Problem Set 6 (Total points: 110 + bonus 50)

Support Vector Machines

In this problem set you will implement an SVM and fit it using quadratic programming. We will use the CVXOPT module to solve the optimization problems.
You may want to start with solving the 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).

Problem 1. [10 points]

Design appropriate matrices to solve the following problem.

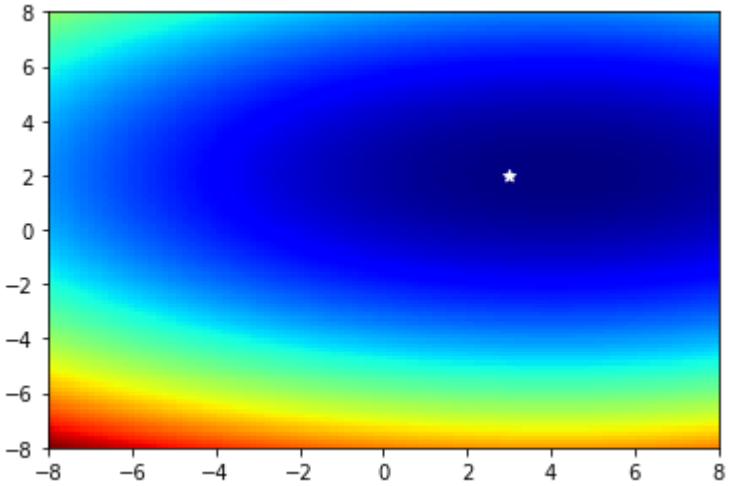
$$\begin{aligned} \min_x \quad & f(x) = x_1^2 + 4x_2^2 - 8x_1 - 16x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 5 \\ & x_1 \leq 3 \\ & x_2 \geq 0 \end{aligned}$$

In [3]:
```
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 =
# G =
# h =
#-------------------------------------------------------------------------
----------------
q = (-8) * matrix([1.,2.])
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 [4]:
```
%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 is the solution not in the minimum?

The optimal solution cannot lie at the minimum as that would violate the constraint because the unconstrained local minimum lies outside of the allowable sub-space. Instead the solution lies on the constraint at the point nearest to 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:

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

Problem 2.1 [60 points]

Your task is to implement this using a 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 [5]:
```
np.bmat([[np.identity(3), np.zeros((3, 1))],
         [np.zeros((2, 3)), -np.ones((2, 1))]])
```

Out[5]:
```
matrix([[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0., -1.],
        [ 0.,  0.,  0., -1.]])
```

In [6]:
```
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 pro
blem described above.
        # We want optimal solution for vector (w, xi, b).

        P = np.bmat([[np.identity(d), np.zeros((d, N + 1))],[np.zeros((N +
1,N + d + 1))]])
        q = np.bmat([[np.zeros((d,1))], [np.ones((N,1))], [np.zeros((1,1
))]])

        g1 = np.diag(Y) @ X @np.bmat([np.identity(d), np.zeros((d,N+1))])
        g2 = np.diag(Y) @ np.bmat([np.zeros((N,d + N)), np.ones((N,1))])
        g3 = np.bmat([np.zeros((N,d)), np.identity(N), np.zeros((N,1))])
        g = g1 + g2 + g3
        G = np.bmat([[g],[np.zeros((N,d)), np.identity(N), np.zeros((N,1
))]])

        h = np.bmat([[np.ones((N,1))],[np.zeros((N,1))]])

        #-----------------------------------------------------------------
------------------------

        P = matrix(P)
        q = matrix(self.C * q)
        G = matrix( (-1) * G)
        h = matrix( (-1) * h)

        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]

        #-----------------------------------------------------------------
------------------------
```

