Problem Set 6 (Total points: 80 + bonus 50)

Support Vector Machines

In this problem set you will implement an SVM and fit it using quadratic programming. We will use CVXOPT module to solve the optimization problems.
You may want to start with solving written problems at the end of this notebook or at least with reading the textbook. It will help a lot in this programming assignment.
Some of the cells will take minutes to run, so feel free to test you code on smaller tasks while you go. Easiest way would be to remove both for-loops and run the code just once.

Quadratic Programming

The standard form of a QP can be formulated as

$$\begin{aligned} \min_{x} \quad & \frac{1}{2} x^T P x + q^T x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

where $\preceq$ is an element-wise $\leq$. CVXOPT solver finds an optimal solution x^*, given a set of matrices P, q, G, h, A, b.

FYI, you can read on the methods to solve quadratic programming problems here (https://en.wikipedia.org/wiki/Quadratic_programming#Solution_methods).

Task 1. [10 points]

Design appropriate matrices to solve the following problem.

$$\begin{aligned} \min_{x} \quad & f(x) = x_1^2 + 4x_2^2 - 8x_1 - 16x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 5 \\ & x_1 \leq 3 \\ & x_2 \geq 0 \end{aligned}$$

In [1]:

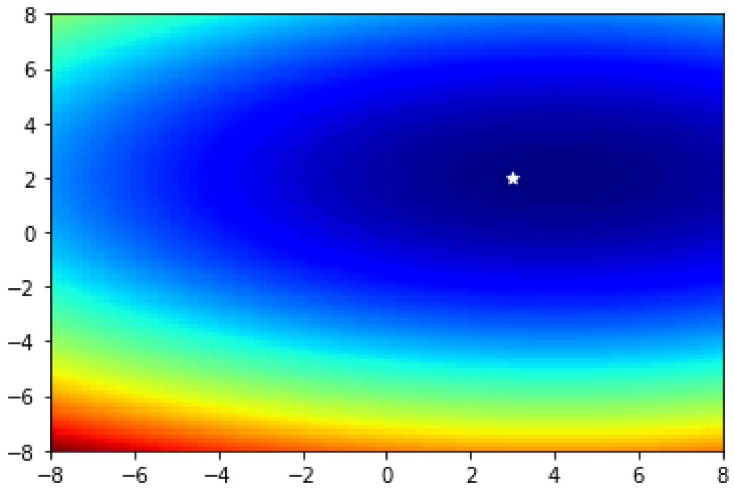
```
from cvxopt import matrix, solvers
# Turns off the printing of CVXOPT solution for the rest of the notebook
solvers.options['show_progress'] = False

P = 2 * matrix([[1., 0.], [0., 4.]])
#-----------------------------------------------------------------------------
# Define q, G, h
q = matrix([[-8.,-16.]])
G = matrix([[1.,1.,0.],[1.,0.,-1]])
h = matrix([[5.,3.,0.]])
#-----------------------------------------------------------------------------

sol = solvers.qp(P, q, G, h)
x1, x2 = sol['x']
print('Optimal x: ({:.8f}, {:.8f})'.format(x1, x2))
```

```
Optimal x: (2.99999993, 1.99927914)
```

Let's visualize the solution

In [2]:

```
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

X1, X2 = np.meshgrid(np.linspace(-8, 8, 100), np.linspace(-8, 8, 100))
F = X1**2 + 4*X2**2 - 8*X1 - 16 * X2

plt.pcolor(X1, X2, F, cmap='jet')
plt.scatter([x1], [x2], marker='*', color='white')
plt.show()
```



Why the solution is not in the minimum?

Linear SVM

Now, let's implement linear SVM. We will do this for a general case, that allows class distributions to overlap **(see Bishop 7.1.1)**.

As a linear model, linear SVM produces classification scores for a given sample x as

$$\hat{y}(x) = w^T \phi(x) + b$$

where $w \in \mathbb{R}^d$, $b \in \mathbb{R}$ are model weights and bias respectively and ϕ is a fixed feature-space transformation. Final label prediction is done by taking the sign of $\hat{y}(x)$.

