CE4708 Artificial Intelligence

Assignment 2

Soft-Margin, Kernel-Based SVM Classifier

Study Abroad Student

Dr. Colin Flanagan

Zophia Raleigh

03 December 2018

Table of Contents

|  |  |
| --- | --- |
|  | Page Number |
| 1. Overview | 2 |
| 1. Functions | 2 |
| * 1. Removed Functions |  |
| * 1. Remaining Functions |  |
| * + 1. rbfKernel -> rbf\_kernel |  |
| * + 1. makeLambdas -> make\_lambdas |  |
| * + 1. makeB ->make\_b |  |
| * + 1. classify -> no naming convention change applied |  |
| * + 1. testClassifier -> test\_classfier |  |
| * + 1. plotContours -> plot\_contours |  |
| * + 1. makeP -> make\_p |  |
| * + 1. activation -> no naming convention change applied |  |
| * + 1. setupMultipliersAndBias -> set\_multipliers\_and\_bias |  |
| 10. modified doTest |  |
| 1. Results | 6 |
| * 1. Misclassification |  |
| * 1. Graphs |  |
| 1. Appendixes | 8 |
| Appendix 1: Output | 8 |
| Appendix 2: Code | 10 |
| Graph 1: Soft-Margin Training Data | 22 |
| Graph 2: Soft-Margin Testing Data | 22 |
| Graph 3: Hard-Margin Training Data | 23 |
| Graph 4: Hard-Margin Testing Data | 23 |

**I. Overview**

The goal of this assignment was to “develop a program in Python to implement a soft-margin, kernel-based, Support Vector Machine (SVM) Classifier” (This can be found in the assignments tab of the class Sulis site under “soft-margin, kernel-based, svm classifier”). The class was also given the parameters of using a Radial Basis Function kernel with a sigma^2 of 0.25, C parameters of one and 10^6, and we were told to use the Python CVXOPT library and QP-solver within the program.

Dr. Flanagan provided the class with a set of code for a soft-margin, kernel based, Support Vector Machine (SVM) Classifier to start the project. With this as a base point, I was able to derive functions to utilize a radial basis function (rbf) kernel for both a soft-margin SVM and a hard-margin SVM on provided training and testing data. After working out the code, my program produced four output graphs and two misclassifications reports. Each graph corresponds to one of the following: soft-margin testing data, soft-margin training data, hard-margin testing data, and hard-margin training data.

**II. Functions**

*A. Removed Functions*

As stated above, Dr. Flanagan provided the class with fifteen functions and their associated comments. Of these fifteen, I decided to utilize nine and additionally modify the doTest function using the function’s internal code without the identifiable function call. The removed functions were, however, helpful to see while gaining insight on how SVMs work.

Each function was removed for different reasons, which I shall explain, but to begin, the functions I removed were: polyKernel, linKernel, makeBinarySequence, makeXor, makeAnd, and doTest.

[P]olyKernel and linKernel were both removed, because the assignment directed the class to use a radial basis function kernel which was supplied as rbfKernel. [M]akeBinarySequence was removed because it did not directly impact what was needed for this assignment and would have only cluttered the code. [M]akeXor and makeAnd were removed because this assignment did not include an XOR or AND logic within its parameters. And finally, doTest was only truly removed in name. I was able to refigure the body code within the function so that I could figuratively call the function twice in the same program, thus solving both the soft and the hard margin problems.

*B. Remaining Functions*

Before truly diving into each of the remaining functions, a few general modifications to the code needed to be made specifically for the purposes of this assignment. The first hurdle I decided to tackle was adding a C value to the appropriate functions. I chose to default the C Value to 10^6 or 1,000,000.0 in the parameters of the all relevant functions, in effect defaulting the SVM to hard-margin operation. The other blanket modification made before refactoring the naming conventions was on every function which called on makeLambdas (make\_lambdas) to ensure the make\_lambdas function interacted with the program correctly.

Of the remaining functions, I decided to change the naming convention from camelCase to a lower\_case\_format for appropriate function and variable names. The main purpose of doing this was to force myself to become intimately familiar with the provided code and to deeply analyze how the various functions interact with each other.

I will now explain the changes made within each of the remaining functions.

*1. rbfKernel -> rbf\_kernel*

The only major change to this function was setting the sigma2 to 0.25, this was specified by the instructions of the professor and allows the program to run without me having to manually specify a sigma2 value for the subsequent steps requiring it.

*2. makeLambdas -> make\_lambdas*

The first noticeable change, outside of the previously stated blanket changes, within this function was a modification to the constraints from a polynomial kernel to a rbf kernel. Further, within the main body of the code, for loops were added to change the shape of the q and h matrices to account for the soft margin operation within the QP Solver.