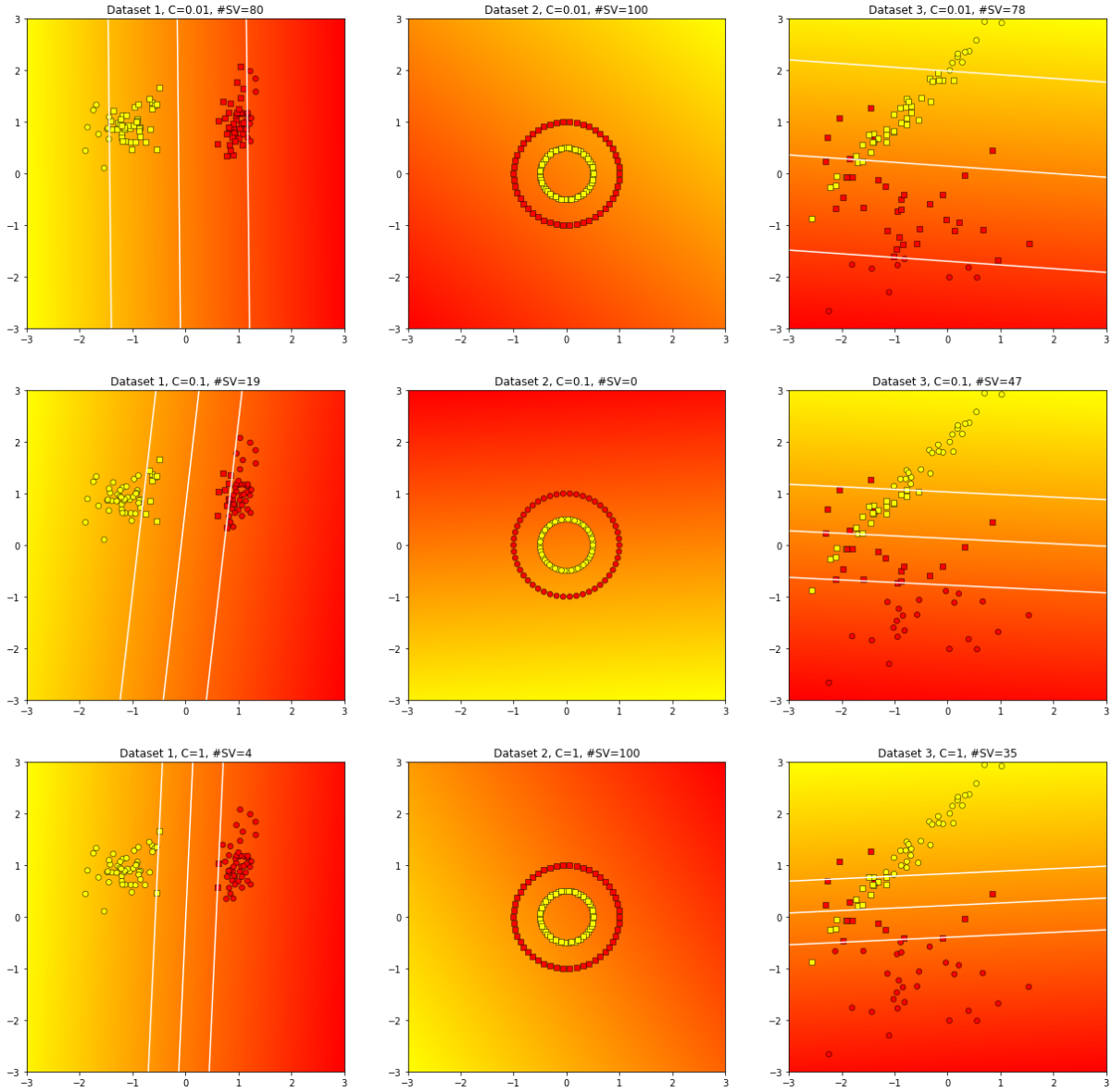
```
        # Find support vectors. Must be a boolean array of length N having
 True for support
        # vectors and False for the rest.

        margin = Y * (X @ self.weights_ + self.bias_)
        self.support_vectors = np.isclose(margin, 1 - self.xi_)
        #-----------------------------------------------------------------
------------------------

    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.
        """
        if self.transform is not None:
            X = self.transform(X)

        y_hat = np.dot(X,self.weights_) + self.bias_
        return y_hat.flatten()

    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 [7]:
```
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_per_class=1, random_state=1)
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_per_class=1, random_state=4)

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

In [8]:
```
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='a
utumn', 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, linestyles='solid')

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

