A7_ResponseP

April 17, 2022

Problem 1

Recall that a machine learning model is classified as a kenerlized method when all operations with respect to the input data, $\mathbf{x}$, are inner products. Or rather, if we want to use a kernel method to predict $\hat{y}$, then our prediction contains the inner product $\mathbf{x_i^T x}$, where $\mathbf{x_i}$ is the training data and $\mathbf{x}$ is the data we wish to make a prediction for.

With this in mind, we wish to transform our current prediction equation $\hat{y} = w^T x$ into $\hat{y} = k(x^*, x)$, where k is a kernel which measures the closeness of two x points. Clearly, this is just a matter of substituting some function in for the weights, w^T.

In ridge regression, recall that we find the weights with the following minimization problem:

$$\min_w \left[\frac{1}{2} \sum_i (y_i - w^T x_i) + \frac{\lambda}{2} w^T w \right],$$

where λ is a regularization coefficient which enforces *weight decay*. To better see how we can turn $w^T x$ into a kernelized method, however, it is helpful to write w in it's matrix algebra form:

$$w = \left(\lambda I + \Phi^T \Phi\right)^{-1} \Phi^T y.$$

Now, define $S = \left(\lambda I + \Phi^T \Phi\right)^{-1}$, and recall that $\Phi^T y$ can also be written as

$$\Phi^T(\mathbf{x}) y = \begin{bmatrix} \vdots & \vdots & & \vdots \\ \phi(x_1) & \phi(x_2) & \cdots & \phi(x_n) \\ \vdots & \vdots & & \vdots \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} \phi(x_i) y_i,$$

where ϕ are arbitrary basis functions.

Now, let's use what we have written for w to perform a "kernel trick" on our current prediction equation:

$$\begin{aligned}\hat{y} = w^T x &= x^T w \\ &= xS\Phi^T y \\ &= \sum_i xS\phi(x_i)y_i \\ &= \sum_i k(x, x_i)y_i.\end{aligned}$$

We can observe a few qualities about our kernel. Firstly, note that the prediction equation itself is now a linear combination of the kernel function and our previous training labels. Consequently, when we now wish to predict the value corresponding to some feature vector x, we do so by calculating its proximity to our training feature vectors, $x_i,\ i = 1, \ldots, N$, and then use these distance values to take a weighted average of our training labels, $y_i,\ i = 1, \ldots, N$.

Secondly, note that while we previously defined $S = \left(\lambda I + \Phi^T \Phi\right)^{-1}$, which clearly consists of some finite number of basis functions, our new kernel function, $k(\cdot, \cdot)$, corresponds to an infinite number of basis functions.

Problem 2

(a)

In order to best illustrate the power of the kernel function, I will implement the Sequential Minimization Optimization (SMO) algorithm with the flexibility to either use an inner product or a Gaussian kernel function as its method of point proximity measurement.

[ ]:
```
import numpy as np
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio
from sklearn import datasets
import matplotlib.pyplot as plt
%matplotlib inline
pio.renderers.default = "notebook+pdf"
```

In order to avoid many if statements or duplicate code, a kernel class and its corresponding function are implemented. A kernel object can either be decalared as Gaussian, or will default to a simple inner product.

[ ]:
```
class Kernel:

    def __init__(self, type, s):
        self.type = type
        self.s = s

    def call_kernel(self, x, xquery):
        if (self.type == 'gaussian'):
            return np.exp(-0.5/s**2 * np.sum((x - xquery)**2))
        else:
```

```
            return np.dot(x.T, xquery).sum()
```

[ ]:
```
def discriminant_function(a, y, x, b, xquery, kernel):
    f = b
    n = len(y)

    for i in range(0, n):
        f = f + a[i] * y[i] * kernel.call_kernel(x[:, i], xquery)
    return f
```

[ ]:
```
def smo_simplified(X, y, C, tol, max_passes, kernel):

    n = len(y)

    a = np.zeros((n, 1))
    b = 0

    passes = 0
    while (passes < max_passes):
        num_changed_alphas = 0
        for i in range(0, n):
            Ei = discriminant_function(a, y, X, b, X[:, i], kernel) - y[i]
            if ((y[i]*Ei < -tol and a[i] < C) or (y[i]*Ei > tol and a[i] > 0)):

                # select j ~= i randomly
                temp = np.random.permutation(n)
                j = temp[0]
                if (j == i):
                    j = temp[2]

                Ej = discriminant_function(a, y, X, b, X[:, j], kernel) - y[j]

                ai = a[i]
                aj = a[j]

                # calculate L and H
                if (y[i] != y[j]):
                    L = max(0, aj - ai)
                    H = min(C, C + aj - ai)
                else:
                    L = max(0, ai + aj - C)
                    H = min(C, ai + aj)
                if (L == H):
                    continue

                eta = 2 * kernel.call_kernel(X[:, i].T, X[:, j]) - kernel.
↪call_kernel(X[:, i].T, X[:, i]) - kernel.call_kernel(X[:, j].T, X[:, j])
```

```
                if (eta >= 0):
                    continue

                # calculate a[j] and clip
                a[j] = a[j] - y[j] * (Ei - Ej) / eta
                if (a[j] > H):
                    a[j] = H
                elif (a[j] < L):
                    a[j] = L

                if abs(a[j] - a[i]) < 1e-5:
                    continue

                # calculate a[i]
                a[i] = a[i] + y[i]*y[j]*(aj - a[j])

                # compute b
                b1 = b - Ei - y[i] * (a[i] - ai) * kernel.call_kernel(X[:, i].
↪T, X[:, i]) - y[j] * (a[j] - aj) * kernel.call_kernel(X[:, i].T, X[:, j])
                b2 = b - Ej - y[i] * (a[i] - ai) * kernel.call_kernel(X[:, i].
↪T, X[:, j]) - y[j] * (a[j] - aj) * kernel.call_kernel(X[:, j].T, X[:, j])
                if (0 < a[i] < C):
                    b = b1
                elif (0 < a[j] < C):
                    b = b2
                else:
                    b = (b1 + b2)/2

                num_changed_alphas += 1

        if num_changed_alphas == 0:
            passes = passes + 1
        else:
            passes = 0

        return a, b
```

We will also declare a function to be used to plot the entirety of our model, or to be more specific: - The training data - The contours surrounding the training data (decision regions) - The support vectors

[ ]:
```
def plot_decision_boundary(alphas, X, b, kernel, ax, c, resolution=100):
        """Plots the model's decision boundary on the input axes object.
        Range of decision boundary grid is determined by the training data.
        Returns decision boundary grid and axes object (`grid`, `ax`)."""

        colors = ('b', 'k', 'r')
```

```
        levels = (-1, 0, 1)

        # Generate coord. grid and evaluate model over entirety of grid
        xrange = np.linspace(X[:,0].min(), X[:,0].max(), resolution)
        yrange = np.linspace(X[:,1].min(), X[:,1].max(), resolution)

        grid = [[discriminant_function(alphas, y, X.T, b, np.array([xr, yr]),␣
↪kernel)
        for xr in xrange] for yr in yrange]
        grid = np.array(grid).reshape(len(xrange), len(yrange))

        ax.contour(xrange, yrange, grid, levels=levels, linewidths=(1, 1, 1),
                   linestyles=('--', '-', '--'), colors=colors)
        # Training Data
        ax.scatter(X[:,0], X[:,1],
                   c=y, cmap=plt.cm.jet, lw=0, alpha=0.25)

        if (c != 0):
                title_name = "s = %s; C = %s" % (kernel.s, c)
                ax.set_title(title_name)

        return grid, ax
```

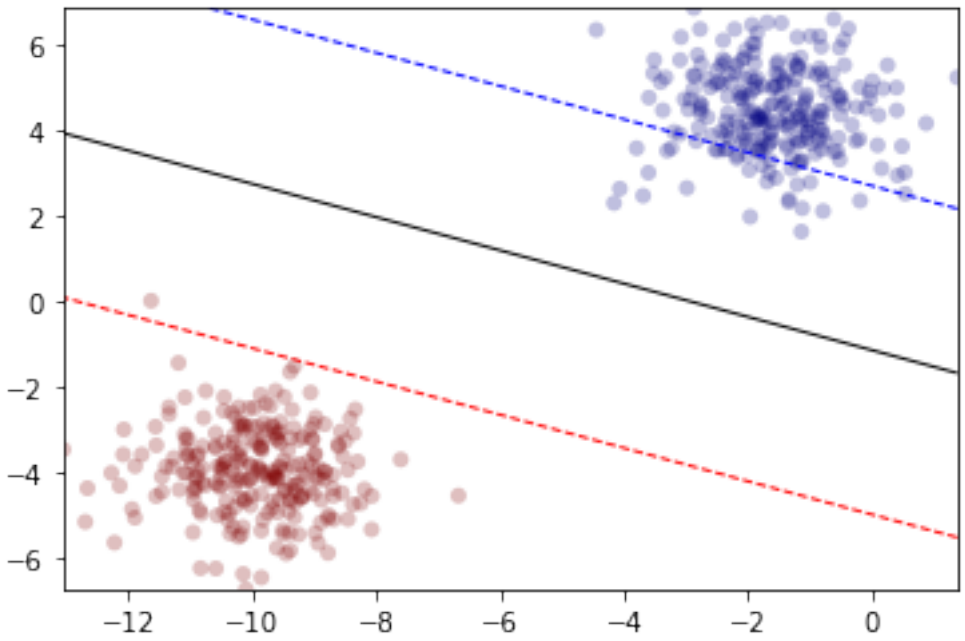
[ ]:
```
# Declare the parameters for the SMO algorithm
C = 1
tol = 1e-6
max_passes = 3000
```

(b)

We will begin our exploration with a non-interesting example: a simple maximum margin classifier. Here we will instantiate linearly-separable data, as well as a kernel object which is designed to measure distance using the inner product.

[ ]:
```
X_train, y = datasets.make_blobs(n_samples=500, centers=2,
                         n_features=2, random_state=1)
y = np.where(y == 0, -1, y)

lin_kernel = Kernel('linear', 0)

alphas, b = smo_simplified(X_train.T, y, C, tol, max_passes, lin_kernel)

b = b.item(0)
```

[ ]:
```
fig, ax = plt.subplots()
grid, ax = plot_decision_boundary(alphas, X_train, b, lin_kernel, ax, 0)
```



As we expected, running the SMO algorithm with the inner product where the training data can be divided into two linearly-separable classes results in a linear discriminant, which does in fact do a good job dividing the decision regions.

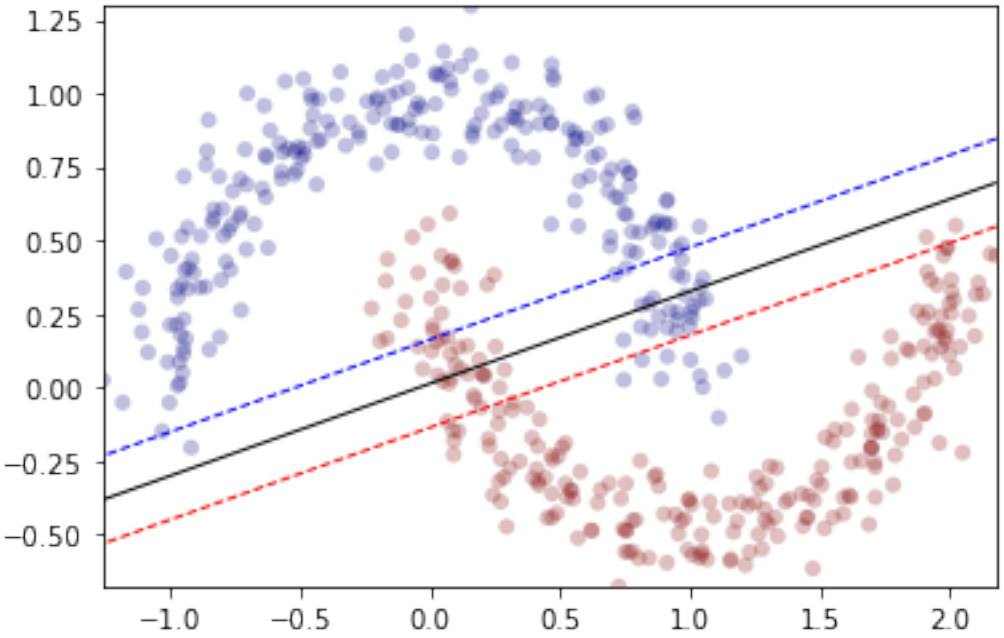
Now, however, let's see how the inner product SMO implementation fairs against training data which is not linearly separable (see plot below).

[ ]:
```
# Initialize Data
X_train, y = datasets.make_moons(n_samples=500, noise=0.1,
                         random_state=1)
y = np.where(y == 0, -1, y)
```

[ ]:
```
data = {
    'x1' : X_train[:, 0],
    'x2' : X_train[:, 1],
    'y' : y
}

df = pd.DataFrame(data)

fig2 = px.scatter(df, x = "x1", y = 'x2', color = 'y')
fig2.show()
```

[ ]:
```
a1, b1 = smo_simplified(X_train.T, y, C, tol, max_passes, lin_kernel)
a1 = np.reshape(a1, len(a1))
```

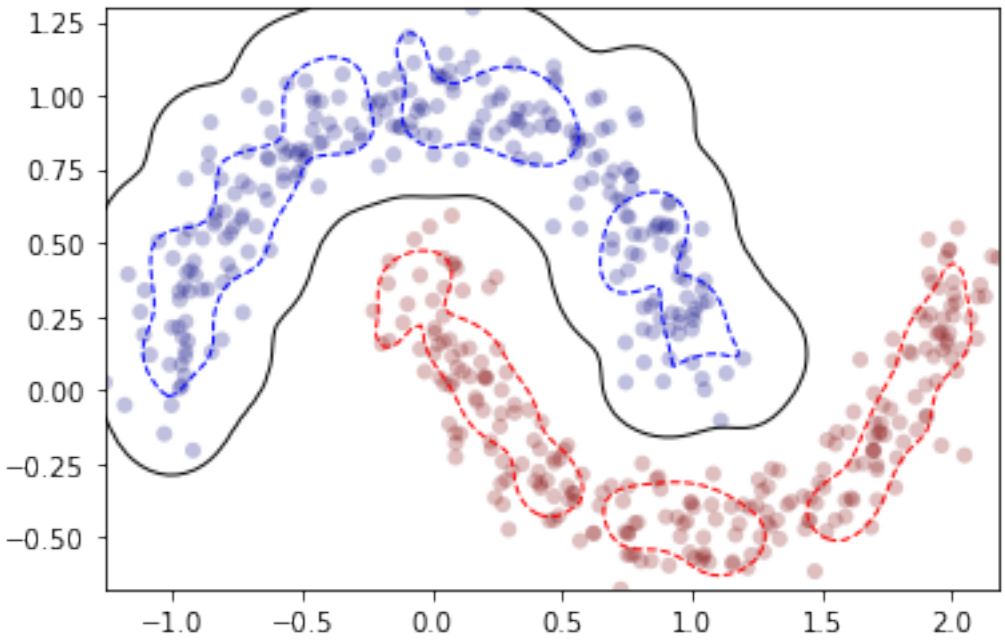
[ ]:
```
fig, ax = plt.subplots()
grid, ax = plot_decision_boundary(a1, X_train, b1, lin_kernel, ax, 0)
```



This time our inner product approach to SMO does not work as well. If we imagine these two classes representing cancer vs. non-cancer patients, clearly we should be motivated to upgrade our prediction capability.

Let's try to classify the same data again, but this time using a Gaussian kernel function.

[ ]:
```
# Instantiate a Gaussian Kernel with sd = 0.1
gauss_kernel = Kernel('gaussian', 0.1)

a1, b1 = smo_simplified(X_train.T, y, C, tol, max_passes, gauss_kernel)
a1 = np.reshape(a1, len(a1))

fig, ax = plt.subplots()
grid, ax = plot_decision_boundary(a1, X_train, b1, gauss_kernel, ax, 0)
```



It is rather needless to say that by replacing the inner product with a Gaussian kernel our classification becomes much more accurate. Our two decision regions (established by the black line), almost perfectly capture the data as broken up into its respective classes.

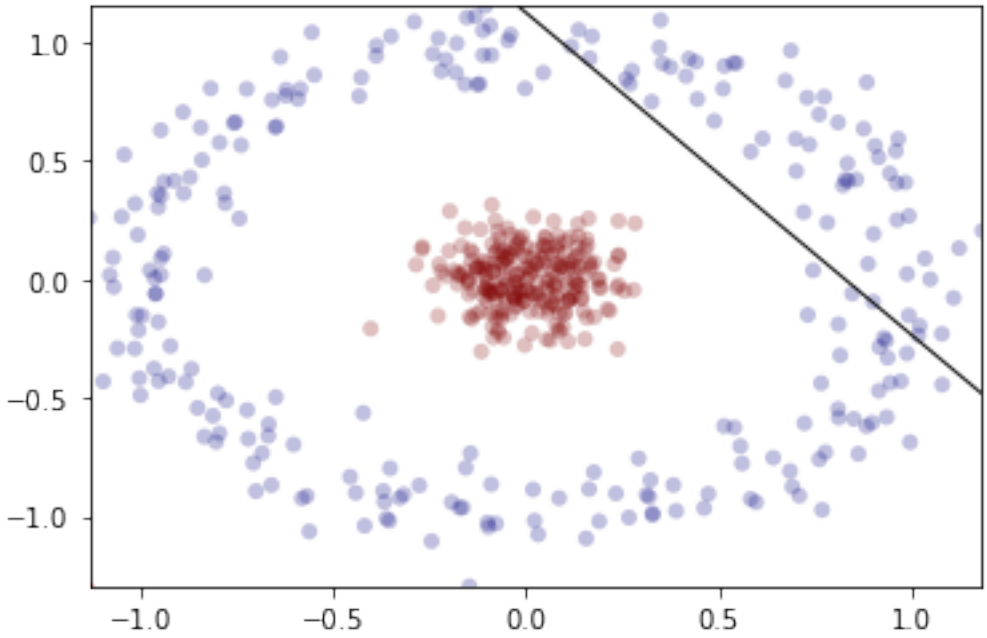
For good measure let's observe one more example: data whose classes are concentric circles (plot below).

[ ]:
```
X_train, y = datasets.make_circles(n_samples=500, noise=0.1,
                                   factor=0.1,
                                   random_state=1)
y = np.where(y == 0, -1, y)

data = {
    'x1' : X_train[:, 0],
    'x2' : X_train[:, 1],
    'y' : y
}

df = pd.DataFrame(data)

fig = px.scatter(df, x = "x1", y = 'x2', color = 'y')
fig.show()
```

Once again we will first attempt to construct the decision regions using a linear discriminant.

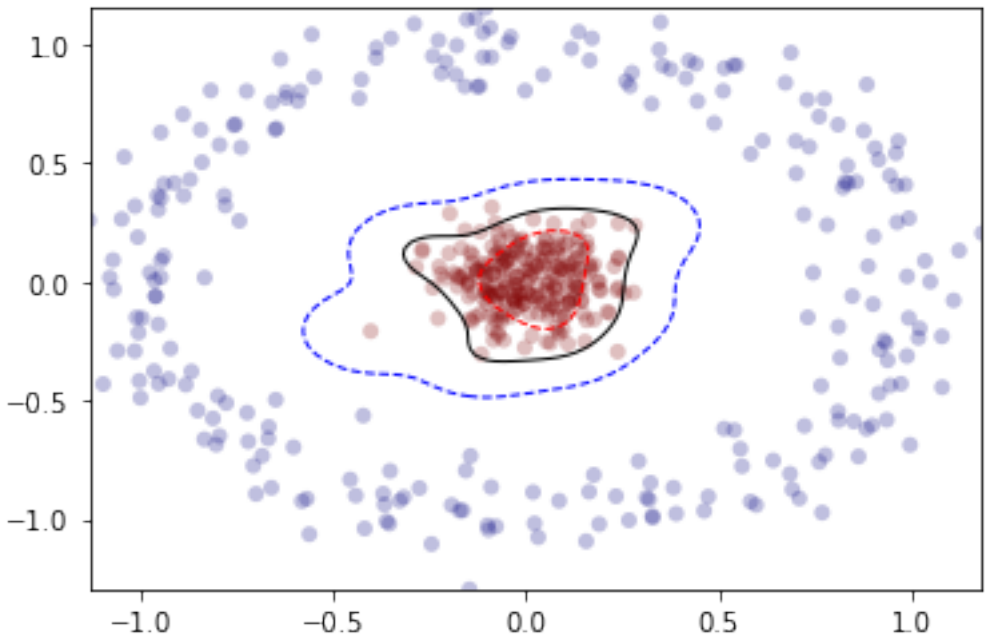
[ ]:
```
alphas, b = smo_simplified(X_train.T, y, C, tol, max_passes, lin_kernel)
alphas = np.reshape(alphas, len(alphas))
```

[ ]:
```
fig, ax = plt.subplots()
grid, ax = plot_decision_boundary(alphas, X_train, b, lin_kernel, ax, 0)
```



As expected, this is pretty horrendous. There isn't much of a point to harping on this any longer, so let's go ahead and see how the Gaussian kernel rectifies our pathetic first attempt.

[ ]:
```
alphas, b = smo_simplified(X_train.T, y, C, tol, max_passes, gauss_kernel)
alphas = np.reshape(alphas, len(alphas))

fig, ax = plt.subplots()
grid, ax = plot_decision_boundary(alphas, X_train, b, gauss_kernel, ax, 0)
```



As was expected, our new decision regions do a much better job classifying the data.

(c)

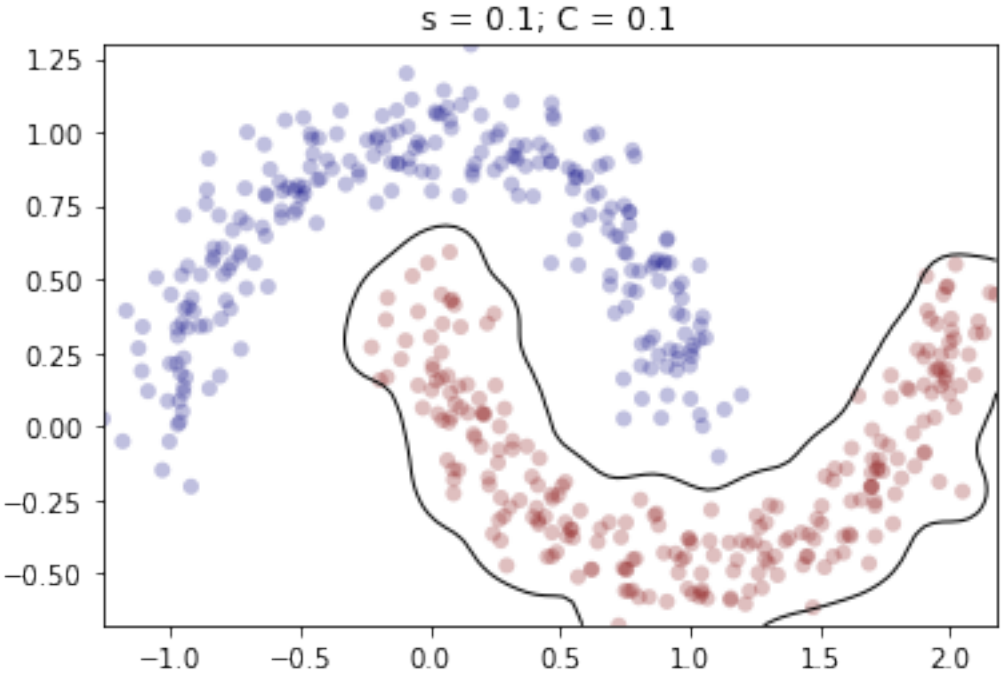
We will now examine building SVMs with varying values for the regularization term C and the standard deviation of the Gaussian kernel, s.

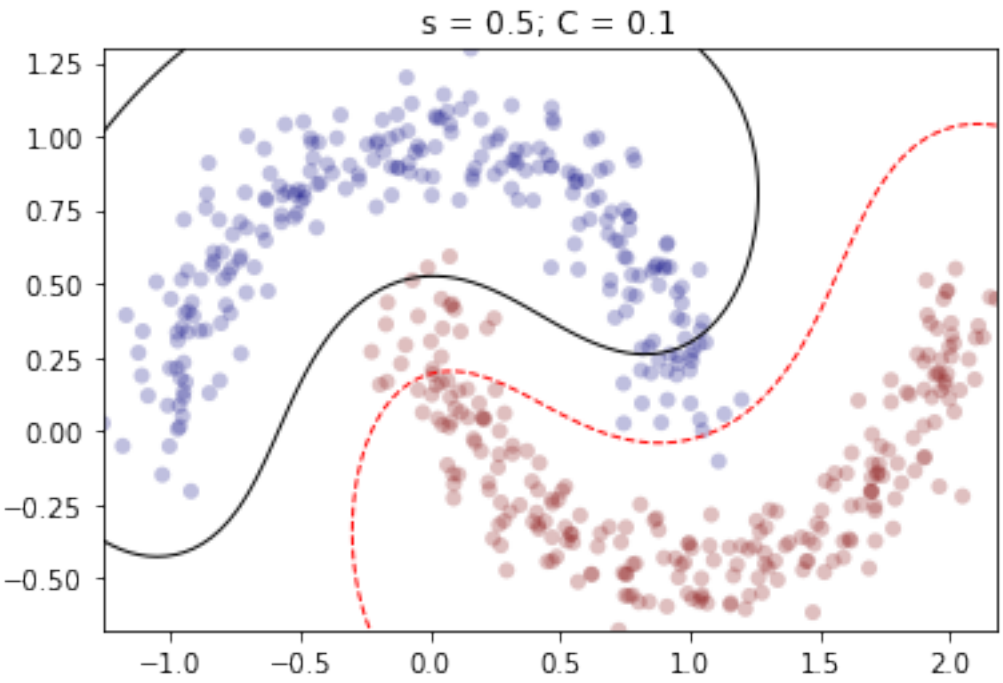
As we increase C, we should expect to see our decision bounds more closely define the decision regions. Or, to be more technical, we will *overfit* our data. This phenomena occurs as when C grows in value more lagrangian multipliers, α, will become active, and thus more vectors become *support vectors*. The more support vectors we have the more flexible our discriminant becomes.

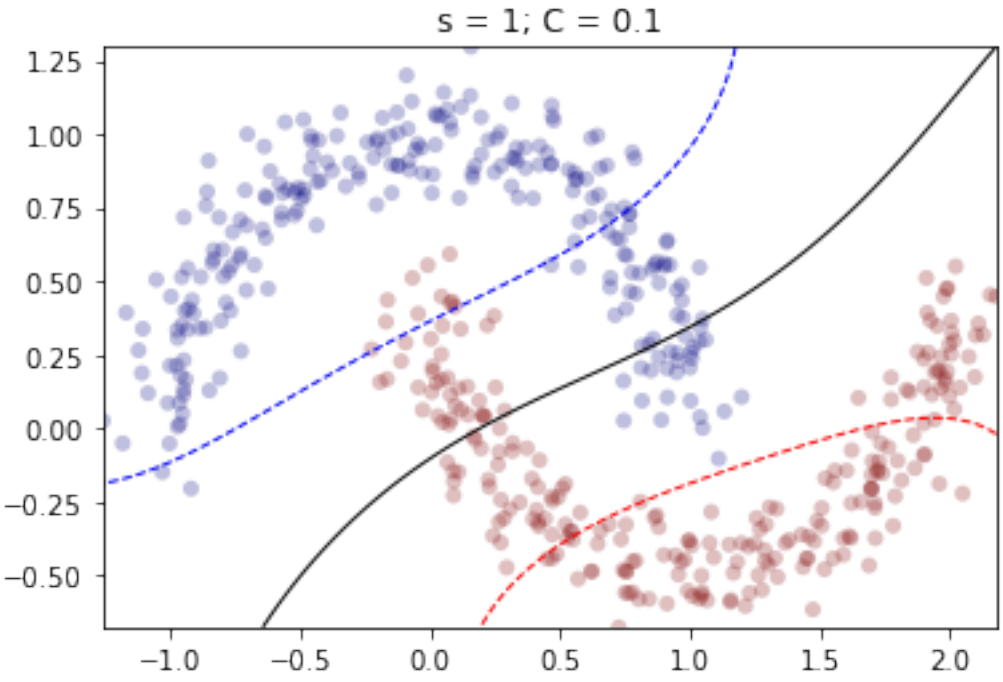
Increasing s has the opposite effect. As s, or rather, the Gaussian kernel's standard deviation, grows in value, the discriminant becomes more smooth. Essentially, when s is large, the points further away from our test point, x^*, have more influence in the prediction. Mathematically speaking, the linear combination of our target data, which serves as our prediction, becomes more evenly weighted.

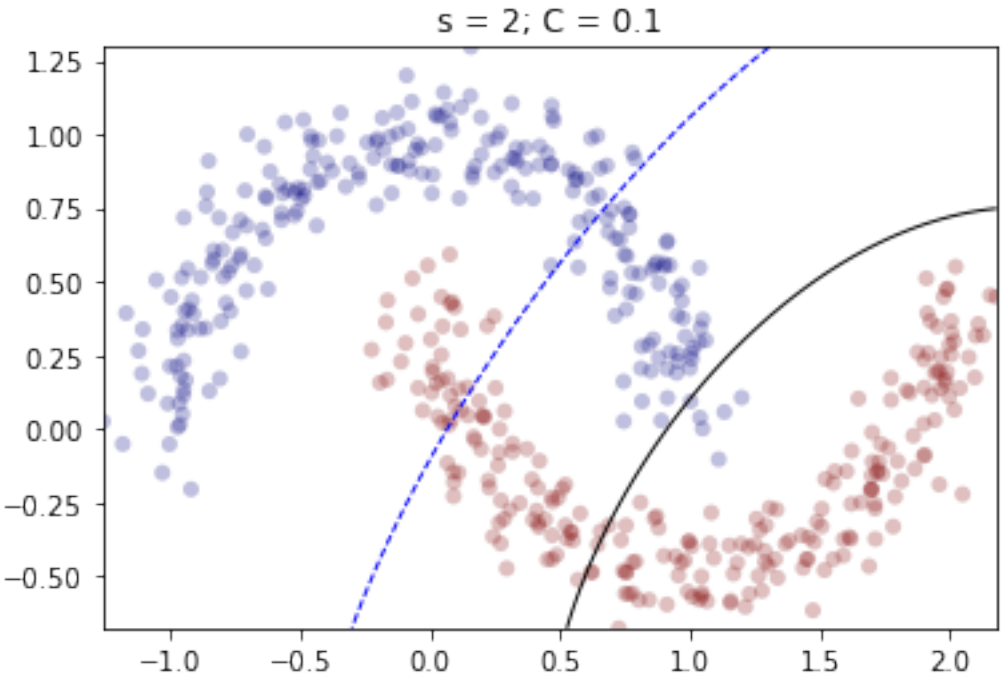
[ ]:
```
X_train, y = datasets.make_moons(n_samples=500, noise=0.1,
                         random_state=1)
y = np.where(y == 0, -1, y)

C_array = [0.1, 0.5, 1, 2]
s_array = [0.1, 0.5, 1, 2]

for c in C_array:
    for s in s_array:
```

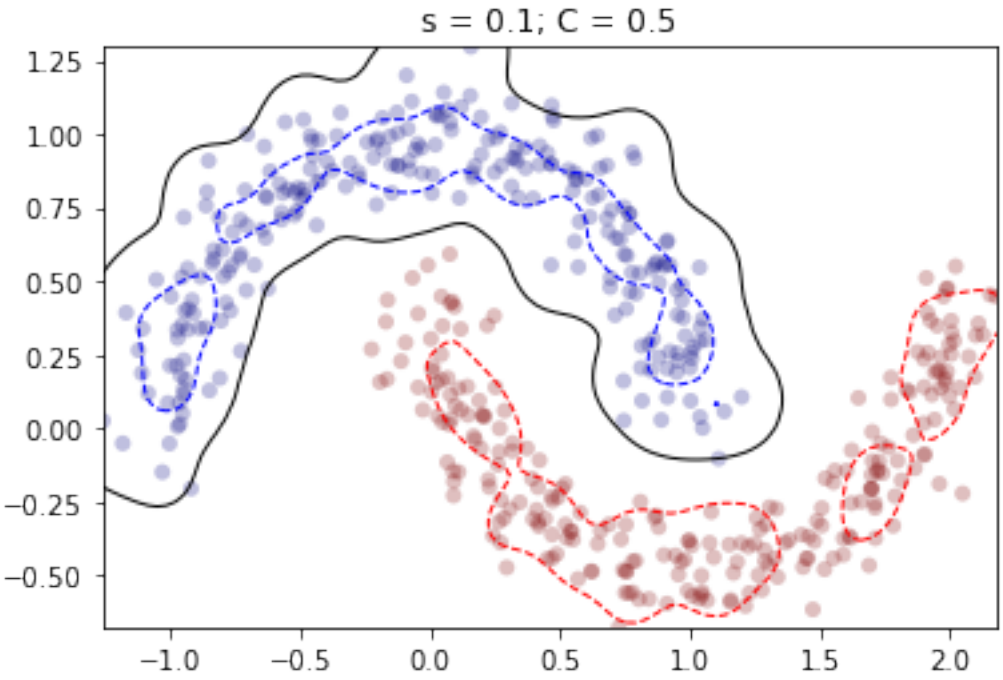
```
        kernel = Kernel('gaussian', s)

        alphas, b = smo_simplified(X_train.T, y, c, tol, max_passes, kernel)
        alphas = np.reshape(alphas, len(alphas))

        to_plot, ax = plt.subplots()
        grid, ax = plot_decision_boundary(alphas, X_train, b, kernel, ax, c)
```

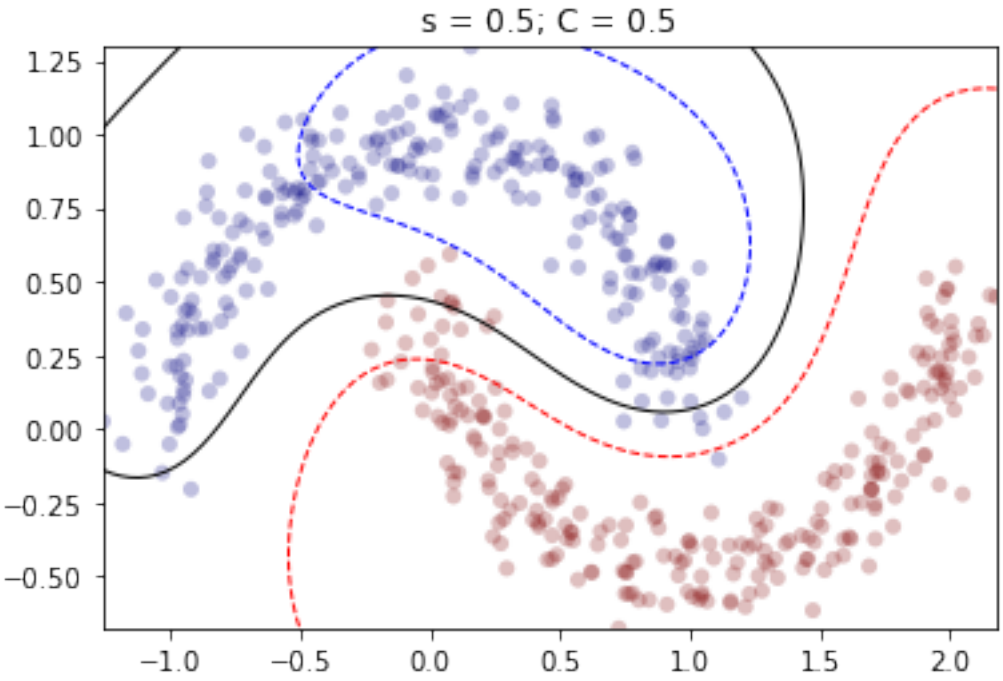


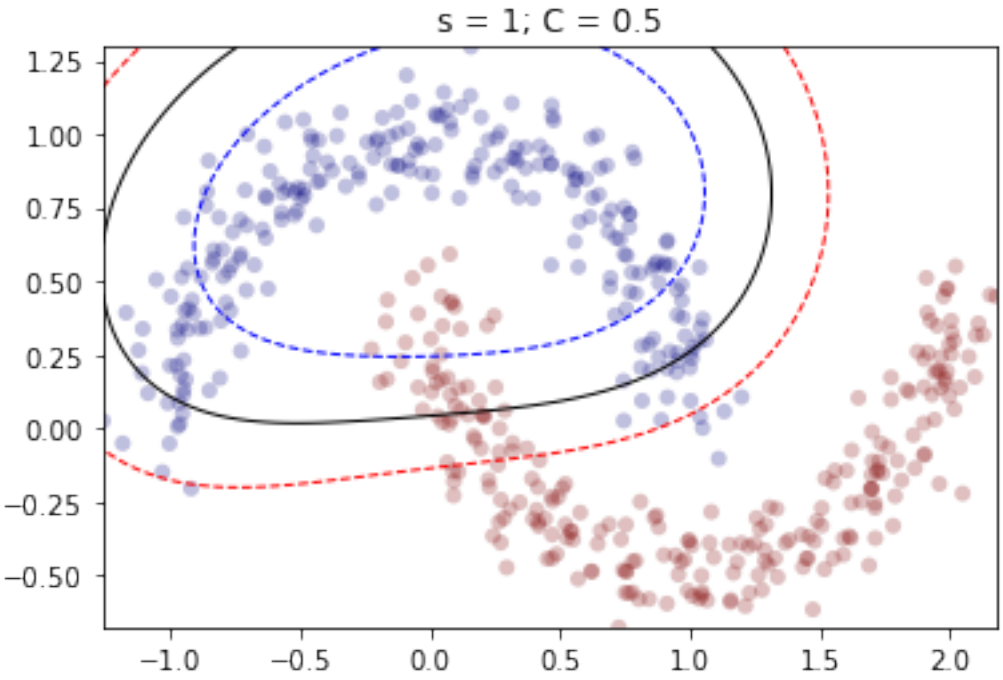

s = 0.5; C = 0.1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

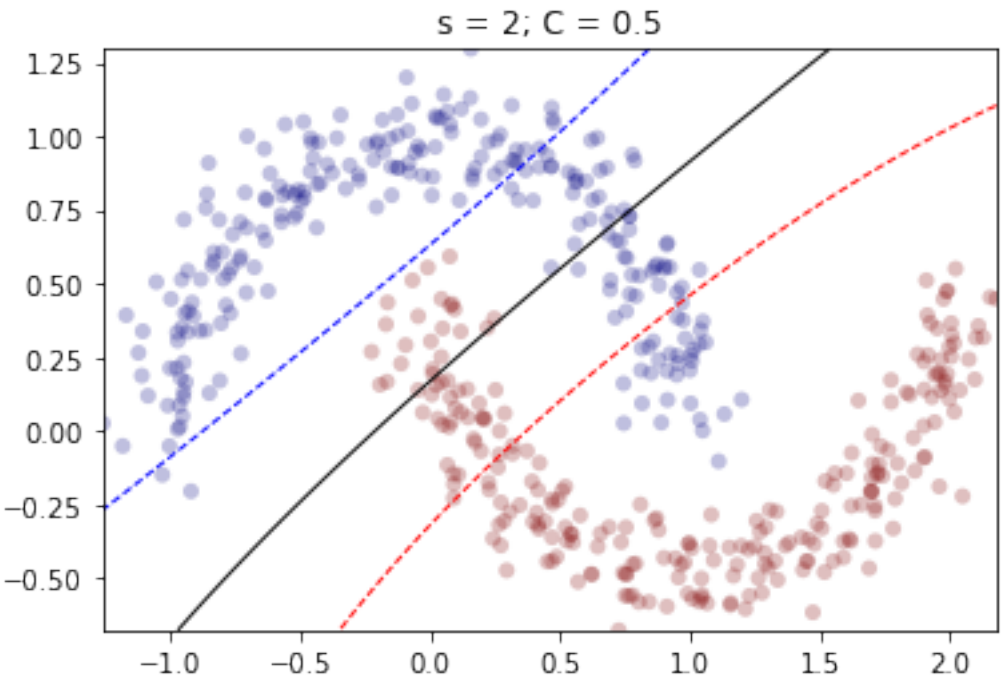

s = 1; C = 0.1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

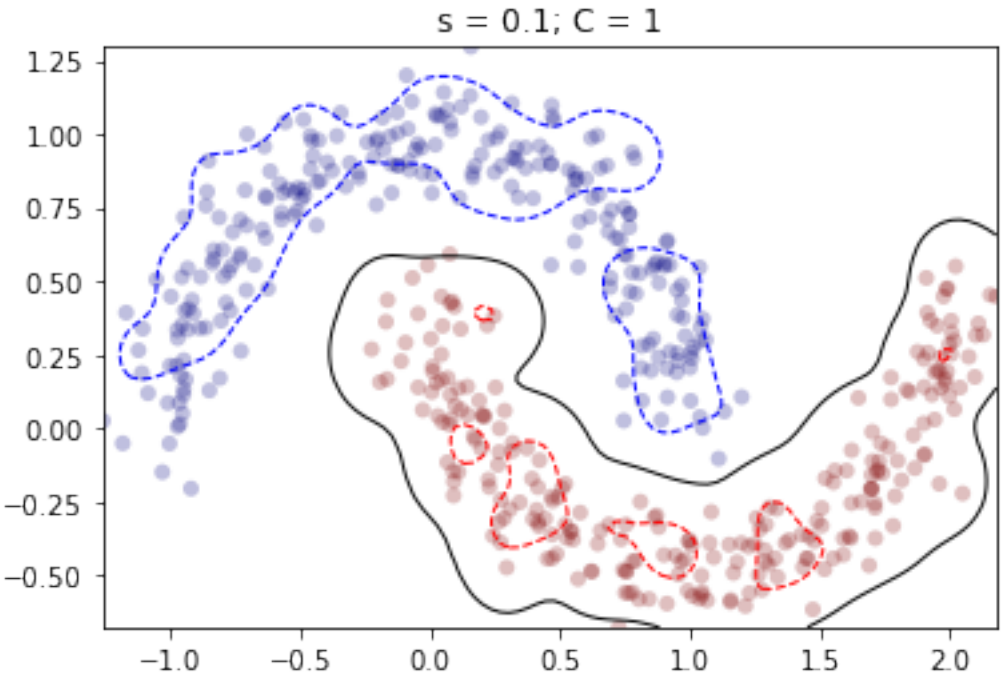

s = 2; C = 0.1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

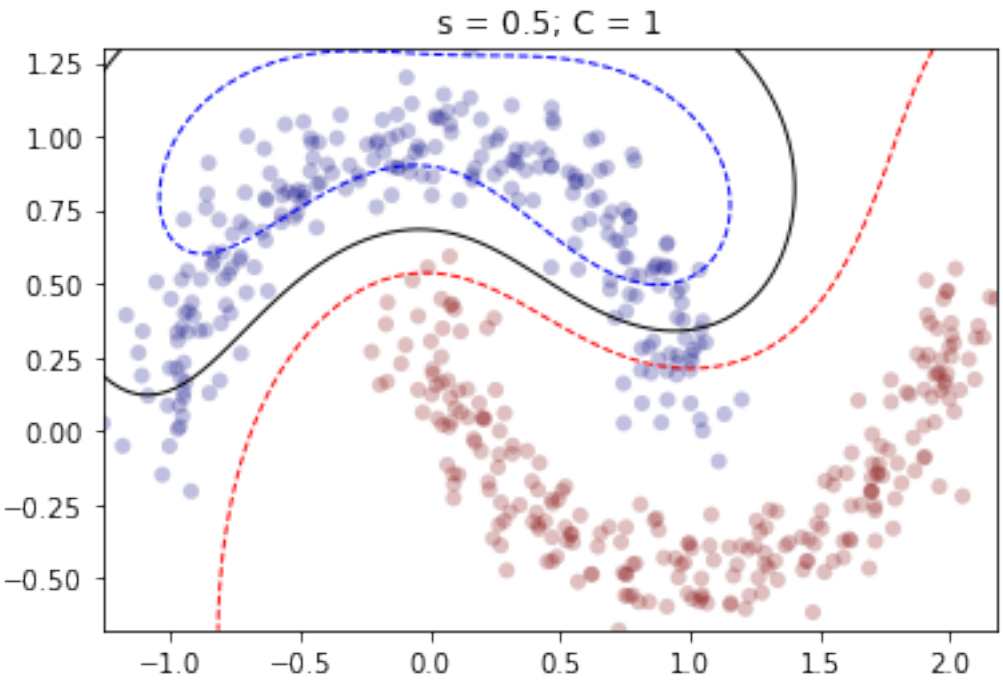

s = 0.1; C = 0.5
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

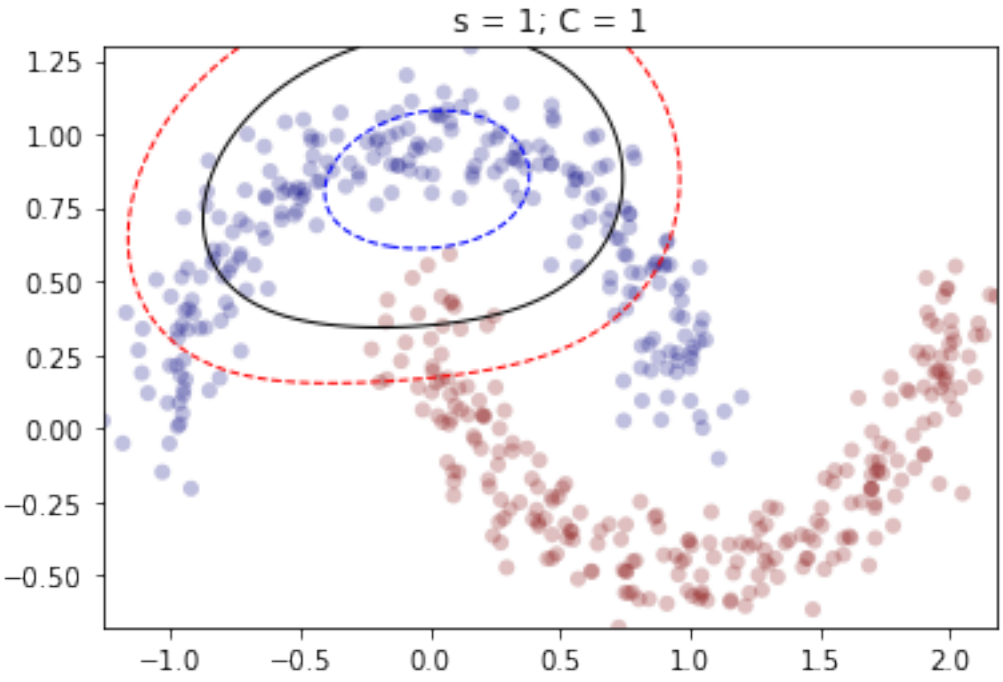

s = 0.5; C = 0.5
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

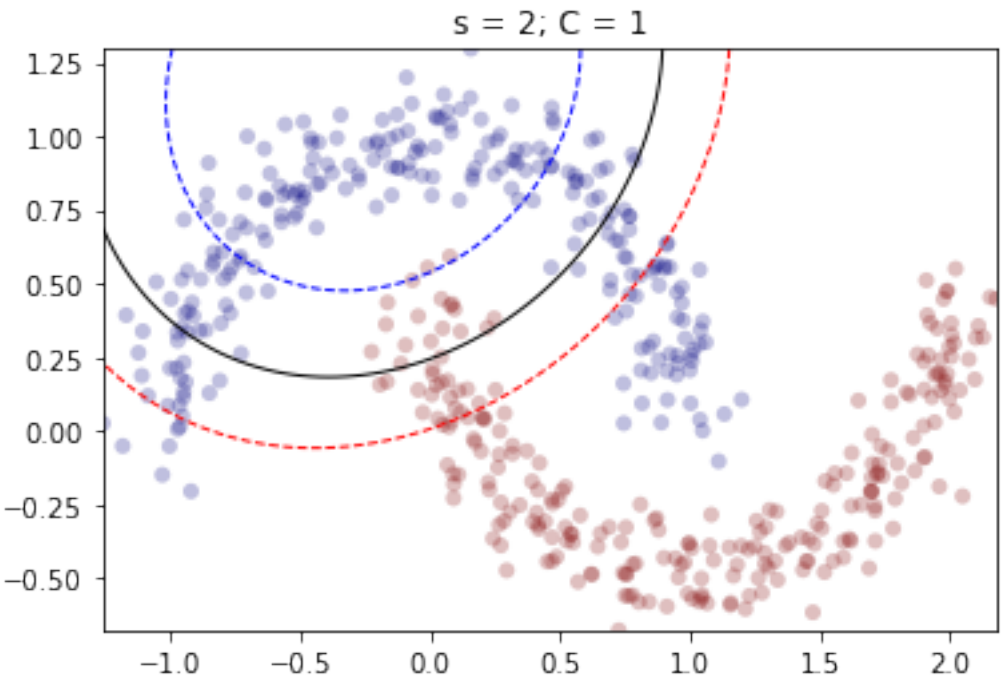

s = 1; C = 0.5
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

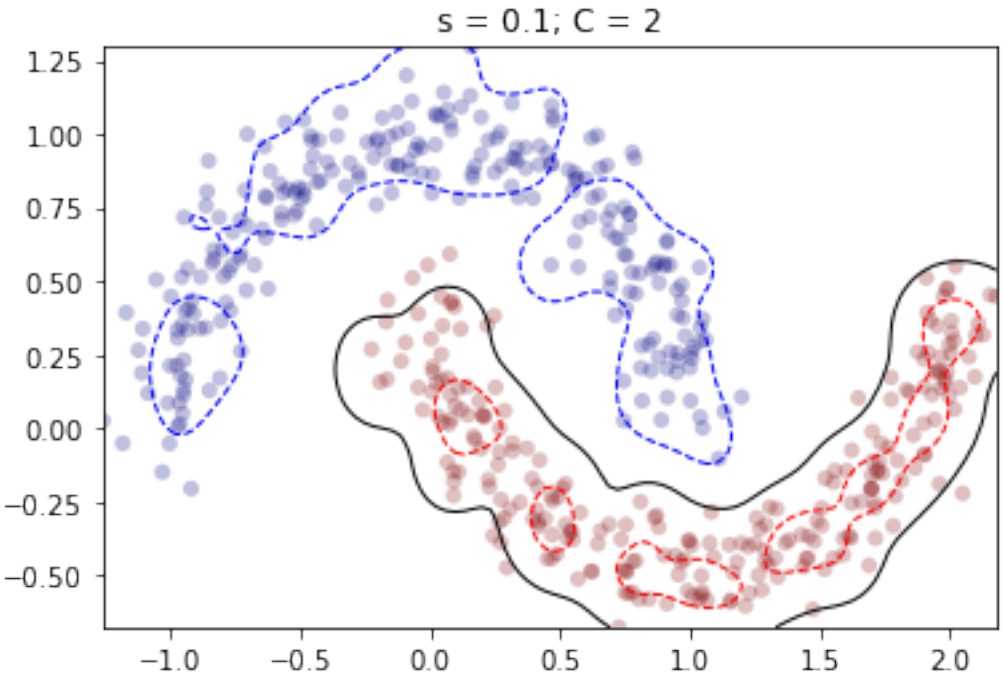

s = 2; C = 0.5
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

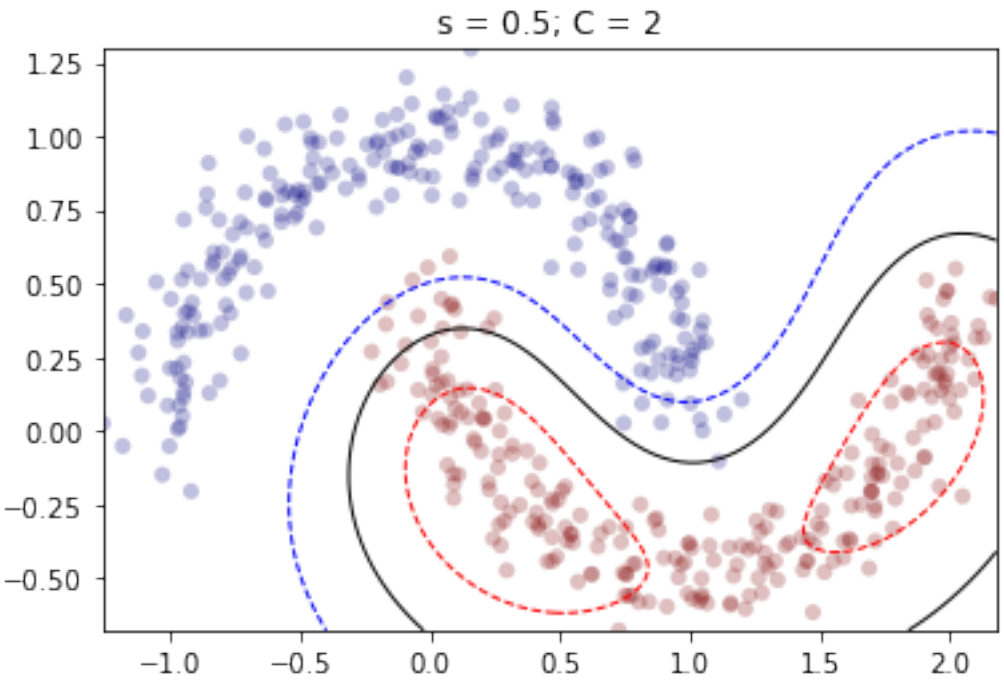

s = 0.1; C = 1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

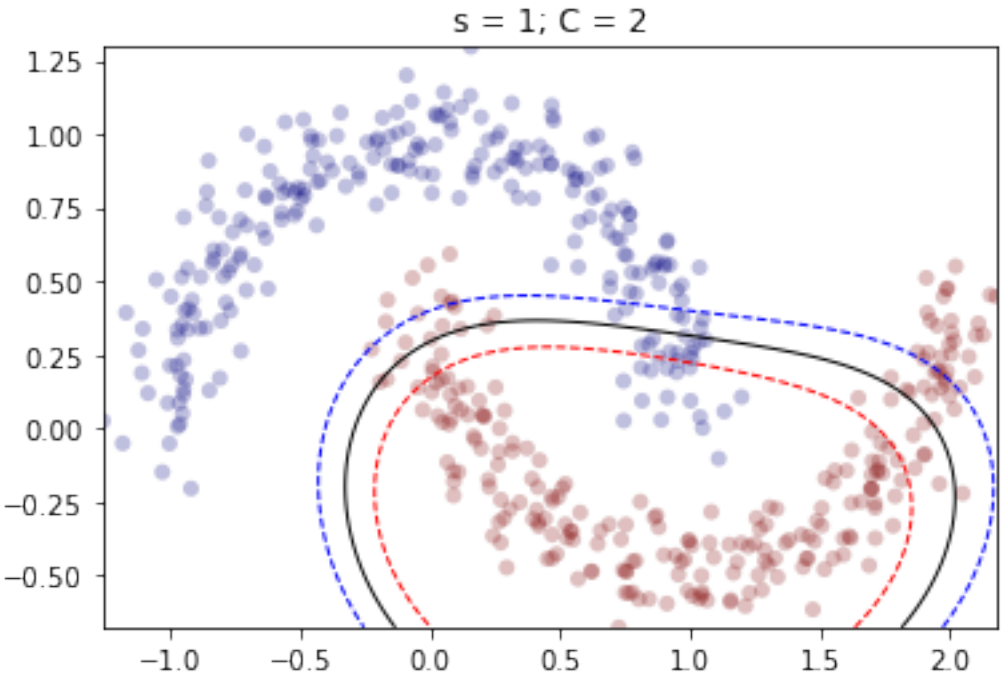

s = 0.5; C = 1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

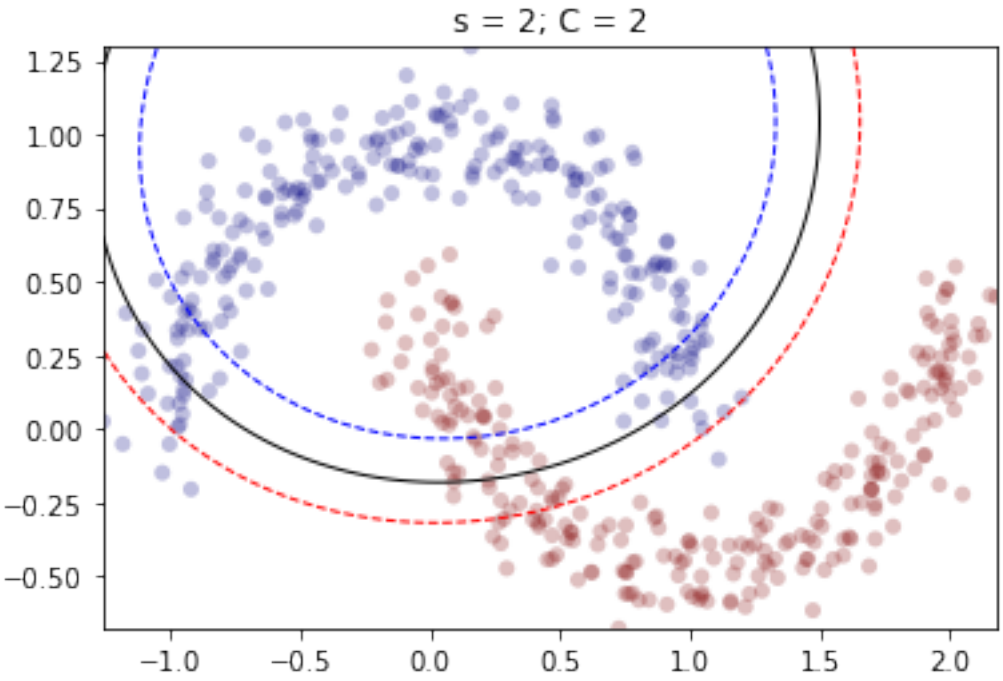

s = 1; C = 1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


s = 2; C = 1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


s = 0.1; C = 2
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


s = 0.5; C = 2
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


s = 1; C = 2
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


s = 2; C = 2
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0