Title: Assignment 2: Develop a program in Python to implement a soft-margin, kernel- based, Support Vector Machine (SVM) Classifier. Use a Radial Basis Function kernel with $\sigma^2 = 0.25$. Test your program using a provided dataset from the Sulis site and two values of the C parameter, C = 1 and C = 10^6. Use the Python CVXOPT library to implement the QP-solver in your program

Author: Oisin Watkins: 15156176

Date: 10/11/18

Introduction

As stated in the title, the goal of this second assignment is to create two different Support Vector Machines (SVM's) with two different softness parameter values (the two values listed for C) with the aim of comparing the accuracy of each machine on both the training data and testing data. Comparing the results from both datasets will lead to a discussion on "Overfitting"/ "Overtraining" SVM's and some conclusions about what overfitting leads to.

Coding an entire SVM from scratch is quiet a difficult task. Along with the CVXOPT library for python, example code was given on the Sulis site which implements a purely hard margin SVM. This original code, "kernelsvm.py", formed the basis of the code which was implemented to achieve the assignment goal. This python script defined various functions and listed multiple examples of the use of each function. For this project, only the following functions were required:

1. rbfKernel -> This applied the Radial Basis Function to the input vectors.
2. makeLambdas -> Function calculates the support vectors, or the Lambda values, for a particular classification problem using the QP solver from CVXOPT
3. makeB -> Calculates the bias given the Lambda values and the input and output values for training
4. makeP -> Makes the P matrix required for the QP solver
5. classify -> Performs classification of an input based on the training data and the Lagrange multipliers
6. testClassifier -> Tests the performance of a classifier on training data by running classify on every point
7. plotContours -> Plots the decision boundary, the +1 region and the -1 region
8. activation -> Returns the activation level of a point
9. setupMultipliersAndBias -> If no Lagrange multipliers or bias are given to the function this will make new ones.

In following sections details will be given as to what issues these functions have with regards creating a soft margin machine with them, and how said issues can be addressed. Other issues, such as plotting data and analysing performance will also be discussed.

Code:

This section will detail each of the functions listed in the introduction, as well as detailing the changes made to each, then providing a copy of the new function.

rbfKernel:

➔ Very little needed changing in this function, the functionality is all in order as it is. The only change made was to default the σ^2 value to 0.25 as required by the assignment. This only means that the sigma value does not have to be specified in all subsequent functions.

```
def rbfKernel(v1, v2, sigma2=0.25):
    assert len(v1) == len(v2)
    assert sigma2 >= 0.0
    mag2 = sum(map(lambda x, y: (x - y) * (x - y), v1, v2))   ## Squared mag of diff.
    return exp(-mag2 / (2.0 * sigma2))
```

makeLambdas:

- ➔ It is within this function that the soft margin functionality of the SVM is implemented. By default this function returns a tuple containing the status of the QP solver's attempts to find a set of Lagrange multipliers to solve the Wolfe Dual equation. If successful, the Lagrange multipliers are returned in the value Ls.
- ➔ Changing this function required first accepting the softness parameter, C, as a parameter and setting it to default to 1.0. Next the "h" column vector had to be doubled in length, from the length of the training output vector to twice this length. The second half of the column is filled with the C value. Next the "G" matrix had to also be doubled in length, from an n x n to a 2n x n matrix, where n is the length of the training output vector. The upper half of G is the negative identity matrix, the lower half is the identity. The rest of the function performs as normal.

```
def makeLambdas(Xs, Ts, K=rbfKernel, C=1.0):
    "Solve constrained maximaization problem and return list of l's."
    P = makeP(Xs, Ts, K)  ## Build the P matrix.
    ## print(P)                    ## Debugging display of P matrix.
    n = len(Ts)
    q = matrix(-1.0, (n, 1))  ## This builds an n-element column
    ## vector of -1.0's (note the double-
    ## precision constant).
    h = matrix(0.0, (2 * n, 1))  ## 2n-element column vector of zeros.
    for i in range(n):  ## Set values from index n to end
        h[i + n] = C  ## equal to the softness parameter

    G = matrix(0.0, (2 * n, n))  ## These lines generate G, a
    # G[::(n+1)] = -1.0            ## 2n x n matrix with -1.0's on its
    for i in range(2 * n):  ## first diagonal and 1.0 on its second diagonal.
        for j in range(n):
            if i < n:
                if j == i:
                    G[i, j] = -1.0
            if i >= n:
                if j == (i - n):
                    G[i, j] = 1.0

    A = matrix(Ts, (1, n), tc='d')  ## A is an n-element row vector of
    ## training outputs.

    ##
    ## Now call "qp". Details of the parameters to the call can be
    ## found in the online cvxopt documentation.
    ##
    r = solvers.qp(P, q, G, h, A, matrix(0.0))  ## "qp" returns a dict, r.
    ##
    ## print(r)                    ## Dump entire result dictionary
    ##                             ## to terminal.
    ##
    ## Return results. Return a tuple, (Status,Ls).  First element is
    ## a string, which will be "optimal" if a solution has been found.
    ## The second element is a list of Lagrange multipliers for the problem,
    ## rounded to six decimal digits to remove algorithm noise.
    ##
    Ls = [round(l, 6) for l in list(r['x'])]  ## "L's" are under the 'x' key.
return (r['status'], Ls)
```