```
C:\Users\timpc\Anaconda3\lib\site-packages\ipykernel_launcher.py:26: UserWa
rning: No contour levels were found within the data range.
```



Why does the number of support vectors decrease as C increases?

As the value of C is increased, we increase the penalty of mis-classification, which results in bringing the margin closer to the decision boundary as we are being more restrictive. Hence as the margin get's smaller the number of support vectors decreases.

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 [9]:
```
model.weights_
```

Out[9]:
```
array([-0.0784521 ,  1.62264867])
```

In [10]:
```
model.bias_
```

Out[10]:
```
-0.3528510092782581
```

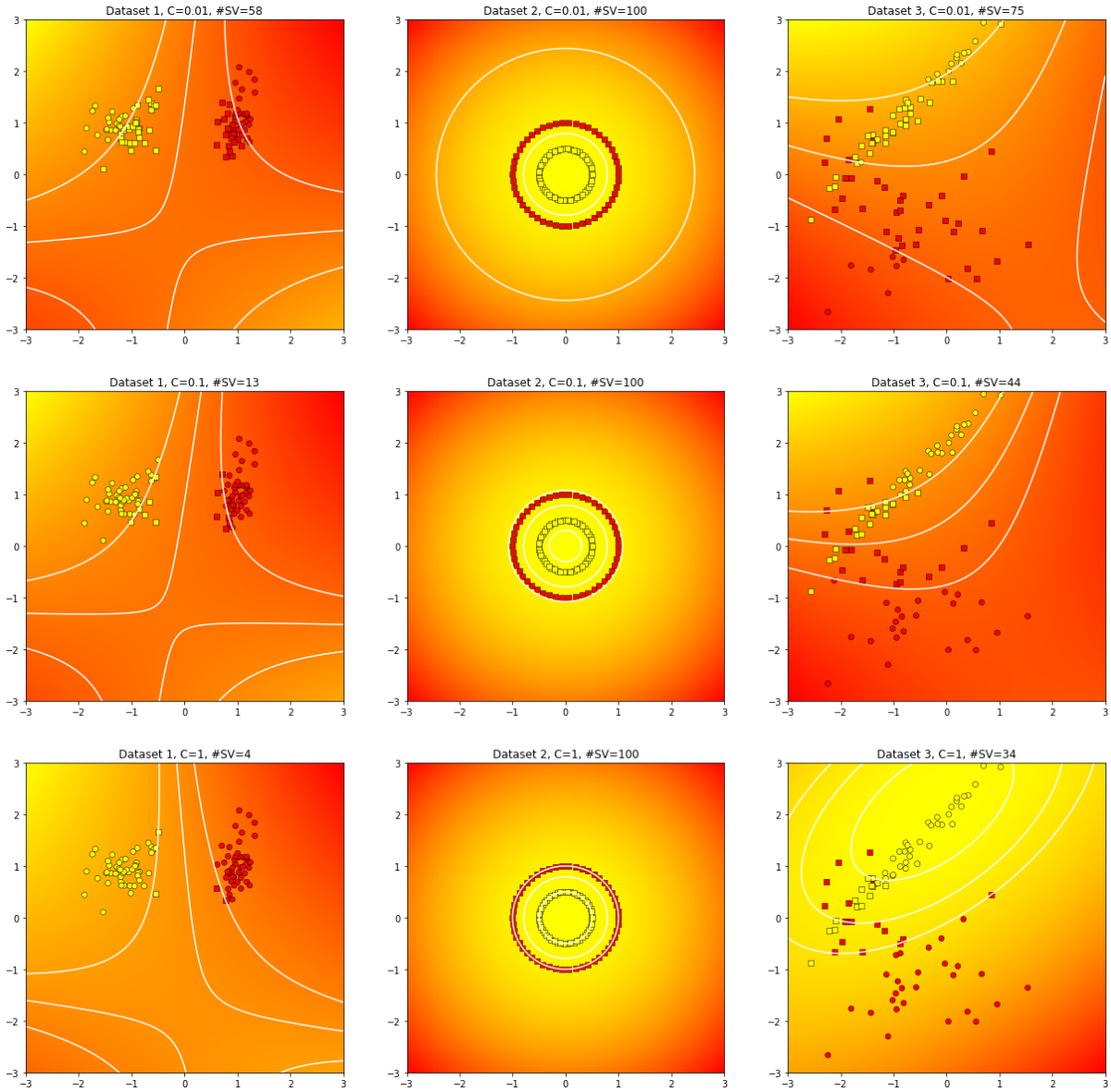
Problem 2.2 [10 points]

Even using a linear SVM, we are able to separate data that is 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 [11]:
```
def append_second_order(X):
    """Given array Nx[x1, x2] return Nx[x1, x2, x1^2, x2^2, x1x2]."""
    x1 = np.array(X[:,0])
    x2 = np.array(X[:,1])
    new_X = np.column_stack((x1,x2,x1**2,x2**2,x1*x2))
    return new_X

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

In [12]:
```
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='a
utumn', 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, linestyles='solid')

        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

$$\begin{aligned} \max_{\alpha} \quad & \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m k(x_n, x_m) \\ \text{subject to} \quad & 0 \leq \alpha_n \leq C, \quad n = 1, \ldots, N \\ & \sum_{n=1}^{N} \alpha_n y_n = 0 \end{aligned}$$

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

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

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 the support vectors (but not all the training samples) in order to make a prediction.

Problem 3.1 [40 points]

Implement KernelSVM

Hints

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

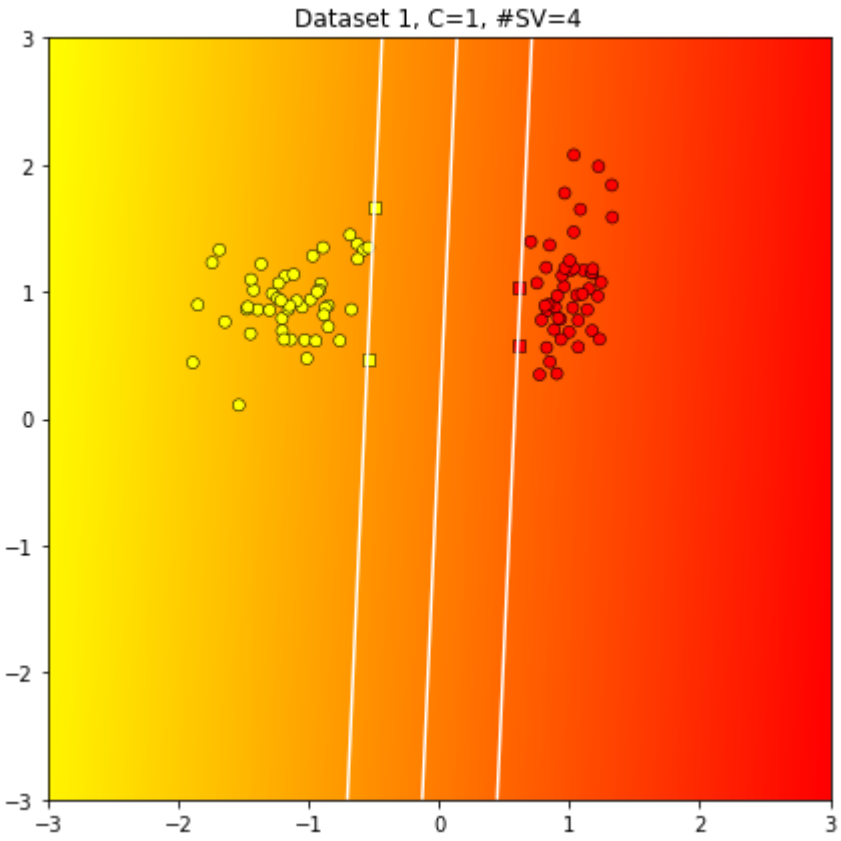
In [14]:
```
def linear_kernel(x,y):
    return np.dot(x.T,y)


class KernelSVM(BaseEstimator):
    def __init__(self, C, kernel=linear_kernel):
        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 as t_i in Bishop.
        """
        N = len(Y)
        Y = Y.reshape((-1,1)) # reshape Y

        K = np.zeros((N,N))
        for i in range(N):
            for j in range(N):
                K[i,j] = self.kernel(X[i], X[j])

        #-------------------------------------------------------------------
------------------------
        # Construct appropriate matrices here to solve the optimization pro
blem described above.
        P = matrix(Y * Y.T * K)
        q = -1.* matrix(np.ones(N))
        G = matrix(np.bmat([[-1.*np.identity(N)],[1.*np.identity(N)]]))
        h = matrix(np.bmat([[np.zeros((N,1))],[1.*self.C*np.ones((N,1))]]))
        A = matrix(1.* Y.reshape((1,-1)))
        b = matrix(np.zeros(1))

        #-------------------------------------------------------------------
------------------------

        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 support
        # vectors and False for the rest.
        self.support_vectors = self.alpha_ > 1e-5
        #-------------------------------------------------------------------
------------------------

        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

        sum_ = 0
        for i in range(self.n_sv):
            sum_ += Y[i]
            for j in range(self.n_sv):
                k_sv = self.kernel(self.X_sup[i],self.X_sup[j])
                sum_ = sum_ -  k_sv*self.alpha_sup[j]*self.Y_sup[j]

        self.bias_ = 1/self.n_sv * sum_
        print("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.
        """
        alpha_sup = self.alpha_sup
        X_sup = self.X_sup
        Y_sup = self.Y_sup
        n_sv = len(X_sup)
        N = X.shape[0]

        print("Num sup vec.",n_sv)
        y_hat = np.zeros(N)

        for i in range(N):
            sum_ = 0
            data_point = X[i]
            for j in range(n_sv):
                kernel_val = self.kernel(X_sup[j],data_point)
                sum_ += alpha_sup[j]*Y_sup[j]*kernel_val

            y_hat[i] =  sum_ + self.bias_

        return y_hat.flatten()

    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 the dot product as a kernel function. What should we expect in this case?

In [15]:
```
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='autum
n',
                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.shap
e)
plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zor
der=1, linestyles='solid')

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

```
bias [0.00778873]
Num sup vec. 4
```



Problem 3.2 [5 points]

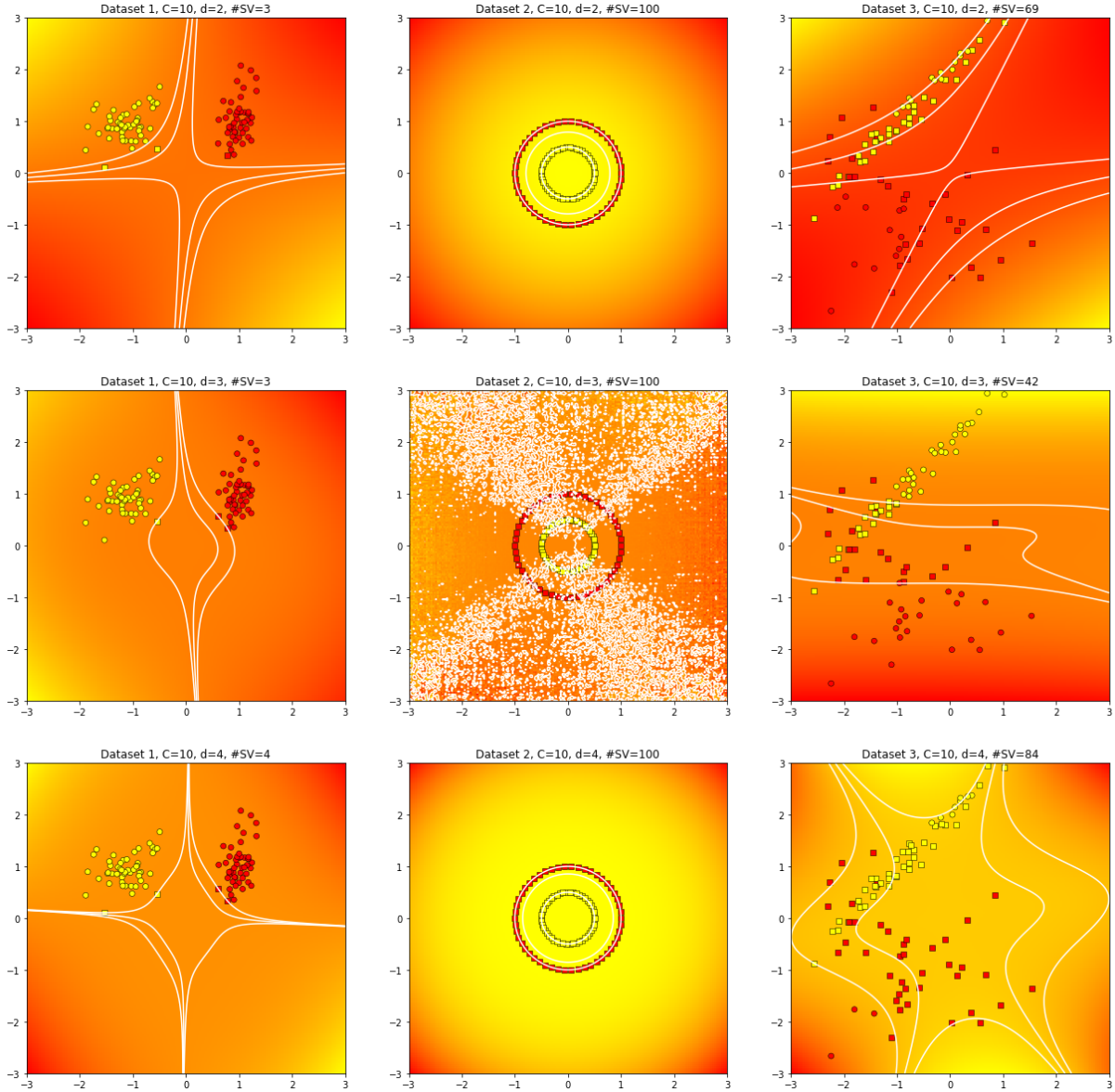
Implement a polynomial kernel function (wiki (https://en.wikipedia.org/wiki/Polynomial_kernel)).

In [17]:
```
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 scalar.
        """
        return (np.dot(x.T,y) + c)**d
    return kernel

assert polynomial_kernel(d=2, c=1)(np.array([1, 2]), np.array([3, 4])) == 1
44, 'Polynomial kernel implemented incorrectly'
```

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

In [18]:
```
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='a
utumn', 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, linestyles='solid')

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

```
bias [-0.77647617]
Num sup vec. 3
bias [1.66666719]
Num sup vec. 100
bias [-1.01665147]
Num sup vec. 69
bias [0.39128129]
Num sup vec. 3
bias [5.32907052e-17]
Num sup vec. 100
bias [-0.10472251]
Num sup vec. 42
bias [0.3846276]
Num sup vec. 4
bias [1.13333375]
Num sup vec. 100
bias [0.1401246]
Num sup vec. 84
```



Task 3.3 [5 points]