It is important to note, as stated previously, that changing these matrices affects almost every other function within the program. Ensuring this function was correct, produced a huge step forward towards producing a viable output.

*3. makeB -> make\_b*

The only changes made to this function was the change in the kernel function and the addition of the c value.

*4. Classify -> no naming convention change applied*

The only changes made to this function was the change in the kernel function and the addition of the c value.

*5. testClassifier -> test\_classifier*

The first things done to this function were the blanket changes of the kernel and c value. Next, I had to modify the function to accept unseen data which was done through the constraints input\_testing=None and output\_testing=None. Then, I optionally tested on the new data and allowed for misclassifications during testing. It is also this function that produces the misclassification output report for training and testing data.

Each of these code change steps were done through for loops and if/else statements which can be seen in Appendix 2: Code.

*6. plotContours -> plot\_contours*

[P]lot\_contours required a change in the kernel function and additional constraints of the c value, plot\_new\_data, new\_x, and new\_t.

The entire purpose of this function is to physically plot the contours found on the output graphs. Utilizing a for loop and if/else statements, I was able to cycle through the provided data to produce graphs which will be discussed in the next section.

*7. makeP -> make\_p*

There were no changes made to this function. Its given set up works for the given assignment.

*8. Activation ->no naming convention change applied*

There were no changes made to this function. Its given set up works for the given assignment.

*9. setupMultipliersAndBias -> set\_multipliers\_and\_bias*

The only changes made to this function was the change in the kernel function and the addition of the c value.

*10. modified doTest*

As I stated previously, I was able to take the given doTest function and reconfigure the code to work for the purposes of this assignment. By taking away the need to call the function, I guaranteed that the code would run. Then, I copied the process of the code twice, once accounting for a c value of 1.0 and once accounting for a c value of 1,000,000.0. This way I ensured I had outputs for both a soft-margin and a hard-margin SVM. Additionally, this code also checked to see if the output\_testing data and testing\_output matched.

Overall, this section of the code ensured the program is tied together and produces a viable output for the user. To review the code for this section, please see Appendix 2: Code.

**III. Results**

*A. Misclassification*

The entire output from running this SVM can be found in Appendix 1: Output. For this section, I am more interested in discussing the relevance of the specific misclassification numbers.

For the soft-margin SVM, the misclassification output Appendix 1: Output is:

Soft margin training misclassification 9

Soft margin testing misclassification 162

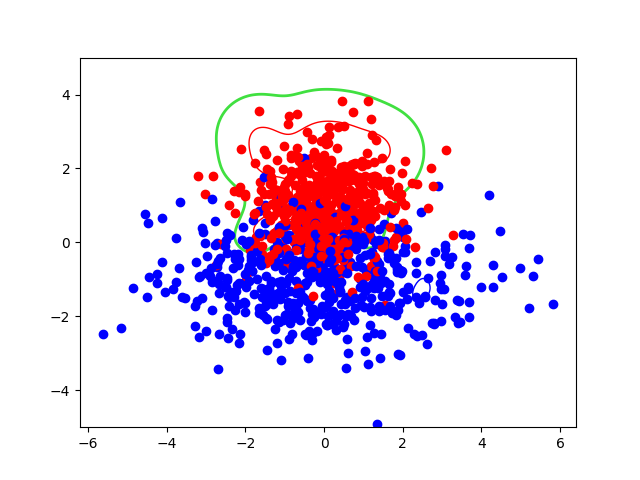
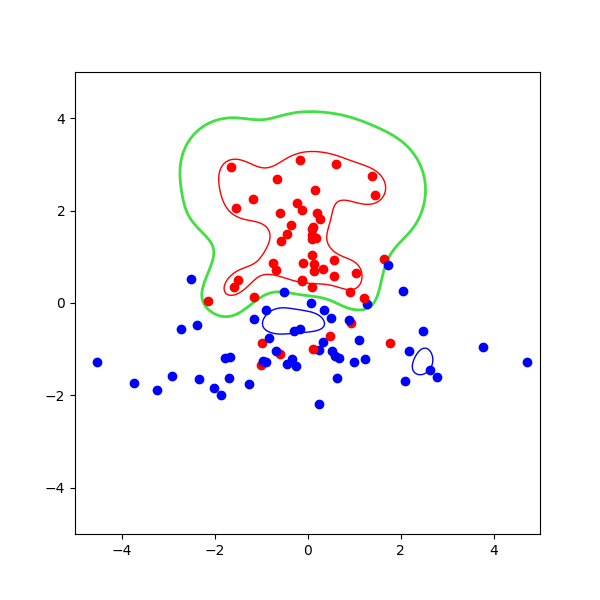
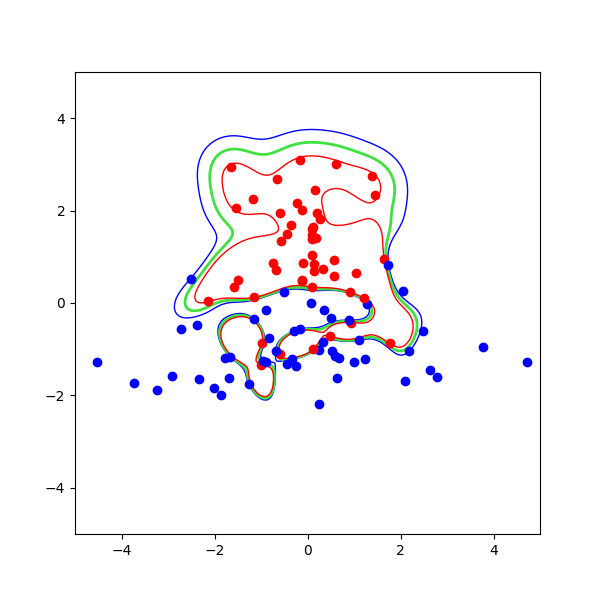
For the hard-margin SVM, the misclassification output from Appendix 1: Output is:

Hard margin training data misclassification 0

Hard margin testing data misclassification 247

There are two main parts to look at when evaluating these outputs. First, we want to look at the training misclassification amounts. From this, one might assume that the hard-margin SVM is better since there are no misclassifications for it while there are 9 misclassifications for the soft-margin. However, when one then evaluates the testing data it becomes clear that the soft-margin machine is in fact superior.

The reason for this is that the hard margin SVM will train to ensure that all known data points (training data) provided produce zero mistakes, so the machine creates incredibly specific boundary lines. In juxtaposition, the soft-margin machine wants as few misclassifications as possible, but is more concerned with creating boundaries that allow for future unseen data (testing data) to have a higher likelihood of still falling within the correct boundary limits.

As long as there is still data to be run through the machine in the future (living in a world without perfect information, aka most worlds) it is often more advantageous to utilize a soft-margin machine which allows outliers to be misclassified thus creating more of a broad accuracy, rather than building strict boundaries that do not allow for a growth in data with time.

B. Graphs

Once the code is run, the program outputs four graphs. Each corresponds to different parameters regarding the input data (testing versus training) and which cycle of the edited doTest function the program is on. What this means is that the graphical output is produced in the following order: soft-margin training data, then soft-margin testing data, then hard-margin training data, then finally hard-margin testing data. Each graph, in order, can be seen in the following appendix.

**IV. Appendixes**

Appendix 1: Output

pcost dcost gap pres dres

0: -3.5168e+01 -2.1070e+02 7e+02 2e+00 8e-16

1: -2.7207e+01 -1.2784e+02 1e+02 7e-16 8e-16

2: -3.1099e+01 -4.6042e+01 1e+01 8e-16 4e-16

3: -3.3063e+01 -3.6402e+01 3e+00 8e-16 5e-16

4: -3.3712e+01 -3.4361e+01 6e-01 1e-16 5e-16

5: -3.3868e+01 -3.3991e+01 1e-01 2e-15 6e-16

6: -3.3899e+01 -3.3919e+01 2e-02 3e-15 5e-16

7: -3.3905e+01 -3.3906e+01 4e-04 1e-15 5e-16

8: -3.3906e+01 -3.3906e+01 1e-05 7e-16 6e-16

Optimal solution found.