Given a set of training samples $x_n \in \mathbb{R}^d$, $n \in 1, \ldots, N$, with the corresponding labels $y_i \in \{-1, 1\}$ linear SVM is fit (*i.e.* parameters w and b are chosen) by solving the following constrained optimization task:

$$\begin{aligned} \min_{w,\xi,b} \quad & \frac{1}{2}\|w\|^2 + C\sum_{n=1}^{N} \xi_n \\ \text{subject to} \quad & y_n \hat{y}(x_n) \geq 1 - \xi_n, \qquad n = 1, \ldots, N \\ & \xi_n \geq 0, \qquad n = 1, \ldots, N \end{aligned}$$

Task 2.1 [60 points]

Your task is to implement this using QP solver by designing appropriate matrices P, q, G, h.

Hints

1. You need to optimize over w, ξ, b. You can simply concatenate them into $\chi = (w, \xi, b)$ to feed it into QP-solver. Now, how to define the objective function and the constraints in terms of χ? (For example, $b_1 + b_2$ can be obtained from vector $(a_1, b_1, b_2, c_1, c_2)$ by taking the inner product with $(0, 1, 1, 0, 0)$).
2. You can use `np.bmat` to construct matrices. Like this:

In [3]:

```
np.bmat([[np.identity(3), np.zeros((3, 1))],
         [np.zeros((2, 3)), -np.ones((2, 1))]])
```

Out[3]:

```
matrix([[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0., -1.],
        [ 0.,  0.,  0., -1.]])
```

In [4]:

```
from sklearn.base import BaseEstimator

class LinearSVM(BaseEstimator):
    def __init__(self, C, transform=None):
        self.C = C
        self.transform = transform

    def fit(self, X, Y):
        """Fit Linear SVM using training dataset (X, Y).

        :param X: data samples of shape (N, d).
        :param Y: data target labels of size (N). Each label is either 1 or -1.
        """
        # Apply transformation (phi) to X
        if self.transform is not None:
            X = self.transform(X)
        d = len(X[0])
        N = len(X)

        #-------------------------------------------------------------------------
        # Construct appropriate matrices here to solve the optimization problem describ
        # We want optimal solution for vector (w, xi, b).
        P = matrix(np.bmat([[np.identity(d),np.zeros((d,N+1))],[np.zeros((N+1,d+N+1))]]
        q = matrix(np.bmat([[np.zeros((1,d)),self.C*np.ones((1,N)),np.zeros((1,1))]])).T

        xy = np.bmat([np.matrix(X[:,d1]*Y).T for d1 in range(d)])
        G = -matrix(np.bmat([[xy,np.identity(N),Y.reshape(N,1)],[np.zeros((N,d)),np.ide
        h = matrix(np.bmat([[-np.ones((N,1))],[(-1)*np.zeros((N,1))]]))
        #-------------------------------------------------------------------------

        sol = solvers.qp(P, q, G, h)
        ans = np.array(sol['x']).flatten()
        self.weights_ = ans[:d]
        self.xi_ = ans[d:d+N]
        self.bias_ = ans[-1]

        #-------------------------------------------------------------------------
        # Find support vectors. Must be a boolean array of length N having True for sup
        # vectors and False for the rest.
        self.support_vectors = self.xi_ >= 0
        #-------------------------------------------------------------------------

    def predict_proba(self, X):
        """
        Make real-valued prediction for some new data.
        :param X: data samples of shape (N, d).
        :return: an array of N predicted scores.
        """
        # return y_hat
        if self.transform is not None:
            X = self.transform(X)
        return np.dot(self.weights_,X.T) + self.bias_

    def predict(self, X):
        """
        Make binary prediction for some new data.
        :param X: data samples of shape (N, d).
        :return: an array of N binary predicted labels from {-1, 1}.
        """
```

```
        return np.sign(self.predict_proba(X))
```

Let's see how our LinearSVM performs on some data.

