Problem Set 6: Support Vector Machines

This assignment requires a working IPython Notebook installation.

Only PDF files are accepted for submission. To print this notebook to a pdf file, you can go to "File" -> "Download as" -> "PDF via LaTex(.pdf)" or simply use "print" in browser.

Total: 100 points (+ 50 bonus points)

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 Problem 1 and reading the textbook, this will help a lot in the programming assignment.

Problem 1 [30 pts]

1.1 Dual Representations [10 pts]

In class we saw that the SVM classifier with parameters $\mathbf{w}, b$ trained on n points $\{\mathbf{x}_i, y_i\}$ can be expressed in either the "primal" form

$$h(\mathbf{x}) = sign(\mathbf{w^T x} + b)$$

or the "dual" form

$$h(\mathbf{x}) = sign(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x_i^T x} + b)$$

The dual form involves a "kernel function" which evaluates dot products $\mathbf{x_i^T x}$ between the input point and the training points. We can think of these values as similarities of the input $\mathbf{x}$ to the training points.

It turns out that many linear models we have seen before can be re-cast into an equivalent "dual representation" in which the predictions are also based on linear combinations of a kernel function evaluated at the training data points.

This is described in sections 6.0-6.1 in Bishop. Read it 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 in your own words: *how can the regularized least-squares regression be formulated in the dual form?*

We can reformulate the least squared algorithm in terms of the parameter vector a instead of working with parameter vector w

1.2 Kernels [10 pts]

Read Section 6.2 and verify the results (6.13) and (6.14) for constructing valid kernels, i.e. prove the kernels constructed by (6.13) and (6.14) are valid.

From Bishop: function $k(x, x')$ is a valid kernel if matrix K, with elements from $k(x_n, x_m)$, is positive semi-definite for all x_n Thus:

(6.13) $k(x, x') = ck_1(x, x')$ with c > 0 constant. And we also have $k_1(x, x')$ is a valid kernel, thus multiplying $k_1(x, x')$ with constant $c > 0$ would not change the semi-definite status of the kernel. Therefore $ck_1(x, x')$ is valid

(6.14) $k(x, x') = f(x)k_1(x, x')f(x')$ with f(.) being any function

$f(x)k_1(x, x')f(x') = f(x)[\phi(x)^T\phi(x')]f(x') = [f(x)\phi](x)^T[f(x')\phi](x') = \phi^{'}(x)^T\phi^{'}(x')$ is a valid kernel, since $\phi(x)^T\phi(x')$ is valid

1.3 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.

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

Assume b = 0, thus we have $t_n(w^T\phi(x_n)) \geq 1$ This inequality is not changed with constant $\gamma > 0$

$$\gamma * t_n(w^T\phi(x_n)) \geq \gamma$$

Therefore the solution for maximum margin hyperplane is unchanged

Implementing the SVM

Note, some of the code in cells below 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 Quadratic Program (QP) can be formulated as

$$\min_{x} \quad \frac{1}{2}x^TPx + q^Tx \tag{1}$$
$$\text{subject to} \quad Gx \preceq h \tag{2}$$
$$Ax = b \tag{3}$$

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

FYI, you can read about the methods for solving quadratic programming problems here (optional).

Problem 2 [10 points]

Design appropriate matrices to solve the following QP problem:

$$\min_{x} \quad f(x) = x_1^2 + 4x_2^2 - 8x_1 - 16x_2 \tag{4}$$
$$\text{subject to} \quad x_1 + x_2 \leq 5 \tag{5}$$
$$x_1 \leq 3 \tag{6}$$
$$x_2 \geq 0 \tag{7}$$

Hint: first notice that if $x = [x_1, x_2]^T$ and P is a matrix

$$\begin{vmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{vmatrix}$$

then $x^TPx = p_{11}x_1^2 + (p_{12} + p_{21})x_1x_2 + p_{22}x_2^2$. We have filled in the correct P below.

In [32]:
```
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.])
#print(q)
#print(G)
#print(h)
#--------------------------------------------------------------------------------
```

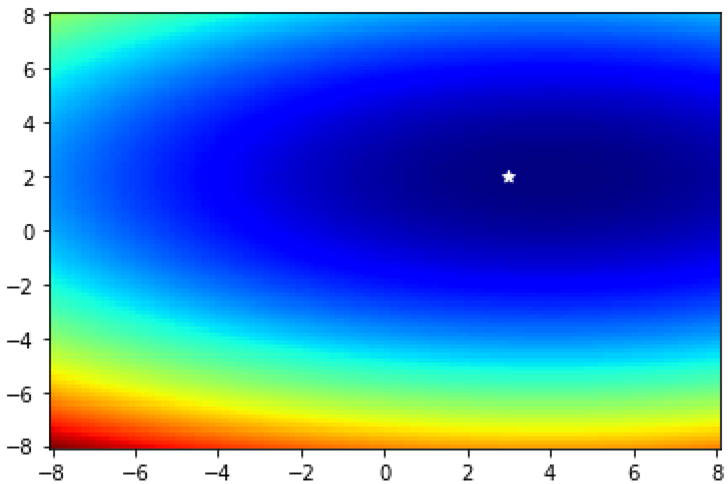
```
sol = solvers.qp(P, q, G, h)
x1, x2 = sol['x']
print('Optimal x: ({:.8f}, {:.8f})'.format(x1, x2))
```

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

Let's visualize the solution

In [33]:

```
%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', shading="auto")
plt.scatter([x1], [x2], marker='*', color='white')
plt.show()
```



Why is the solution not in the minimum? Because of constraint conditions x1 + x2 <= 5; x1 <= 3; x2 >= 0

Linear SVM

Now, let's implement linear SVM. We will do this for the general case that allows class distributions to overlap, i.e. the linearly non-separable case **(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:

$$\begin{aligned} \min_{w,\xi,b} \quad & \frac{1}{2}||w||^2 + C\sum_{n=1}^{N}\xi_n & & \quad (8) \\ \text{subject to} \quad & y_n\hat{y}(x_n) \geq 1 - \xi_n, & n = 1, \ldots, N & \quad (9) \\ & \xi_n \geq 0, & n = 1, \ldots, N & \quad (10) \end{aligned}$$

Note that the above is a quadratic programming problem.

Problem 3.1 [50 points]

Your task is to implement the linear SVM above using a QP solver by designing appropriate matrices P, q, G, h. Complete the code below.

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 [34]:
```
np.bmat([[np.identity(3), np.zeros((3, 1))],
         [np.zeros((2, 3)), -np.ones((2, 1))]])
```

Out[34]:
```
matrix([[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0., -1.],
        [ 0.,  0.,  0., -1.]])
```

In [135…
```
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 described above.
        # 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)), np.zeros((N+1,N+1))]]))
        q = matrix(np.bmat([[np.zeros([1, d]), self.C * np.ones([1, N]),np.zeros([1, 1])]])).T
        G = matrix(np.bmat([[-np.dot(np.diag(Y), X), -np.identity(N), -np.array(Y).reshape((N, 1))],
                            [np.zeros((N, d)), -np.identity(N), np.zeros((N, 1))]]))
        h = matrix(np.bmat([[-np.ones([N, 1])],[np.zeros([N, 1])]]))

        #print(g,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]
        #print(ans.shape)
        #print(self.xi_.shape)
        #-----------------------------------------------------------------------------------------
        # Find support vectors. Must be a boolean array of length N having True for support
        # vectors and False for the rest.
        margin = Y * (X.dot(model.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.
```

```
        """
        # return y_hat
        #pass
        if self.transform is not None:
            X = self.transform(X)
        y_hat = X.dot(self.weights_) + self.bias_
        return y_hat

    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 [136…

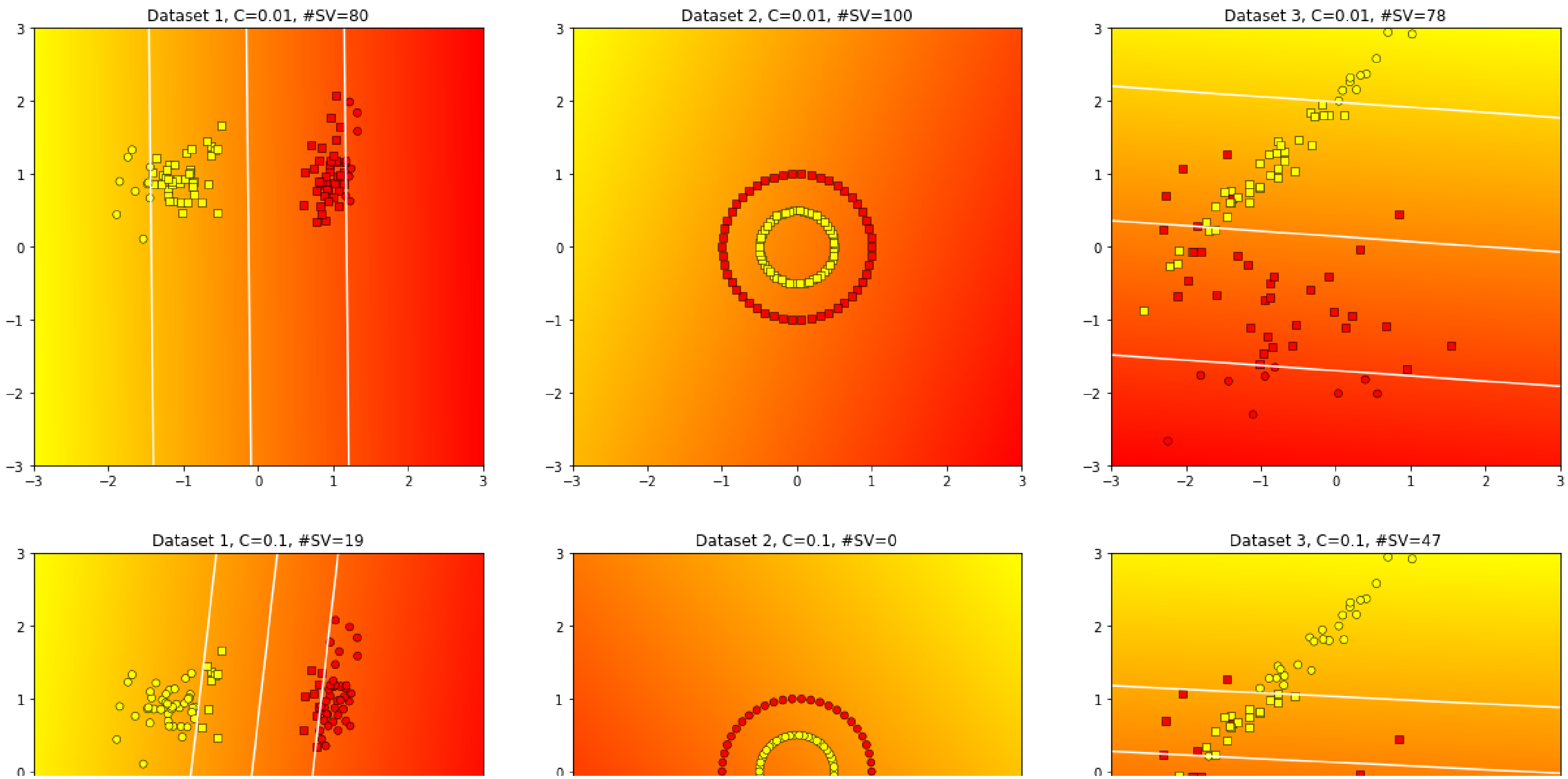
```
from sklearn.datasets import make_classification, make_circles
X = [None, None, None]
y = [None, None, None]
X[0], y[0] = make_classification(n_samples=100, n_features=2, n_redundant=0, n_clusters_per_class=1, random_state=1)
X[1], y[1] = make_circles(n_samples=100, factor=0.5)
X[2], y[2] = make_classification(n_samples=100,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

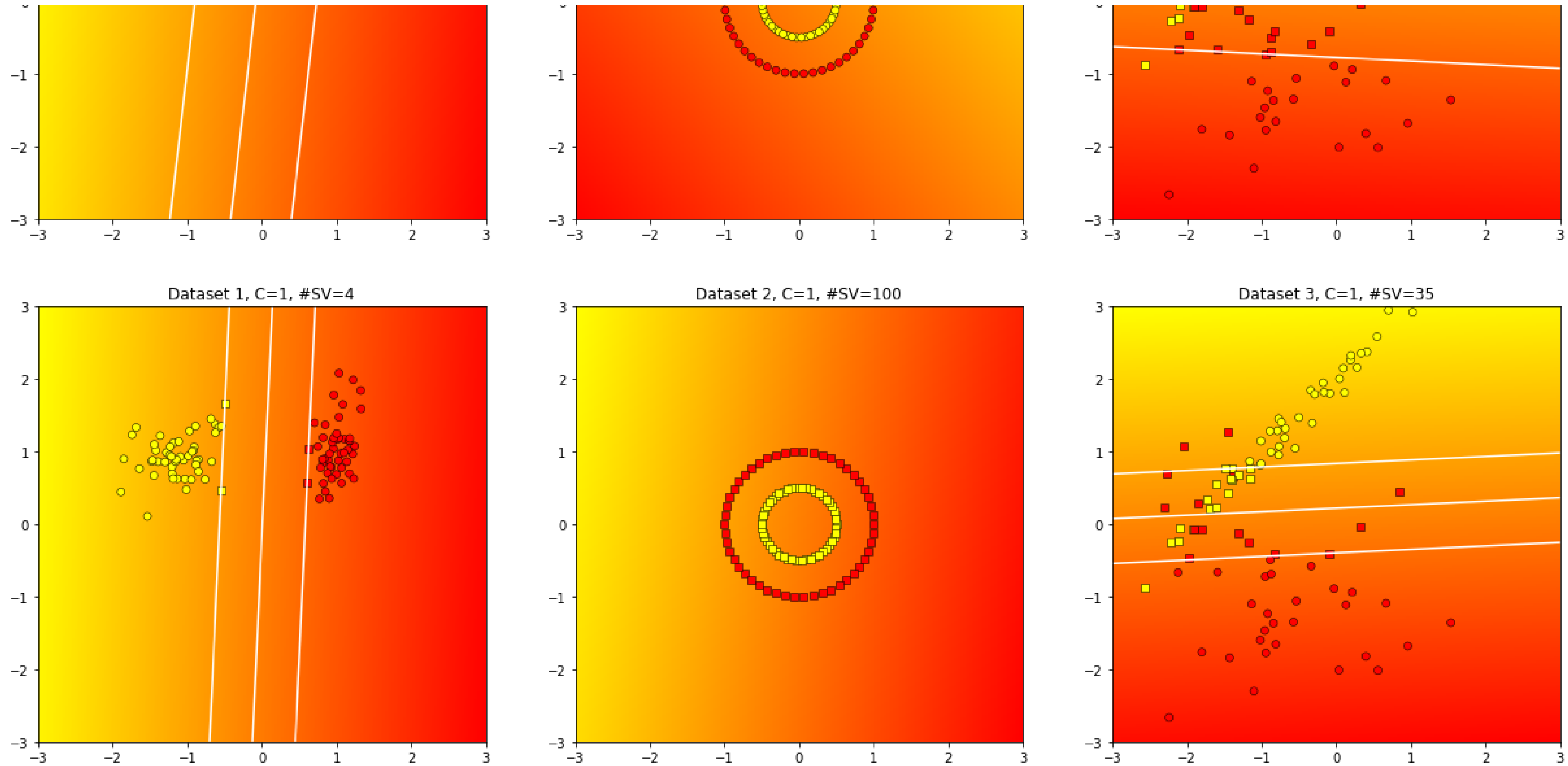
In [137…

```
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, shading="auto")
        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()
```

```
<ipython-input-137-81e3971017ed>:26: UserWarning: No contour levels were found within the data range.
  plt.contour(xx, yy, zz, levels=(-1, 0, 1,), colors='w', linewidths=1.5, zorder=1, linestyles='solid')
```





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

For debug purposes: the very last model must have almost the same weights and bias as:

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

$$b = -0.3528510092782581$$

In [113…

```
model.weights_
```

Out[113…

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

In [114…

```
model.bias_
```

Out[114…

```
-0.352851009278258
```

Problem 3.2 [10 points]

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

Implement the following feature transformation $\phi(x_1, x_2) = (x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1 x_2)$ and re-run your SVM.

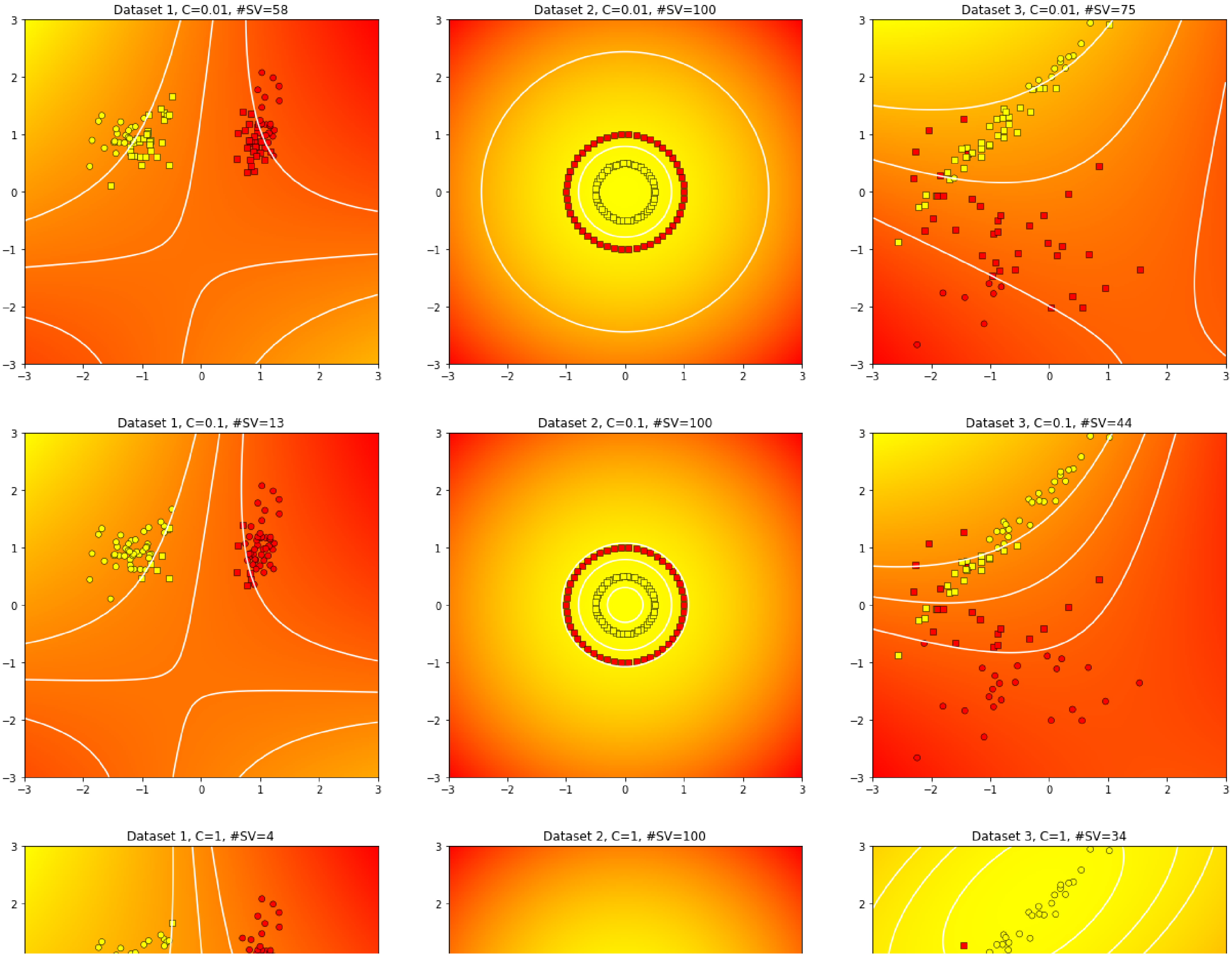
In [118…

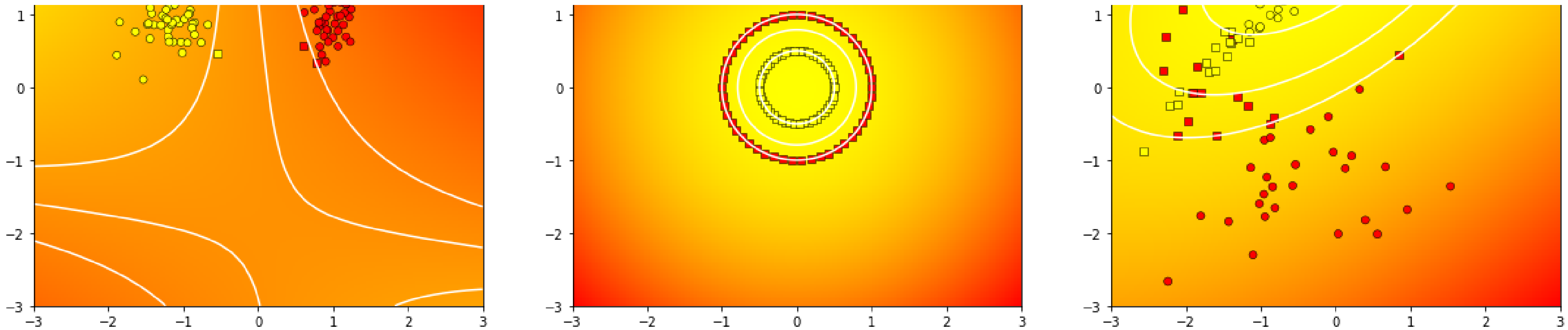
```
def append_second_order(X):
    """Given array Nx[x1, x2] return Nx[x1, x2, x1^2, x2^2, x1x2]."""
    # return new_X
    #pass
    return np.concatenate((X, X * X, (X[:, :1] * X[:, 1:])), axis = 1)

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

In [119…

```
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, shading="auto")
        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

$$\begin{aligned} \max_{\alpha} \quad & \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m k(x_n, x_m) && (11) \\ \text{subject to} \quad & 0 \le \alpha_n \le C, \quad n = 1, \dots, N && (12) \\ & \sum_{n=1}^{N} \alpha_n y_n = 0 && (13) \end{aligned}$$

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 4.1 [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 [120…

```
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 as t_i in Bishop.
        """
        N = len(Y)

        #------------------------------------------------------------------------------------------
        # Construct appropriate matrices here to solve the optimization problem described above.

        P = matrix([Y[i] * Y[j] * self.kernel(X[i], X[j]) for i in range(N) for j in range(N)], (N,N))
        #print(P)
        q = matrix(-np.ones(N))
        G = matrix(np.bmat([[-1. * np.identity(N)], [1. * np.identity(N)]]))
        h = matrix([0.] * N + [self.C] * N)
        A = matrix(1. * Y, (1, N))
        b = matrix(0.)
        #------------------------------------------------------------------------------------------

        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]
```

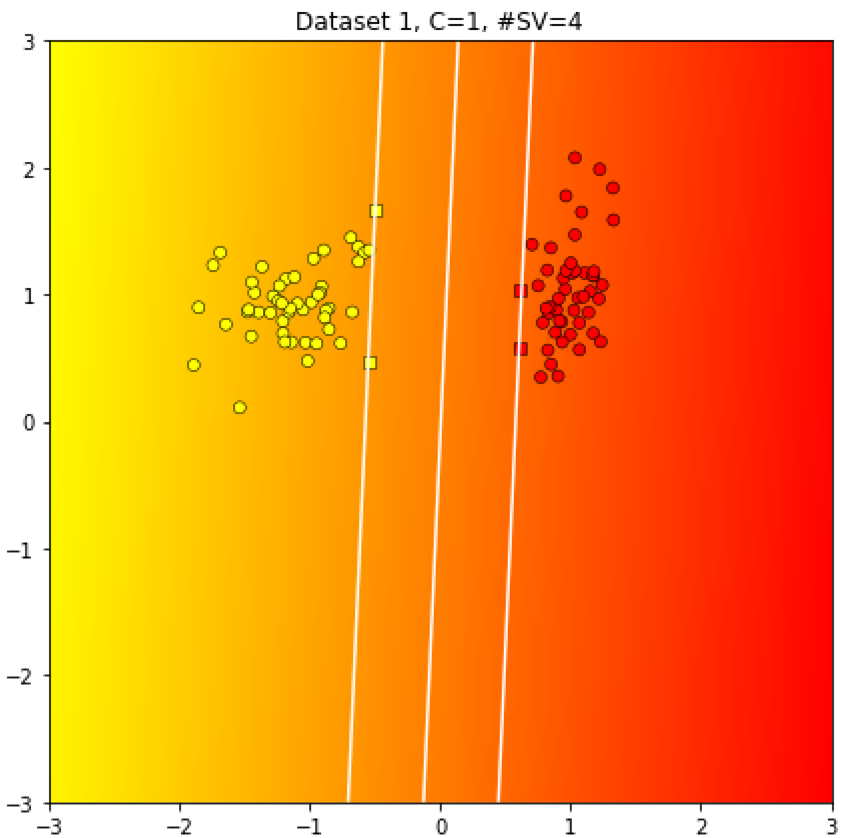
```
        self.Y_sup = Y[sv_ind]
        self.alpha_sup = self.alpha_[sv_ind]
        self.n_sv = len(sv_ind)

        #-----------------------------------------------------------------------------------------
        # Compute bias

        self.bias_ = np.mean([self.Y_sup[i] - np.sum([self.alpha_sup[j] * self.Y_sup[j] * self.kernel(self.X_sup[i], self
        #-----------------------------------------------------------------------------------------

    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
        return [np.sum([self.alpha_sup[i] * self.Y_sup[i] * self.kernel(self.X_sup[i], X[j]) for i in range(self.n_sv)])

    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 [121…

```
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, shading="auto")
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()
```



Problem 4.2 [5 points]

Implement a polynomial kernel function (wiki).

In [122…

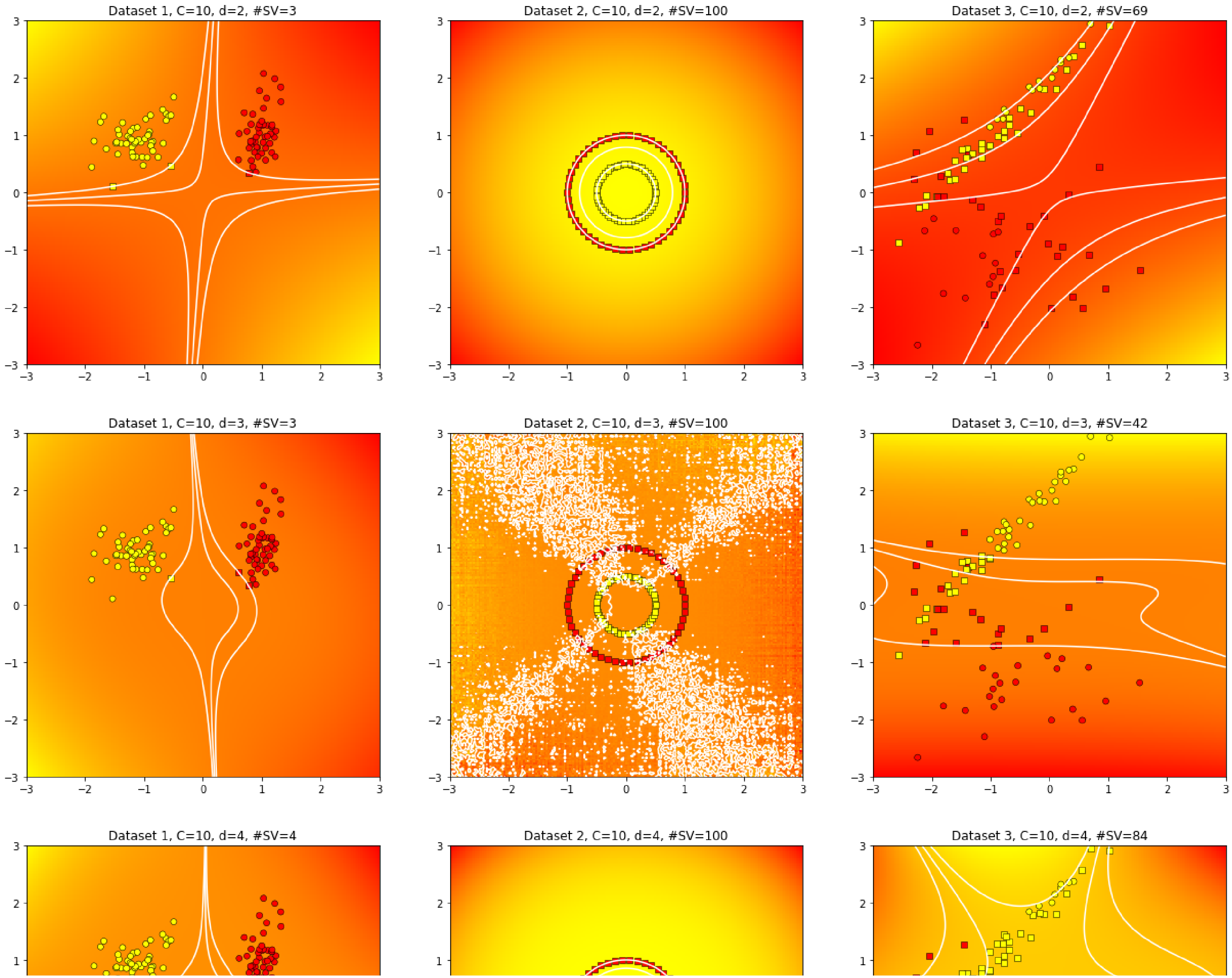
```
def polynomial_kernel(d, c=0):
    """Returns a polynomial kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [polynomial kernel of degree d with bias parameter c] of x and y. A scalar.
        """
        #pass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

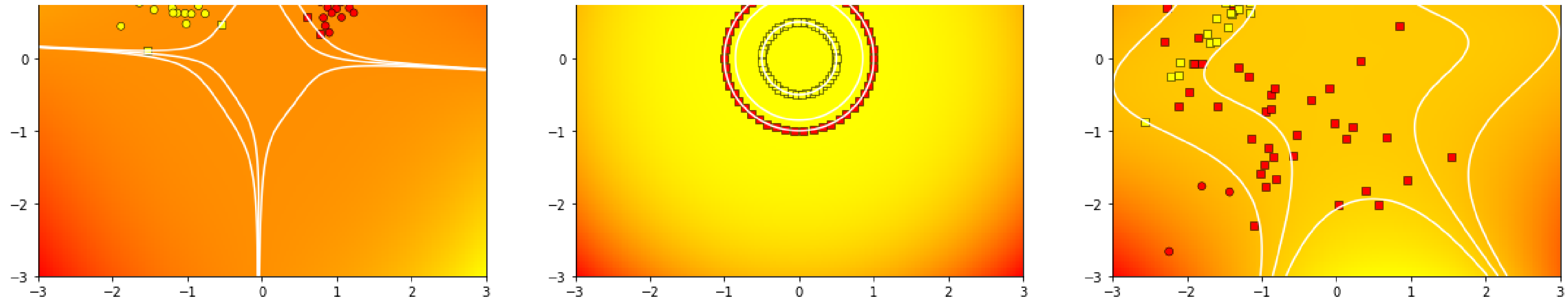
Let's see how it performs. This might take some time to run.

In [123…

```
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], 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, shading="auto")
        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))
```





Problem 4.3 [5 points]

Finally, you need to implement a **radial basis function** kernel (wiki).

In [124…

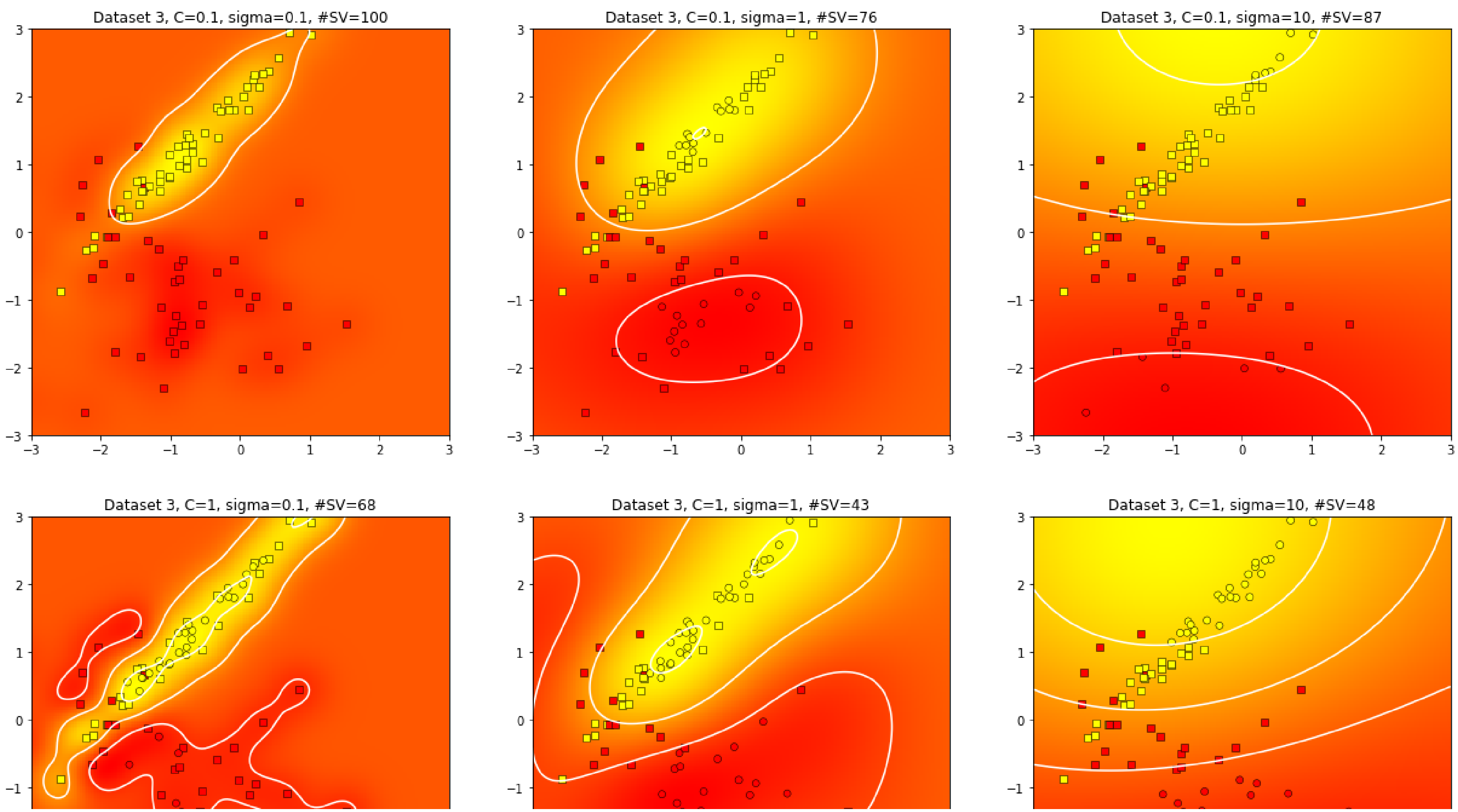
```
def RBF_kernel(sigma):
    """Returns an RBF kernel FUNCTION."""
    def kernel(x, y):
        """
        :param x: vector of size L
        :param y: vector of size L
        :return: [rbf kernel with parameter sigma] of x and y. A scalar.
        """
        return np.exp(-np.sum((x - y) ** 2)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

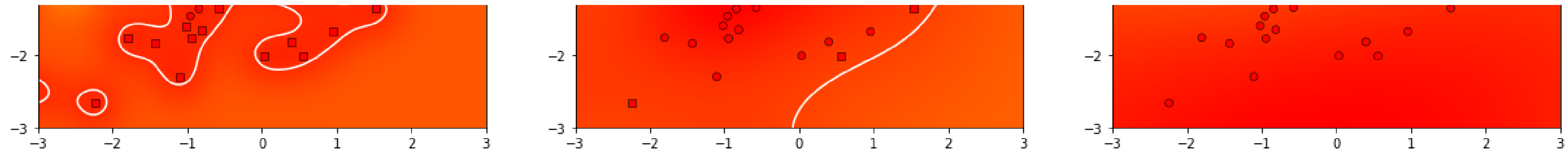
Let's see how it performs. This might take some time to run.

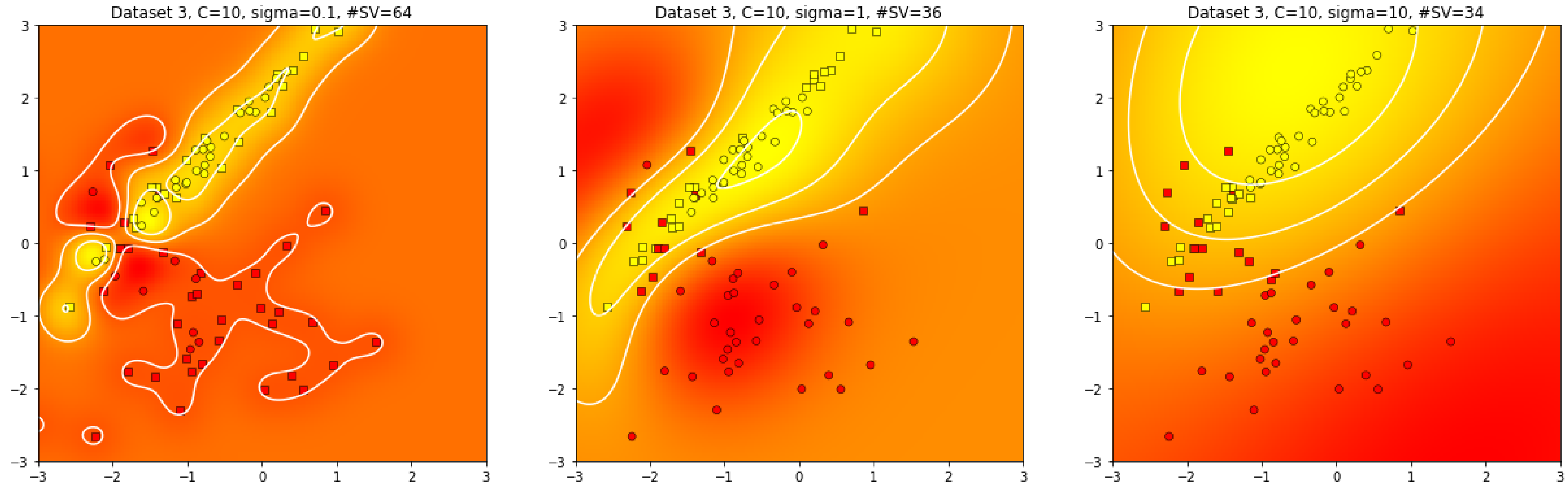
In [125…

```
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], 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, shading="auto")
        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, 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 [17]:

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

```
Init signature: SVC(C=1.0, kernel='rbf', degree=3, gamma='auto_deprecated', coef0=0.0, shrinking=True, probability=False,
tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=Non
e)
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 matrix
    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 change
    to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of
    'auto' is used as a default indicating that no explicit value of gamma
    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 data
        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 work
        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=None)
        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.
```

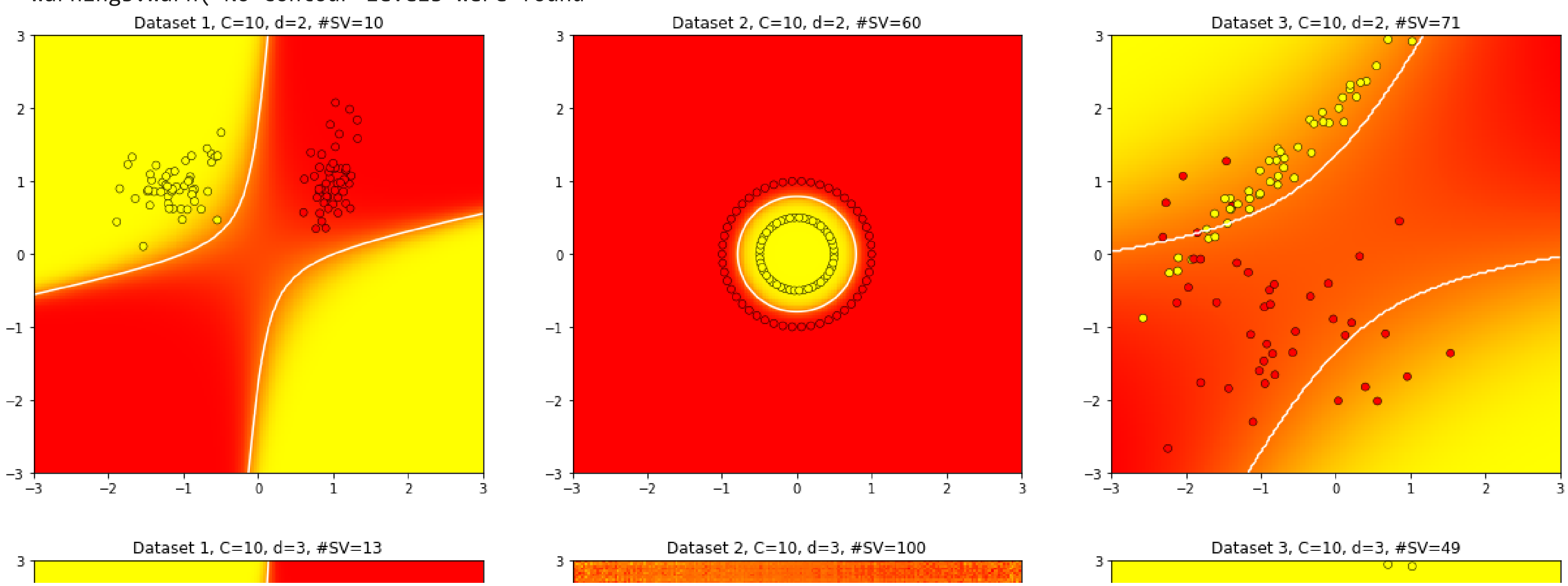
```
n_support_ : array-like, dtype=int32, shape = [n_class]
    Number of support vectors for each class.

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 the
    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='rbf',
    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\classes.py
Type:           ABCMeta
```

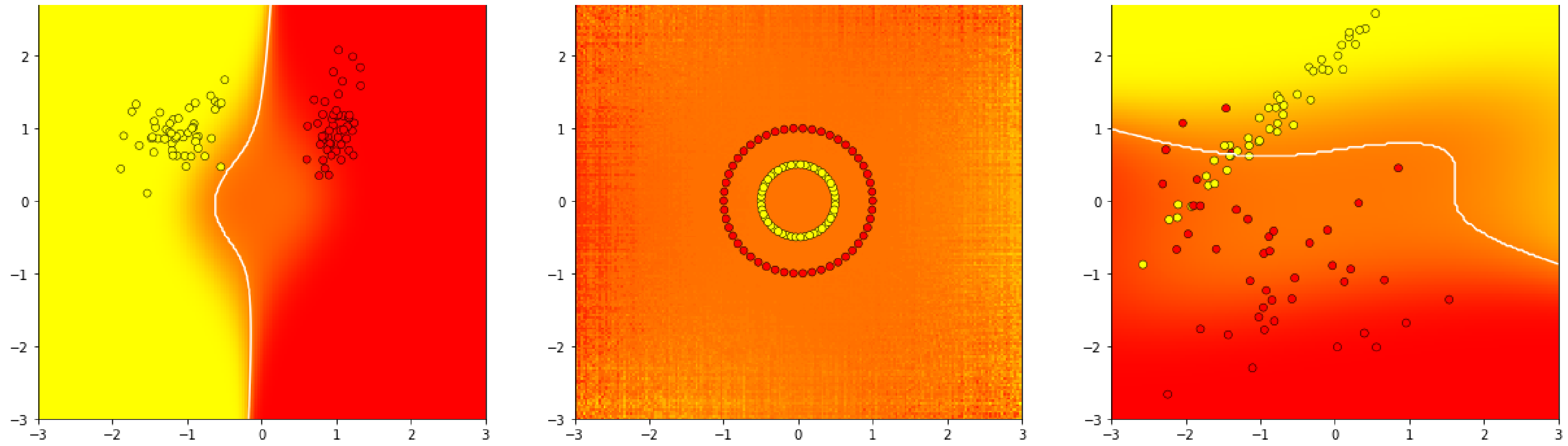
In [18]:

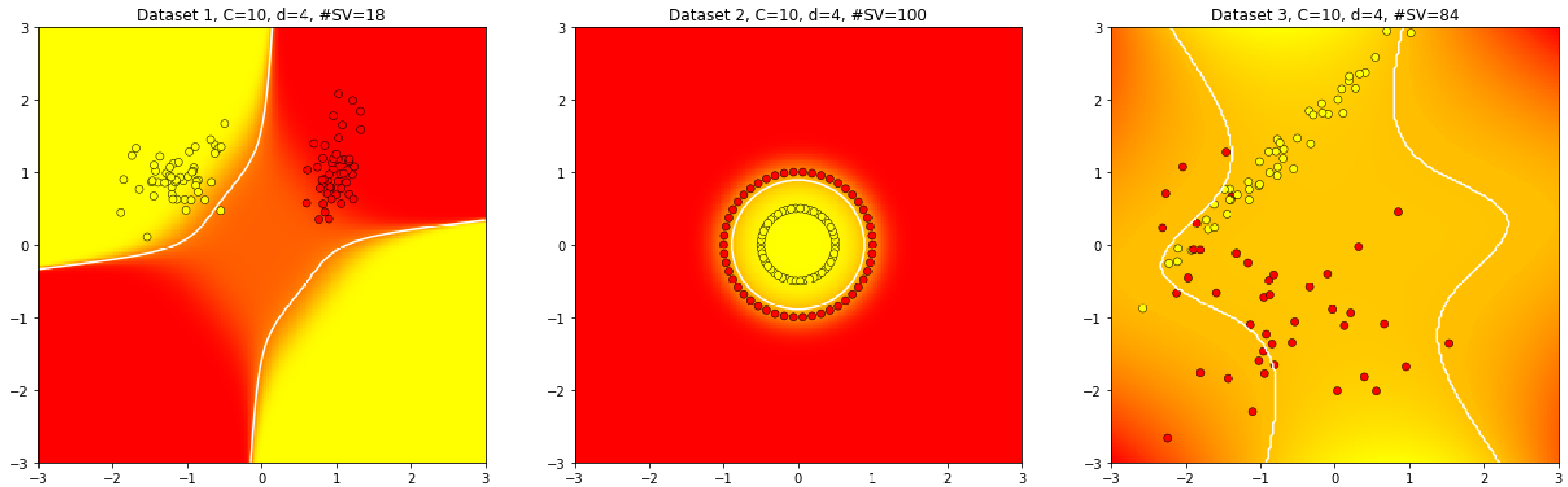
```
plot_i = 0
```

```
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, shading="auto")
        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: UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```






Dataset 1, C=10, d=4, #SV=18
Dataset 2, C=10, d=4, #SV=100
Dataset 3, C=10, d=4, #SV=84