Check PASSED

Soft margin training misclassification 9

Soft margin testing misclassification 162

pcost dcost gap pres dres

0: 3.7528e+18 -2.6468e+19 3e+19 7e-07 3e-07

1: 1.0677e+18 -2.9348e+18 4e+18 6e-07 3e-07

2: 2.2072e+17 -4.4426e+17 7e+17 1e-07 2e-07

3: 3.7830e+16 -6.6426e+16 1e+17 7e-07 2e-07

4: 8.8771e+15 -1.5584e+16 2e+16 4e-07 1e-07

5: 1.4256e+15 -2.3189e+15 4e+15 3e-07 7e-08

6: 2.1473e+14 -2.5993e+14 5e+14 1e-07 4e-08

7: 3.0984e+13 -3.5076e+13 7e+13 4e-08 3e-08

8: 4.4504e+12 -4.9419e+12 9e+12 2e-08 9e-09

9: 6.3826e+11 -7.0419e+11 1e+12 1e-08 3e-09

10: 9.1485e+10 -1.0071e+11 2e+11 2e-10 8e-10

11: 1.3109e+10 -1.4422e+10 3e+10 1e-09 2e-10

12: 1.8778e+09 -2.0668e+09 4e+09 2e-10 1e-10

13: 2.6879e+08 -2.9643e+08 6e+08 2e-10 6e-11

14: 3.8401e+07 -4.2591e+07 8e+07 1e-11 2e-11

15: 5.4570e+06 -6.1480e+06 1e+07 4e-11 7e-12

16: 7.6260e+05 -8.9926e+05 2e+06 3e-12 3e-12

17: 9.9729e+04 -1.3729e+05 2e+05 6e-12 1e-12

18: 7.9426e+03 -2.4937e+04 3e+04 3e-12 5e-13

19: -4.6022e+03 -8.8689e+03 4e+03 2e-12 5e-13

20: -6.2588e+03 -7.2745e+03 1e+03 7e-13 4e-13

21: -6.6904e+03 -6.8251e+03 1e+02 4e-13 3e-13

22: -6.7414e+03 -6.7526e+03 1e+01 2e-12 3e-13

23: -6.7455e+03 -6.7482e+03 3e+00 2e-13 3e-13

24: -6.7468e+03 -6.7475e+03 6e-01 2e-13 3e-13

25: -6.7471e+03 -6.7471e+03 2e-02 2e-12 4e-13

26: -6.7471e+03 -6.7471e+03 2e-04 2e-12 3e-13

Optimal solution found.

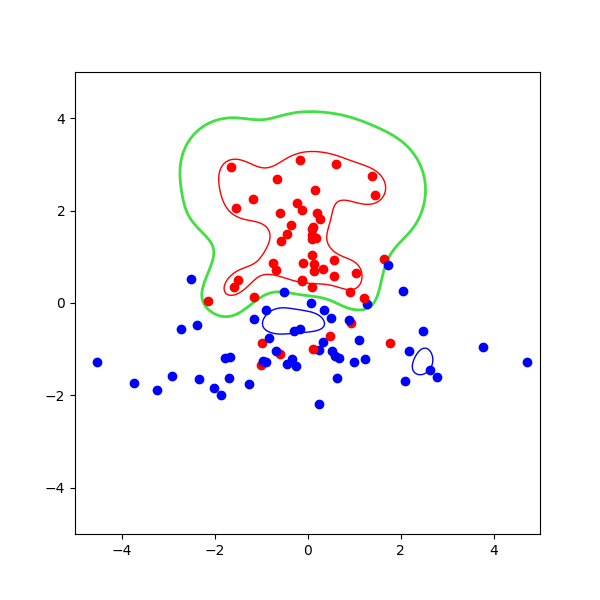
Check PASSED

Hard margin training data misclassification 0

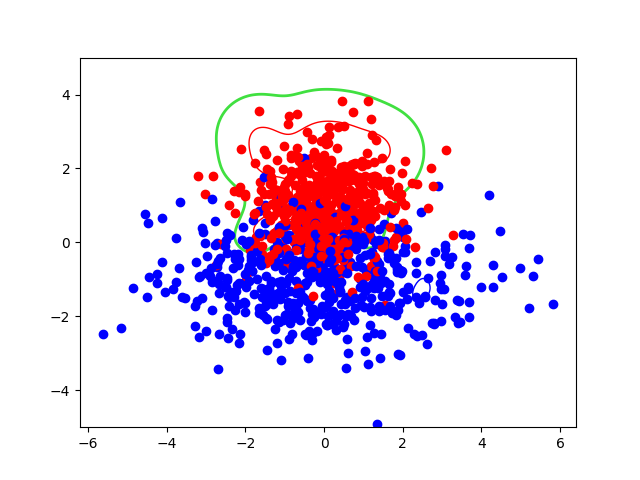
Hard margin testing data misclassification 247