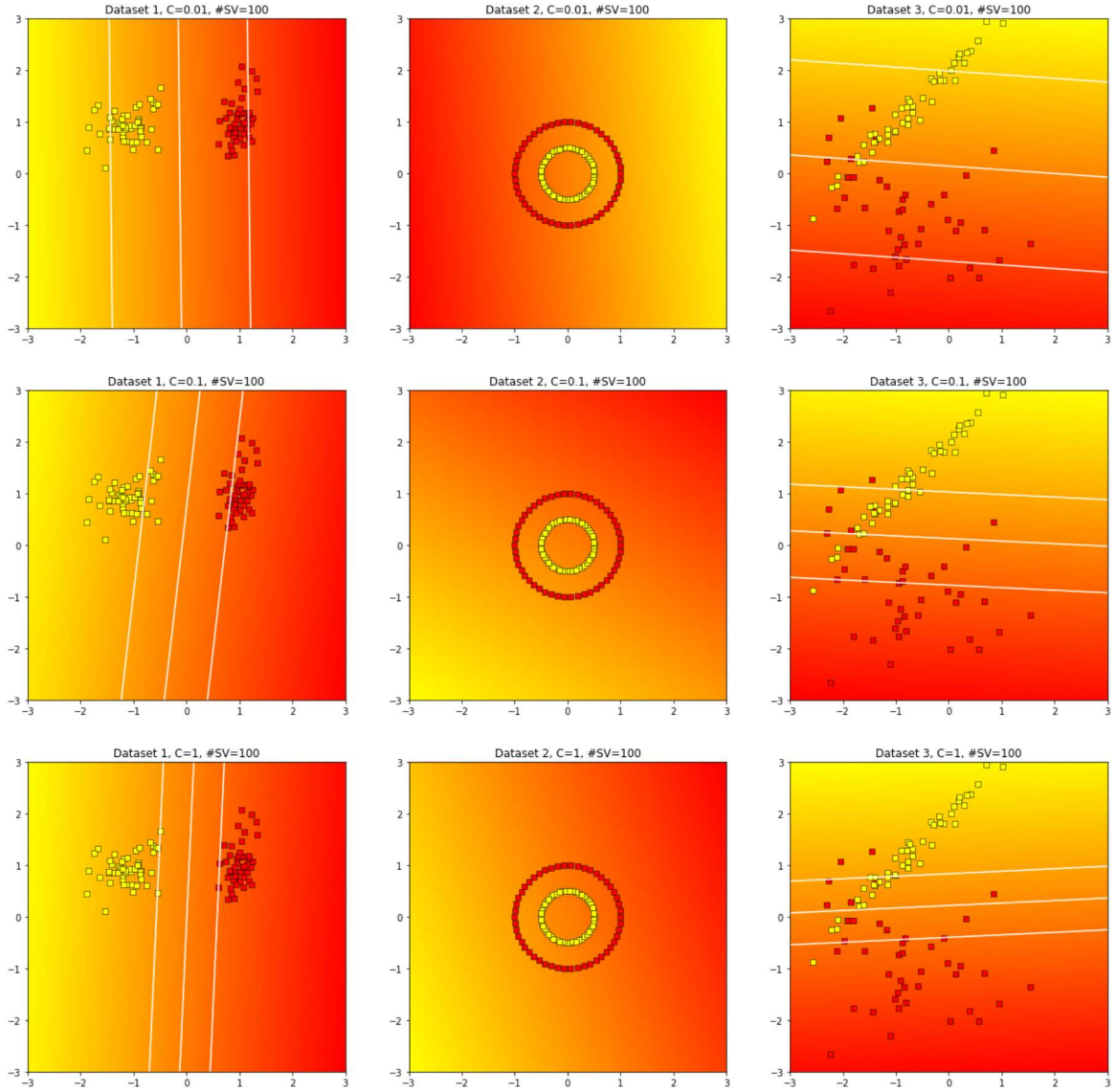
In [5]:

```
from sklearn.datasets import make_classification, make_circles
X = [None, None, None]
y = [None, None, None]
X[0], y[0] = make_classification(n_samples=100, n_features=2, n_redundant=0, n_clusters
X[1], y[1] = make_circles(n_samples=100, factor=0.5)
X[2], y[2] = make_classification(n_samples=100, n_features=2, n_redundant=0, n_clusters

# Go from {0, 1} to {-1, 1}
y = [2 * yy - 1 for yy in y]
```

