

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel → Restart) and then **run all cells** (in the menubar, select Cell → Run All).

Make sure you fill in any place that says YOUR CODE HERE or "YOUR ANSWER HERE", as well as your name and collaborators below:

```
NAME = "Ayush Koirala"  
ID = "St122802"
```

▼ Lab 07: Support Vector Machines

Today we'll look at the SVM maximum margin classification problem and how we can implement the optimization in Python.

We'll use the cvxopt quadratic programming optimizer in Python.

Later in the lectures, we'll see that more specialized algorithms such as Sequential Minimal Optimization implemented by the machine learning libraries are more effective for large SVM problems.

▼ Linearly separable case: Direct solution using quadratic programming

If we assume that the data are linearly separable, we can use the following setup for the optimization:

- The data are pairs $(\mathbf{x}^{(i)}, y^{(i)})$ with $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \{-1, 1\}$.
- The hypothesis is

$$h_{\mathbf{w},b}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b > 0 \\ -1 & \text{otherwise} \end{cases}$$

- The objective function is

$$\mathbf{w}^*, b^* = \operatorname{argmax}_{\mathbf{w},b} \gamma,$$

where γ is the minimum geometric margin for the training data:

$$\gamma = \min_i \gamma^{(i)}$$

and $\gamma^{(i)}$ is the geometric margin for training example i , i.e., the signed distance of $\mathbf{x}^{(i)}$ from the decision boundary, with positive distances indicating that the point is on the correct side of the boundary and negative distances indicating that the point is on the incorrect side of the boundary:

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \right)^\top \mathbf{x}^{(i)} + \frac{b}{\|\mathbf{w}\|} \right).$$

- As we derived in lecture, to find the optimal \mathbf{w}, b according to the objective function above, we can in its place solve the constrained optimization problem

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{subject to} \quad & y^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1, i \in 1..m \end{aligned}$$

So let's do it! In case you don't already have the cvxopt pip package installed:

```
# Run in case there is no cvxopt installed. You can also do this in a separate terminal.
```

```
!https_proxy=http://192.41.170.23:3128 http_proxy=http://192.41.170.23:3128 pip install cvxopt
```

```
Collecting cvxopt
```

```
  Downloading cvxopt-1.3.0-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.7 MB)
```

```
    |████████████████████████████████████████| 12.7 MB 1.6 MB/s eta 0:00:01
```

```
Installing collected packages: cvxopt
```

```
Successfully installed cvxopt-1.3.0
```

```
import numpy as np
import matplotlib.pyplot as plt
import cvxopt
```

Here are some sample data. In this example, we've planted some specific points that will serve as the support vectors so that we know the "correct" answer for the maximum margin hyperplane.

```
Xf = np.matrix([[ 164.939, 163.431, 157.554, 152.785, 156.385, 159.242, 156.281, 164.411, 157.308, 159.579 ],
                 [ 56.927, 48.945, 45.678, 45.969, 40.896, 46.848, 42.225, 42.380, 42.150, 49.739 ]]).T;
Xm = np.matrix([[ 168.524, 171.597, 179.469, 176.063, 180.939, 177.011, 183.284, 180.549, 176.502, 185.392 ],
                 [ 64.353, 61.793, 74.552, 69.851, 74.730, 75.871, 79.170, 79.753, 64.923, 73.665 ]]).T;
X = np.concatenate([Xf, Xm],0);
y = np.concatenate([-np.matrix(np.ones([10,1])),np.matrix(np.ones([10,1]))]);
```

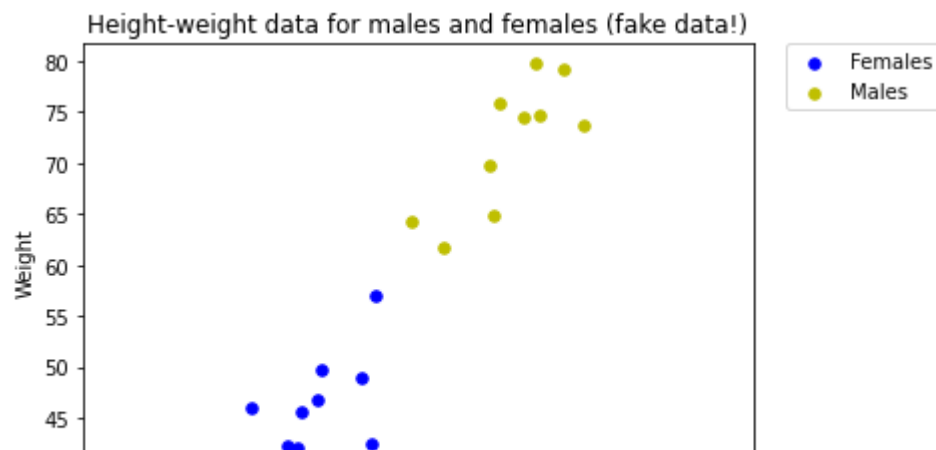
```
print(X.shape)
print(y.shape)
```

```
(20, 2)
(20, 1)
```

```
# Plot the data
def plot_mf(Xf,Xm):
    axes = plt.axes()
    females_series = plt.scatter(np.array(Xf[:,0]), np.array(Xf[:,1]), s=30, c='b', marker='o', label='Females')
    males_series = plt.scatter(np.array(Xm[:,0]), np.array(Xm[:,1]), s=30, c='y', marker='o', label='Males')
    axes.set_aspect('equal', 'datalim')
    plt.xlabel('Height')
    plt.ylabel('Weight')
    plt.title('Height-weight data for males and females (fake data!)')
    plt.legend(handles=[females_series, males_series], bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    return axes

def plot_w(axes, w, b):
    ylim = axes.get_ylim()
    xlim = axes.get_xlim()
    p1 = (xlim[0], - (w[0,0] * xlim[0] + b) / w[1,0])
    p2 = (xlim[1], - (w[0,0] * xlim[1] + b) / w[1,0])
    plt.plot((p1[0],p2[0]), (p1[1],p2[1]), 'r-')

plot_mf(Xf, Xm)
plt.show()
```



▼ Exercise 1 (in lab): Linearly separable data (total of 25 points)

Take the example data and SVM optimization code using cvxopt from the exercise in lecture. Verify that you can find the decision boundary for such "easy" cases. Show your results in your lab report. We'll go step by step below.

▼ Exercise 1.1 Create SVM function using cvopt (5 points)

You'll want to ensure that Q is symmetric by setting it to $(Q + Q^T)/2$. Then create an `args` variable containing Q and c , each converted from numpy to cvxopt using `cvxopt.matrix(numpy_matrix)`. If A is supplied, you should add A and B to the list of arguments, and if E is supplied, you should add E and d to the list of arguments. Finally, you can call `cvxopt.solvers.qp(*args)`. If the status returned is `optimal` return the valid solution else return `None`.

```
def cvxopt_solve_qp(Q, c, A=None, B=None, E=None, d=None):
    # Fill your code value in 'None'
    # Some 'None' can be avoided.
    Q_new = .5 * (Q + Q.T)
    args = [cvxopt.matrix(Q_new), cvxopt.matrix(c)]
    if A is not None:
        args.extend([cvxopt.matrix(A), cvxopt.matrix(B)])
    if E is not None:
        args.extend([cvxopt.matrix(E), cvxopt.matrix(d)])
```

```

sol = cvxopt.solvers.qp(*args)
if sol is not None and 'optimal' not in sol['status']:
    return None
x = np.array(sol['x']).reshape((Q.shape[1],))
# YOUR CODE HERE
#raise NotImplementedError()
return x

# Test function: Do not remove
Q_test = np.array([[1, 0],[0, 0]])
c_test = np.zeros([2])
A_test = np.array([[15., 1.],[14., 1.], [13., 1], [-5.,-1.],[-6.,-1.],[-8,-1]])
B_test = -np.ones([6])
try:
    x_test = cvxopt_solve_qp(Q_test, c_test, A_test, B_test)
    print('x_test:', x_test)
except:
    assert False, "cvxopt_solve_qp is incorrect"

print("success!")
# End Test function

```

```

      pcost      dcost      gap    pres    dres
0:  2.8800e-02  1.0464e+00  1e+01  1e+00  3e+01
1:  1.8859e-01 -8.5852e-01  1e+00  2e-01  3e+00
2:  1.6523e-01  3.7852e-02  1e-01  5e-16  2e-15
3:  8.2300e-02  7.2465e-02  1e-02  3e-16  2e-15
4:  8.0043e-02  7.9915e-02  1e-04  7e-16  4e-15
5:  8.0000e-02  7.9999e-02  1e-06  3e-16  8e-16
6:  8.0000e-02  8.0000e-02  1e-08  5e-16  2e-15
Optimal solution found.
x_test: [-0.40000001  4.20000012]
success!

```

Expected result (or look-alike):

```

pcost dcost gap pres dres
0: 2.8800e-02 1.0464e+00 1e+01 1e+00 3e+01

```

```

1: 1.8859e-01 -8.5852e-01 1e+00 2e-01 3e+00
2: 1.6523e-01 3.7852e-02 1e-01 4e-16 4e-16
3: 8.2300e-02 7.2465e-02 1e-02 6e-16 3e-15
4: 8.0043e-02 7.9915e-02 1e-04 7e-16 2e-16
5: 8.0000e-02 7.9999e-02 1e-06 2e-16 1e-15
6: 8.0000e-02 8.0000e-02 1e-08 8e-16 1e-15

```

Optimal solution found.

```
x_test: [-0.40000001 4.20000012]\
```

▼ Exercise 1.2: Find Q, c, A, B for input into cvxopt_solve_qp function (10 points)

Now you should use the `cvxopt_solve_qp` function to find \mathbf{w} and \mathbf{b} for the data \mathbf{X} , \mathbf{y} prepared earlier. You'll need to set up numpy arrays for \mathbf{Q} , \mathbf{c} , \mathbf{A} , and \mathbf{B} that when passed to the QP solver will give us the optimal decision boundary.

```

Q = None
c = None
A = None
B = None
# YOUR CODE HERE
#raise NotImplementedError()
def QcAB_matrix(X, y):
    diagonal_el = np.ones(shape=(X.shape[1])).tolist()
    diagonal_el.append(0)
    Q = np.diag(diagonal_el)
    # print(Q)
    c = np.zeros(shape=X.shape[1]+1)
    # print(c)
    A = -np.multiply(y, np.insert(X, X.shape[1], 1, axis = 1))
    # print(X.shape)
    # print(A.shape)
    # print(A)
    B = -np.ones(shape=X.shape[0])

    return Q,c,A,B

```

```

Q,c,A,B = QcAB_matrix(X, y)
print(Q.shape,c.shape,A.shape,B.shape)
def wts_cvxopt(Q, c, A, B):
    x = cvxopt_solve_qp(Q, c, A, B)
    x = np.array(x)
    x = x.squeeze()

    return x
x = wts_cvxopt(Q, c, A, B)
#x = cvxopt_solve_qp(Q, c, A, B)

```

```

(3, 3) (3,) (20, 3) (20,)
      pcost      dcost      gap      pres      dres
0:  1.4721e-03  6.5053e+00  5e+01  2e+00  4e+02
1:  1.0012e-02 -4.7161e+00  1e+01  6e-01  1e+02
2:  2.6180e-02 -4.8172e+00  7e+00  2e-01  6e+01
3:  3.9767e-02 -4.5363e-01  5e-01  1e-02  2e+00
4:  3.5404e-02  1.8200e-02  2e-02  5e-15  9e-14
5:  3.1392e-02  3.0877e-02  5e-04  5e-15  6e-13
6:  3.1250e-02  3.1245e-02  5e-06  6e-15  2e-13
7:  3.1249e-02  3.1248e-02  5e-08  5e-15  2e-13
Optimal solution found.

```

```

print('Q:\n', Q)
print('c:\n', c)
print('A:\n', A[7:13])
print('B:\n', B)
print('x:\n', x)
# Test function: Do not remove
assert Q.shape == (3, 3) and Q[2,2] == Q[0,1] and Q[2,0] == 0 and Q[0,0] == Q[1,1] and Q[0,0] == 1, 'Q value is incorrect'
assert c.shape == 3 or c.shape == (3,) or c.shape == (3,1), 'Size of c is incorrect'
assert np.all((c == 0)), 'c value is incorrect'
assert A.shape == (20,3), 'Size of A is incorrect'
assert np.max(A[:,2]) == 1 and np.min(A[:,2]) == -1, 'A value is incorrect'
assert not np.array_equal(np.round(A[:,0:2],1), np.round(X,1)), 'A value is incorrect'
assert np.array_equal(np.round(x,1), np.round([0.16001143, 0.19207647, -38.32646165],1)), 'x value is incorrect'

```

```
print("success!")
# End Test function
```

```
Q:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 0.]]
c:
[0. 0. 0.]
A:
[[ 164.411  42.38    1.    ]
 [ 157.308  42.15    1.    ]
 [ 159.579  49.739    1.    ]
 [-168.524 -64.353   -1.    ]
 [-171.597 -61.793   -1.    ]
 [-179.469 -74.552   -1.    ]]
B:
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
-1. -1.]
x:
[ 0.16001143  0.19207647 -38.32646165]
```

Expected Result (or look-alike):

```
Q:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 0.]]
c:
[0. 0. 0.]
A:
[[ 164.411 42.38 1.]
 [ 157.308 42.15 1.]
 [ 159.579 49.739 1.]
 [-168.524 -64.353 -1.]
 [-171.597 -61.793 -1.]
 [-179.469 -74.552 -1.]]
```


B:

[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]

x:

[0.16001143 0.19207647 -38.32646165]\

▼ Exercise 1.3: Use x from above to find w and b (5 points)

Now you've got the optimal parameter vector for the QP problem. Extract \mathbf{w} and b , and be sure to normalize \mathbf{w} to unit length and b accordingly.

```
w = None
b = None
# YOUR CODE HERE
#raise NotImplementedError()
def wts_bias(x):
    w = x[:-1] / np.sqrt(np.sum(np.square(x[:-1])))
    b = x[-1] / np.sqrt(np.sum(np.square(x[:-1])))
    w = w.reshape(-1,1)
    return w,b

w, b = wts_bias(x)

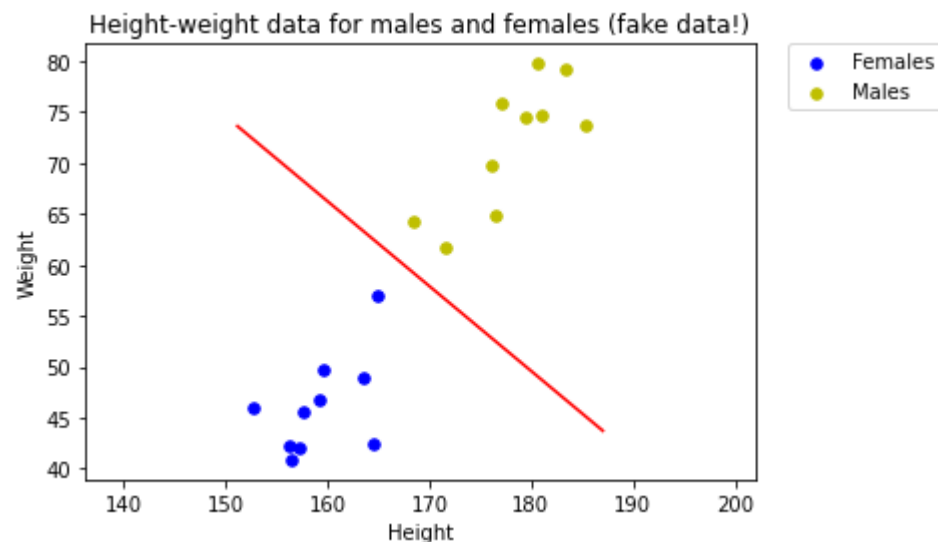
print('Optimal w: [%f %f] b: %f' % (w[0,0],w[1,0],b))

axes = plot_mf(Xf, Xm)
plot_w(axes, w, b)
plt.show()

# Test function: Do not remove
assert w.shape == 2 or w.shape ==(2,) or w.shape == (2,1), 'Size of w is incorrect'
assert w[0] > 0 and w[1] > 0 and w[0] <= 1 and w[1] <= 1, 'w value is incorrect'
assert isinstance(b, (float, int)), 'Type of b is incorrect'
assert b < 0, 'b value is incorrect'
```

```
print("success!")
# End Test function
```

Optimal w: [0.640061 0.768324] b: -153.309495



success!

Expected result (Or look-alike):

Optimal w: [0.640061 0.768324] b: -153.309495



Here's some code to get the prediction accuracy of a linear SVM with parameters \mathbf{w} and b already identified:

```
def predict_linear(X,w,b):
    s = X@w+b
    s[s >= 0] = 1
    s[s < 0] = -1
    return s
y_pred = predict_linear(X,w,b)
```

```
accuracy = np.sum(y_pred==y)/y.size
print(accuracy)
```

1.0

▼ Exercise 2 (in lab): non-separable data (5 points)

Take the example of the annulus from the logistic regression lab. Verify that cvxopt cannot find a decision boundary for this case. Show your results in your lab report.

Note: You don't need to separate the data into train/test subsets.

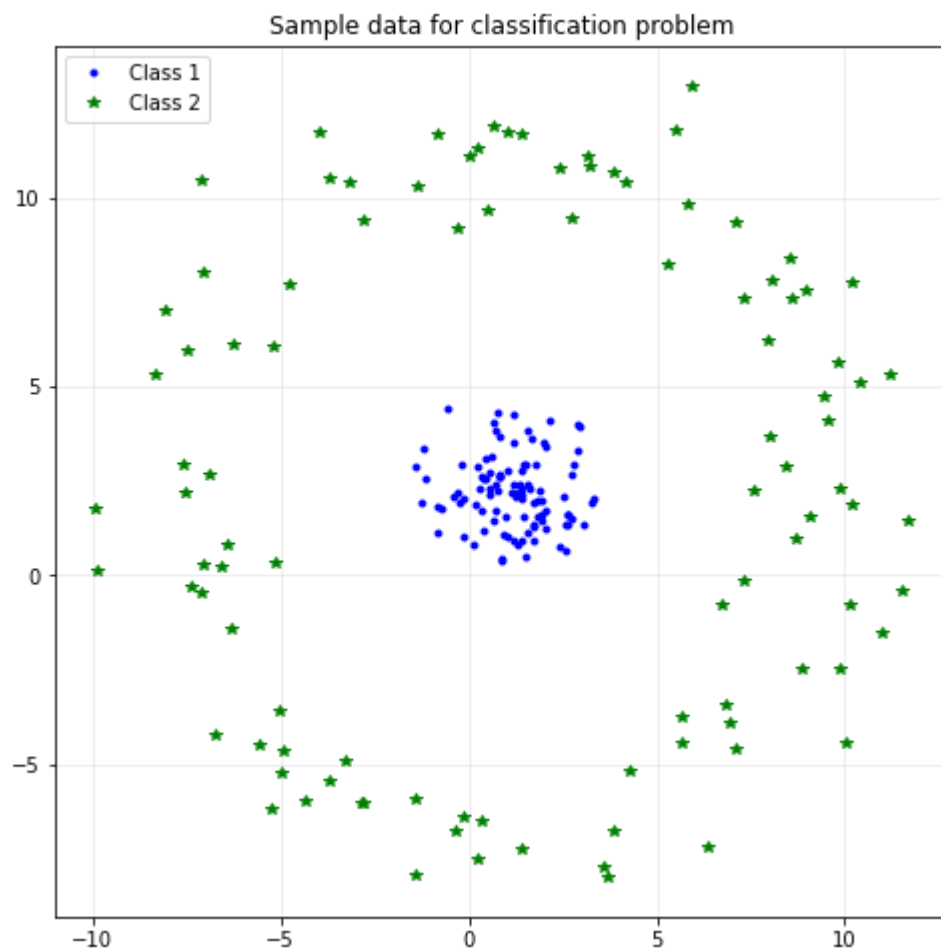
```
# Generate data for class 1
mu_1 = np.array([1.0, 2.0])
sigma_1 = 1
num_sample = 100
cov_mat = np.matrix([[sigma_1,0],[0,sigma_1]])
X1 = np.random.multivariate_normal(mean= mu_1, cov=cov_mat, size = num_sample)

# Generate data for class 2
angle = np.random.uniform(0, 2*np.pi, num_sample)
d = np.random.normal(np.square(3*sigma_1),np.square(.5*sigma_1),num_sample)
X2 = np.array([X1[:,0] + d*np.cos(angle), X1[:,1] + d*np.sin(angle)]).T

# Combine X1 and X2 into single dataset
X_annulus = np.concatenate([X1, X2],axis = 0)
y_annulus = np.append(-np.ones(num_sample),np.ones(num_sample))[:,np.newaxis]
```

```
def plot_annulus(X1, X2):
    ax = plt.axes()
    plt.title('Sample data for classification problem')
    plt.grid(axis='both', alpha=.25)
    plt.plot(X1[:,0],X1[:,1],'b.', label = 'Class 1')
    plt.plot(X2[:,0],X2[:,1],'g*', label = 'Class 2')
    plt.legend(loc=2)
    ax.set_aspect('equal', 'datalim')
    return ax
```

```
fig1 = plt.figure(figsize=(8,8))  
plot_annulus(X1, X2)  
plt.show()
```



▼ Exercise 2.1: Fill in Q , c , A , B for the annulus data and find w and b

Here you just have to repeat the same process as in Exercise 1 but with the annulus data generated above.

```

# Try to use try/catch to get output
get_error = False
try:
    Q = None
    c = None
    A = None
    B = None
    x = None
    w = None
    b = None
    # YOUR CODE HERE
    #raise NotImplementedError()
    Q,c,A,B = QcAB_matrix(X_annulus, y_annulus)
    x = wts_cvxopt(Q, c, A, B)
    w, b = wts_bias(x)
    print(w,b)
    output_str = 'Optimal w: [%f %f] b: %f' % (w[0,0],w[1,0],b)
    get_error = False
except Exception as e:
    output_str = e
    get_error = True

```

	pcost	dcost	gap	pres	dres
0:	5.3378e-05	1.9953e+02	2e+02	2e+00	9e-14
1:	4.9266e-09	4.7031e+02	5e+00	1e+00	1e-13
2:	4.6773e-13	4.2569e+04	5e+00	1e+00	2e-11
3:	4.6746e-17	3.7606e+08	4e+02	1e+00	2e-07
4:	4.6746e-21	3.3213e+14	4e+06	1e+00	9e-02
5:	4.6747e-25	2.9334e+22	3e+12	1e+00	9e+06
6:	1.7579e-26	2.5357e+32	9e+20	1e+00	1e+17
7:	6.2082e-26	4.7388e+40	2e+29	1e+00	3e+25
8:	4.8796e-26	2.0275e+48	7e+36	1e+00	1e+33
9:	5.2943e-26	2.0492e+56	7e+44	1e+00	8e+40
10:	5.1334e-26	8.2093e+63	3e+52	1e+00	9e+48
11:	5.5289e-26	7.6531e+71	3e+60	1e+00	3e+56
12:	5.2598e-26	3.3004e+79	1e+68	1e+00	1e+64
13:	5.2316e-26	3.2019e+87	1e+76	1e+00	1e+72
14:	5.3209e-26	1.3401e+95	5e+83	1e+00	2e+79
15:	5.2276e-26	1.2983e+103	5e+91	1e+00	4e+87
16:	5.3931e-26	5.5205e+110	2e+99	1e+00	2e+95

```

17:  5.1806e-26  5.3267e+118  2e+107  1e+00  3e+102
18:  5.4461e-26  2.2750e+126  8e+114  1e+00  2e+111
19:  5.1570e-26  2.1852e+134  8e+122  1e+00  6e+118
20:  5.4985e-26  9.4038e+141  3e+130  1e+00  3e+126
21:  5.1395e-26  8.9741e+149  3e+138  1e+00  2e+134
22:  5.5491e-26  3.8935e+157  1e+146  1e+00  2e+142
23:  5.1220e-26  3.6924e+165  1e+154  1e+00  1e+150
24:  5.6043e-26  1.6165e+173  6e+161  1e+00  inf
25:  5.1096e-26  1.5211e+181  5e+169  1e+00  inf
26:  5.6727e-26  6.7424e+188  2e+177  1e+00  inf
27:  5.1023e-26  6.2797e+196  2e+185  1e+00  inf
28:  5.7680e-26  2.8389e+204  1e+193  1e+00  inf
29:  5.0779e-26  2.6047e+212  9e+200  1e+00  inf
30:  5.8753e-26  1.1888e+220  4e+208  1e+00  inf
31:  5.1363e-26  1.0507e+228  4e+216  1e+00  inf
32:  5.9577e-26  4.7856e+235  2e+224  1e+00  inf
33:  5.3954e-26  3.9321e+243  1e+232  1e+00  inf
34:  5.8770e-26  1.8083e+251  6e+239  1e+00  inf
35:  5.5936e-26  1.4311e+259  5e+247  1e+00  inf
36:  5.5976e-26  6.6257e+266  2e+255  1e+00  inf
37:  5.3757e-26  5.6862e+274  2e+263  1e+00  inf
38:  5.4439e-26  2.4820e+282  9e+270  1e+00  inf
39:  5.0426e-26  2.3373e+290  8e+278  1e+00  inf

```

```
print(output_str)
```

```
# Test function: Do not remove
```

```
assert Q.shape == (3, 3) and Q[2,2] == Q[0,1] and Q[2,0] == 0 and Q[0,0] == Q[1,1] and Q[0,0] == 1, 'Q value is incorrect'
```

```
assert c.shape == 3 or c.shape == (3,) or c.shape == (3,1), 'Size of c is incorrect'
```

```
assert np.all((c == 0)), 'c value is incorrect'
```

```
assert A.shape == (200,3), 'Size of A is incorrect'
```

```
assert str(output_str) == 'domain error' or "'NoneType' object is not subscriptable" or get_error, 'Output incorrect'
```

```
print("success!")
```

```
# End Test function
```

```
domain error
success!
```

Expected result: Show that there was an error and that the optimization did not converge.

▼ Generalized Lagrangian optimization for SVMs

Now we consider the generalized Lagrangian for the SVM. This technique is suitable for solving problems of the form

$$\begin{aligned} \min_{\mathbf{w}} \quad & f(\mathbf{w}) \\ \text{subject to} \quad & g_i(\mathbf{w}) \leq 0, i \in 1..k \\ & h_i(\mathbf{w}) = 0, i \in 1..l \end{aligned}$$

The generalized Lagrangian is

$$\mathcal{L}(\mathbf{w}, \alpha, \beta) = f(\mathbf{w}) + \sum_{i=1}^k \alpha_i g_i(\mathbf{w}) + \sum_{i=1}^l \beta_i h_i(\mathbf{w}),$$

which has been cleverly arranged to be equal to $f(\mathbf{w})$ whenever \mathbf{w} satisfies the constraints and ∞ otherwise.

Primal and dual Lagrangian problems

The primal problem is to find

$$p^* = \min_{\mathbf{w}} \theta_{\mathcal{P}}(\mathbf{w}) = \min_{\mathbf{w}} \max_{\alpha, \beta, \alpha_i \geq 0} \mathcal{L}(\mathbf{w}, \alpha, \beta)$$

and the dual problem is to find

$$d^* = \max_{\alpha, \beta, \alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta) = \max_{\alpha, \beta, \alpha_i \geq 0} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \alpha, \beta).$$

If f is convex, the g_i 's are affine, the h_i 's are convex, and the g_i 's are strictly feasible, it turns out that the solutions to the primal and dual problem are the same, and the KKT conditions hold:

$$\begin{aligned} \frac{\partial}{\partial w_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) &= 0, i \in 1..n \\ \frac{\partial}{\partial \beta_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) &= 0, i \in 1..l \\ \alpha_i^* g_i(\mathbf{w}^*) &= 0, i \in 1..k \\ g_i(\mathbf{w}^*) &\leq 0, i \in 1..k \\ \alpha_i^* &\geq 0, i \in 1..k \end{aligned}$$

Solving the dual Lagrangian problem

The dual problem (finding α maximizing the Lagrangian with the optimal \mathbf{w} turns out to be easiest to solve and gives us the ability to use kernels other than the linear (inner product) kernel.

We first solve for \mathbf{w} assuming fixed α and β (we don't have equality constraints though, so no need for β).

We need to rewrite the SVM constraints in the necessary form with $g_i(\mathbf{w}) = 0$. For the SVM, the constraints are

$$g_i(\mathbf{w}, b) = -y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) + 1 \geq 0.$$

Using that definition of $g_i(\mathbf{w}, b)$, we obtain the Lagrangian

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^m \alpha_i \left[y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) - 1 \right]$$

Taking the gradient of \mathcal{L} with respect to \mathbf{w} and setting it to 0, we obtain

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0,$$

which gives us

$$\mathbf{w} = \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)}.$$

From $\frac{\partial \mathcal{L}}{\partial b} = 0$, we obtain

$$\sum_{i=1}^m \alpha_i y^{(i)} = 0,$$

which is interesting (think about what it means also considering that $\alpha_i > 0$ only for examples on the margin. Unfortunately it doesn't help us find b ! In any case, we plug this definition for the optimal \mathbf{w} into the original Lagrangian, to obtain

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j (\mathbf{x}^{(i)})^\top \mathbf{x}^{(j)} - b \sum_{i=1}^m \alpha_i y^{(i)}.$$

We already know that the last term is 0, so we get

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle.$$

OK! We've eliminated \mathbf{w} and b from the optimization. Now we just need to maximize \mathcal{L} with respect to α . This gives us the final (dual) optimization problem

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{such that} \quad & \alpha_i \geq 0, i \in 1..m \\ & \sum_{i=1}^m \alpha_i y^{(i)} = 0 \end{aligned}$$

This turns out to be QP again!

Aside: once we solve for α , we obtain \mathbf{w} according to the equation above, then it turns out that the optimal b can be obtained as in the lecture notes.

▼ QP solution to dual problem

We need to negate our objective function to turn the max (SVM formulation) into a min (QP formulation).

For the second term of $W(\alpha)$, first let \mathbf{K} be the kernel matrix with $K_{ij} = \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$. Then $\alpha^\top \text{diag}(\mathbf{y}) \mathbf{K} \text{diag}(\mathbf{y}) \alpha$ gives us the summation in the second term ($\text{diag}(\mathbf{y})$ is just the square diagonal matrix with \mathbf{y} as its diagonal).

The (negated) first term of $W(\alpha)$ can be written in QP form with $\mathbf{c} = [-1 \quad -1 \quad \dots]^\top$.

So that gives us our QP setup:

$$\begin{aligned} \mathbf{Q} &= \text{diag}(\mathbf{y}) \mathbf{K} \text{diag}(\mathbf{y}) \quad \mathbf{c} = \begin{bmatrix} -1 \\ -1 \\ \vdots \end{bmatrix} \\ \mathbf{A} &= -\mathbf{I}_{m \times m} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix} \\ \mathbf{E} &= \mathbf{y}^\top \quad \mathbf{d} = [0]. \end{aligned}$$

OK, now the code:

```

m = X.shape[0];
n = X.shape[1];

# Transform data set so that each attribute has a
# mean of 0 and a standard deviation of 1

def preprocess(X):
    means = X.mean(0);
    scales = 1/np.std(X,0);
    Xh = np.concatenate([X.T,np.ones([1,20])],0);
    Tm = np.matrix(np.eye(3));
    Tm[0:2,2:3] = -X.mean(0).T;
    Ts = np.matrix(np.eye(3));
    Ts[0:2,0:2] = np.diagflat(scales);
    T = Ts*Tm;
    XX = (T * Xh);
    XX = XX[0:2,:].T;
    return XX, T;

# RBF/Gaussian kernel

def gauss_kernel(X):
    sigma = 0.2
    m = X.shape[0];
    K = np.matrix(np.zeros([m,m]));
    for i in range(0,m):
        for j in range(0,m):
            K[i,j] = (X[i,:] - X[j,:]).reshape(1,-1) @ (X[i,:] - X[j,:]).reshape(-1,1)
    K = np.exp(-K/(2*sigma*sigma))
    return K;

def linear_kernel(X):
    m = X.shape[0];
    K = np.matrix(np.zeros([m,m]));
    for i in range(0,m):
        for j in range(0,m):
            K[i,j] = (X[i,:].reshape(1,-1)@X[j,:].reshape(-1,1))
    return K;

```

▼ Exercise 3 (in lab): linearly separable data (15 points)

Take the example data from Exercise 1. Verify that you can use the dual optimization to find the decision boundary for such "easy" cases. Show your results in your lab report.

▼ Exercise 3.1: Find Q, c, A, b, E, d for input into cvxopt_solve_qp function (10 points)

Here you need to set up the QP variables appropriate to get the solution to the dual optimization problem. When we pass them to cvxopt, we should get α^* , the optimal Lagrangian parameters. We know from inspection which examples in the original data will be the support vectors with non-zero α_i 's.

```
Q = None
c = None
A = None
b = None
E = None
d = None

K = linear_kernel(X)
def QcABEd(X, y, K):
    Q = np.diag(np.array(y).squeeze()) @ K @ np.diag(np.array(y).squeeze()) #shape mxm
    print(Q.shape)

    c = -np.ones(shape = (X.shape[0],)) #shape of c -> (m,)
    print(c.shape)

    A = -np.eye(N = X.shape[0]) #shape of A -> (m,m)    its an identity matrix
    print(A.shape)

    B = np.zeros(shape = (X.shape[0],)) #shape (m, )
    print(B.shape)

    E = y.T
    print(E.shape)
```

```

d = np.zeros(shape = (1,))
print(d.shape)

return Q, c, A, B, E, d

Q, c, A, B, E, d = QcABEd(X,y, K)
def alphastar(Q, c, A, B, E, d):
    x = cvxopt_solve_qp(Q, c, A, B, E, d)
    x = np.array(x)
    x = x.squeeze()
    return x

# YOUR CODE HERE
#raise NotImplementedError()
alpha_star = cvxopt_solve_qp(Q, c, A, B, E, d)

(20, 20)
(20,)
(20, 20)
(20,)
(1, 20)
(1,)
      pcost      dcost      gap      pres      dres
0: -2.7646e+00 -4.9725e+00 5e+01 6e+00 2e+00
1: -6.4101e+00 -3.8299e+00 1e+01 2e+00 6e-01
2: -5.0055e+00 -1.2719e+00 7e+00 8e-01 2e-01
3: -5.1552e-02 -4.0648e-02 5e-01 3e-02 1e-02
4: -1.8200e-02 -3.5404e-02 2e-02 8e-18 2e-13
5: -3.0877e-02 -3.1392e-02 5e-04 3e-18 2e-13
6: -3.1245e-02 -3.1250e-02 5e-06 5e-18 2e-13
7: -3.1248e-02 -3.1249e-02 5e-08 6e-18 3e-13
Optimal solution found.

```

```

print('Q rank: %d' % np.linalg.matrix_rank(Q))
print("Optimal alpha:\n", alpha_star)

# Test function: Do not remove
assert Q.shape == (20, 20), 'Size of Q is incorrect'

```

```

assert np.linalg.matrix_rank(Q) == 2, 'Q rank is incorrect'
assert np.all((c == -1)), 'c value is incorrect'
assert A.shape == (20,20), 'Size of A is incorrect'
assert np.all((B == 0)), 'b value is incorrect'
assert np.array_equal(np.round(E,1), np.round(y.T,1)), 'E value is incorrect'
assert d.shape == (1,) or d.shape == 1 or d.shape == (1,1) , 'Size of d is incorrect'
assert np.all((d == 0)), 'd value is incorrect'
assert alpha_star.shape == (20,) or alpha_star.shape == 20 or alpha_star.shape == (20,1), 'Size of alpha_star is incorrect'

print("success!")
# End Test function

```

```

Q rank: 2
Optimal alpha:
[3.12484796e-02 1.13821985e-09 7.68004003e-10 6.22346942e-10
 6.43114906e-10 8.93974185e-10 6.60695981e-10 4.31409827e-10
 6.68557407e-10 1.19689394e-09 1.56332821e-02 1.56151999e-02
 5.00606721e-10 6.71834910e-10 4.89168003e-10 4.93113730e-10
 4.90935291e-10 4.85894510e-10 9.54348935e-10 4.42078646e-10]
success!

```

Expected Result (or look a like):

Q rank: 2

Optimal alpha:

```

[3.12484796e-02 1.13821985e-09 7.68004003e-10 6.22346942e-10
 6.43114906e-10 8.93974184e-10 6.60695980e-10 4.31409827e-10
 6.68557407e-10 1.19689394e-09 1.56332821e-02 1.56151999e-02
 5.00606721e-10 6.71834910e-10 4.89168003e-10 4.93113729e-10
 4.90935291e-10 4.85894509e-10 9.54348935e-10 4.42078646e-10]

```

▼ Exercise 3.2: write get_wb function (5 points)

Next, try to write the `get_wb()` function that, given the optimal α_i 's, returns the corresponding \mathbf{w} and b . If you get stuck, refer to the handout from lecture, but it will be better for you to try to figure it out for yourself.

```

def get_wb(X, y, alpha, K):
    # Find the support vectors
    S = alpha > 1e-6
    XS = None
    yS = None
    alphaS = None
    alphaSyS = None
    w = None
    # Find b
    KS = None
    NS = None
    b = None
    # Normalize w,b
    scalef = None
    w = None
    b = None

    # YOUR CODE HERE
    S = alpha > 1e-6
    XS = X[S]
    yS = y[S]
    alphaS = alpha[S]
    #    print(alphaS)
    alphaSyS = np.multiply(alpha, S.astype(int))
    w = X.T @ np.multiply(alphaSyS.reshape(-1,1), y.reshape(-1,1))
    #    print(w.shape)

    #find b
    KS = linear_kernel(XS)
    #    print(KS)
    NS = XS.shape[0]
    b = 1/NS * np.sum(yS - KS @ np.multiply(alphaS.reshape(-1,1), yS.reshape(-1,1)))

    #Normalize w,b
    scalef = np.sqrt(np.sum(np.square(w)))
    w = w/scalef
    b = b/scalef

```

```
#raise NotImplementedError()
return w,b
```

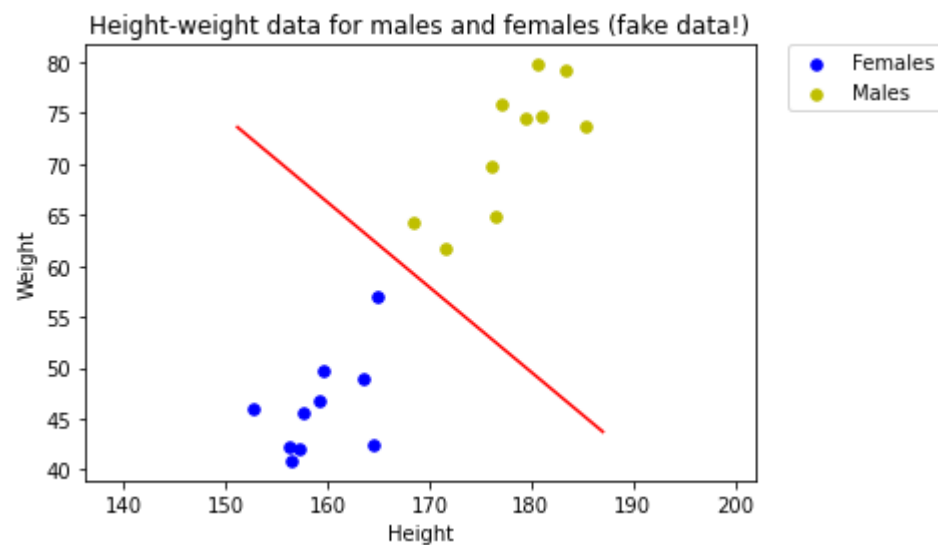
```
# Test function: Do not remove
w,b = get_wb(X, y, alpha_star, K)

print("Optimal w: [%f,%f] b: %f" % (w[0],w[1],b))

axes = plot_mf(Xf, Xm)
plot_w(axes, w, b)
plt.show()

print("success!")
# End test function
```

Optimal w: [0.640062,0.768324] b: -153.309583



success!

Expected Result (Or look-alike):

Optimal w: [0.640062,0.768324] b: -153.309583



```
y_pred = predict_linear(X,w,b)
accuracy = np.sum(y_pred==y)/y.size
print(accuracy)
```

1.0

▼ Exercise 4.1 (in lab): non-separable data, linear kernel (15 points)

Again, take the example of the annulus from the logistic regression lab. Verify that the dual optimization with the linear kernel still cannot find a decision boundary for this case. Show your results in your lab report. As before, you'll want to set up Q , c , A , B , E , and d appropriately but now for the dual optimization problem.

```
(m, n) = X_annulus.shape
print(X_annulus.shape)
K_annulus = linear_kernel(X_annulus);

Q_annulus = None
c = None
A = None
B = None
E = None
d = None

# YOUR CODE HERE
Q_annulus, c, A, B, E, d = QcABEd(X_annulus, y_annulus, K_annulus)
#raise NotImplementedError()
```

```
(200, 2)
(200, 200)
(200,)
(200, 200)
(200,)
(1, 200)
(1,)
```



```
alpha_star_annulus = cvxopt_solve_qp(Q_annulus, c, A, B, E, d)

print("Optimal alpha:", alpha_star_annulus)
# Test function: Do not remove
assert alpha_star_annulus is None, "alpha_star_annulus cannot be calculated."

print("success!")
# End test function
```

```
      pcost      dcost      gap      pres      dres
0: -1.9953e+02 -4.2177e+02 2e+02 2e-14 2e+00
1: -4.7031e+02 -4.7557e+02 5e+00 2e-14 1e+00
2: -4.2569e+04 -4.2574e+04 5e+00 1e-12 1e+00
3: -3.7185e+08 -3.7185e+08 9e+02 9e-08 1e+00
4: -5.2447e+08 -5.2448e+08 1e+03 1e-07 1e+00
Terminated (singular KKT matrix).
Optimal alpha: None
success!
```

```
get_error = False
try:
    w,b = get_wb(X_annulus, y_annulus, alpha_star_annulus, K_annulus)

    output_str = "Optimal w: [%f,%f] b: %f" % (w[0],w[1],b)
    plot_mf(Xf,Xm)
    plot_w(w,b)
    get_error = False
except Exception as e:
    output_str = str(e)
    get_error = True

print(output_str)
# Test function: Do not remove
assert str(output_str) == 'domain error' or "'NoneType' object is not subscriptable" or get_error, 'Output incorrect'

print("success!")
# End Test function
```

```
'>' not supported between instances of 'NoneType' and 'float'
success!
```

Expected Result: Error due to `alpha_star` being `None`.

Exercise 5.1 (in lab): "easy" non-separable data, Gaussian (RBF) kernel with non-overlapping data (10 points)

Now we'll use the Gaussian (radial basis function) kernel instead of the linear kernel implemented in the code above and verify that we can correctly solve the easy non-separable data where a transformation to a feature space makes the data linearly separable. We already know that a simple polar transformation will give us linearly separable data, but it turns out we can achieve the same thing using the RBF kernel.

By the way, you might think about which solution is better for our annulus problem: the explicit transformation or the RBF kernel. See how many support vectors we get before you decide.

```
(m, n) = X_annulus.shape
K_annulus = None
Q_annulus = None
c = None
A = None
B = None
E = None
d = None

# YOUR CODE HERE
K_annulus = gauss_kernel(X_annulus)
print("K_annulus shape", K_annulus.shape)
Q_annulus, c, A, B, E, d = QcABEd(X_annulus, y_annulus, K_annulus)
#raise NotImplementedError()
```

```
K_annulus shape (200, 200)
(200, 200)
```

```
(200,)
(200, 200)
(200,)
(1, 200)
(1,)
```

```
alpha_star_annulus = cvxopt_solve_qp(Q_annulus, c, A, B, E, d)

print('Q rank: %d' % np.linalg.matrix_rank(Q_annulus))
print("Optimal alpha:")
print(alpha_star_annulus[:5])

# Test function: Do not remove
assert np.linalg.matrix_rank(Q_annulus) > 2, "Q rank is incorrect"
assert alpha_star_annulus is not None, "alpha_star_annulus cannot be calculated."

print("success!")
# End test function
```

```
      pcost      dcost      gap      pres      dres
0: -5.1216e+01 -1.4134e+02  9e+01  7e-15  2e+00
1: -5.9281e+01 -6.7206e+01  8e+00  1e-15  3e-01
2: -6.2294e+01 -6.3927e+01  2e+00  2e-14  4e-02
3: -6.2420e+01 -6.2689e+01  3e-01  4e-14  4e-03
4: -6.2445e+01 -6.2476e+01  3e-02  5e-14  1e-16
5: -6.2449e+01 -6.2450e+01  1e-03  3e-14  1e-16
6: -6.2449e+01 -6.2449e+01  4e-05  1e-14  1e-16
Optimal solution found.
Q rank: 200
Optimal alpha:
[6.51094755e-01 1.31888241e-05 8.09516286e-01 8.81162639e-01
 1.22571096e-05]
success!
```

Expected result (or look-alike):

```
pcost dcost gap pres dres
0: -5.0795e+01 -1.3928e+02 9e+01 1e-15 2e+00
1: -5.9073e+01 -6.6466e+01 7e+00 2e-14 3e-01
```

2: -6.2076e+01 -6.3563e+01 1e+00 1e-14 4e-02

3: -6.2184e+01 -6.2423e+01 2e-01 8e-15 5e-03

4: -6.2205e+01 -6.2232e+01 3e-02 3e-14 5e-04

5: -6.2209e+01 -6.2209e+01 7e-04 4e-14 6e-06

6: -6.2209e+01 -6.2209e+01 1e-05 1e-14 7e-08

Optimal solution found.

Q rank: 200

Optimal alpha:

[5.77224260e-01 1.31648362e+00 1.17475881e+00 4.38419640e-01 8.16578932e-07]

With the nonlinear kernel, we no longer have a \mathbf{w} and b in the input space. First, let's revise our prediction method using just support vectors, their α_i 's, and their labels:

```
def predict(x, X, y, alpha):
    s = []
    sigma = 0.2
    for j in range(x.shape[0]):
        ss = 0
        for i in range(X.shape[0]):
            ss += alpha[i]*y[i]*np.exp(-(X[i]-x[j])@(X[i]-x[j]))/(2*sigma*sigma))
        s.append(ss)
    s = np.array(s)
    s[s >= 0] = 1
    s[s < 0] = -1
    return s
```

```
y_pred = predict(X_annulus, X_annulus, y_annulus, alpha_star_annulus)
np.sum(y_annulus == y_pred)/y_annulus.size
```

1.0

Another problem is graphing the decision boundary. Since the decision boundary is in feature space, which is infinite dimensional in the case of the RBF kernel, we can't exactly graph it! Instead, consider the 2-dimensional input space, let's obtain the prediction of the model at each point in a regular grid over the space, then let's plot the resulting mesh on top of the data.

```
x_series = np.linspace(-15, 15, 100)
y_series = np.linspace(-15, 15, 100)

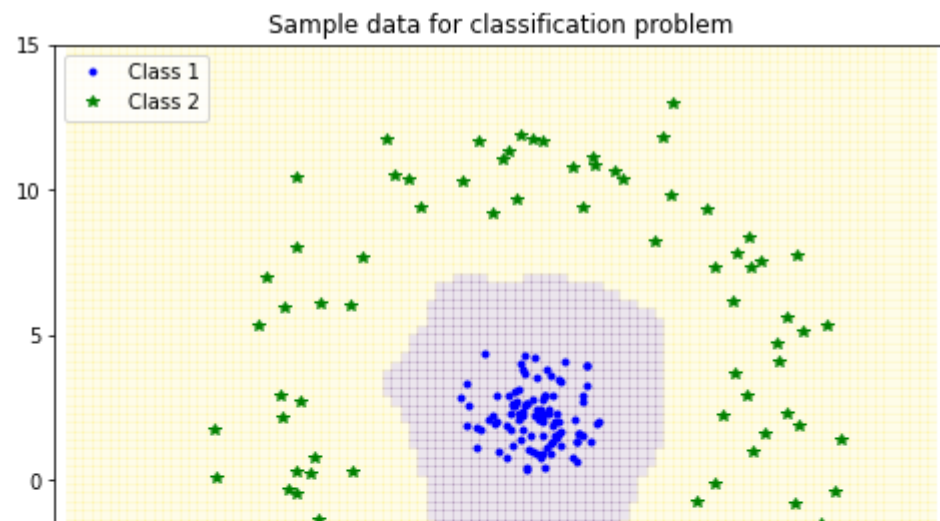
x_mesh, y_mesh = np.meshgrid(x_series, y_series)

x_mesh = x_mesh.reshape(-1, 1)
y_mesh = y_mesh.reshape(-1, 1)

mesh = np.append(x_mesh, y_mesh, axis=1)
y_pred = predict(mesh, X_annulus, y_annulus, alpha_star_annulus)

x_mesh = x_mesh.reshape(100, 100)
y_mesh = y_mesh.reshape(100, 100)
y_pred = y_pred.reshape(100, 100)

fig1 = plt.figure(figsize=(8,8))
ax = plot_annulus(X1, X2)
ax.pcolormesh(x_mesh, y_mesh, y_pred, cmap='viridis', shading='auto', alpha=0.1)
plt.show()
```



Expected Result:

 lab06-3.png



▼ Exercise 6 (take home): more difficult non-separable data

Now find or generate a dataset in which the decision boundary is nonlinear AND the data overlap along that nonlinear boundary. Show that the result.

```
import numpy as np
import matplotlib.pyplot as plt
import cvxopt

np.random.seed(32)

mu_1 = np.array([1.0, 2.0])
sigma_1 = 1
num_sample = 100
cov_mat = np.matrix([[sigma_1, 0], [0, sigma_1]])
X = np.random.multivariate_normal(mean= mu_1, cov=cov_mat, size = num_sample)
```

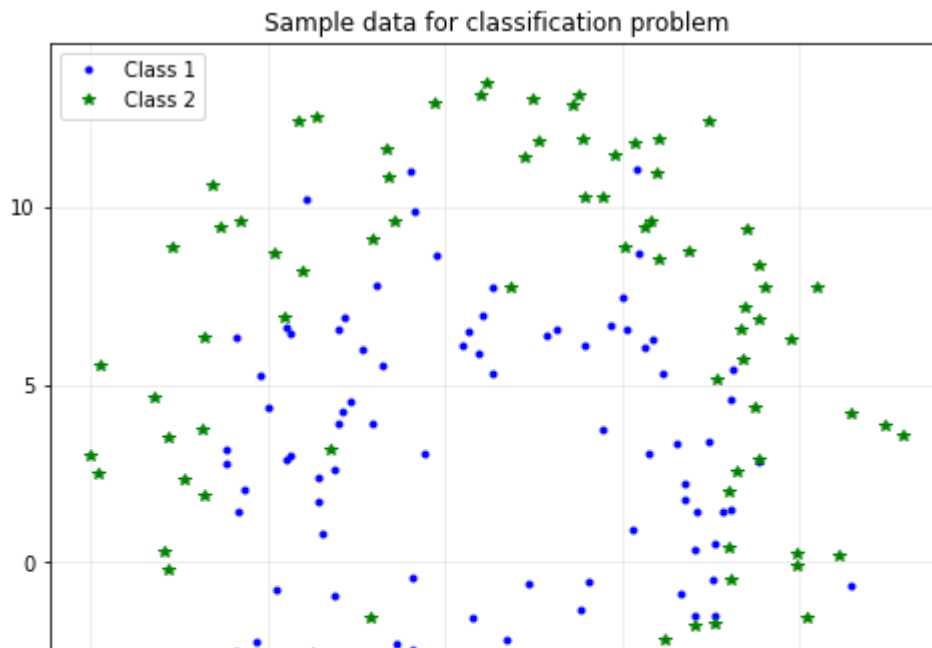
```
# Generate data for class 1
angle = np.random.uniform(0, 2*np.pi, num_sample)
d = np.random.normal(np.square(2.5*sigma_1),np.square(1*sigma_1),num_sample)
X1 = np.array([X[:,0] + d*np.cos(angle), X[:,1] + d*np.sin(angle)]).T

# Generate data for class 2
angle = np.random.uniform(0, 2*np.pi, num_sample)
d = np.random.normal(np.square(3*sigma_1),np.square(1.2*sigma_1),num_sample)
X2 = np.array([X[:,0] + d*np.cos(angle), X[:,1] + d*np.sin(angle)]).T

# Combine X1 and X2 into single dataset
X_annulus = np.concatenate([X1, X2],axis = 0)
y_annulus = np.append(-np.ones(num_sample),np.ones(num_sample))[:,np.newaxis]

def plot_annulus(X1, X2):
    ax = plt.axes()
    plt.title('Sample data for classification problem')
    plt.grid(axis='both', alpha=.25)
    plt.plot(X1[:,0],X1[:,1],'b.', label = 'Class 1')
    plt.plot(X2[:,0],X2[:,1],'g*', label = 'Class 2')
    plt.legend(loc=2)
    ax.set_aspect('equal', 'datalim')
    return ax

fig1 = plt.figure(figsize=(8,8))
plot_annulus(X1, X2)
plt.show()
```



```
def gausskernel(X, sigma):
    m = X.shape[0];
    K = np.matrix(np.zeros([m,m]));
    for i in range(0,m):
        for j in range(0,m):
            K[i,j] = (X[i,:] - X[j,:]).reshape(1,-1) @ (X[i,:] - X[j,:]).reshape(-1,1) # ||X-Z||^2 = (X-Z).T @ (X-Z)
    K = np.exp(-K/(2*sigma*sigma))
    return K;
```

```
def quadparams(X, y, K, c):
    Q = np.diag(np.array(y).squeeze()) @ K @ np.diag(np.array(y).squeeze()) #shape mxm
    C = -np.ones(shape = (X.shape[0],)) #shape of c -> (m,)
    A = np.eye(N = X.shape[0]) #shape of A -> (m,m) its an identity matrix
    B = np.zeros(shape = (X.shape[0],)) #shape (m, )
    B = B + c
    E = y.T
    d = np.zeros(shape = (1,))
    return Q, C, A, B, E, d
```

```
def alphastar(Q, c, A, B, E, d):
```



```

    x = cvxopt_solve_qp(Q, c, A, B, E, d)
    x = np.array(x)
    x = x.squeeze()
    return x

def predict(x, X, y, alpha, sigma): #prediction part in lecture note (my note)
    s = []
    for j in range(x.shape[0]): #loop for each points to predict, here x represent test sets
        ss = 0
        for i in range(X.shape[0]): #loop for each points in training set, X represent training sets
            ss += alpha[i]*y[i]*np.exp(-(X[i]-x[j])@(X[i]-x[j]))/(2*sigma*sigma))
        s.append(ss)
    s = np.array(s)
    s[s >= 0] = 1
    s[s < 0] = -1
    return s

```

```

s = [0.5,1,2,3] #RBF parameter, greater the value less curvature
cc = [0.01, 0.1, 1, 10] #higher c -> more training error
for sigma in s:
    for c in cc:
        K_annulus = gausskernel(X_annulus, sigma)
        print("K_annulus shape", K_annulus.shape)
        Q_annulus, C, A, B, E, d = quadparams(X_annulus, y_annulus, K_annulus, c)

        alpha_star_annulus = cvxopt_solve_qp(Q_annulus, C, A, B, E, d)

        y_pred = predict(X_annulus, X_annulus, y_annulus, alpha_star_annulus, sigma)
        accuracy = np.sum(y_annulus == y_pred)/y_annulus.size
        print(f"Accuracy is for c={c} and sigma={sigma} is: {accuracy}")

#plot
x_series = np.linspace(-15, 15, 100)
y_series = np.linspace(-15, 15, 100)

x_mesh, y_mesh = np.meshgrid(x_series, y_series)

```

```
x_mesh = x_mesh.reshape(-1, 1)
y_mesh = y_mesh.reshape(-1, 1)

mesh = np.append(x_mesh, y_mesh, axis=1)
y_pred = predict(mesh, X_annulus, y_annulus, alpha_star_annulus, sigma)

x_mesh = x_mesh.reshape(100, 100)
y_mesh = y_mesh.reshape(100, 100)
y_pred = y_pred.reshape(100, 100)

fig1 = plt.figure(figsize=(8,8))
ax = plot_annulus(X1, X2)
ax.pcolormesh(x_mesh, y_mesh, y_pred, cmap='viridis', shading='auto', alpha=0.1)
plt.show()
```

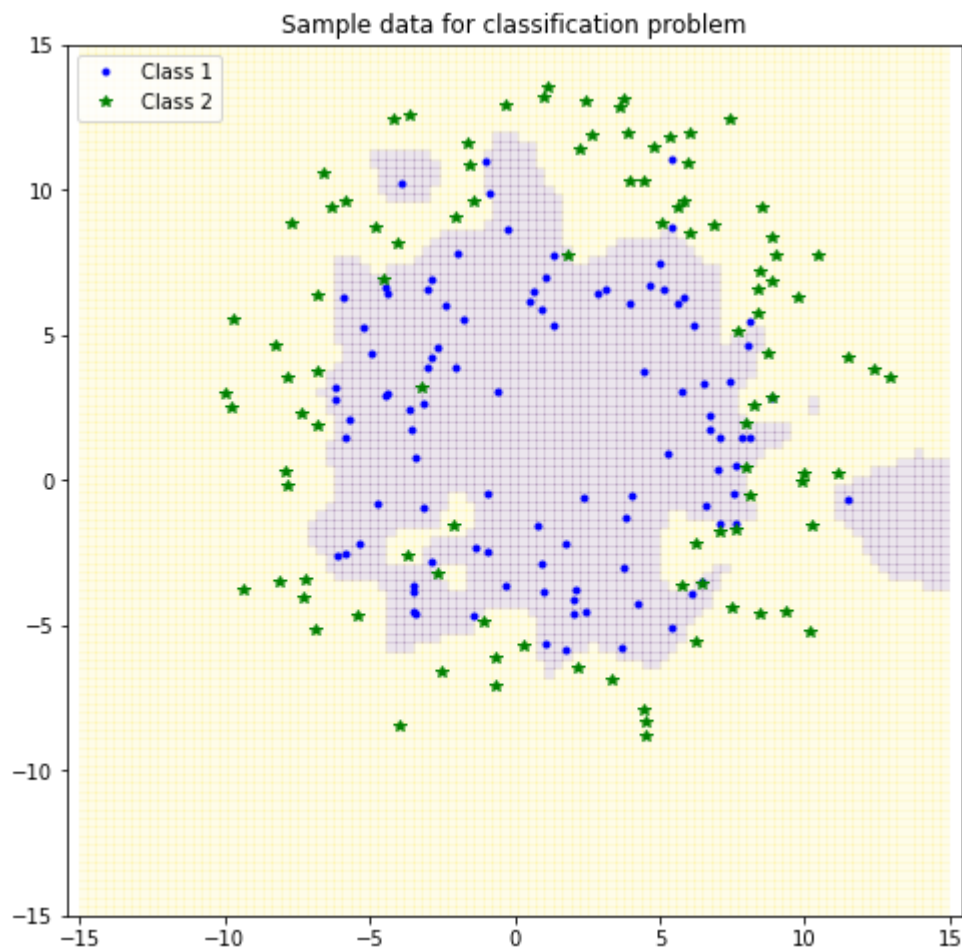


K_annulus shape (200, 200)

	pcost	dcost	gap	pres	dres
0:	-7.4524e+01	-2.3563e+01	1e+02	3e+01	3e-16
1:	-4.6230e+01	-1.2772e+01	2e+01	6e+00	2e-16
2:	-8.5391e+00	-2.1408e+00	8e+00	1e+00	6e-16
3:	-2.0865e+00	-1.9871e+00	1e-01	2e-02	4e-16
4:	-1.9880e+00	-1.9870e+00	1e-03	2e-04	5e-16
5:	-1.9870e+00	-1.9870e+00	1e-05	2e-06	3e-16
6:	-1.9870e+00	-1.9870e+00	1e-07	2e-08	5e-16

Optimal solution found.

Accuracy is for c=0.01 and sigma=0.5 is: 0.96



K_annulus shape (200, 200)

	pcost	dcost	gap	pres	dres
0:	-7.8970e+01	-3.5401e+01	1e+02	2e+01	3e-16
1:	-5.4114e+01	-2.7101e+01	1e+01	4e+00	2e-16

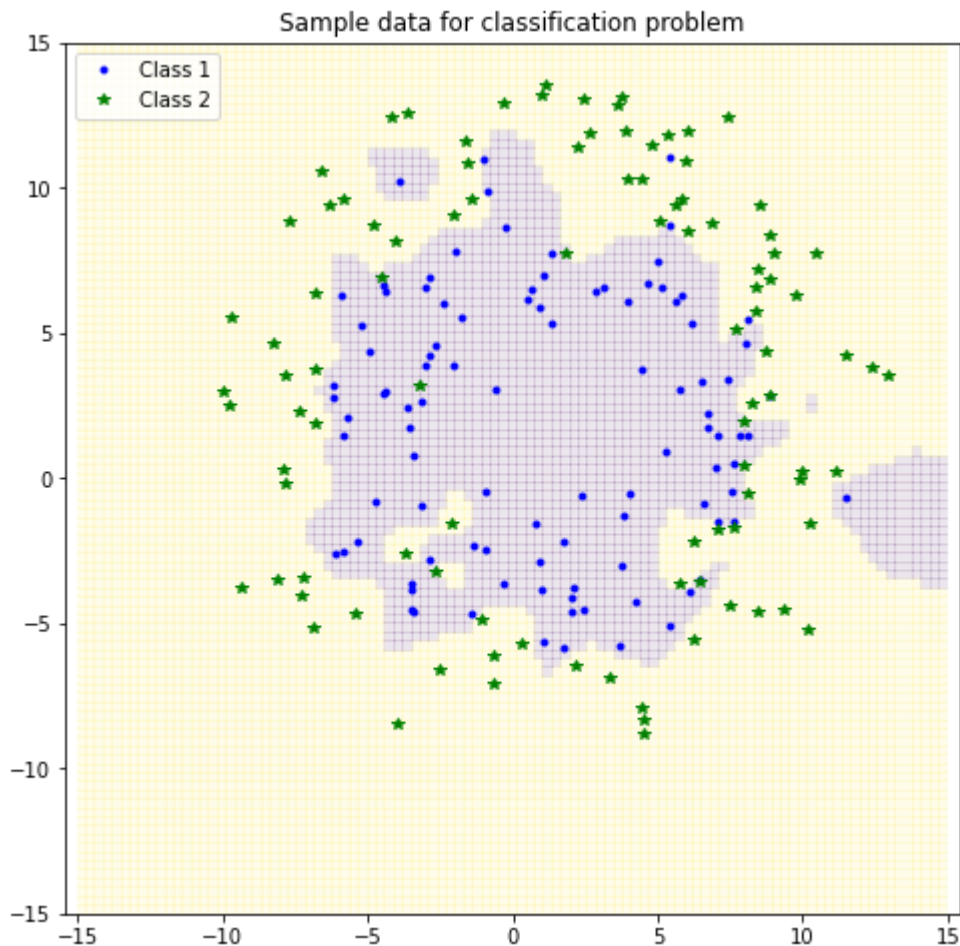
```

1: -3.4114e+01 -2.7191e+01 1e+01 4e+00 2e-10
2: -2.8390e+01 -1.9195e+01 8e+00 1e+00 4e-16
3: -1.8660e+01 -1.8702e+01 2e-01 8e-03 4e-16
4: -1.8701e+01 -1.8702e+01 2e-03 8e-05 2e-16
5: -1.8702e+01 -1.8702e+01 2e-05 8e-07 2e-16
6: -1.8702e+01 -1.8702e+01 2e-07 8e-09 2e-16

```

Optimal solution found.

Accuracy is for $c=0.1$ and $\sigma=0.5$ is: 0.96



K_annulus shape (200, 200)

```

pcost      dcost      gap      pres      dres
0: -1.0368e+02 -9.3284e+01 7e+02 2e+00 2e+00
1: -4.8154e+01 -1.4574e+02 1e+02 3e-14 5e-16
2: -8.5416e+01 -1.0051e+02 2e+01 9e-15 2e-16
3: -9.0691e+01 -9.2585e+01 2e+00 2e-14 2e-16
4: -9.1395e+01 -9.1568e+01 2e-01 1e-14 2e-16
5: -9.1461e+01 -9.1460e+01 0e+00 1e-14 2e-16

```

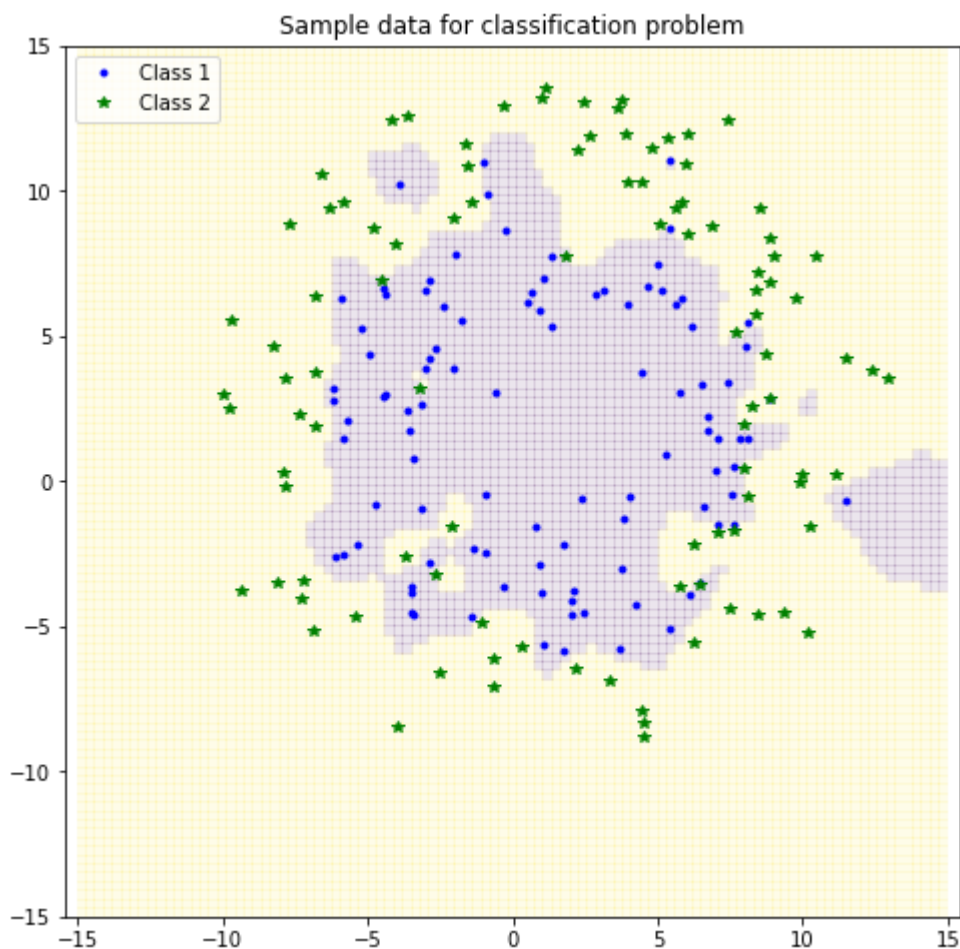
```

5: -9.1461e+01 -9.1469e+01 9e-03 1e-14 2e-16
6: -9.1465e+01 -9.1465e+01 4e-04 3e-14 2e-16
7: -9.1465e+01 -9.1465e+01 2e-05 1e-14 2e-16

```

Optimal solution found.

Accuracy is for $c=1$ and $\sigma=0.5$ is: 0.97



K_annulus shape (200, 200)

	pcost	dcost	gap	pres	dres
0:	1.6242e+03	-1.5704e+03	6e+03	3e-01	9e+00
1:	-1.0471e+02	-7.5797e+02	7e+02	4e-02	1e+00
2:	-1.6288e+02	-2.4567e+02	8e+01	3e-03	1e-01
3:	-1.6667e+02	-1.7333e+02	7e+00	2e-04	8e-03
4:	-1.6721e+02	-1.6739e+02	2e-01	6e-06	2e-04
5:	-1.6724e+02	-1.6724e+02	5e-03	9e-08	3e-06
6:	-1.6724e+02	-1.6724e+02	2e-04	9e-10	3e-08
7:	-1.6724e+02	-1.6724e+02	3e-06	9e-12	3e-10