Appendix 2: Code

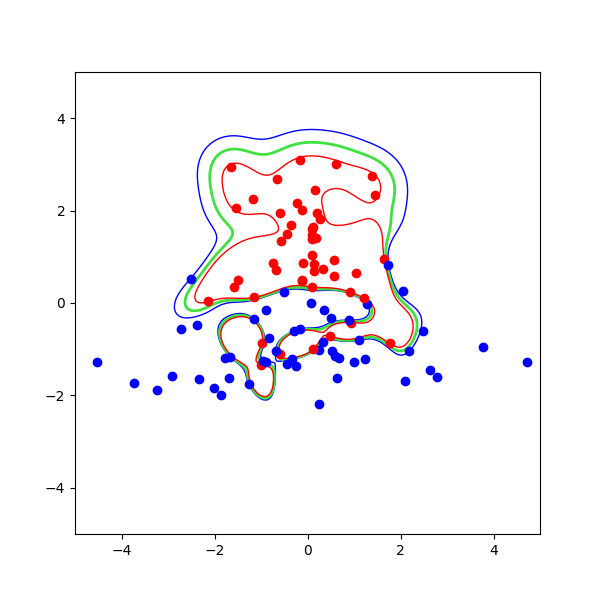
1. from cvxopt import matrix, solvers
2. from math import exp
3. import matplotlib.pyplot as plt
4. import numpy as np
5. from numpy import arange, array
6. training\_data = np.loadtxt('training\_data.txt', skiprows=1)
7. training\_input, training\_output = training\_data[:, :2], training\_data[:, 2]
8. testing\_data = np.loadtxt('testing\_data.txt', skiprows=1)
9. testing\_input, testing\_output = testing\_data[:, :2], testing\_data[:, 2]
10. # rbf\_kernel(x,y,s2) -- Radial basis function kernel exp(-||x-y||^2/2\*sigma^2).
11. # further represented as k
12. # Arguments:
13. #
14. # x,y -- n-element vectors.
15. #
16. # s2 -- Variance of the R.B.F. kernel.
17. # Squared standard deviation, sigma^2.
18. #
19. # Returns: Scalar value of kernel for the 2 input vectors.
20. def rbf\_kernel(v1, v2, sigma2=0.25):
21. assert len(v1) == len(v2)
22. assert sigma2 >= 0.0
23. mag2 = sum(map(lambda x, y: (x - y) \* (x - y), v1, v2))
24. return exp(-mag2 / (2.0 \* sigma2))
25. # input\_training -- A list of vectors (lists) representing
26. # the input training patterns for this
27. # problem.
28. #
29. # e.g., [[0,0],[0,1],[1,0],[1,1]], the set
30. # of binary training patterns.
31. #
32. # output -- A list of desired outputs. These must
33. # be values in the set {-1,+1}. If there
34. # are n input vectors in x, there must be
35. # exactly n values in t.
36. #
37. # e.g., [-1,-1,-1,1] -- the outputs
38. # for a 2-input AND function.
39. #
40. # k -- A Kernel function.
41. # c -- The softness variable
42. def make\_lambdas(input\_training, output, k=rbf\_kernel, c=1000000.0):
43. """Solve constrained maximaization problem and return list of l's."""
44. p = make\_p(input\_training, output, k)
45. n = len(output)
46. q = matrix(-1.0, (n, 1))
47. h = matrix(0.0, (2 \* n, 1))
48. for i in range(n):
49. h[i + n] = c
50. g = matrix(0.0, (2 \* n, n))
51. for i in range(2 \* n):
52. for j in range(n):
53. if i < n:
54. if j == i:
55. g[i, j] = -1.0
56. if i >= n:
57. if j == (i - n):
58. g[i, j] = 1.0
59. a = matrix(output, (1, n), tc='d')
60. r = solvers.qp(p, q, g, h, a, matrix(0.0))
61. # print(r)
62. output = [round(l, 6) for l in list(r['x'])]
63. return (r['status'], output)
64. # make\_b -- Given the set of training vectors, "Xs", the set of
65. # training responses "Ts", and the set of Lagrange
66. # multipliers for the problem "Ls", return the bias for
67. # the classifier.
68. #
69. # Arguments:
70. #
71. # input\_training -- Inputs, as "input\_training" in makeLambdas.
72. #
73. # output -- A list of desired outputs. As "output" in
74. # makeLambdas.
75. #
76. # lang\_list -- A list of Lagrange multipliers, the
77. # solution to the constrained optimaztion
78. # of W(L) as returned by a call to
79. # makeLambdas. N.B., if this argument is
80. # None (the default), this routine will call
81. # generateLambdas automatically.
82. #
83. # k -- A Kernel function.
84. def make\_b(input\_training, output, lang\_list=None, k=rbf\_kernel, c=1000000.0):
85. """Generate the bias given input\_traing, Output and (optionally) Ls and K"""
86. lang\_list, dummy\_b = set\_multipliers\_and\_bias(input\_training, output, lang\_list,
87. 0.0, k, c)
88. sv\_count = 0
89. b\_sum = 0.0
90. for n in range(len(output)):
91. if lang\_list[n] >= 1e-10:
92. sv\_count += 1
93. b\_sum += output[n]
94. for i in range(len(output)):
95. if lang\_list[i] >= 1e-10:
96. b\_sum -= lang\_list[i] \* output[i] \* k(input\_training[i], input\_training[n])
97. return b\_sum / sv\_count
98. # classify -- Classify an input vector using the Lagrange
99. # multipliers for the problem, the set of training
100. # inputs, "Xs", the set of desired outputs, "Ts" and
101. # bias "b", classify a vector "x".
102. #
103. # Arguments:
104. #
105. # x -- An input vector to classify (a list of values).
106. #
107. # input\_training -- A list of the training input vectors (hence a list
108. # of n-element lists).
109. #
110. # output -- A list of desired outputs. As "Ts" in
111. # makeLambdas.
112. #
113. # lang\_list -- A list of Lagrange multipliers.
114. #