In [6]:

```
C_values = [0.01, 0.1, 1]

plot_i = 0
plt.figure(figsize=(len(X) * 7, len(C_values) * 7))
for C in C_values:
    for i in range(len(X)):
        plot_i += 1
        plt.subplot(len(C_values), len(X), plot_i)
        #-------------------------------------------------------------------------
        model = LinearSVM(C=C)
        #-------------------------------------------------------------------------
        model.fit(X[i], y[i])
        sv = model.support_vectors
        n_sv = sv.sum()
        if n_sv > 0:
            plt.scatter(X[i][:, 0][sv], X[i][:, 1][sv], c=y[i][sv], cmap='autumn', mark
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        if n_sv < len(X[i]):
            plt.scatter(X[i][:, 0][~sv], X[i][:, 1][~sv], c=y[i][~sv], cmap='autumn',
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        xvals = np.linspace(-3, 3, 200)
        yvals = np.linspace(-3, 3, 200)
        xx, yy = np.meshgrid(xvals, yvals)
        zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()]), xx.shape)
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=

        plt.xlim([-3, 3])
        plt.ylim([-3, 3])
        plt.title('Dataset {}, C={}, #SV={}'.format(i + 1, C, n_sv))
plt.show()
```

```
e:\Users\ljsPC\Anaconda3\lib\site-packages\matplotlib\contour.py:1243: U
serWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



Why the number of support vectors decreases as C increases?

For debug purposes. Very last model must have almost the same weights and bias:

$$w = \begin{pmatrix} -0.0784521 \\ 1.62264867 \end{pmatrix}$$
$$b = -0.3528510092782581$$

In [7]:

```
model.weights_
```

Out[7]:

```
array([-0.0784521 ,  1.62264867])
```

In [8]:

```
model.bias_
```

Out[8]:

```
-0.3528510092782581
```

Task 2.2 [10 points]

Even using linear SVM, we are able to separate data that linearly inseparable by using feature transformation.

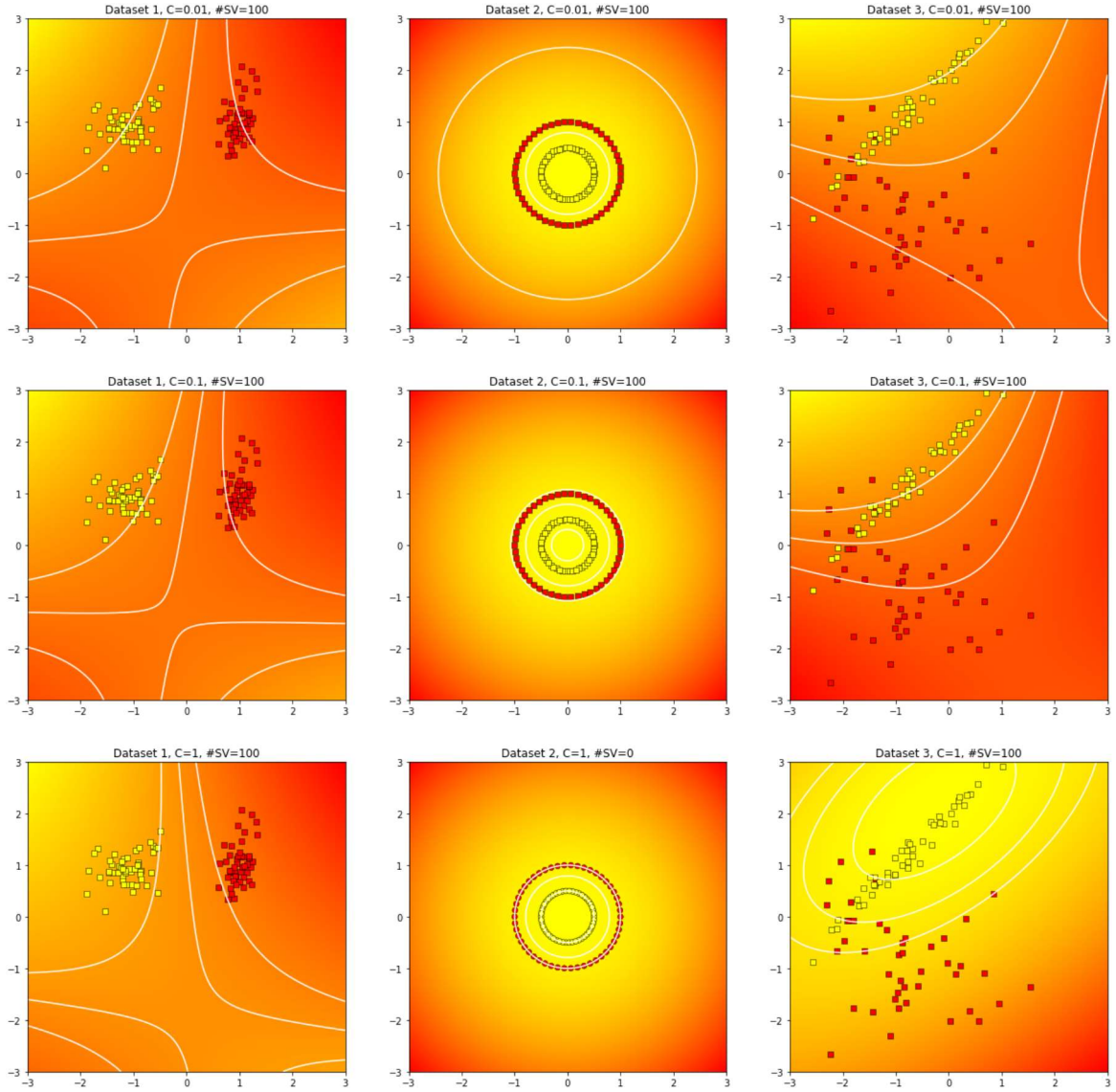
Implement the following feature transformation $\phi(x_1, x_2) = (x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1 x_2)$

In [9]:

```
def append_second_order(X):
    """Given array Nx[x1, x2] return Nx[x1, x2, x1^2, x2^2, x1x2]."""
    # return new_X
    X_0 = X[:,0].reshape(-1, 1)
    X_1 = X[:,1].reshape(-1, 1)
    new_X = np.concatenate([X_0, X_1, X_0*X_0, X_1*X_1, X_0*X_1], axis=1)

    return new_X

assert np.all(append_second_order(np.array([[1, 2]])) == np.array([[1, 2, 1, 4, 2]])),
```

In [10]:

```
plot_i = 0
C_values = [0.01, 0.1, 1]
plt.figure(figsize=(len(X) * 7, len(C_values) * 7))
for C in C_values:
    for i in range(len(X)):
        plot_i += 1
        plt.subplot(len(C_values), len(X), plot_i)
        #------------------------------------------------------------------------
        model = LinearSVM(C=C, transform=append_second_order)
        #------------------------------------------------------------------------
        model.fit(X[i], y[i])
        sv = model.support_vectors
        n_sv = sv.sum()
        if n_sv > 0:
            plt.scatter(X[i][:, 0][sv], X[i][:, 1][sv], c=y[i][sv], cmap='autumn', mark
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        if n_sv < len(X[i]):
            plt.scatter(X[i][:, 0][~sv], X[i][:, 1][~sv], c=y[i][~sv], cmap='autumn',
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        xvals = np.linspace(-3, 3, 200)
        yvals = np.linspace(-3, 3, 200)
        xx, yy = np.meshgrid(xvals, yvals)
        zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()]), xx.shape)
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=

