Machine Learning and Data Mining
University of Jean Monnet, Saint Etienne

Advanced Machine Learning
Practical Assignment 1

Muaz Twaty
Abdul Rahman Zakarya

K-PCA

November 3, 2018

In this file we will study the Kernel-PCA

1 Steps:

1.1 We pick a kernel

1.2 We construct the normalized kernel matrix of the data (dimension mŒm):

1.2.1

$$K_{zeromean} = K - 2 * 1_{\frac{1}{n}} K + 1_{\frac{1}{n}} K 1_{\frac{1}{n}}$$

1.3 We solve an eigenvalue problem:

1.3.1

$$K\alpha_i = \lambda_i \alpha_i$$

1.4 For any data point (new or old), we can represent it as:

1.4.1

$$y_j = \Sigma_i(\alpha_{ij} K(x, x_i)), j = 1, .., d$$

```
In [2]: import matplotlib.pyplot as plt
        from sklearn.datasets import make_moons
        from numpy import ones, exp, loadtxt, tanh
        from numpy.linalg import eig, norm
        from sklearn.preprocessing import normalize, scale
        import numpy as np

        VERBOSE = False
        def __DEBUG(msg):
                if VERBOSE: print(msg)

        fig = 1
        K2_SIGMA = 0.007

        def liner_kernel(X,Y):
                return np.dot(X,Y)
        def polynomial_kernel(X, Y):
                return (X.T.dot(Y) + 1) ** 2

        def gaussian_kernel(X, Y):
                return exp( (-1 * (norm(X - Y) ** 2)) / (2 * (K2_SIGMA ** 2)) )
```

The function that compute the zero mean Garm matrix:

```
In [3]: def k_matrix(A, kernel):

                n = A.shape[0]
                d = A.shape[1]
                K = ones((n, n))
                for i in range(n):
                        for j in range(n):
                                K[i, j] = kernel(A[i], A[j])

                K_SUM = K.sum() / (n ** 2)
                K_SUMROWS = K.sum(axis=1) / n

                K_ = ones((n, n))
                for i in range(n):
                        for j in range(n):
                                K_[i, j] = K[i, j] - K_SUMROWS[i] - K_SUMROWS[j] + K_SUM

                return K_
```