makeP:

➔ Makes the P matrix for the QP solver. No changes made here.

```
def makeP(xs, ts, K):
    """Make the P matrix given the list of training vectors,
       desired outputs and kernel."""
    N = len(xs)
    assert N == len(ts)
    P = matrix(0.0, (N, N), tc='d')
    for i in range(N):
        for j in range(N):
            P[i, j] = ts[i] * ts[j] * K(xs[i], xs[j])
    return P
```

makeB:

➔ Makes the bias for the SVM. Only changed to accept the softness parameter and to feed this value into all subsequent function calls.

```
def makeB(Xs, Ts, Ls=None, K=rbfKernel, C=1.0):
    "Generate the bias given Xs, Ts and (optionally) Ls and K"
    ## Note 0.0 bias parameter in call to setup (can't setup bias
    ## within bias setup routine, would lead to infinite regress.
    Ls, dummyB = setupMultipliersAndBias(Xs, Ts, Ls, 0.0, K, C)
    ## Calculate bias as average over all support vectors (non-zero
    ## Lagrange multipliers).
    sv_count = 0
    b_sum = 0.0
    for n in range(len(Ts)):
        if Ls[n] >= 1e-10:  ## 1e-10 for numerical stability.
            sv_count += 1
            b_sum += Ts[n]
            for i in range(len(Ts)):
                if Ls[i] >= 1e-10:
                    b_sum -= Ls[i] * Ts[i] * K(Xs[i], Xs[n])

    return b_sum / sv_count
```

classify:

➔ This function classifies an input vector into +1 and -1 categories. Once again, the only change needed for this function was to allow it to accept the C values as a parameter and add this value to all relevant function calls

```
def classify(x, Xs, Ts, Ls=None, b=None, K=rbfKernel, verbose=True, C=1.0):
    "Classify an input x into {-1,+1} given support vectors, outputs and L."
    Ls, b = setupMultipliersAndBias(Xs, Ts, Ls, b, K, C)
    ## Do classification.  y is the "activation level".
    y = activation(x, Xs, Ts, Ls, b, K)
    if verbose:
        print("{} {:8.5f}  -->".format(x, y), end=' ')
        if y > 0.0:
            print("+1")
        elif y < 0.0:
            print("-1")
        else:
            print("0  (ERROR)")
    if y > 0.0:
        return +1
    elif y < 0.0:
        return -1
    else:
        return 0
```

testClassifier:

➔ This function originally tested the performance of a classifier on the training data. This functionality has remained, as it is useful. However, added to it is the ability to test previously unseen data, accepting the C value as a parameter as well as now having 3 return values: “good” = has the classifier passed the test; “train_Misclass” = no. misclassifications on training data; “test_Misclass” = no. misclassifications on testing data. Regardless of whether or not new data is given to this function, it will always test the performance of the SVM on the training data.

```
def testClassifier(Xs, Ts, test_x=None, test_t=None, do_Test=False, Ls=None, b=None, K=rbfKernel, verbose=True, C=1.0):
    "Test a classifier specifed by Lagrange mults, bias and kernel on all Xs/Ts pairs."
    assert len(Xs) == len(Ts)
    Ls, b = setupMultipliersAndBias(Xs, Ts, Ls, b, K, C)
    ## Do classification test.
    good = True
    train_MisClass = 0
    test_MisClass = 0
    for i in range(len(Xs)):
        c = classify(Xs[i], Xs, Ts, Ls, b, K, verbose, C)
        if c != Ts[i]:
            if verbose:
                print("Misclassification: input {}, output {:d}, "
                      "expected {:d}".format(Xs[i], c, Ts[i]))
            train_MisClass += 1
    if train_MisClass >= int(len(Ts) / 2.0):
        good = False

    print('Training status is: ' + str(good))

    if do_Test and good and test_x != None and test_t != None:
        for i in range(len(test_x)):
            test_C = classify(test_x[i], Xs, Ts, Ls, b, K, verbose, C)
            if test_C != test_t[i]:
                if verbose:
                    print("Misclassification: input {}, output {:d}, "
                          "expected {:d}".format(Xs[i], c, Ts[i]))
                test_MisClass += 1
        if test_MisClass >= int(len(test_t) / 2.0):
            good = False
    return good, train_MisClass, test_MisClass
```

plotContours:

➔ This function plots the decision boundary, the +1 region and the -1 region of any particular SVM. This function originally couldn’t plot data other than the training data. The ability to plot previously unseen data without changing the 3 original contours was the main change made to this function. Like most others it was also given the C value as a parameter.

```
def plotContours(Xs, Ts, Ls=None, b=None, K=rbfKernel, labelContours=False, labelPoints=False, minRange=-0.6,
                 maxRange=1.6, step=0.05, C=1.0, plotNewData=False, newX=None, newT=None):
    "Plot contours of activation function for a 2-d classifier, e.g. 2-input XOR."
    assert len(Xs) == len(Ts)
    assert len(Xs[0]) == 2  ## Only works with a 2-d classifier.
    Ls, b = setupMultipliersAndBias(Xs, Ts, Ls, b, K, C)
    ## Build activation level array.
    xs = arange(minRange, maxRange + step / 2.0, step)
    ys = arange(minRange, maxRange + step / 2.0, step)
    als = array([[activation([y, x], Xs, Ts, Ls, b, K) for y in ys] for x in xs])
    CS = plt.contour(xs, ys, als, levels=(-1.0, 0.0, 1.0), linewidths=(1, 2, 1), colors=('blue', '#40e040', 'red'))
    if ~plotNewData and newX is None and newT is None:
        for i, t in enumerate(Ts):
            if t < 0:
                col = 'blue'
            else:
                col = 'red'
            if labelPoints:
                ## print("Plotting %s (%d) as %s"%(Xs[i],t,col))
                plt.text(Xs[i][0] + 0.1, Xs[i][1], "%s: %d" % (Xs[i], t), color=col)
            plt.plot([Xs[i][0]], [Xs[i][1]], marker='o', color=col)
    elif plotNewData:
        for i, t in enumerate(newT):
            if t < 0:
                col = 'blue'
            else:
                col = 'red'
            if labelPoints:
                ## print("Plotting %s (%d) as %s"%(Xs[i],t,col))
                plt.text(newX[i][0] + 0.1, newX[i][1], "%s: %d" % (newX[i], t), color=col)
            plt.plot([newX[i][0]], [newX[i][1]], marker='o', color=col)

    ## Generate labels for contours if flag 'labelContours' is set to
    ## strings 'manual' or 'auto'.  Manual is manual labelling, auto is
    ## automatic labelling (which can mess up if hidden behind data
    ## points).
    if labelContours == 'manual':
        plt.clabel(CS, fontsize=9, manual=True)
    elif labelContours == 'auto':
        plt.clabel(CS, fontsize=9)
    plt.show()
```

activation:

➔ The “activation” function returns the activation level of any given point given the Lagrange multipliers, training inputs and outputs and bias. This function did not need to be changed

```
def activation(X, Xs, Ts, Ls, b, K):
    """Return activation level of a point X = [x1,x2,....] given
       training vectors, training (i.e., desired) outputs, Lagrange
       multipliers, bias and kernel."""
    y = b
    for i in range(len(Ts)):
        if Ls[i] >= 1e-10:
            y += Ls[i] * Ts[i] * K(Xs[i], X)
    return y
```

setupMultipliersAndBias:

➔ This function acts as a type of failsafe; if no Lagrange multipliers or bias a provided to this function it will create new ones. This is why it is called at the beginning of nearly every function. The only changes made here allowed for the incorporation of the C value into all function calls within setupMultipliersAndBias.

```
def setupMultipliersAndBias(Xs, Ts, Ls=None, b=None, K=rbfKernel, C=1.0):
    ## No Lagrange multipliers supplied, generate them.
    if Ls == None:
        status, Ls = makeLambdas(Xs, Ts, K, C)
        ## If Ls generation failed (non-seperable problem) throw exception
        if status != "optimal": raise Exception("Can't find Lambdas")
        print("Lagrange multipliers:", Ls)
    ## No bias supplied, generate it.
    if b == None:
        b = makeB(Xs, Ts, Ls, K, C)
        print("Bias:", b)
    return Ls, b
```

The Remainder of the Script:

What remains of the code imports the training and testing data and then creates, trains and tests two independent SVM's, one being soft-margined (C=1) and one being hard-margined ($C=10^6$). The process is straightforward; Create a machine using one of the C values. This script runs the soft margin machine first. Then provided the QP solver found an optimal solution, create a bias value. The characteristics of this device are now saved into the Ls and b values. Using these values, run testClassifier on both the training and testing data. This will return the status of the classifier (if it passed the test or not) and two numbers corresponding to how many misclassifications occurred in training and how many occurred in testing. These are printed to the screen. Next both the training and testing points are plotted over the decision boundary and contours marking the +/-1 regions. Once the plotting is finished, a new machine is made with $C=10^6$. All the same testing is performed, the same numbers printed to the screen and the same plots created.

```
training_File = open('training-dataset-aut-2017.txt', 'r')
train = [line.split(',') for line in training_File.readlines()]
train_Vals, train_Xs, train_Ts = [[0]] * len(train), [[0, 0]] * len(train), [[0]] * len(train)

testing_File = open('testing-dataset-aut-2017.txt', 'r')
test = [line.split(',') for line in testing_File.readlines()]
test_Vals, test_Xs, test_Ts = [[0]] * len(test), [[0, 0]] * len(test), [[0]] * len(test)

for i in range(len(train)):
    train_Vals[i] = train[i][0].split()
    train_Xs[i] = train_Vals[i][0:2]
    train_Ts[i] = train_Vals[i][2]

for i in range(len(test)):
    test_Vals[i] = test[i][0].split()
    test_Xs[i] = test_Vals[i][0:2]
    test_Ts[i] = test_Vals[i][2]

for i in range(len(test)):
    test_Ts[i] = float(test_Ts[i])
    for j in range(2):
        test_Xs[i][j] = float(test_Xs[i][j])

for i in range(len(train)):
    train_Ts[i] = float(train_Ts[i])
    for j in range(2):
        train_Xs[i][j] = float(train_Xs[i][j])

""" Now the training and testing data have been saved as floats """
```

```
K = rbfKernel
title = ''
contours = True
minRange = -5
maxRange = 5
step = 0.05
C = 1.0

status, Ls = makeLambdas(train_Xs, train_Ts, K, C)
print("  Result status:", status)
print("  L vector:", Ls)
```

```
if status == "optimal":
    b = makeB(train_Xs, train_Ts, Ls, K, C)
    print("  bias:", b)
    trained, train_Misclass, test_Misclass = testClassifier(train_Xs, train_Ts, test_Xs, test_Ts, do_Test=True, Ls=Ls,
                                                            b=b, K=K, verbose=False, C=C)
    if trained:
        print("  Check PASSED")
        if contours:
            if title:
                t = title
            else:
                t = ""
            plt.figure(t, figsize=(6, 6))
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C)
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C, plotNewData=True,
                         newX=test_Xs,
                         newT=test_Ts)
    else:
        print("  Check FAILED: Classifier does not work correctly on inputs")
        if contours:
            if title:
                t = title
            else:
                t = ""
            plt.figure(t, figsize=(6, 6))
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C)
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C, plotNewData=True,
                         newX=test_Xs,
                         newT=test_Ts)
    print("\n\n")
    print("    Soft margin machine misclassified on training data " + str(train_Misclass) + " times\n")
    print("    Soft margin machine misclassified on testing data " + str(test_Misclass) + " times\n")
    print("\n\n")
```

```
C = 1000000.0

status, Ls = makeLambdas(train_Xs, train_Ts, K, C)
print("  Result status:", status)
print("  L vector:", Ls)
```

```
status, Ls = makeLambdas(train_Xs, train_Ts, K, C)
print("  Result status:", status)
print("  L vector:", Ls)

if status == "optimal":
    b = makeB(train_Xs, train_Ts, Ls, K, C)
    print("  bias:", b)
    trained, train_Misclass, test_Misclass = testClassifier(train_Xs, train_Ts, test_Xs, test_Ts, do_Test=True, Ls=Ls,
                                                             b=b, K=K, verbose=False, C=C)
    if trained:
        print("  Check PASSED")
        if contours:
            if title:
                t = title
            else:
                t = ""
            plt.figure(t, figsize=(6, 6))
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C)
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C, True, test_Xs,
                         test_Ts)
    else:
        print("  Check FAILED: Classifier does not work correctly on inputs")
        if contours:
            if title:
                t = title
            else:
                t = ""
            plt.figure(t, figsize=(6, 6))
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C)
            plotContours(train_Xs, train_Ts, Ls, b, K, False, False, minRange, maxRange, step, C, True, test_Xs,
                         test_Ts)
    print("\n\n")
    print("    Hard margin machine misclassified on training data " + str(train_Misclass) + " times\n")
    print("    Hard margin machine misclassified on testing data " + str(test_Misclass) + " times\n")
    print("\n\n")
```

Results:

Giving a fair and honest account of how this machine performs involves discussing the printed output of this script as well as viewing the plotted outputs from testing.