115. # b -- The classifier bias, as generated by makeB.
116. #
117. # k -- A Kernel function.
118. #
119. # verbose -- Controls whether or not the routine
120. # prints details about the current classification to
121. # the terminal as well as returning a status
122. # value. Defaults to True.
123. def classify(x, input\_training, output, lang\_list=None, b=None, k=rbf\_kernel,
124. verbose=True, c=1000000.0):
125. """Classify an input x into {-1,+1} given support vectors, outputs and L."""
126. lang\_list, b = set\_multipliers\_and\_bias(input\_training, output, lang\_list, b, k, c)
127. y = activation(x, input\_training, output, lang\_list, b, k)
128. if verbose:
129. print("{} {:8.5f} -->".format(x, y), end=' ')
130. if y > 0.0:
131. print("+1")
132. elif y < 0.0:
133. print("-1")
134. else:
135. print("0 (ERROR)")
136. if y > 0.0:
137. return +1
138. elif y < 0.0:
139. return -1
140. else:
141. return 0
142. # testClassifier(input\_training,output,lang\_list,b,K,verbose)
143. # --Test a classifier by checking to see if its response
144. # to every training input Xs[i] is the desired output
145. # Ts[i].
146. #
147. # Arguments:
148. #
149. # input\_training -- A list of vectors (lists) representing
150. # the input training patterns for this
151. # problem.
152. #
153. # output -- A list of desired outputs. These must
154. # be values in the set {-1,+1}.
155. # lang\_list -- A list of Lagrange multipliers.
156. #
157. # b -- The classifier bias, as generated by makeB.
158. #
159. # k -- A Kernel function.
160. #
161. # verbose -- Controls whether or not the routine
162. # prints details of misclassifications to the
163. # terminal as well as returning a status
164. # value. Defaults to True.
165. # Note: If good is false test failed
166. def test\_classifier(input\_training, output\_training, lang\_list=None, b=None,
167. k=rbf\_kernel, verbose=True, do\_test=False,
168. c=1000000.0, input\_testing=None, output\_testing=None):
169. """Test a classifier specifed by Lagrange mults, bias and kernel on all Xs/Ts
170. pairs."""
171. assert len(input\_training) == len(output\_training)
172. lang\_list, b = set\_multipliers\_and\_bias(input\_training, output\_training, lang\_list, b,
173. k, c)
174. good = True
175. train\_mis\_class = 0
176. test\_mis\_class = 0
177. for i in range(len(input\_training)):
178. train\_c = classify(input\_training[i], input\_training, output\_training, lang\_list, b,
179. k, verbose, c)
180. if train\_c != output\_training[i]:
181. if verbose:
182. print("Misclassification: input {}, output {:d}, "
183. "expected {:d}".format(input\_training[i], train\_c, output\_training[i]))
184. train\_mis\_class += 1
185. if train\_mis\_class >= int(len(output\_training) / 2.0):
186. good = False
187. if good and do\_test:
188. for i in range(len(input\_testing)):
189. test\_c = classify(input\_testing[i], input\_training, output\_training, lang\_list, b,
190. k, verbose, c)
191. if test\_c != output\_testing[i]:
192. if verbose:
193. print("Misclassificaion: input {}, output {:d},"
194. "expected {:d}".format(input\_testing, test\_c, output\_testing[i]))
195. test\_mis\_class += 1
196. if test\_mis\_class >= int(len(output\_testing) / 2.0):
197. good = False
198. return good, train\_mis\_class, test\_mis\_class
199. # plotContours(input\_training,output,lang\_list,b,K,labelContours,labelPoints)
200. # -- Plot the contours of the decision boundary and
201. # +ve/-ve margins of a trained nonlinear SVM. Also plots
202. # points from the training set (x,y) and t.
203. #
204. # N.B. Must be a 2-d classification problem (clearly
205. # can't plot contours of a higher-dimensional problem).
206. #
207. #
208. # Arguments:
209. #
210. # input\_training -- A list of vectors (lists) representing
211. # the input training patterns for this
212. # problem.
213. #
214. # e.g., [[0,0],[0,1],[1,0],[1,1]], the set
215. # of binary training patterns.
216. #
217. # output -- A list of desired outputs. These must
218. # be values in the set {-1,+1}. If there
219. # are n input vectors in x, there must be
220. # exactly n values in t.
221. #
222. # e.g., [-1,-1,-1,1] -- the outputs