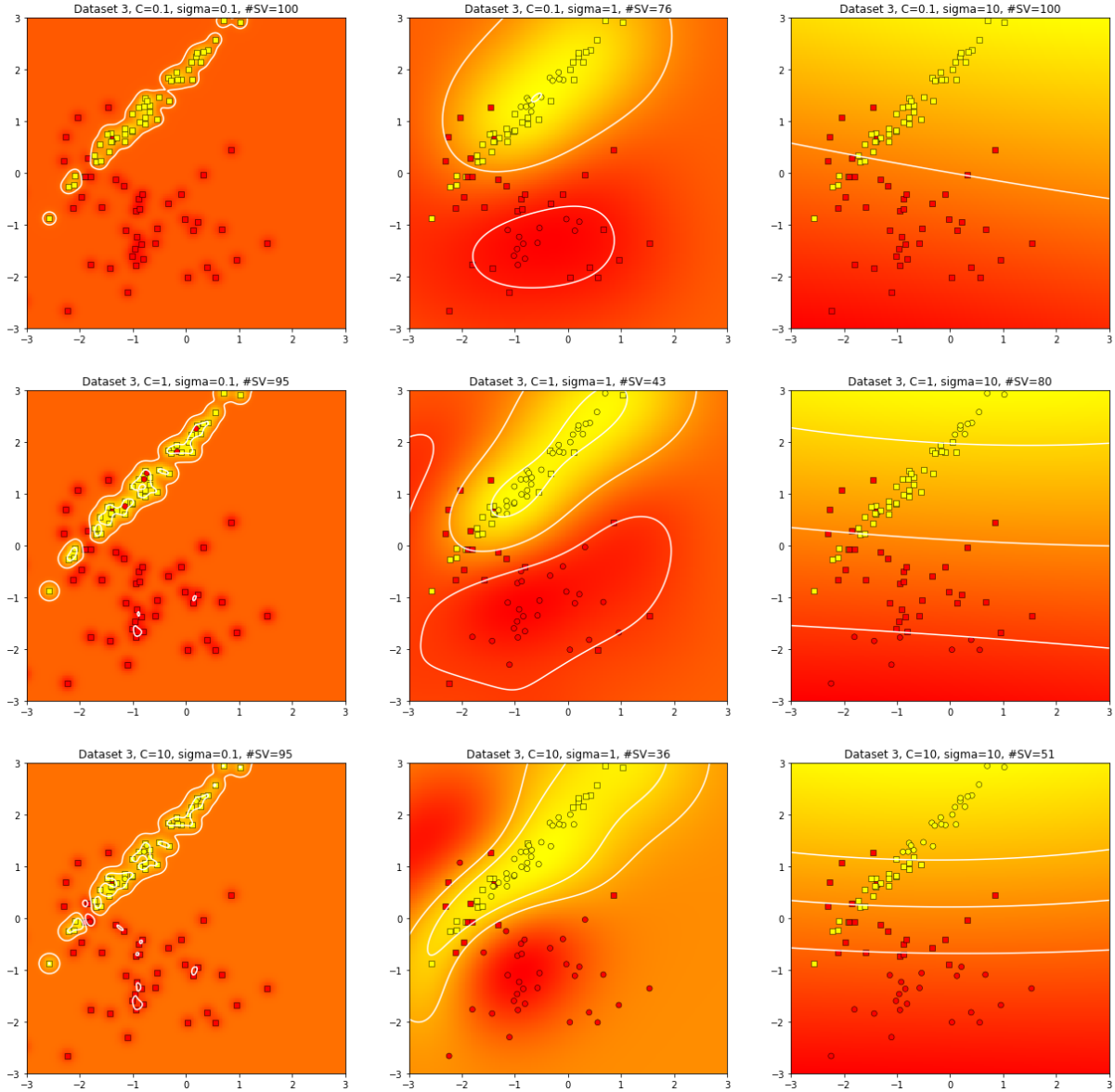
Finally, you need to implement a **radial basis function** kernel (wiki (https://en.wikipedia.org/wiki/Radial_basis_function_kernel)).

In [19]:
```
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.
        """
        diff_squared = np.dot((x-y).T,(x-y))
        exponent = -0.5*(1/sigma**2)*diff_squared
        return np.exp(exponent)
    return kernel
```

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

In [20]:
```
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='a
utumn', 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, linestyles='solid')

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

```
bias [-0.04583153]
Num sup vec. 100
bias [-0.36152901]
Num sup vec. 76
bias [-0.02354633]
Num sup vec. 100
bias [-0.1695614]
Num sup vec. 95
bias [-0.55622428]
Num sup vec. 43
bias [-0.25353059]
Num sup vec. 80
bias [-0.18012244]
Num sup vec. 95
bias [-1.66051546]
Num sup vec. 36
bias [-1.15895975]
Num sup vec. 51
```



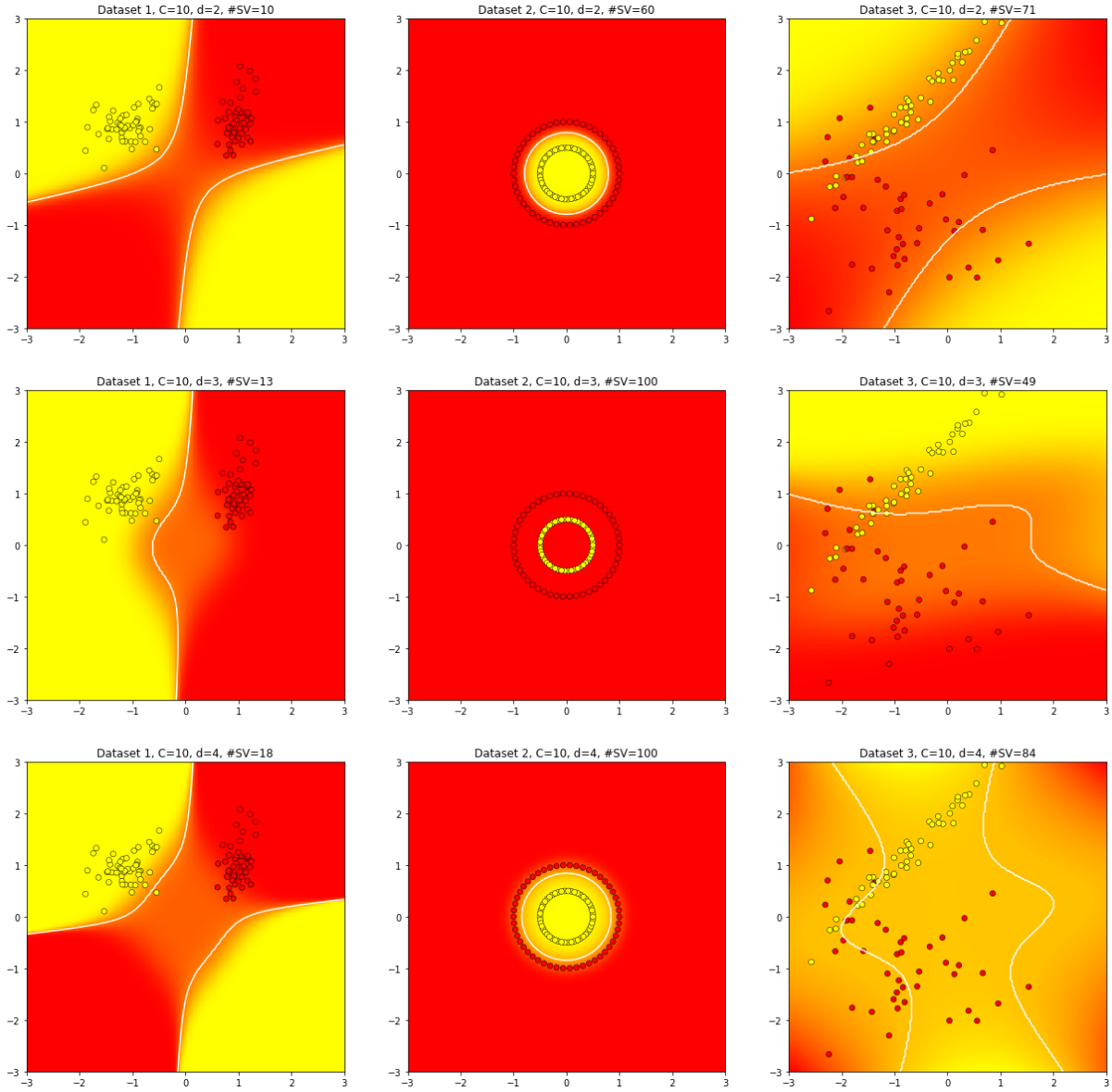
Well done!

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 [21]:
```
from sklearn.svm import SVC
SVC?
```

In [22]:
```
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', linewidt
hs=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()])
[:, 1] * 2 - 1, xx.shape)
        plt.pcolormesh(xx, yy, zz, cmap='autumn', zorder=0)
        plt.contour(xx, yy, zz, levels=(-1., 0., 1.), colors='w', linewidth
s=1.5, zorder=1, linestyles='solid')