➔ The Printed Output:

Copied here is the entire command line output after running the script.

C:\Users\Fierce-PC\Anaconda3\python.exe "D:/Semester 7/CE4708 - Artificial Intelligence/Projects/SVM.py"

	pcost	dcost	gap	pres	dres
0:	-3.5185e+01	-2.1242e+02	8e+02	2e+00	8e-16
1:	-2.7204e+01	-1.2871e+02	1e+02	4e-16	7e-16
2:	-3.1141e+01	-4.6276e+01	2e+01	7e-16	7e-16
3:	-3.3146e+01	-3.6512e+01	3e+00	1e-15	6e-16
4:	-3.3808e+01	-3.4451e+01	6e-01	1e-15	6e-16
5:	-3.3965e+01	-3.4094e+01	1e-01	1e-15	4e-16
6:	-3.4001e+01	-3.4010e+01	9e-03	8e-16	5e-16

7: -3.4004e+01 -3.4004e+01 3e-04 1e-15 6e-16

8: -3.4004e+01 -3.4004e+01 1e-05 2e-15 6e-16

Optimal solution found.

Result status: optimal

L vector: [0.606579, 0.599871, 0.849971, 0.548477, 1.0, 0.665069, 1.0, 1.0, 0.799057, 0.0, 1.9e-05, 1.0, 0.001585, 0.058799, 0.0, 1e-06, 0.466352, 0.379469, 1.0, 1.0, 0.311318, 0.362665, 1.0, 1.0, 1.0, 0.82238, 0.297113, 1e-06, 1.0, 0.0, 1.0, 0.335579, 1.0, 3e-06, 0.0, 0.999998, 0.0, 0.767597, 1e-06, 0.289405, 0.773843, 0.071023, 0.752459, 0.403548, 0.73413, 0.0, 0.508179, 0.382412, 0.000165, 1.0, 0.484688, 1.0, 0.0, 0.855588, 1e-06, 0.615112, 0.109157, 0.0, 0.0, 0.940972, 0.0, 0.608914, 2e-06, 0.26352, 0.0, 0.652727, 1.0, 1.0, 3e-06, 0.124253, 1.0, 1e-06, 0.000409, 0.378344, 0.99913, 0.0, 1.0, 0.150965, 1.0, 0.599525, 0.0, 1.0, 0.894839, 0.0, 1e-06, 0.331082, 2e-06, 0.0, 0.714331, 0.422286, 0.0, 0.0, 1.0, 0.614258, 1.0, 1.0, 2e-06, 1.0, 1e-06, 0.059489]

bias: -0.09352109255748722

Training status is: True

Check PASSED

Soft margin machine misclassified on training data 9 times

Soft margin machine misclassified on testing data 163 times

pcost dcost gap pres dres

0: 3.7924e+12 -2.6825e+13 3e+13 8e-10 3e-10

1: 1.0775e+12 -2.9555e+12 4e+12 6e-10 3e-10

2: 2.2228e+11 -4.4656e+11 7e+11 9e-10 2e-10

3: 3.8117e+10 -6.7061e+10 1e+11 4e-10 1e-10

4: 8.9398e+09 -1.5706e+10 2e+10 4e-10 1e-10

5: 1.4353e+09 -2.3397e+09 4e+09 9e-11 8e-11

6: 2.1600e+08 -2.6250e+08 5e+08 7e-11 6e-11

7: 3.1065e+07 -3.5513e+07 7e+07 5e-11 2e-11

8: 4.4213e+06 -5.0434e+06 9e+06 5e-11 6e-12

9: 6.1666e+05 -7.3502e+05 1e+06 3e-12 2e-12

10: 7.9528e+04 -1.1285e+05 2e+05 3e-13 1e-12

11: 5.2128e+03 -2.1288e+04 3e+04 8e-13 6e-13

12: -5.0321e+03 -8.3726e+03 3e+03 4e-12 3e-13

13: -6.3633e+03 -7.1714e+03 8e+02 1e-12 4e-13

14: -6.6991e+03 -6.8061e+03 1e+02 3e-13 4e-13

15: -6.7429e+03 -6.7505e+03 8e+00 4e-13 3e-13

16: -6.7459e+03 -6.7479e+03 2e+00 2e-13 4e-13

17: -6.7469e+03 -6.7473e+03 4e-01 6e-14 4e-13

18: -6.7471e+03 -6.7471e+03 1e-02 3e-14 4e-13

19: -6.7471e+03 -6.7471e+03 2e-04 2e-12 3e-13

Optimal solution found.