        plt.xlim([-3, 3])
        plt.ylim([-3, 3])
        plt.title('Dataset {}, C={}, #SV={}'.format(i + 1, C, n_sv))
plt.show()
```



Bonus part (Optional)

Dual representation. Kernel SVM

The dual representation of the maximum margin problem is given by

$$\max_{\alpha} \quad \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m k(x_n, x_m)$$

$$\text{subject to} \quad 0 \leq \alpha_n \leq C, \quad n = 1, \ldots, N$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0$$

In this case bias b can be computed as

$$b = \frac{1}{|\mathcal{S}|} \sum_{n \in \mathcal{S}} \left(y_n - \sum_{m \in \mathcal{S}} \alpha_m y_m k(x_n, x_m) \right),$$

and the prediction turns into

$$\hat{y}(x) = \sum_{n \in S} \alpha_n y_n k(x_n, x) + b.$$

Everywhere above k is a kernel function: $k(x_1, x_2) = \phi(x_1)^T \phi(x_2)$ (and the trick is that we don't have to specify ϕ, just k).

Note, that now

1. We want to maximize the objective function, not minimize it.
2. We have equality constraints. (That means we should use A and b in qp-solver)
3. We need access to support vectors (but not all the training samples) in order to make a prediction.

Task 3.1 [40 points]

Implement KernelSVM

Hints

1. What is the variable we are optimizing over?
2. How to maximize a function given a tool for minimization?
3. What is a definition of support vector in dual representation?

In [ ]:

```
class KernelSVM(BaseEstimator):
    def __init__(self, C, kernel=np.dot):
        self.C = C
        self.kernel = kernel

    def fit(self, X, Y):
        """Fit Kernel SVM using training dataset (X, Y).

        :param X: data samples of shape (N, d).
        :param Y: data target labels of size (N). Each label is either 1 or -1. Denoted
        """
        N = len(Y)

        #-------------------------------------------------------------------------------
        # Construct appropriate matrices here to solve the optimization problem describ
        # P =
        # q =
        # G =
        # h =
        # A =
        # b =
        #-------------------------------------------------------------------------------

        sol = solvers.qp(P, q, G, h, A, b)
        self.alpha_ = np.array(sol['x']).flatten()

        #-------------------------------------------------------------------------------
        # Find support vectors. Must be a boolean array of length N having True for supp
        # vectors and False for the rest.
        # self.support_vectors =
        #-------------------------------------------------------------------------------

        sv_ind = self.support_vectors.nonzero()[0]
        self.X_sup = X[sv_ind]
        self.Y_sup = Y[sv_ind]
        self.alpha_sup = self.alpha_[sv_ind]
        self.n_sv = len(sv_ind)

        #-------------------------------------------------------------------------------
        # Compute bias
        # self.bias_ =
        #-------------------------------------------------------------------------------

    def predict_proba(self, X):
        """
        Make real-valued prediction for some new data.
        :param X: data samples of shape (N, d).
        :return: an array of N predicted scores.
        """
        # return y_hat
        pass

    def predict(self, X):
        """
        Make binary prediction for some new data.
        :param X: data samples of shape (N, d).
        :return: an array of N binary predicted labels from {-1, 1}.
        """
        return np.sign(self.predict_proba(X))
```

We can first test our implementation by using dot product as a kernel function. What should we expect in this case?

In [ ]:

```
C = 1
i = 0

plt.figure(figsize=(7, 7))
#-------------------------------------------------------------------------
model = KernelSVM(C=C, kernel=np.dot)
#-------------------------------------------------------------------------
model.fit(X[i], y[i])
sv = model.support_vectors
n_sv = sv.sum()
if n_sv > 0:
    plt.scatter(X[i][:, 0][sv], X[i][:, 1][sv], c=y[i][sv], cmap='autumn', marker='s',
                linewidths=0.5, edgecolors=(0, 0, 0, 1))
if n_sv < len(X[i]):
    plt.scatter(X[i][:, 0][~sv], X[i][:, 1][~sv], c=y[i][~sv], cmap='autumn',
                linewidths=0.5, edgecolors=(0, 0, 0, 1))
xvals = np.linspace(-3, 3, 200)
yvals = np.linspace(-3, 3, 200)
xx, yy = np.meshgrid(xvals, yvals)
zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()]), xx.shape)
plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=1, lines

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.title('Dataset {}, C={}, #SV={}'.format(i + 1, C, n_sv))
plt.show()
```

Task 3.2 [5 points]

Implement polynomial kernel function (wiki (https://en.wikipedia.org/wiki/Polynomial_kernel)).

In [ ]:

```
def polynomial_kernel(d, c=0):
    """Returns a polynomial kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [polynomial kernel of degree d with bias parameter c] of x and y. A sc
        """
        pass
    return kernel

assert polynomial_kernel(d=2, c=1)(np.array([1, 2]), np.array([3, 4])) == 144, 'Polynom
```

Let's see how it performs. This might take some time to run.

In [ ]:

```
plot_i = 0
C = 10
d_values = [2, 3, 4]
plt.figure(figsize=(len(X) * 7, len(d_values) * 7))
for d in d_values:
    for i in range(len(X)):
        plot_i += 1
        plt.subplot(len(d_values), len(X), plot_i)
        #------------------------------------------------------------------------
        model = KernelSVM(C=C, kernel=polynomial_kernel(d))
        #------------------------------------------------------------------------
        model.fit(X[i], y[i])
        sv = model.support_vectors
        n_sv = sv.sum()
        if n_sv > 0:
            plt.scatter(X[i][:, 0][sv], X[i][:, 1][sv], c=y[i][sv], cmap='autumn', mark
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        if n_sv < len(X[i]):
            plt.scatter(X[i][:, 0][~sv], X[i][:, 1][~sv], c=y[i][~sv], cmap='autumn',
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        xvals = np.linspace(-3, 3, 200)
        yvals = np.linspace(-3, 3, 200)
        xx, yy = np.meshgrid(xvals, yvals)
        zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()]), xx.shape)
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=

        plt.xlim([-3, 3])
        plt.ylim([-3, 3])
        plt.title('Dataset {}, C={}, d={}, #SV={}'.format(i + 1, C, d, n_sv))
```

Task 3.3 [5 points]

Finally, you need to implement **radial basis function** kernel (wiki (https://en.wikipedia.org/wiki/Radial_basis_function_kernel)).

In [ ]:

```
def RBF_kernel(sigma):
    """Returns an RBF kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [rbf kernel with parameter sigma] of x and y. A scalar.
        """
        pass
    return kernel
```

Let's see how it performs. This might take some time to run.

In [ ]:

```
plot_i = 0
C_values = [0.1, 1, 10]
sigma_values = [0.1, 1, 10]
plt.figure(figsize=(len(sigma_values) * 7, len(C_values) * 7))
i = 2
for C in C_values:
    for sigma in sigma_values:
        plot_i += 1
        plt.subplot(len(C_values), len(X), plot_i)
        model = KernelSVM(C=C, kernel=RBF_kernel(sigma))
        model.fit(X[i], y[i])
        sv = model.support_vectors
        n_sv = sv.sum()
        if n_sv > 0:
            plt.scatter(X[i][:, 0][sv], X[i][:, 1][sv], c=y[i][sv], cmap='autumn', mark
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        if n_sv < len(X[i]):
            plt.scatter(X[i][:, 0][~sv], X[i][:, 1][~sv], c=y[i][~sv], cmap='autumn',
                        linewidths=0.5, edgecolors=(0, 0, 0, 1))
        xvals = np.linspace(-3, 3, 200)
        yvals = np.linspace(-3, 3, 200)
        xx, yy = np.meshgrid(xvals, yvals)
        zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()]), xx.shape)
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=

        plt.xlim([-3, 3])
        plt.ylim([-3, 3])
        plt.title('Dataset {}, C={}, sigma={}, #SV={}'.format(i + 1, C, sigma, n_sv))
```

Well done!

Awesome! Now you understand all of the important parameters in SVMs. Have a look at SVM from scikit-learn module and how it is used (very similar to ours).

In [ ]:

```
from sklearn.svm import SVC
SVC?
```

In [ ]:

```
plot_i = 0
C = 10
d_values = [2, 3, 4]
plt.figure(figsize=(len(X) * 7, len(d_values) * 7))
for d in d_values:
    for i in range(len(X)):
        plot_i += 1
        plt.subplot(len(d_values), len(X), plot_i)
        #-------------------------------------------------------------------------
        model = SVC(kernel='poly', degree=d, gamma='auto', probability=True)
        #-------------------------------------------------------------------------
        model.fit(X[i], y[i])
        plt.scatter(X[i][:, 0], X[i][:, 1], c=y[i], cmap='autumn', linewidths=0.5, edge
        xvals = np.linspace(-3, 3, 200)
        yvals = np.linspace(-3, 3, 200)
        xx, yy = np.meshgrid(xvals, yvals)
        zz = np.reshape(model.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1] * 2 -
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1., 0., 1.), colors='w', linewidths=1.5, zorde

        plt.xlim([-3, 3])
        plt.ylim([-3, 3])
        plt.title('Dataset {}, C={}, d={}, #SV={}'.format(i + 1, C, d, len(model.suppor
```

Written Problems

Dual Representations [10 pts]

Read section 6.1 in Bishop, and work through all of the steps of the derivations in equations 6.2-6.9. You should understand how the derivation works in detail. Write down your understanding.

Firstly, by taking the derivative of the loss function with respect to w and set it to zero, we can get a equation for w. Then, we use $a_n = -\frac{1}{\lambda} w^T \phi(x_n) - t_n$ to subtitute the parameter vector w, so that we do not need to work on w anymore. And by using a, we can form a dual representation.

In Dual Representation, the $\phi(x_n)^T \phi(x_m)$ is replaced by the gram matrix K_{nm}, so that we can directly on the kernel instead of the explicit introduction of the feature vector $\phi(x)$.

Then, by taking the derivative of the loss function with a and K_{nm} with respect to a, we can get a equation for a. Then we can rewrite the linear regression model.

Therefore, we can work with kernel K_{nm} instead of the explicit introduction of the feature vector $\phi(x)$, and implicitly use feature spaces of high dimensionality.

Kernels [10 pts]

Read Section 6.2 and Verify the results (6.13) and (6.14) for constructing valid kernels.

$k_1(x, x') = \phi(x)^T \phi(x')$, then $k(x, x') = ck_1(x, x') = c\phi(x)^T \phi(x')$
If we set new $\phi(x)$ as $\phi(x) = \sqrt{c}\phi(x)$ which is still valid (for c > 0).

For $k(x, x') = f(x)k_1(x, x')f(x') = f(x)\phi(x)^T\phi(x')f(x')$, we set $\phi(x) = f(x)\phi(x)$ then we get the valid kernel since $f(x)$ is polynomial.

Maximum Margin Classifiers [10 pts]

Read section 7.1 and show that, if the 1 on the right hand side of the constraint (7.5) is replaced by some arbitrary constant $\gamma > 0$, the solution for maximum margin hyperplane is unchanged.

$$t_n(w^T\phi(x_n) + b) \geq 1$$

$t_n(w^T\phi(x_n) + b) \geq 1$ shows that the data points are classified as true if the equations for them are equal or larger than 1. By setting different value for the point that is closest to the surface, like $t_n(w^T\phi(x_n) + b) \geq C$, we can also set the constant on the right which may not be 1 anymore while the constraintsare still satisfied.

In [ ]: