. x = []
31. **for** i **in** range(len(xstart)):
32. x.append([1, xstart[i][0], xstart[i][1], xstart[i][0]\*\*2, xstart[i][1]\*\*2, xstart[i][0] \* xstart[i][1], \
33. abs(xstart[i][0] - xstart[i][1]), abs(xstart[i][0] + xstart[i][1])])
35. xtest = []
36. **for** i **in** range(len(x2)):
37. xtest.append([1, x2[i][0], x2[i][1], x2[i][0]\*\*2, x2[i][1]\*\*2, x2[i][0] \* x2[i][1], \
38. abs(x2[i][0] - x2[i][1]), abs(x2[i][0] + x2[i][1])])

41. **def** sign(x):
42. **if** x < 0:
43. **return** -1
44. **if** x > 0:
45. **return** 1
46. **if** x == 0:
47. **return** 0
49. **for** k **in** range(4, 9):
50. x3 = []
51. y3 = []
52. **for** z **in** range(25):
53. test = []
54. **for** q **in** range(k):
55. test += [x[z][q]]
56. x3.append(test)
57. y3.append(y[z])
59. left = numpy.matmul(numpy.transpose(x3), x3)
60. leftinv = numpy.linalg.inv(left)
61. w = numpy.matmul(leftinv, numpy.matmul(numpy.transpose(x3), y3))
63. x4 = []
64. y4 = []
65. **for** t **in** range(25, 35):
66. test1 = []
67. **for** r **in** range(k):
68. test1 += [x[t][r]]
69. x4.append(test1)
70. y4.append(y[t])
72. eintotal = 0
73. **for** it **in** range(len(x4)):
74. **if**  sign(numpy.inner(w, x4[it])) != y4[it]:
75. eintotal += 1
77. insample = (eintotal / len(x))
79. **print**("Classification Error for k = " + str(k - 1) + ": " + str(insample))
81. xtest2 = []
82. **for** z **in** range(len(xtest)):
83. test3 = []
84. **for** q **in** range(k):
85. test3 += [xtest[z][q]]
86. xtest2.append(test3)
88. wrong = 0
89. **for** iter2 **in** range(len(xtest)):
90. **if**  sign(numpy.inner(w, xtest2[iter2])) != y2[iter2]:
91. wrong += 1
93. disagree = (wrong / len(xtest))
95. **print**("Out-Of-Sample Error for k = " + str(k - 1) + ": " + str(disagree))

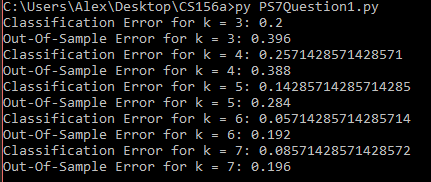
The output is shown:



The smallest classification error is 0.0 at k = 6. The answer is [d].

2. The small out-of-sample error is 0.072 at k = 7. The answer is [e].

3. Using the code before but changing line 52 to range(25, 35) and line 65 to range(25):



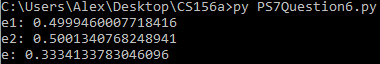
The smallest classification error is 0.057 at k = 6. The answer is [d].

4. The smallest out-of-sample error is 0.192 at k = 6. The answer is [d].

5. The out-of-sample error for k = 6 for Problem 1 is 0.084. The out-of-sample error for k = 6 for Problem 3 is 0.192. This is closest to (0.1, 0.2) respectively. The answer is [b].

6. Since e1 and e2 are independent, random variables distributed uniformly on [0, 1], these values will be expected to be the average: 0.5. Since e = min(e1, e2), it would make sense that e = 0.5; however, e has an optimistic bias which makes it less than 0.5. The answer can be shown through a huge set of data:

1. **import** random
3. e1 = 0
4. e2 = 0
5. e = 0
7. **for** i **in** range (100000000):
8. e1t = random.uniform(0, 1)
9. e1 += e1t
10. e2t = random.uniform(0, 1)
11. e2 += e2t
12. et = min(e1t, e2t)
13. e += et
15. **print**("e1: " + str(e1/100000000))
16. **print**("e2: " + str(e2/100000000))
17. **print**("e: " + str(e/100000000))



The answer is closest to [d].

7. To do this, we need to create constant models and linear models out of the points:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Point Used | Point Used | Model Formula | Error |
| Constant | (-1, 0) | (p, 1) | Y = 0.5 | 0.5 |
| Constant | (p, 1) | (1, 0) | Y = 0.5 | 0.5 |
| Constant | (-1, 0) | (1, 0) | Y = 0 | 1 |
| Linear | (-1, 0) | (p, 1) | Y = (1/(p+1))x + (1/(p+1)) | 2/(p+1) |
| Linear | (p, 1) | (1, 0) | Y = (-1/(1-p))x + (1/(1-p)) | 2/(1-p) |
| Linear | (-1, 0) | (1, 0) | Y = 0 | 1 |

Set these equal:

Solve:

Multiply each side by and simply:

Using the quadratic formula with x = p2:

We can restrict this to real numbers, so we exclude the minus case. When we square root this value, we get the plus/minus case again, but p is said to be greater or equal to zero:

The answer is [c].

8. Work is in the code:

1. **import** random
2. **from** operator **import** add
3. **from** sklearn **import** svm
5. N = int(input("Number of Points: "))
6. runs = 1000
7. actualruns = 0
8. disagree = 0
9. vectors = 0
11. **for** q **in** range(runs):
12. points = []
13. **for** i **in** range(N):
14. points.append([1, [random.uniform(-1,1), random.uniform(-1,1)], 0])
16. x = []
17. **for** l **in** range(N):
18. x.append([points[l][1][0], points[l][1][1]])
20. # creates random list of points with default first and last value
22. point1 = [random.uniform(-1,1), random.uniform(-1,1)]
23. point2 = [random.uniform(-1,1), random.uniform(-1,1)]
24. slope = (point2[1] - point1[1])/(point2[0] - point1[0])
25. intercept = point1[1] - slope \* point1[0]
27. # creates the target function
29. **for** j **in** points:
30. **if** j[1][1]  < j[1][0] \* slope + intercept:
31. j[2] = -1
32. **else**:
33. j[2] = 1
35. y = []
36. **for** k **in** range(N):
37. y.append(points[k][2])
38. check = set(y)
39. **if** len(check) == 1:
40. **continue**
41. actualruns += 1
43. # updates the y value of each point set
45. **def** sign(x):
46. **if** x < 0:
47. **return** -1
48. **if** x > 0:
49. **return** 1
50. **if** x == 0:
51. **return** 0
53. w = [0, 0, 0]
55. misclass = points
57. **while** len(misclass) != 0:
58. iter = random.randint(0, len(misclass) - 1)
59. w = list(map(add, w, [q \* misclass[iter][2] **for** q **in** [misclass[iter][0],\
60. misclass[iter][1][0], misclass[iter][1][1]]]))
61. misclass = []
62. **for** z **in** range(len(points)):
63. **if** sign(w[0] \* points[z][0] + w[1] \* points[z][1][0]\
64. + w[2] \* points[z][1][1]) != points[z][2]:
65. misclass.append(points[z])
67. points2 = []
68. **for** m **in** range(1000):
69. points2.append([1, [random.uniform(-1,1), random.uniform(-1,1)], 0])
71. **for** n **in** points2:
72. **if** n[1][1]  < n[1][0] \* slope + intercept:
73. n[2] = -1
74. **else**:
75. n[2] = 1
77. wrong = 0
79. **for** iter2 **in** range(len(points2)):
80. **if**  sign(w[0] \* points2[iter2][0] + w[1] \* \
81. points2[iter2][1][0] + w[2] \* points2[iter2][1][1]) != points2[iter2][2]:
82. wrong += 1
84. PLAdisagree = (wrong/len(points2))
86. clf = svm.SVC(kernel='linear', C=999999999999999999999999999999999999)
87. clf.fit(x,y)
88. vectors += len(clf.support\_)
90. wrong2 = 0
92. **for** iter3 **in** range(len(points2)):
93. **if**  clf.predict([[points2[iter3][1][0], points2[iter3][1][1]]]) != points2[iter3][2]:
94. wrong2 += 1
96. SVMdisagree = (wrong2/len(points2))
98. **if** PLAdisagree > SVMdisagree:
99. disagree += 1
101. **print**("SVM is better than PLA " + str((disagree/actualruns) \* 100) + "% of the time")
102. **print**("Number of support vectors: " + str(vectors/actualruns))





It was better 63% of the time, so the answer is closest to [c].

9. It was better 59.9% of the time, so the answer is closest to [d].

10. It was 2.997, so the answer is closest to [b].