Result status: optimal

L vector: [0.0, 875.931042, 0.0, 952.391668, 907.018952, 2e-06, 556.298526, 0.0, 2.010434, 0.0, 5.4e-05, 1e-06, 0.0, 0.0, 0.0, 0.0, 1.253169, 2e-06, 22.421726, 2.167659, 1e-06, 0.0, 3020.822533, 120.14932, 21.638947, 0.0, 1e-06, 0.0, 584.879918, 0.0, 1e-06, 33.434821, 49.042793, 0.0, 0.0, 866.315477, 0.0, 9.459986, 3.3e-05, 1e-06, 1.698097, 0.0, 1.903393, 0.0, 0.0, 1e-06, 1.016382, 3e-06, 0.0, 22.640893, 1.440606, 0.0, 0.0, 91.862649, 3e-06, 0.0, 0.29972, 0.0, 9e-06, 2471.474719, 0.0, 2e-06, 1.5e-05, 1e-06, 0.0, 2e-06, 6.277938, 0.0, 0.000301, 0.0, 837.525349, 0.0, 1.109147, 0.0, 2.560559, 0.0, 6.370667, 0.0, 636.178635, 1e-06, 0.0, 1e-06, 2.31874, 0.0, 3e-06, 71.094409, 6.4e-05, 40.199931, 1.811772, 0.0, 0.0, 0.0, 35.248093, 278.67124, 45.938692, 27.369374, 1.4e-05, 883.922091, 6e-06, 1e-06]

bias: -2.06838674893135

Training status is: True

Check PASSED

Hard margin machine misclassified on training data 0 times

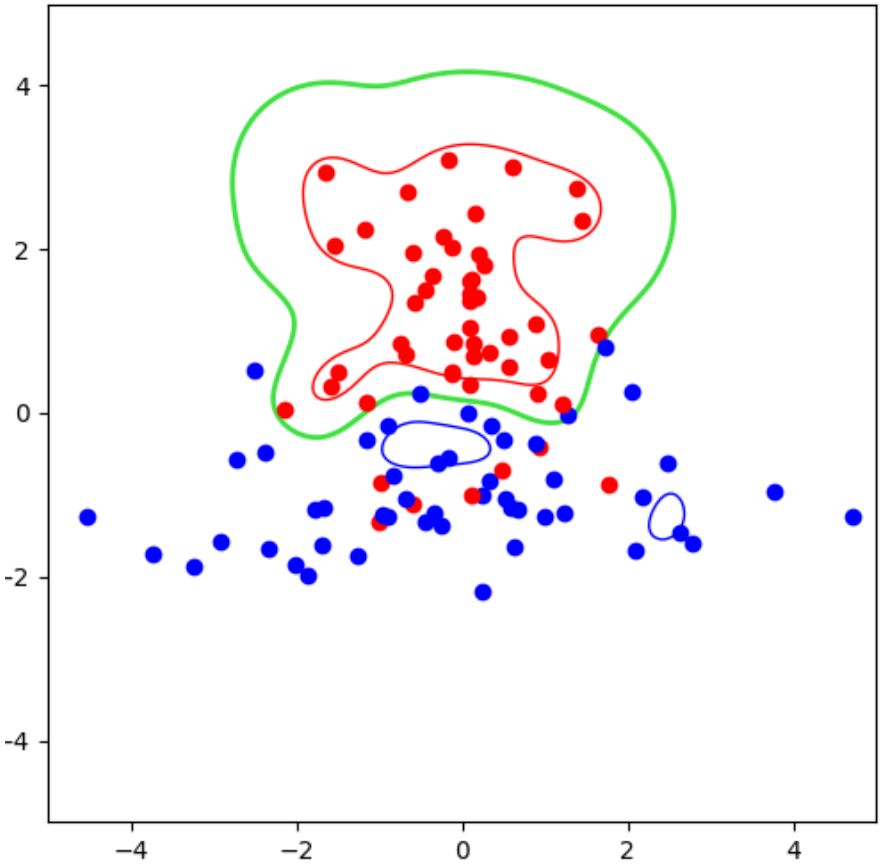
Hard margin machine misclassified on testing data 248 times