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

```
C:\Users\timpc\Anaconda3\lib\site-packages\ipykernel_launcher.py:19: UserWa
rning: No contour levels were found within the data range.
```



4 Written Problems

Problem 4.1 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.

4.1

$$J(w) = \frac{1}{2}\sum_{n=1}^{N}\{w^T\phi(x_n) - t_n\}^2 + \frac{\lambda}{2}w^Tw \quad (6.2)$$

$$\nabla_w J(w) = \frac{1}{2}\sum_{n=1}^{N} 2\{w^T\phi(x_n) - t_n\}\cdot\phi(x_n) + \lambda w = 0 \Rightarrow$$

$$\Rightarrow w = -\frac{1}{\lambda}\sum_{n=1}^{N}\{w^T\phi(x_n) - t_n\}\cdot\phi(x_n) = \quad (6.3)$$

$$= \sum_{n=1}^{N}\left(\frac{\{w^T\phi(x_n) - t_n\}}{-\lambda}\right)\cdot\phi(x_n) = \sum_{n=1}^{N} a_n\phi(x_n) = \Phi^T a$$

Φ – design matrix

$a = (a_1, a_2, \ldots, a_n)^T$

and we defined:

$$(6.4) \quad a = -\frac{1}{\lambda}(w^T\phi(x_n) - t_n)$$

now we substitute $w = \Phi^T a$ into $J(w)$

$$J(\Phi^T a) = \frac{1}{2}\sum_{n=1}^{N}[(\Phi^T a)^T\phi(x_n) - t_n]^2 + \frac{\lambda}{2}(\Phi^T a)^T(\Phi^T a) =$$

$$= \frac{1}{2}\sum_{n=1}^{N}[a^T\Phi\cdot\phi(x_n) - t_n]^2 + \frac{\lambda}{2}a^T\Phi\Phi^T a =$$

$$= \frac{1}{2}\sum_{n=1}^{N}[a^T\Phi\cdot\phi(x_n) - t_n]^T[a^T\Phi\cdot\phi(x_n) - t_n] + \frac{\lambda}{2}a^T\Phi\Phi^T a =$$

$$= \frac{1}{2}a^T\Phi\Phi^T\Phi\Phi^T a - a^T\Phi\Phi^T t + \frac{1}{2}t^Tt + \frac{\lambda}{2}a^T\Phi\Phi^T a \quad (6.5)$$

We define $K = \Phi\Phi^T$ – the Gram matrix

where $K_{nm} = \phi(x_n)^T\phi(x_m) = k(x_n, x_m)$ (6.6)

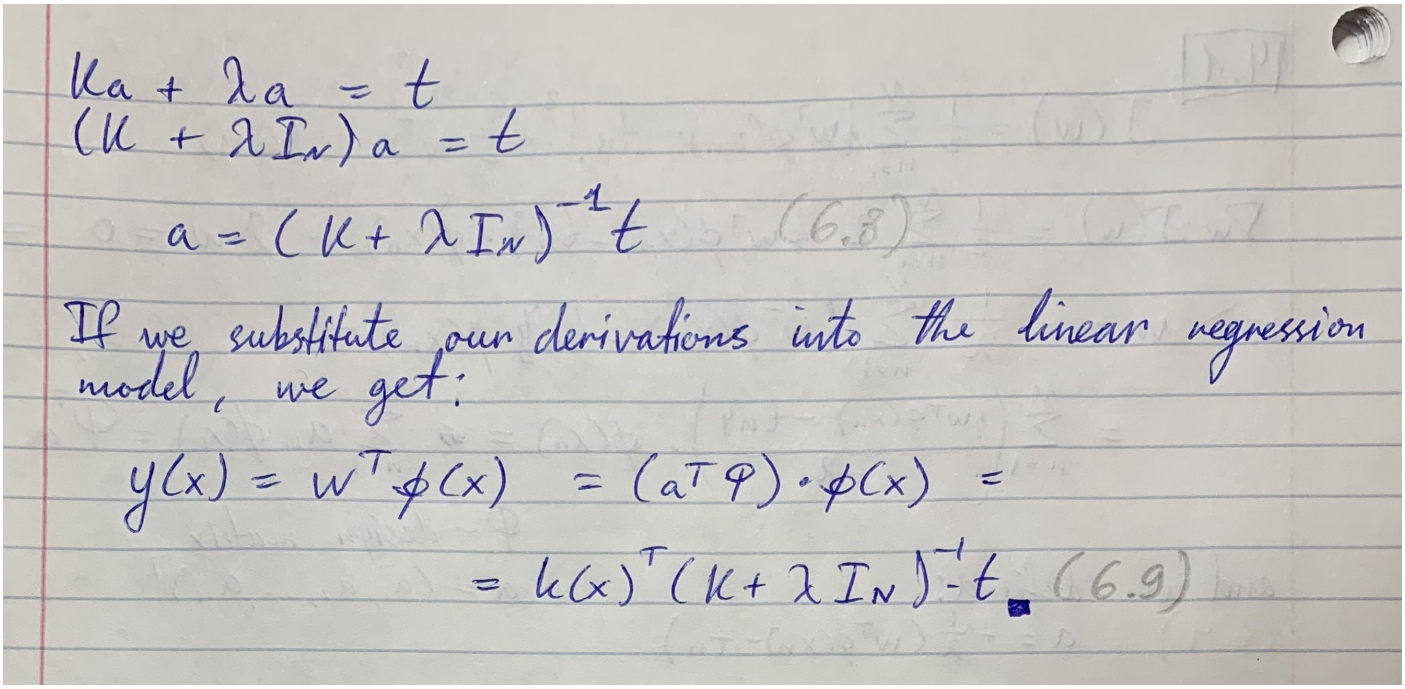
We can now substitute K in $J(a)$

$$J(a) = \frac{1}{2}a^T KKa - a^T Kt + \frac{1}{2}t^Tt + \frac{\lambda}{2}a^T Ka \quad (6.7)$$

We then compute $\nabla_a J(a) = 0$

$$\nabla_a J(a) = KKa - Kt + \lambda Ka = 0$$

$$KKa + \lambda Ka = Kt$$



$$Ka + \lambda a = t$$
$$(K + \lambda I_N)a = t$$
$$a = (K + \lambda I_N)^{-1} t \qquad (6.8)$$

If we substitute our derivations into the linear regression model, we get:

$$y(x) = w^T \phi(x) = (a^T \Phi) \cdot \phi(x) =$$
$$= k(x)^T (K + \lambda I_N)^{-1} t \qquad (6.9)$$

Problem 4.2 Kernels [10 pts]

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

4.2 $k(x, x') = \phi(x)^T \phi(x')$ – by definition

where $\phi(x)$ – are non-linear feature space mapping

then

$k_1(x, x') = \phi_1(x)^T \phi_1(x')$

now consider $k(x, x') = c \cdot k_1(x, x')$

We now must be able to find a valid representation using some feature vectors

if we take $\phi(x) = c^{1/2} \phi_1(x)$ then,

$$k(x, x') = c^{1/2} \phi_1(x)^T \cdot c^{1/2} \phi_1(x) =$$

$$= c \cdot \phi_1(x)^T \cdot \phi_1(x) = c \cdot k_1(x, x')$$

Now we show $k(x, x') = f(x) \cdot k_1(x, x') \cdot f(x')$

$k(x, x') = f(x) \cdot \phi_1(x)^T \phi_1(x) \cdot f(x')$

If we define:

$v(x) = \phi_1(x) \cdot f(x)$, then

$k(x, x') = v(x)^T v(x')$ – it can be written as a scalar product of feature mappings

It means that $k(x, x') = f(x) \cdot k_1(x, x') \cdot f(x')$ is a valid Kernel

Problem 4.3 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.

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