Support_Vector_Data_Descriptors

November 2, 2018

In this file, we will study the topic of Support vectors data descriptors.

1 Steps:

1.1 SVDD without kernels:

A) We generateed our own labeled data set

B) We processed the data without the labels to generate the support vectors.

C) We choose the best value of the constant 'C' according to highest obtained classification accuracy: Outliers are class 1 and others are class 0 (Now we use the labels)

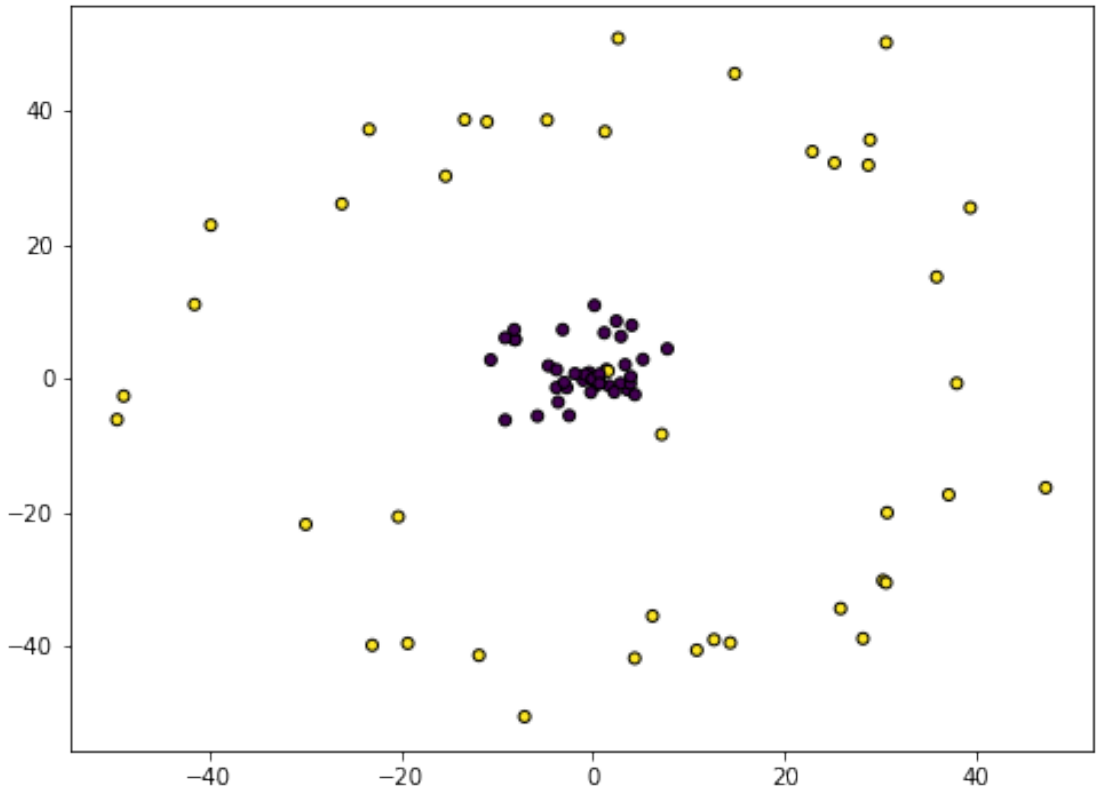
1.2 SVDD with Kernels:

A) We used the moon dataset.

B) we did the same as previous but now there is two parameters to be tuned 'C' and 'Sigma'

2 Read and visualize data

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        X = data = np.loadtxt('few_outlyers_labeled.data')
        Y = X[:,2]
        X = X[:,0:2]
        plt.figure(0, figsize=(8,6))
        plt.clf()
        plt.scatter(X[:,0], X[:,1], c=Y , s=25, edgecolor='k')
        plt.show()
        n = X.shape[0] # number of entries
        m = X.shape[1] # number of features
```



We put some outliers in the data set

3 kerenl function, we will change this later

```
In [4]: def kernel(x,y):
            #no kernel
            return np.dot(x,y)
```

The objective function that we must maximize:
$F(R,a) = R^2 + C\Sigma_i(\epsilon_i)$
The consrains: $|x_i - a|^2 \leq R^2 + \epsilon_i, \epsilon_i \geq 0 \; \forall i$
Using Lagrange multipliers:

$$L(R,a,\alpha_y,\gamma_i,\epsilon_i) = R^2 + C\Sigma_i(\epsilon_i) - \Sigma_i(\alpha_i(R^2 + \epsilon_i - (|x_i|^2 - 2a.x_i + |a|^2)) - \Sigma_i(\gamma_i\epsilon_i))$$

After computing the derivatives and resubstituting them in the previous we get:

$$L = \Sigma_i(\alpha_i(x_i.x_i)) - \Sigma_{i,j}(\alpha_i\alpha_j(x_i.x_j))$$

With the constrains:

$$\Sigma\alpha_i = 1$$

$$0 \leq \alpha_i \leq C$$

```
In [5]: # function to optimize
        def L(a,*args):
            ret = 0
            for i in range(len(a)):
                ret += a[i]*kernel(X[i],X[i])
            for i in range(len(a)):
                for j in range(len(a)):
                    ret -= a[i]*a[j]*kernel(X[i],X[j])
            return -1 * ret
            # we add -1 because we want to mazimize the function instead of minimizing

        # partial derivative of L regarding the alphas
        def dL(a,*args):
            da = np.zeros(len(a))
            for i in range(len(a)):
                da[i] = kernel(X[i],X[i])
                for j in range(len(a)):
                    da[i] -= a[j]*kernel(X[i],X[j])
            return -1 * np.array( da ,float)
            # we add -1 because we want to maximize the function instead of minimizing
```

We will try multiple values for C and choose the best value (according to classification results))

```
In [6]: import scipy.optimize as optimize
        import math
        best_accuracy = 0
        best_C = 0
        best_colors = []
        C0 = [i/10000 for i in range(1,10)]
        C1 = [i/1000 for i in range(1,10)]
        C2 = [i/100 for i in range(1,10)]
        C3 = [i/10 for i in range(1,5)]
        C_arr = C0 + C1 + C2 + C3
```

Note that for each x_i we have three possibilities: $\alpha_i = C$, x_i is an outlier
$0 < \alpha_i < C$, $x_i $ is a suport vector
$\alpha_i = 0$, $x_i $ is not a support vector or an outlier

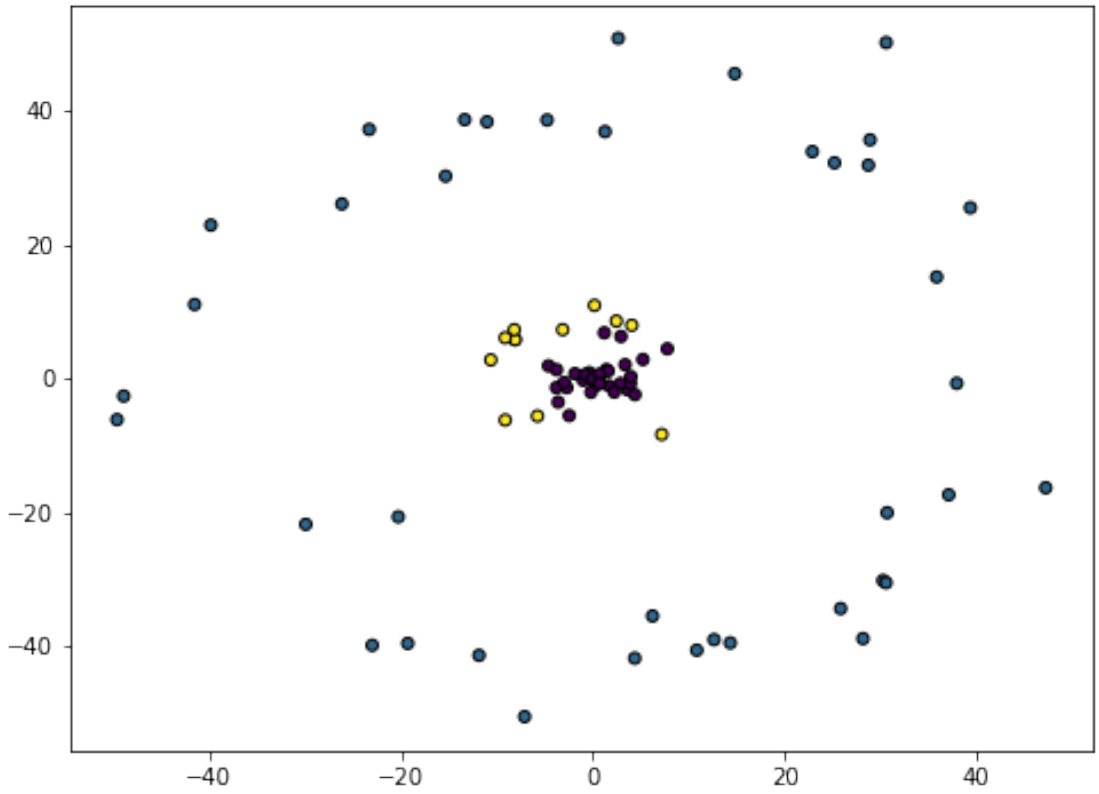
```
In [7]: import sys
        import os

        for C in C_arr:
            x0 = [0 for _ in range(n)] # initial solution
            # solve for alphas
            sys.stdout = open(os.devnull, "w")
            # previous line will prevent the function from printing in the batch console
```

```
alpha = optimize.fmin_slsqp(L,x0, fprime=dL
                            , eqcons=[lambda x: sum(x) - 1 ]
                            , bounds = [(0,C) for _ in range(n)]
                            , full_output =False )
sys.stdout = sys.__stdout__ # this line will restor default setting of printing
if math.isnan(sum(alpha)) or sum(alpha) < 0.9 or sum(alpha) > 1.1:
    continue
eps = 1e-12
# plotting
def calc_colore(i):
    #not SV alpha[i] == 0
    if alpha[i] < eps:
        return 0
    #outlier if alpha[i] >= C
    if alpha[i] > C - eps:
        return 1
    #support vector if alpha[i] < C
    return 3

colores = [calc_colore(i) for i in range(n)]
cur_accuracy = np.mean([(Y[i] == 0 and colores[i] != 1)
                        or (Y[i] == 1 and colores[i] == 1)
                        for i in range(n)])
# save the best results
if cur_accuracy > best_accuracy:
    best_accuracy = cur_accuracy
    best_C = C
    best_colors = colores
```

4 Plot the best results

4.1 Blue points are outliers, yellow are the support vectors

In [8]:
```
plt.figure(0, figsize=(8,6))
plt.clf()
plt.scatter(X[:,0], X[:,1], c=best_colors , s=25, edgecolor='k')
plt.show()
```



5 we but the same previous code in a function to be able to run it for multiple values of the parameters

```
In [9]: import sys
        import os
        import numpy as np
        import matplotlib.pyplot as plt
        import math
        from numpy import ones, exp, loadtxt, tanh
        from numpy.linalg import eig, norm
        SIGMA = 0.01
```

6 The function called 'run' will run the code for a given value of Sigma and will choose the bes value of C

Please not that we computed the pairwise kernel values for all the dataset before running the function and we saved it in an array called kernel.

```
In [10]: def run(X,Y,kernel):
             n = X.shape[0] # number of entries
```

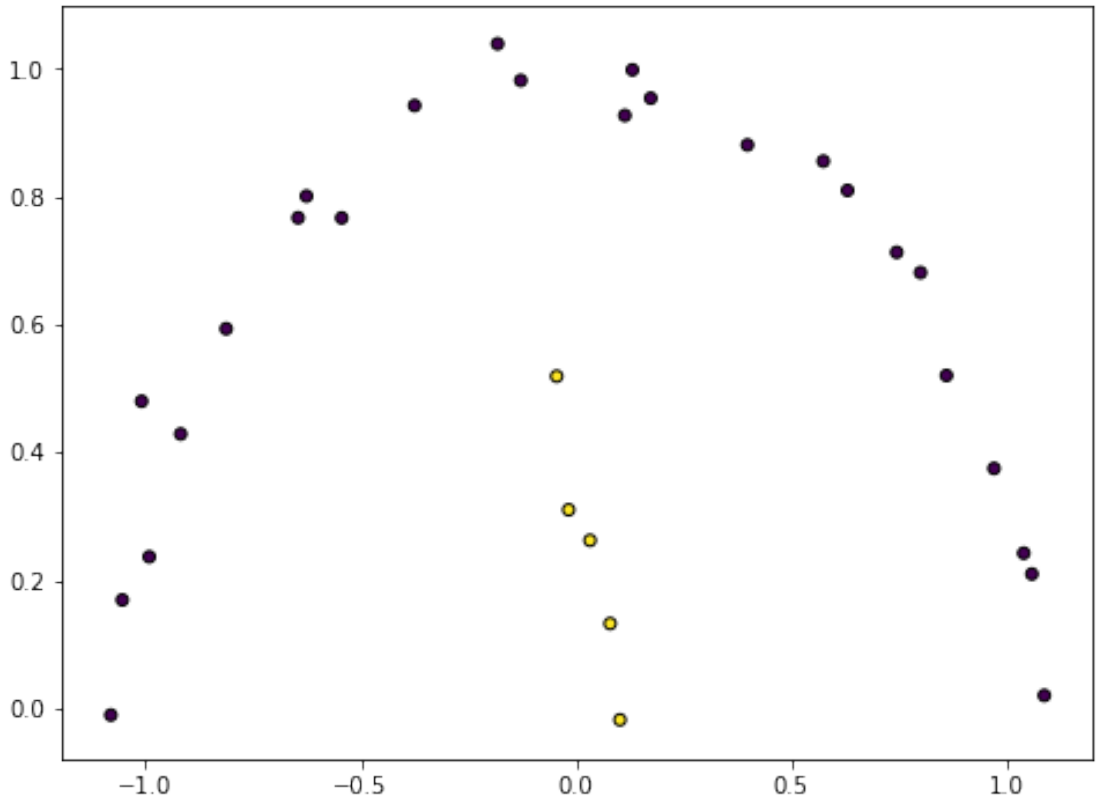
```
m = X.shape[1] # number of features
# function to optimize
def L(a,*args):
    ret = 0
    for i in range(len(a)):
        ret += a[i]*kernel_arr[i,i]
    for i in range(len(a)):
        for j in range(len(a)):
            ret -= a[i]*a[j]*kernel_arr[i,j]
    return -1 * ret
# we add -1 because we want to mazimize the function instead of minimizing

# partial derivative of L regarding the alphas
def dL(a,*args):
    da = np.zeros(len(a))
    for i in range(len(a)):
        da[i] = kernel_arr[i,i]
        for j in range(len(a)):
            da[i] -= a[j]*kernel_arr[i,j]
    return -1 * np.array( da ,float)
# we add -1 because we want to mazimize the function instead of minimizing

import scipy.optimize as optimize

# values of the constant C
best_accuracy = 0
best_C = 0
best_colors = []

C0 = [i/10000 for i in range(1,10)]
C1 = [i/1000 for i in range(1,10)]
C2 = [i/100 for i in range(1,10)]
C3 = [i/10 for i in range(1,5)]

C_arr = C0 + C1 + C2 + C3
global cc
for C in C_arr:
    #print('.',end='')
    cc += 1
    sys.stdout.flush()
    x0 = [0 for _ in range(n)] # initial solution
    sys.stdout = open(os.devnull, "w")
    # previous line will prevent the function from printing in the batch console
    alpha = optimize.fmin_slsqp(L,x0, fprime=dL
                                , eqcons=[lambda x: sum(x) - 1 ]
                                , bounds = [(0,C) for _ in range(n)]
                                , full_output =False )
    sys.stdout = sys.__stdout__ # this line will restor default setting of printi
```

```
            if math.isnan(sum(alpha)) or sum(alpha) < 0.9 or sum(alpha) > 1.1:
                continue
            #print(sum(alpha))

            eps = 1e-12
            # plotting
            def calc_colore(i):
                #not SV alpha[i] == 0
                if alpha[i] < eps:
                    return 0
                #outlyer if alpha[i] >= C
                if alpha[i] > C - eps:
                    return 1
                #support vector if alpha[i] < C
                return 3

            colores = [calc_colore(i) for i in range(n)]
            cur_accuracy = np.mean([(Y[i] == 0 and colores[i] != 1)
                                    or (Y[i] == 1 and colores[i] == 1)
                                    for i in range(n)])
            # save the best results
            if cur_accuracy > best_accuracy:
                best_accuracy = cur_accuracy
                best_C = C
                best_colors = colores

        return best_accuracy,best_colors,best_C
```

7 we generate the data and will run the code for multiple values for Sigma

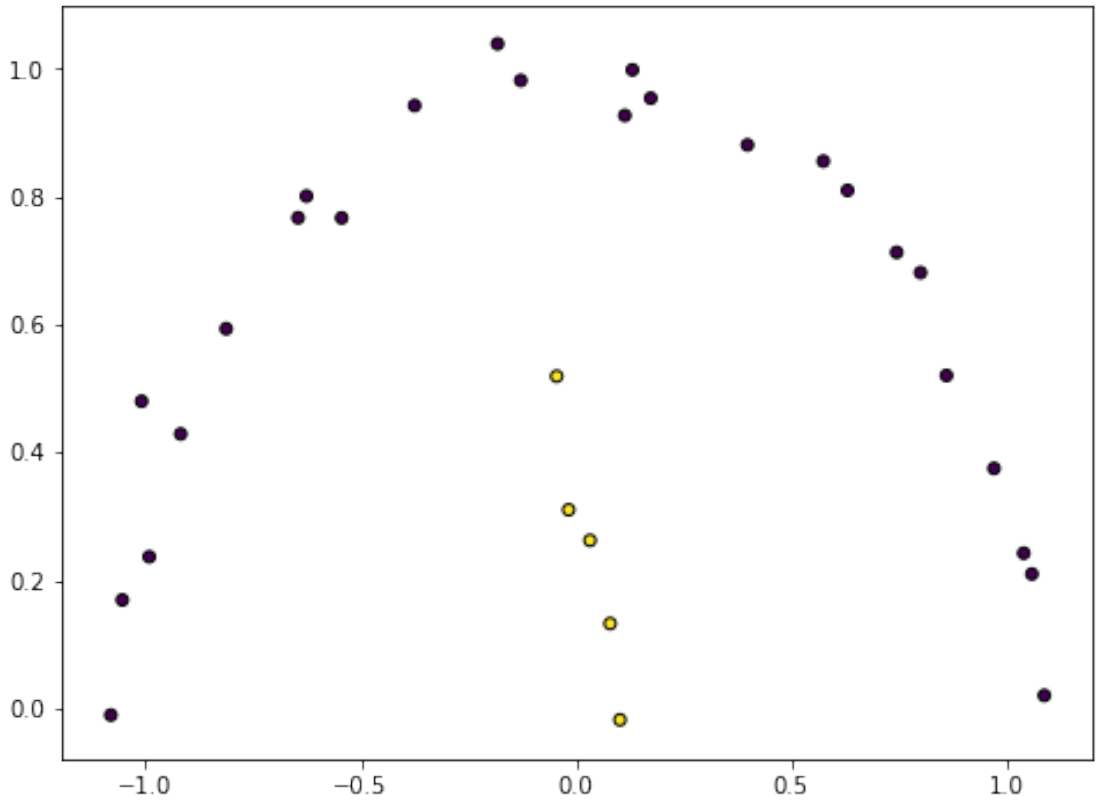
In [11]:
```
from sklearn.datasets import make_moons

X,Y = make_moons(n_samples=50, shuffle=False, noise=.05, random_state=0)
X = X[0:30,:]
Y = Y[:30]
print(Y)

plt.figure(0, figsize=(8,6))
plt.clf()
plt.scatter(X[:,0], X[:,1], c=Y , s=25, edgecolor='k')
plt.show()

total_accuracy = -5
total_colors = []
total_C = 0
total_sigma = 0
```

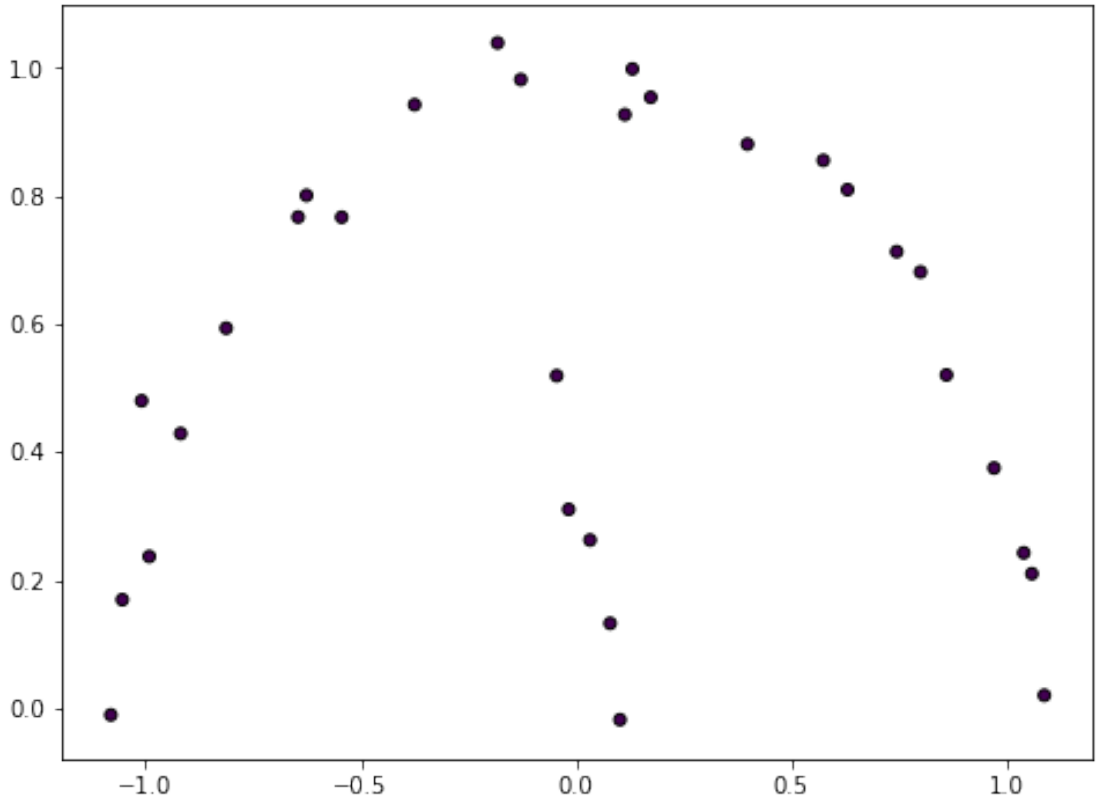
```
from scipy.spatial.distance import pdist, squareform

S0 = [i/10000 for i in range(1,10)]
S1 = [i/1000 for i in range(1,10)]
S2 = [i/100 for i in range(1,10)]
S3 = [i/10 for i in range(1,10)]
S4 = [i for i in range(1,10)]
SIGMA_arr = S0 + S1 + S2 + S3 + S4
```



In [23]:
```
ii = 0
for sg in SIGMA_arr:
    if ii % 10 == 0:
        print(int(ii/10),end = ' ')
        sys.stdout.flush()
    ii += 1
    pairwise_sq_dists = squareform(pdist(X, 'sqeuclidean'))
    kernel_arr = np.exp(-pairwise_sq_dists / sg**2)
    #print()
    #SIGMA = sg
    accuracy,colors,C = run(X,Y,linear)
    #print(accuracy,colors,C)
```

```
        #print(kernel_arr)
        if accuracy > total_accuracy:
            total_accuracy = accuracy
            total_colors = colors
            total_C = C
            total_sigma = sg

print("best acc: ",total_accuracy)
print("colores:",total_colors)
print("total_C",total_C)
print("total_sigma",total_sigma)
print("cc",cc)
plt.figure(0, figsize=(8,6))
plt.clf()
plt.scatter(X[:,0], X[:,1], c=total_colors , s=25, edgecolor='k')
plt.show()
```





8 The result is very bad. We do not know why we did not get a good separation using the rbf kernel with a very big range of the two parameters