Process finished with exit code 0

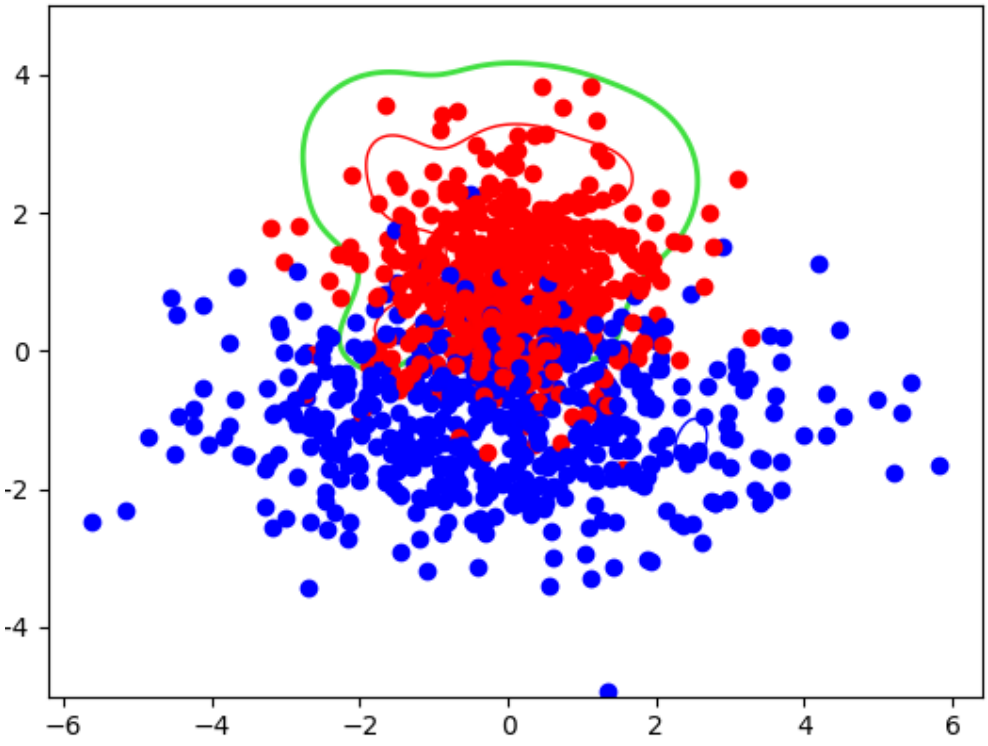
What's most important to note here is that although the soft-margin machine misclassified the training data more frequently that the hard-margin machine (technically infinitely more, as the hard margin machine never misclassified the training data) the hard-margin misclassified the testing data 248 times, whereas the soft-margin machine only misclassified 163 of the testing inputs. This is typical of overtrained SVM's; they perform perfectly on the training inputs, but as they have such a strange (and in this case discontinuous) decision boundary it often leads to far more errors on data which it hasn't seen before.

The next thing to consider are the output plots from the script:

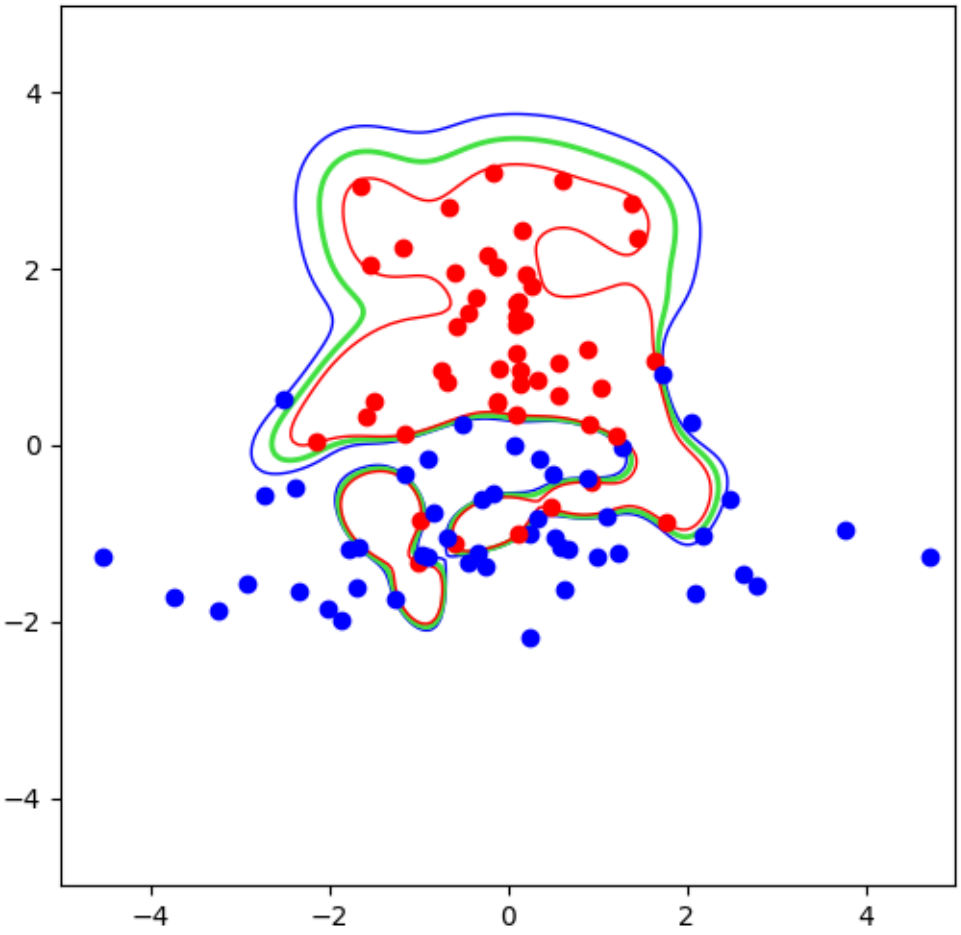
➔ Training: Soft-Margin



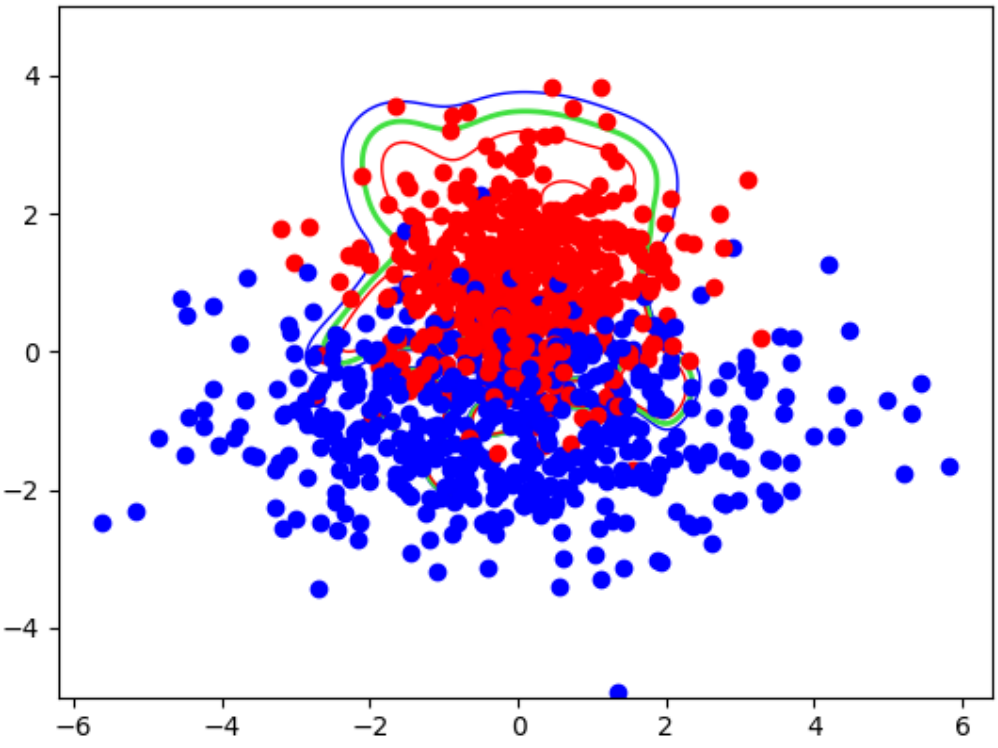
➔ Testing: Soft-Margin



➔ Training: Hard-Margin



➔ Testing: Hard-Margin



Comparing the training plots, it's easy to see how the Hard-Margin machine carefully cuts up the input space such that no training input is misclassified, resulting in a rather odd-looking and discontinuous decision boundary, whereas the Soft-Margin machine results in a more forgiving continuous decision boundary. This does mean that the soft-margin machine misclassifies the training data somewhat, however this machine performs considerably better on testing data than the Hard-Margin machine. This is because the Hard-Margin machine has essentially learned off the training data and is now less fit to classify these datasets. Having created such a "perfect" decision boundary, it has become less sensitive to the testing data's distribution of ± 1 points, whereas the Soft-Margin machine has remained more attuned to the testing data's distribution of ± 1 points.

Conclusions:

Now having created and tested a Soft- and Hard-Margin machine, as well as comparing their performances, one conclusion that can be drawn is that hard-margin machines perform far worse on datasets that overlap in any way. Typically a hard-margin machine will attempt to cut the input space such that the decision boundary is never incorrect, however this can cause issues when the distribution of points in each class is overlapping. In such cases soft-margin machines perform better as they tend to remain more sensitive to the distributions of points in each class, rather than attempting to learn off the training inputs.