# 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)](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 [122]:

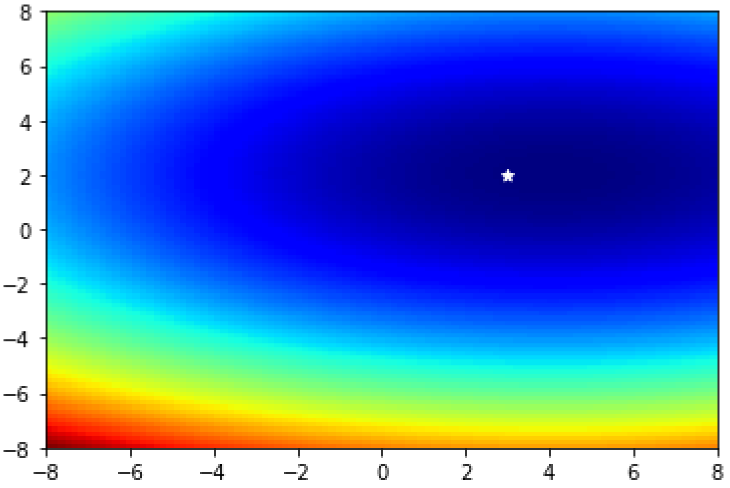
```
from cvxopt import matrix, solvers
# Turns off the printing of CVXOPT solution for the rest of the notebook
solvers.options['show_progress'] = False

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 [123]:

```
%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?

Answer:
The unconstrained local minimum lies outside of the allowable sub-space. Therefore the optimal solution cannot lie at the minimum as that would violate the constraint. Instead the solution lies on the constraint at the point nearest to he 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 $\phi$ 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 $x_n \in \mathbb{R}^d, n \in 1, \dots, N$, 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:

$$\begin{array}{lll} \min\limits_{w,\xi,b} & \frac{1}{2}||w||^2 + C\sum\limits_{n=1}^{N}\xi_n & \\ \text{subject to} & y_n\hat{y}(x_n) \geq 1 - \xi_n, & n = 1, \dots, N \\ & \xi_n \geq 0, & n = 1, \dots, N \end{array}$$

# 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 $w, \xi, b$. 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 $\chi$? (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 [124]: np.bmat([[np.identity(3), np.zeros((3, 1))],
                   [np.zeros((2, 3)), -np.ones((2, 1))]])
```

```
Out[124]: matrix([[ 1.,  0.,  0.,  0.],
                  [ 0.,  1.,  0.,  0.],
                  [ 0.,  0.,  1.,  0.],
                  [ 0.,  0.,  0., -1.],
                  [ 0.,  0.,  0., -1.]])
```

In [125]:

```
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 eithe
r 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 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 havi
ng True for support
        # vectors and False for the rest.
#         margin = 1/np.sqrt(self.weights_.T@self.weights_)
        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.

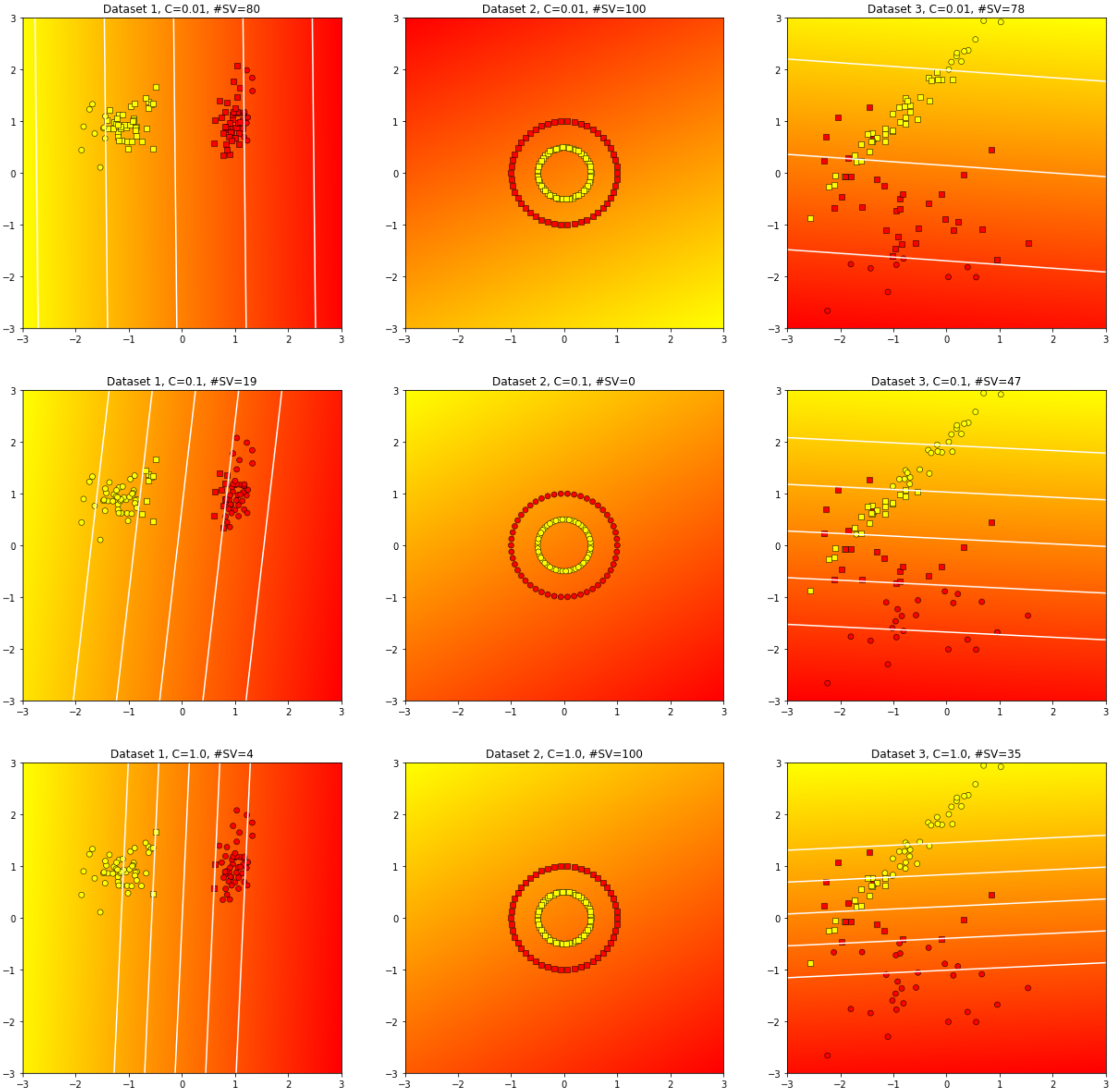
In [126]:

```
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_redundan
t=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_redundan
t=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 [127]:

```
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', 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=(-2,-1, 0, 1,2), 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()
```

```
/Users/danielvaroli/opt/anaconda3/lib/python3.7/site-packages/ipykernel
_launcher.py:26: UserWarning: No contour levels were found within the d
ata range.
```



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

Answer:

As we increase the value of C, we increase the penalty of mis-classification, as a result we bring 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 [128]: model.weights_
```

```
Out[128]: array([-0.0784521 ,  1.62264867])
```

```
In [129]: model.bias_
```

```
Out[129]: -0.3528510092782581
```

## Problem 2.2 [10 points]

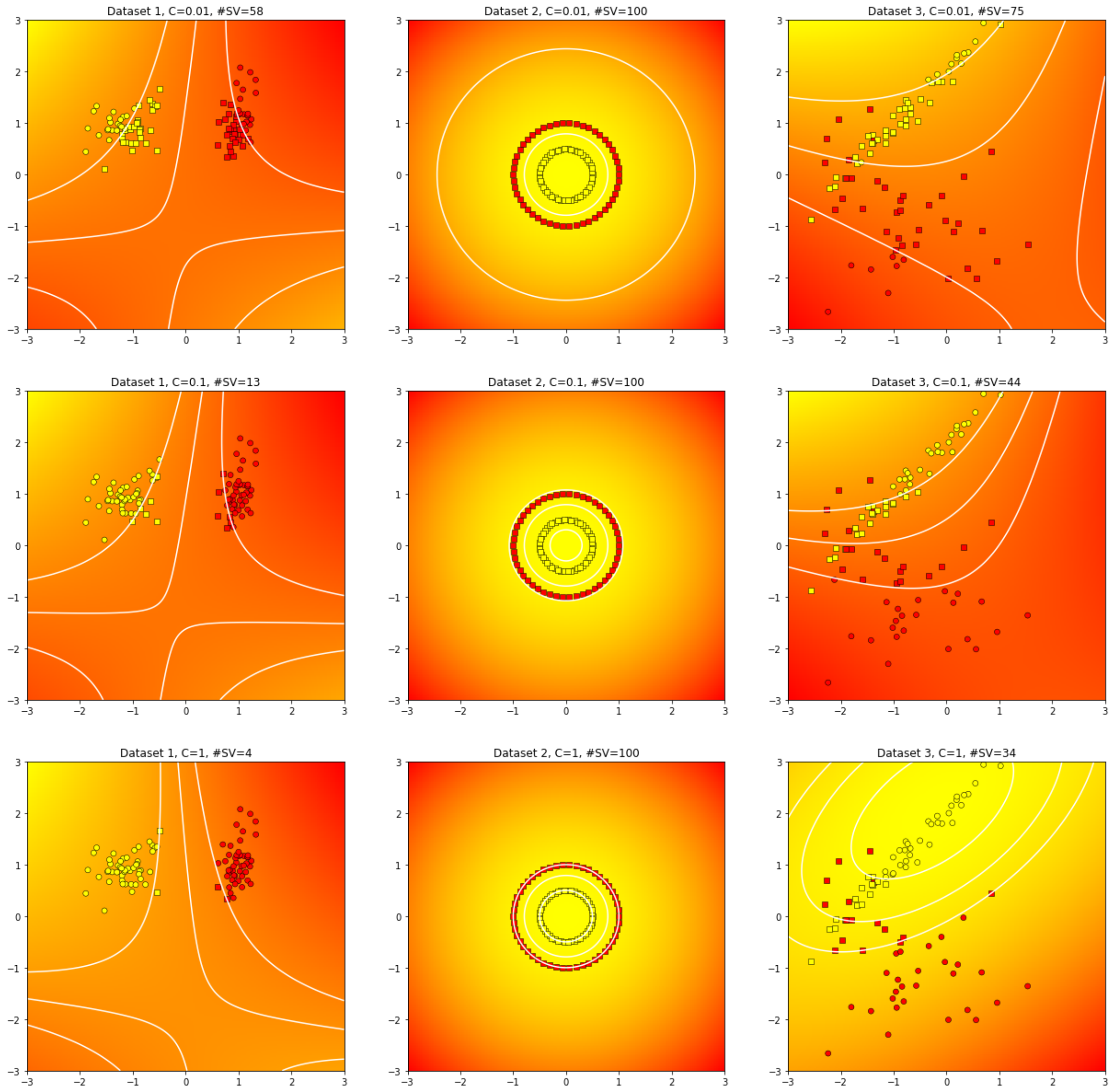
Even using a linear SVM, we are able to separate data that is linearly inseparable by using feature transformation.

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 [130]: 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 [131]:

```
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', 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

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

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

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

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 $\phi$, 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? lagrange multipliers
2. How can we maximize a function given a tool for minimization? Take the negative
3. What is the definition of a support vector in the dual representation?

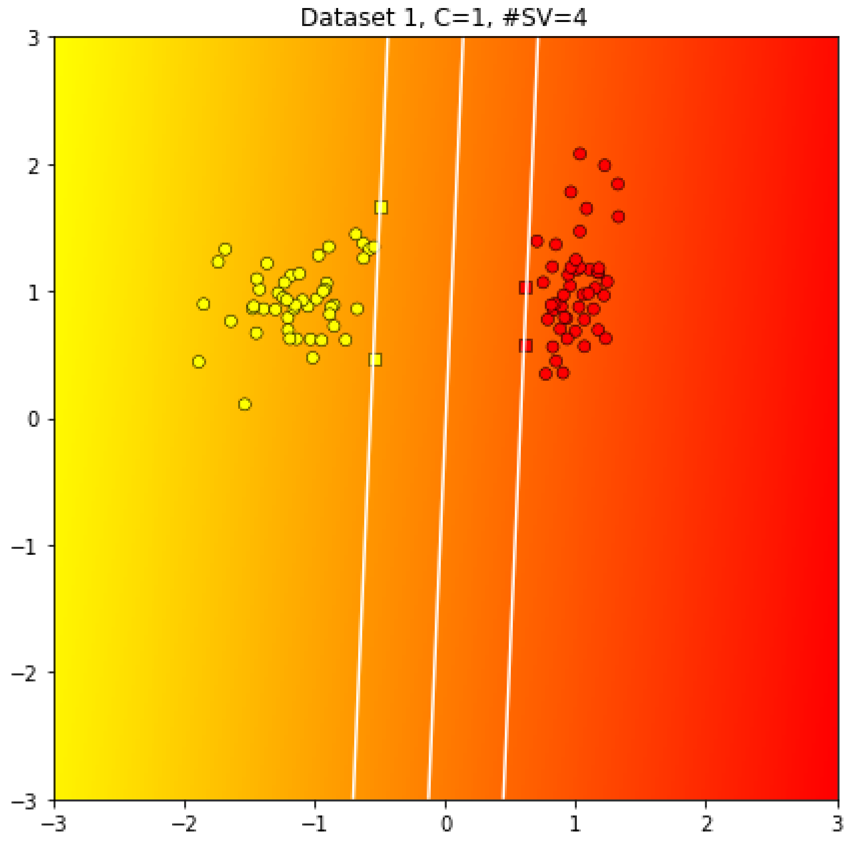
In [193]:
```
class KernelSVM(BaseEstimator):
    def __init__(self, C, kernel=linear_kernel):
        self.C = C
        self.kernel = kernel

    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,num_features = X.shape

        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 problem described above.
        P = matrix(np.outer(Y,Y) * K)
        q = matrix(-1. * np.ones(N) )
        A = 1. * matrix(Y.reshape((1,-1)))
        b = matrix(0.0)
        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))]]))

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

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


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

#         ind = np.arange(len(a))[sv]

        sv_ind = self.support_vectors.nonzero()[0]
        self.alpha_sup = self.alpha_[sv_ind]
        self.X_sup = X[sv_ind]
        self.Y_sup = Y[sv_ind]
        self.n_sv = len(sv_ind)
        print("Num Support Vectors: %d" % self.n_sv)
        #---------------------------------------------------------------
---------------------------
        # Compute bias

        self.bias_ = 0
        for n in range(len(self.alpha_sup)):
            self.bias_ += self.Y_sup[n]
            self.bias_ -= np.sum(self.alpha_sup * self.Y_sup * K[sv_ind[
n],self.support_vectors])

        self.bias_ /= self.n_sv

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


    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.
        """
        y_predict = np.zeros(len(X))
        for i in range(len(X)):
            s = 0
            for alpha_sup, Y_sup, X_sup in zip(self.alpha_sup, self.Y_su
p, self.X_sup):
                s += alpha_sup * Y_sup * self.kernel(X[i], X_sup)
            y_predict[i] = s

        return y_predict + 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))
```

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

In [194]:

```
def linear_kernel(x, y):
    return np.dot(x, y)


C = 1
i = 0

plt.figure(figsize=(7, 7))
#----------------------------------------------------------------------------
model = KernelSVM(C=C, kernel=linear_kernel)
#----------------------------------------------------------------------------
model.fit(X[i], y[i])
sv = model.support_vectors ######### EDITED ##### should be 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, linestyles='solid')

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

```
Num Support Vectors: 4
```



# Problem 3.2 [5 points]

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

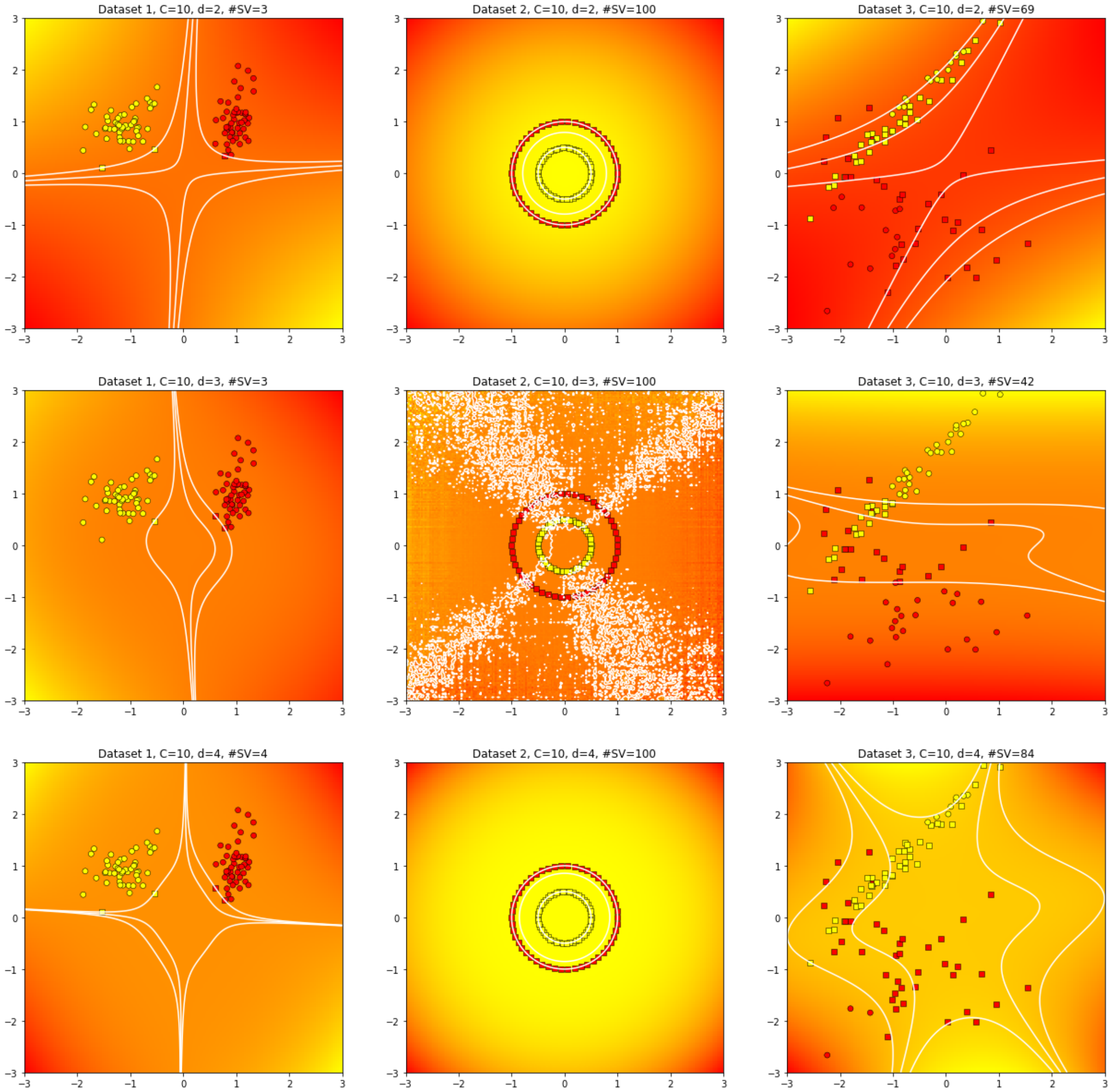
In [195]:
```python
def polynomial_kernel(d, c=0):
    """Returns a polynomial kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [polynomial kernel of degree d with bias parameter c] of x and y. A scalar.
        """
        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])) == 144, 'Polynomial kernel implemented incorrectly'
```

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

In [196]:

```
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', 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], c
map='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', linewidt
hs=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))
```

```
Num Support Vectors: 3
Num Support Vectors: 100
Num Support Vectors: 69
Num Support Vectors: 3
Num Support Vectors: 100
Num Support Vectors: 42
Num Support Vectors: 4
Num Support Vectors: 100
Num Support Vectors: 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 [197]:

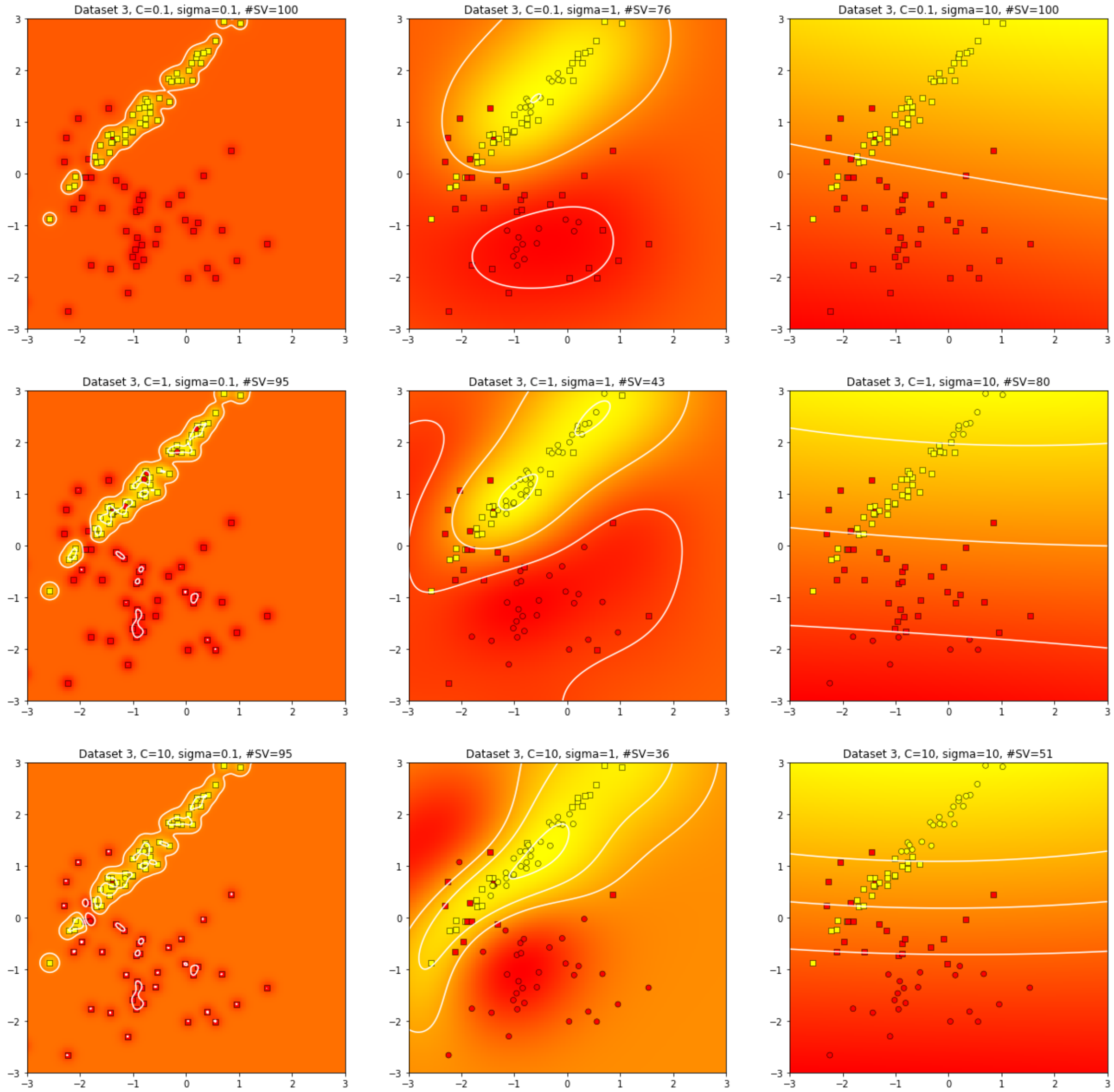
```
def RBF_kernel(sigma):
    """Returns an RBF kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [rbf kernel with parameter sigma] of x and y. A scalar.
        """
        return np.exp(-np.linalg.norm(x-y)**2 / (2 * (sigma ** 2)))
    return kernel
```

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

In [188]:

```
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', 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], c
map='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', linewidt
hs=1.5, zorder=1, linestyles='solid')

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

```
100 support vectors out of 100 points
76 support vectors out of 100 points
100 support vectors out of 100 points
95 support vectors out of 100 points
43 support vectors out of 100 points
80 support vectors out of 100 points
95 support vectors out of 100 points
36 support vectors out of 100 points
51 support vectors out of 100 points
```



Dataset 3, C=0.1, sigma=0.1, #SV=100

Dataset 3, C=0.1, sigma=1, #SV=76

Dataset 3, C=0.1, sigma=10, #SV=100

Dataset 3, C=1, sigma=0.1, #SV=95

Dataset 3, C=1, sigma=1, #SV=43

Dataset 3, C=1, sigma=10, #SV=80

Dataset 3, C=10, sigma=0.1, #SV=95

Dataset 3, C=10, sigma=1, #SV=36

Dataset 3, C=10, sigma=10, #SV=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 [17]:

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

```
Init signature: SVC(C=1.0, kernel='rbf', degree=3, gamma='auto_deprecat
ed', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_siz
e=200, class_weight=None, verbose=False, max_iter=-1, decision_function
_shape='ovr', random_state=None)
Docstring:
C-Support Vector Classification.

The implementation is based on libsvm. The fit time complexity
is more than quadratic with the number of samples which makes it hard
to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided
kernel functions and how `gamma`, `coef0` and `degree` affect each
other, see the corresponding section in the narrative documentation:
:ref:`svm_kernels`.

Read more in the :ref:`User Guide <svm_classification>`.

Parameters
----------
C : float, optional (default=1.0)
    Penalty parameter C of the error term.

kernel : string, optional (default='rbf')
    Specifies the kernel type to be used in the algorithm.
    It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'
or
    a callable.
    If none is given, 'rbf' will be used. If a callable is given it is
    used to pre-compute the kernel matrix from data matrices; that matr
ix
    should be an array of shape ``(n_samples, n_samples)``.

degree : int, optional (default=3)
    Degree of the polynomial kernel function ('poly').
    Ignored by all other kernels.

gamma : float, optional (default='auto')
    Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

    Current default is 'auto' which uses 1 / n_features,
    if ``gamma='scale'`` is passed then it uses 1 / (n_features * X.std
())
    as value of gamma. The current default of gamma, 'auto', will chang
e
    to 'scale' in version 0.22. 'auto_deprecated', a deprecated version
of
    'auto' is used as a default indicating that no explicit value of ga
mma
    was passed.

coef0 : float, optional (default=0.0)
    Independent term in kernel function.
    It is only significant in 'poly' and 'sigmoid'.
```

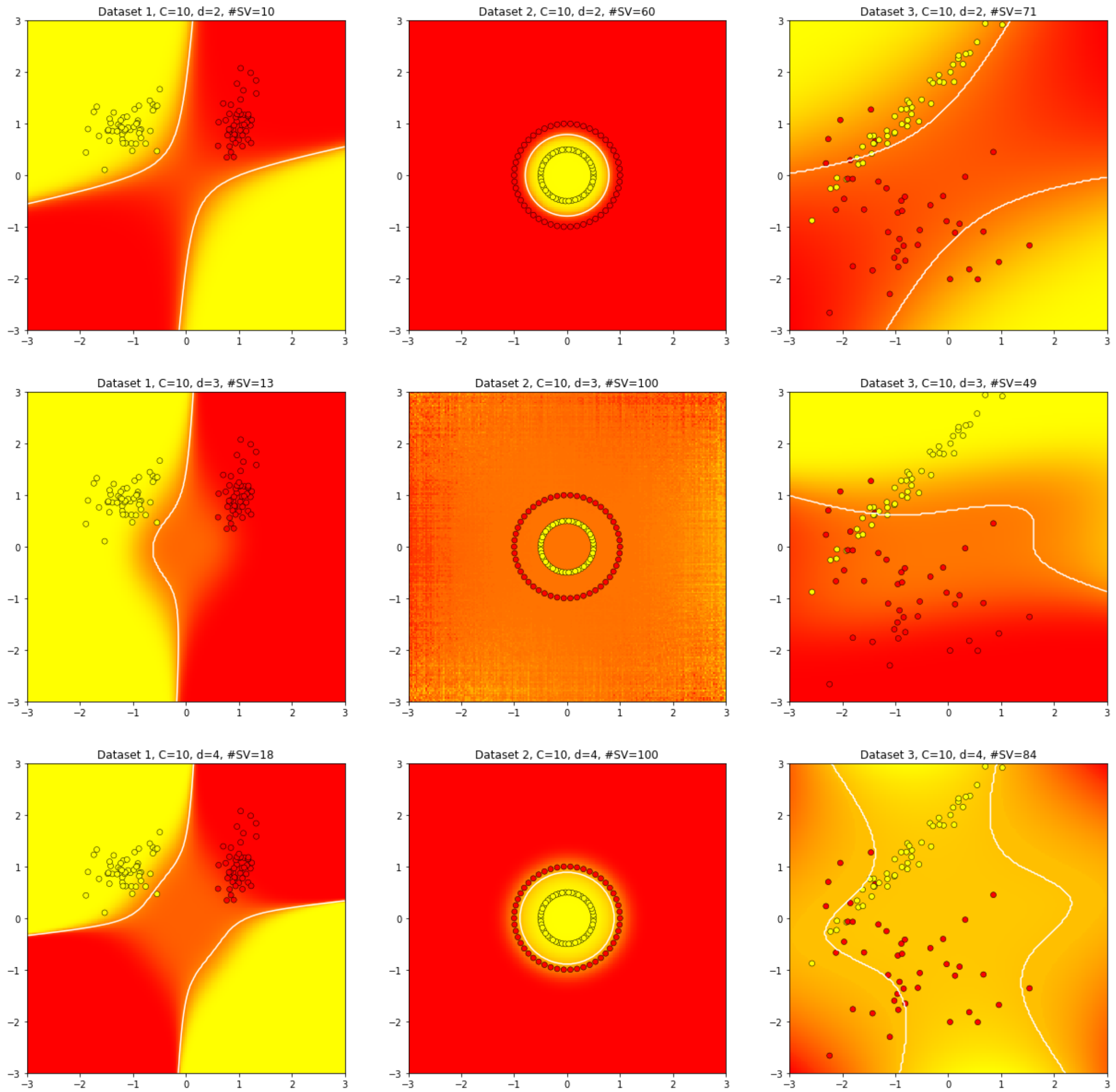
```
shrinking : boolean, optional (default=True)
    Whether to use the shrinking heuristic.

probability : boolean, optional (default=False)
    Whether to enable probability estimates. This must be enabled prior
    to calling `fit`, and will slow down that method.

tol : float, optional (default=1e-3)
    Tolerance for stopping criterion.

cache_size : float, optional
    Specify the size of the kernel cache (in MB).

class_weight : {dict, 'balanced'}, optional
    Set the parameter C of class i to class_weight[i]*C for
    SVC. If not given, all classes are supposed to have
    weight one.
    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input da
ta
    as ``n_samples / (n_classes * np.bincount(y))``

verbose : bool, default: False
    Enable verbose output. Note that this setting takes advantage of a
    per-process runtime setting in libsvm that, if enabled, may not wor
k
    properly in a multithreaded context.

max_iter : int, optional (default=-1)
    Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape : 'ovo', 'ovr', default='ovr'
    Whether to return a one-vs-rest ('ovr') decision function of shape
    (n_samples, n_classes) as all other classifiers, or the original
    one-vs-one ('ovo') decision function of libsvm which has shape
    (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one
    ('ovo') is always used as multi-class strategy.

    .. versionchanged:: 0.19
        decision_function_shape is 'ovr' by default.

    .. versionadded:: 0.17
       *decision_function_shape='ovr'* is recommended.

    .. versionchanged:: 0.17
       Deprecated *decision_function_shape='ovo' and None*.

random_state : int, RandomState instance or None, optional (default=Non
e)
    The seed of the pseudo random number generator used when shuffling
    the data for probability estimates. If int, random_state is the
    seed used by the random number generator; If RandomState instance,
    random_state is the random number generator; If None, the random
    number generator is the RandomState instance used by `np.random`.

Attributes
----------
```

```
support_ : array-like, shape = [n_SV]
    Indices of support vectors.

support_vectors_ : array-like, shape = [n_SV, n_features]
    Support vectors.

n_support_ : array-like, dtype=int32, shape = [n_class]
    Number of support vectors for each class.

dual_coef_ : array, shape = [n_class-1, n_SV]
    Coefficients of the support vector in the decision function.
    For multiclass, coefficient for all 1-vs-1 classifiers.
    The layout of the coefficients in the multiclass case is somewhat
    non-trivial. See the section about multi-class classification in th
e
    SVM section of the User Guide for details.

coef_ : array, shape = [n_class * (n_class-1) / 2, n_features]
    Weights assigned to the features (coefficients in the primal
    problem). This is only available in the case of a linear kernel.

    `coef_` is a readonly property derived from `dual_coef_` and
    `support_vectors_`.

intercept_ : array, shape = [n_class * (n_class-1) / 2]
    Constants in decision function.

Examples
--------
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = SVC(gamma='auto')
>>> clf.fit(X, y) #doctest: +NORMALIZE_WHITESPACE
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rb
f',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]

See also
--------
SVR
    Support Vector Machine for Regression implemented using libsvm.

LinearSVC
    Scalable Linear Support Vector Machine for classification
    implemented using liblinear. Check the See also section of
    LinearSVC for more comparison element.
File:           c:\miniconda\envs\cvx\lib\site-packages\sklearn\svm\cla
sses.py
Type:           ABCMeta
```

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 = 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, 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', 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, len(model.support_vectors_)))
```

```
C:\miniconda\envs\cvx\lib\site-packages\matplotlib\contour.py:1243: Use
rWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```


Dataset 1, C=10, d=2, #SV=10
Dataset 2, C=10, d=2, #SV=60
Dataset 3, C=10, d=2, #SV=71
Dataset 1, C=10, d=3, #SV=13
Dataset 2, C=10, d=3, #SV=100
Dataset 3, C=10, d=3, #SV=49
Dataset 1, C=10, d=4, #SV=18
Dataset 2, C=10, d=4, #SV=100
Dataset 3, C=10, d=4, #SV=84

# 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.

# Problem 4.2 Kernels [10 pts]

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

# 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.

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

# Problem 4.1

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

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

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

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

$$= \Phi^T a \qquad \underline{(6.3)} \qquad\qquad a = (a_1, a_2 \dots a_N)^T$$

we define $a_n = \left(-\frac{1}{\lambda}(w^T\phi(x_n) - t_n)\right)$ $\underline{(6.4)}$

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

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

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

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

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

we now define the Gram matrix $\underline{\underline{K}} = \underline{\underline{\Phi}}\,\underline{\underline{\Phi}}^T$

where the $(n,m)$ element is

$$K_{n,m} = \phi(x_n)^T \phi(x_m) = k(x_n, x_m) \qquad \underline{(6.6)}$$

$\Rightarrow$ we can rewrite $J(a)$ as

$$J(a) = \frac{1}{2} a^T \left(\Phi \Phi^T\right)\left(\Phi \Phi^T\right) a - a^T \left(\Phi \Phi^T\right) t + \frac{1}{2} t^T t + \frac{\lambda}{2} a^T \left(\Phi \Phi^T\right) a$$

$$= \frac{1}{2} a^T K K a - a^T K t + \frac{1}{2} t^T t + \frac{\lambda}{2} a^T K a \qquad \underline{(6.7)}$$

we now compute $\nabla_a J(a)$ and set it equal to $0$

$\Rightarrow$

$$\nabla_a J(a) = K K a - K t + \lambda K a = 0$$

$$\Rightarrow \quad K K a + \lambda K a = K t$$

$$K a + \lambda a = t$$

$$(K + \lambda I_N) a = t$$

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

We now substitute (6.8) into the linear Regression model given by $y(x) = w^T \phi(x)$

tova $w = \Phi^T a$

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

$\underline{(6.9)}$

# Problem 4.2

a) consider the definition of a Kernel given by

$\kappa(x,x') = \phi(x)^T \phi(x')$ , where $\phi(x)$ - are a non-linear feature space mapping.

consider $\kappa_1(x,x') = \phi_1(x)^T \phi_1(x')$ , a kernel

now consider $\kappa(x,x') = c \cdot \kappa_1(x,x')$ , if $\kappa$ is a valid kernel we must be able to find a valid representation using some feature vectors

if we take $\phi(x) = c^{\frac{1}{2}} \phi_1(x)$ we can rewrite

$$\boxed{\begin{aligned} \kappa(x,x') &= c^{\frac{1}{2}} \phi_1(x)^T \cdot c^{\frac{1}{2}} \phi_1'(x) = c \cdot \phi_1(x)^T \phi_1(x) \\ &= c \cdot \kappa_1(x,x') \end{aligned}}$$

$\Rightarrow$ $c \cdot \kappa_1(x,x')$ - is a valid kernel

b) Now show that $\kappa(x,x') = f(x)\,\kappa_1(x,x')\,f(x')$ is a valid kernel, where $f(x) \rightarrow$ any function

we can rewrite $\kappa(x,x')$

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

if we define $v(x) = \phi_1(x) \cdot f(x)$, we can write

$$k(x,x') = v(x)^T v(x')$$

$\Rightarrow$ $k(x,x')$ can be written as an scalar product of feature mappings $\Rightarrow$

$$k(x,x') = f(x)\, k_1(x,x')\, f(x') \text{ is a valid kernel}$$

# Problem 4.3

suppose $\exists\ w_1, b_1$ that satisfy the solution to equation (7.5) and minimize equation (7.3). Our constraint will have the form

$$t_n(w_1^T\phi(x_n)+b_1)\geq 1$$

then suppose we have a constraint

$$t_n(w^T\phi(x_n)+b)\geq\gamma \qquad ,\ \gamma\geq 0$$

if we rescale our variables such that $w=\gamma w_1$ and $b=\gamma b_1$, we will still minimize (7.3) and our constraint would become

$$t_n((\gamma w_1)^T\phi(x_n)+\gamma b_1)\geq\gamma$$

$$\cancel{\gamma}\cdot t_n(w_1^T\phi(x_n)+b_1)\geq\cancel{\gamma}$$

$$t_n(w_1^T\phi(x_n)+b_1)\geq 1$$

therefore replacing 1 on the r.h.s with $\gamma>0$, does not change the solution for the maximum margin hyperplane