223. # for a 2-input AND function.
224. #
225. # lang\_list -- A list of Lagrange multipliers.
226. #
227. # b -- The classifier bias, as generated by makeB.
228. #
229. # -- A Kernel function.
230. #
231. # label\_contours -- Controls whether or not the contour
232. # lines for the decision boundary and margins are
233. # labelled or not. If False (the default), they
234. # are not. If set to the string 'auto', labels
235. # are automatically applied (which can result
236. # in visually unappealing plots, as the labels
237. # can end up hhidden behind data points). If
238. # set to string 'manual', this flag tells the
239. # routine to allow the user to place contour
240. # labels interactively.
241. #
242. # labelPoints -- Controls whether or not the routine
243. # plots the (x,y) and expected values of training
244. # points. Defaults to False.
245. def plot\_contours(input\_training, output, lang\_list=None, b=None, k=rbf\_kernel,
246. label\_contours=False,
247. label\_points=False,
248. min\_range=-0.6, max\_range=1.6, step=0.05, c=1000000.0,
249. plot\_new\_data=False, new\_x=None, new\_t=None):
250. """Plot contours of activation function for a 2-d classifier, e.g. 2-input XOR."""
251. assert len(input\_training) == len(output)
252. assert len(input\_training[0]) == 2
253. lang\_list, b = set\_multipliers\_and\_bias(input\_training, output, lang\_list, b, k, c)
254. xs = arange(min\_range, max\_range + step / 2.0, step)
255. ys = arange(min\_range, max\_range + step / 2.0, step)
256. als = array([[activation([y, x], input\_training, output, lang\_list, b, k) for y in ys] for
257. x in xs])
258. cs = plt.contour(xs, ys, als, levels=(-1.0, 0.0, 1.0), linewidths=(1, 2, 1),
259. colors=('blue', '#40e040', 'red'))
260. if plot\_new\_data:
261. for i, t in enumerate(new\_t):
262. if t < 0:
263. col = 'blue'
264. else:
265. col = 'red'
266. if label\_points:
267. plt.text(new\_x[i][0] + 0.1, new\_x[i][1], "%s: %d" % (new\_x[i], t),
268. color=col)
269. plt.plot([new\_x[i][0]], [new\_x[i][1]], marker='o', color=col)
270. else:
271. for i, t in enumerate(output):
272. if t < 0:
273. col = 'blue'
274. else:
275. col = 'red'
276. if label\_points:
277. plt.text(input\_training[i][0] + 0.1, input\_training[i][1], "%s: %d" %
278. (input\_training[i], t), color=col)
279. plt.plot([input\_training[i][0]], [input\_training[i][1]], marker='o', color=col)
280. if label\_contours == 'manual':
281. plt.clabel(cs, fontsize=9, manual=True)
282. elif label\_contours == 'auto':
283. plt.clabel(cs, fontsize=9)
284. plt.show()
285. # make\_p -- Generates the P matrix for a kernel SVM problem. See the
286. # comments associated with generateLambdas below for a
287. # discussion of the form and role of the P matrix.
288. def make\_p(xs, ts, k):
289. """Make the P matrix given the list of training vectors, desired outputs and
290. kernel."""
291. n = len(xs)
292. assert n == len(ts)
293. p = matrix(0.0, (n, n), tc='d')
294. for i in range(n):
295. for j in range(n):
296. p[i, j] = ts[i] \* ts[j] \* k(xs[i], xs[j])
297. return p
298. # def activation -- Calculate the "activation level" of the classifier in
299. # response to a given input. This is the input to the
300. # step nonlinearity that is the classifier output.
301. #
302. # y = b + sum\_i Ls[i]\*Ts[i]\*K(Xs[i],X)
303. #
304. # for nonzero Lagrange multipliers.
305. # Used by routines "classify" and "plotContours".
306. #
307. # Arguments:
308. #
309. # x -- Input point ot be classified (a list).
310. # input\_training -- List of training points (list of lists).
311. # output -- List of training (i.e. desired) outputs.
312. # lang\_list -- List of Lagrange multipliers for problem.
313. # b -- Bias for problem.
314. # k -- Kernel function.
315. def activation(x, input\_training, output, lang\_list, b, k):
316. """Return activation level of a point X = [x1,x2,....] given
317. training vectors, training (i.e., desired) outputs, Lagrange
318. multipliers, bias and kernel."""
319. y = b
320. for i in range(len(output)):
321. if lang\_list[i] >= 1e-10:
322. y += lang\_list[i] \* output[i] \* k(input\_training[i], x)
323. return y
324. # set\_multipliers\_and\_bias -- A utility to set up the Lagrange
325. # multipliers and bias for a problem if they are not
326. # already established in the calling routine.
327. #
328. # Arguments:
329. #
330. # input\_training -- List of training points (list of lists).
331. # output -- List of training (i.e. desired) outputs.
332. # lang\_list -- List of Lagrange multipliers for problem. If this
333. # is None on the call, multipliers will be generated.
334. # b -- Bias for problem. If this is None on the call, a
335. # bias will be generated.
336. # k -- Kernel function.
337. def set\_multipliers\_and\_bias(input\_training, output, lang\_list=None, b=None,
338. k=rbf\_kernel, c=1000000.0):
339. if lang\_list is None:
340. status, lang\_list = make\_lambdas(input\_training, output, k, c)
341. if status != "optimal": raise Exception("Can't find Lambdas")
342. # print("Lagrange multipliers:", lang\_list)
343. if b is None:
344. b = make\_b(input\_training, output, lang\_list, k, c)
345. # print("Bias:", b)
346. return lang\_list, b
347. # soft margin
348. k = rbf\_kernel
349. title = ''
350. contours = True
351. min\_range = -5
352. max\_range = 5
353. step = 0.05
354. c = 1.0
355. status, lang\_list = make\_lambdas(training\_input, training\_output, k, c)
356. """Run a test problem."""
357. if status == "optimal":
358. b = make\_b(training\_input, training\_output, lang\_list, k, c)
359. # print(" bias:", b)
360. Passed, softmargin\_training\_missed, softmargin\_testing\_missed =
361. test\_classifier(training\_input, training\_output,
362. lang\_list, b, k, c=c,
363. verbose=False,
364. do\_test=True,
365. input\_testing=testing\_input,
366. output\_testing=testing\_output)
367. if Passed:
368. print(" Check PASSED")
369. if contours:
370. if title:
371. t = title
372. else:
373. t = ""
374. plt.figure(t, figsize=(6, 6))
375. plot\_contours(training\_input, training\_output, lang\_list, b, k, False, False,
376. min\_range, max\_range, step)
377. plot\_contours(training\_input, training\_output, lang\_list, b, k, False, False,
378. min\_range, max\_range, step,
379. plot\_new\_data=True, new\_x=testing\_input, new\_t=testing\_output)
380. else:
381. print(" Check FAILED: Classifier does not work correctly on training inputs for
382. a soft margin")
383. print("\n\n")
384. print("Soft margin training misclassification " + str(softmargin\_training\_missed))
385. print("Soft margin testing misclassification " + str(softmargin\_testing\_missed))
386. print("\n\n")
387. # hard margin
388. k = rbf\_kernel
389. title = ''
390. contours = True
391. minRange = -5
392. maxRange = 5
393. step = 0.05
394. c = 1000000000.0
395. status, lang\_list = make\_lambdas(training\_input, training\_output, k, c)
396. """Run a test problem."""
397. if status == "optimal":
398. b = make\_b(training\_input, training\_output, lang\_list, k, c)
399. # print(" bias:", b)
400. Passed, hardmargin\_training\_missed, hardmargin\_testing\_missed =
401. test\_classifier(training\_input, training\_output,
402. lang\_list, b, k, verbose=False,
403. c=c,
404. do\_test=True,
405. input\_testing=testing\_input,
406. output\_testing=testing\_output)
407. if Passed:
408. print(" Check PASSED")
409. if contours:
410. if title:
411. t = title
412. else:
413. t = ""
414. plt.figure(t, figsize=(6, 6))
415. plot\_contours(training\_input, training\_output, lang\_list, b, k, False, False,
416. min\_range, max\_range, step)
417. plot\_contours(training\_input, training\_output, lang\_list, b, k, False, False,
418. min\_range, max\_range, step,
419. plot\_new\_data=True, new\_x=testing\_input, new\_t=testing\_output)
420. else:
421. print(" Check FAILED: Classifier does not work correctly on training inputs for
422. a soft margin")
423. print("\n\n")
424. print("Hard margin training data misclassification " +
425. str(hardmargin\_training\_missed))
426. print("Hard margin testing data misclassification " + str(hardmargin\_testing\_missed))

Graph 1: Soft-Margin Training Data

Graph 2: Soft-Margin Testing Data



Graph 3: Hard-Margin Training Data



Graph 4: Hard-Margin Testing Data