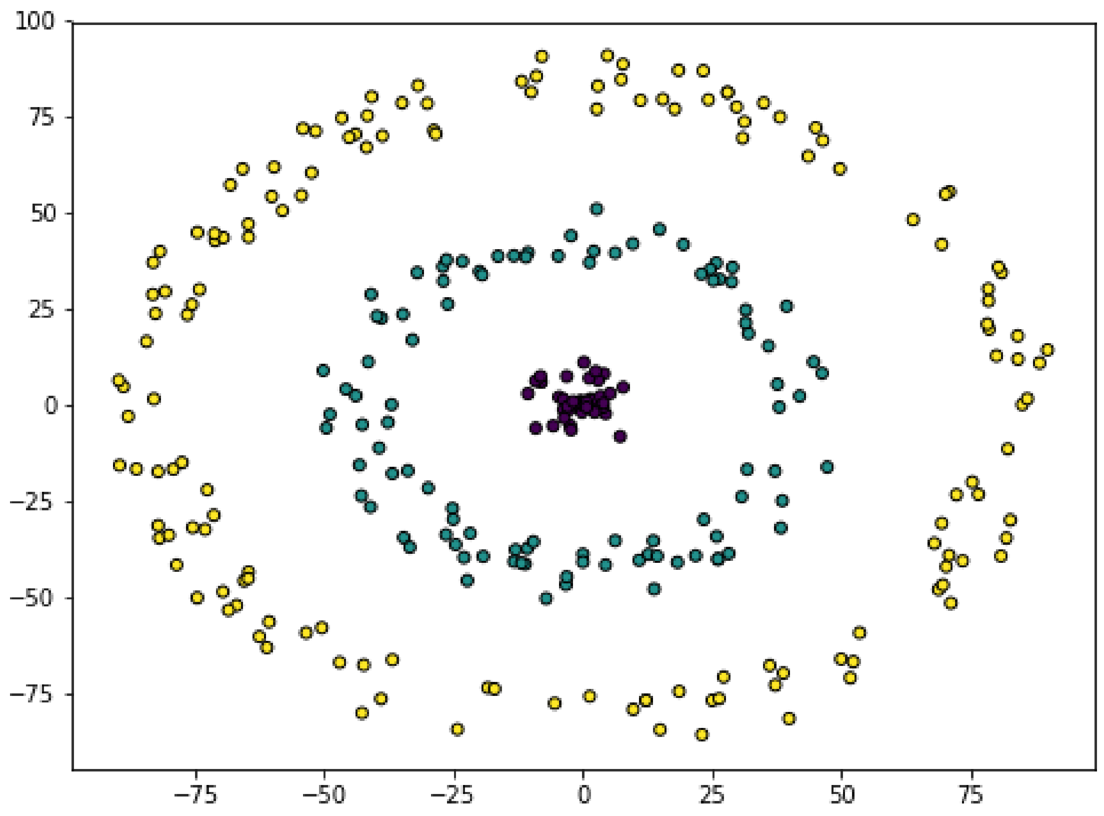
In this version of the K-PCA, we will generate only d component where d is the original dimensionality of the dataset

```
In [6]: def kpca(A, kernel):
                n = A.shape[0]
                d = A.shape[1]
                # calculate the kernelized matrix of data
                K = k_matrix(A, kernel)

                # eigendecoposition of kernelized covariance matrix
                eig_values, eig_vectors = eig(K)
                idx = eig_values.argsort()[::-1]
                eig_values = eig_values[idx]
                eig_vectors = eig_vectors[:,idx]

                # project data (only the first d component) d: the number of features in the o
                sub_eig_vectors = eig_vectors[0:d,:]
                #__DEBUG("sub_eig_vectors = \n" +  str(sub_eig_vectors))

                A_new = ones((n,d))
                for i in range(n):
                        for j in range(d):
```

```
                    temp = 0
                    for z in range(n):
                            temp += eig_vectors[j,z]*kernel(A[i],A[z])
                    A_new[i,j] = temp

        #__DEBUG("A_new = \n" + str(A_new))
        return A_new
```

Read and visulize the data:

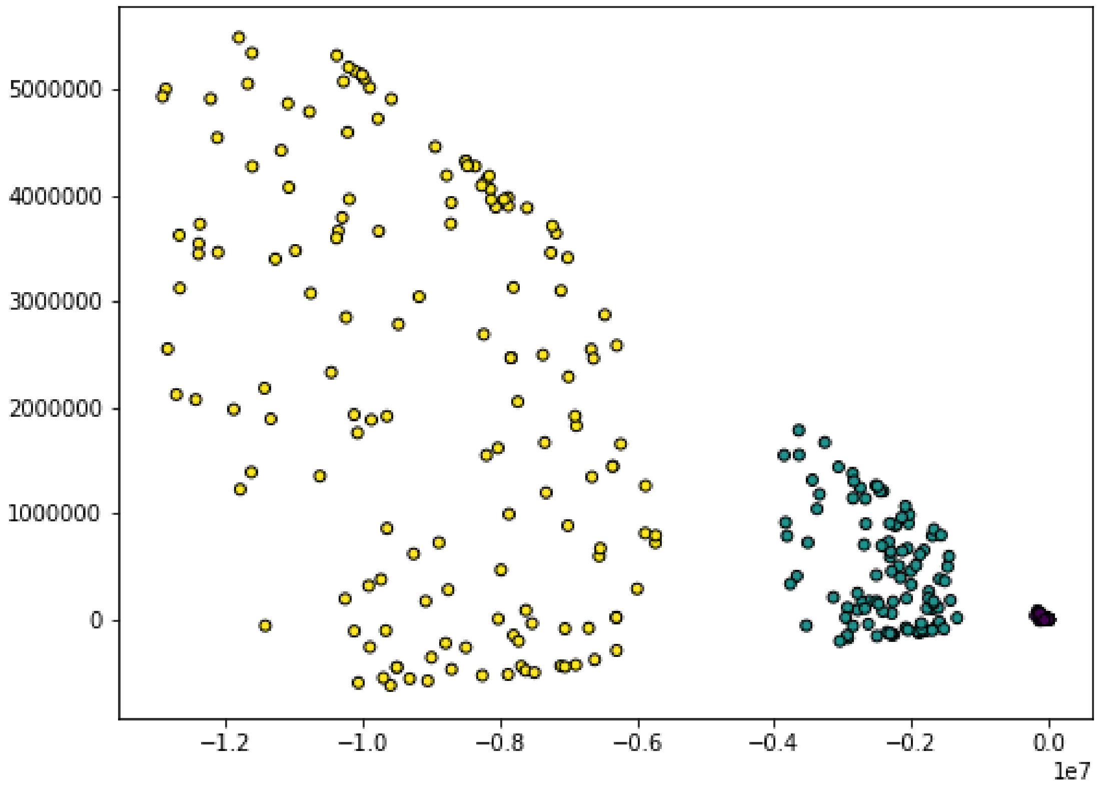
```
In [7]: from numpy import array
        #VERBOSE = True

        data = loadtxt("data.data")
        #__DEBUG("data : \n" + str(data))
        d = data.shape[1]
        A = data[:,0:(d-1)]
        Y = data[:,(d-1):d].T[0]

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



Using the polunomial kernel:

```
In [8]: A_new = kpca(A, polynomial_kernel)
        plt.figure(fig, figsize=(8, 6))
        plt.clf()
        plt.scatter(A_new[:, 0], A_new[:, 1], c=Y,s=25, edgecolor='k')
        plt.show()
        fig += 1
```

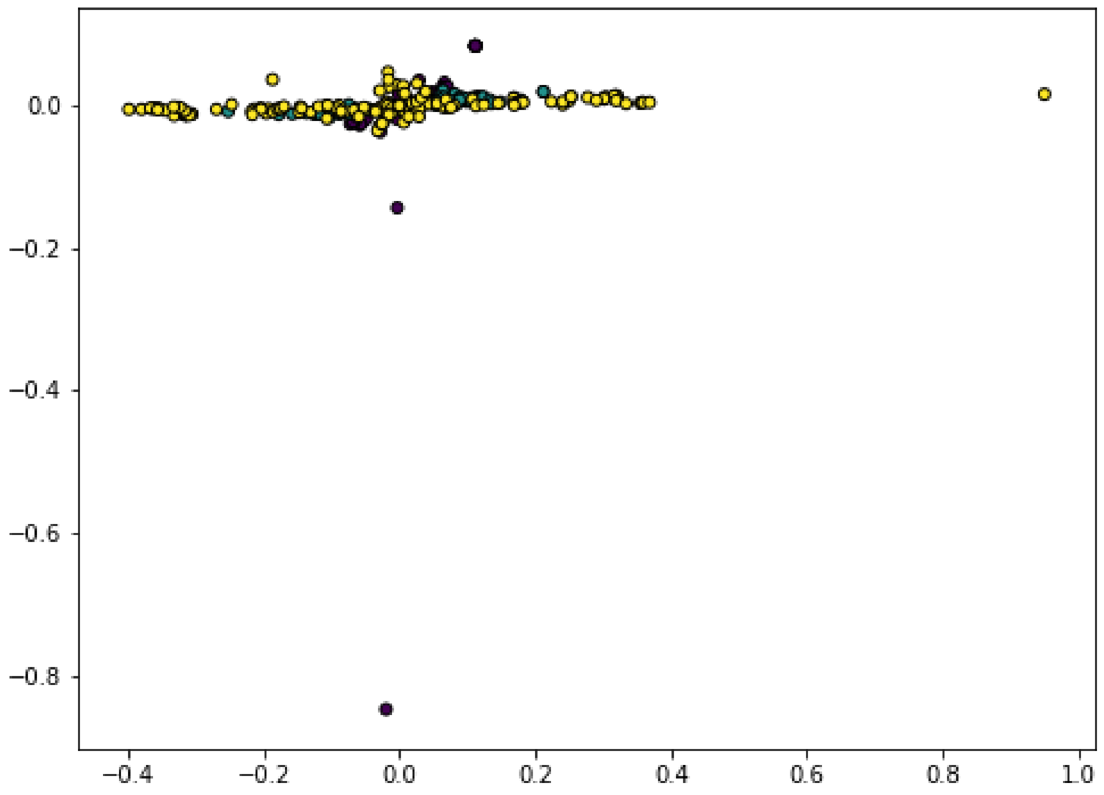
```
C:\Users\Muaz\Anaconda3\lib\site-packages\ipykernel_launcher.py:23: ComplexWarning: Casting co
```

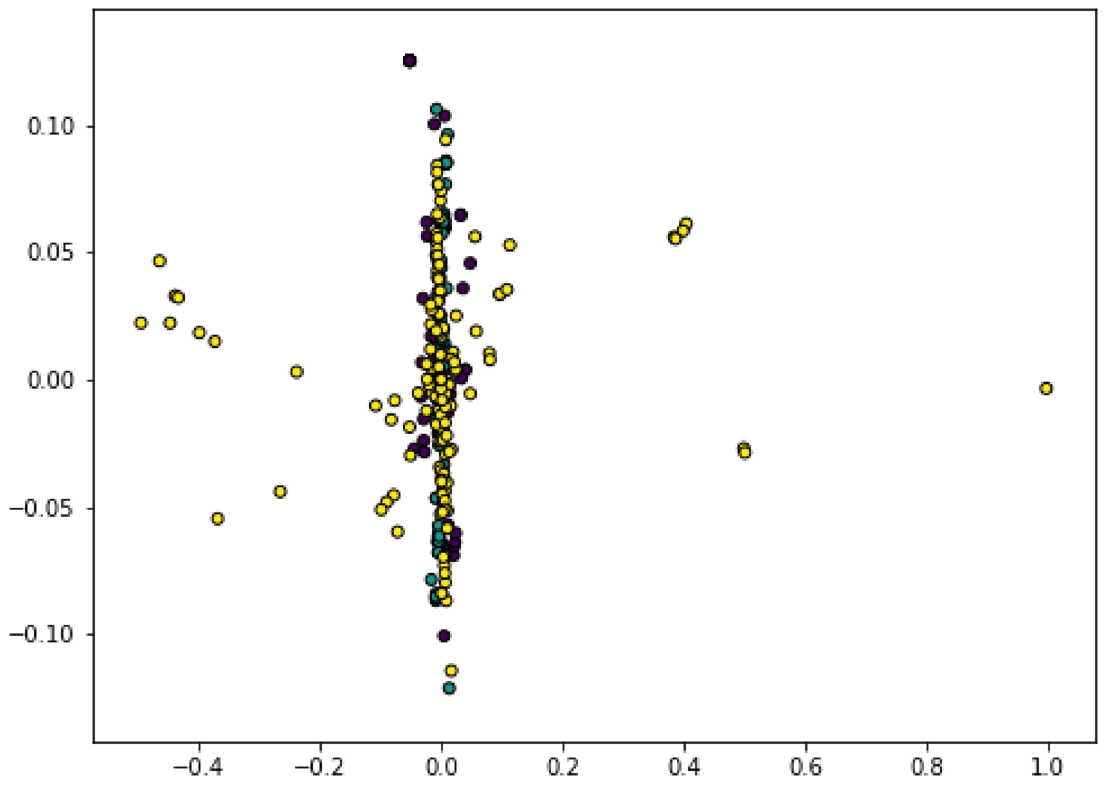


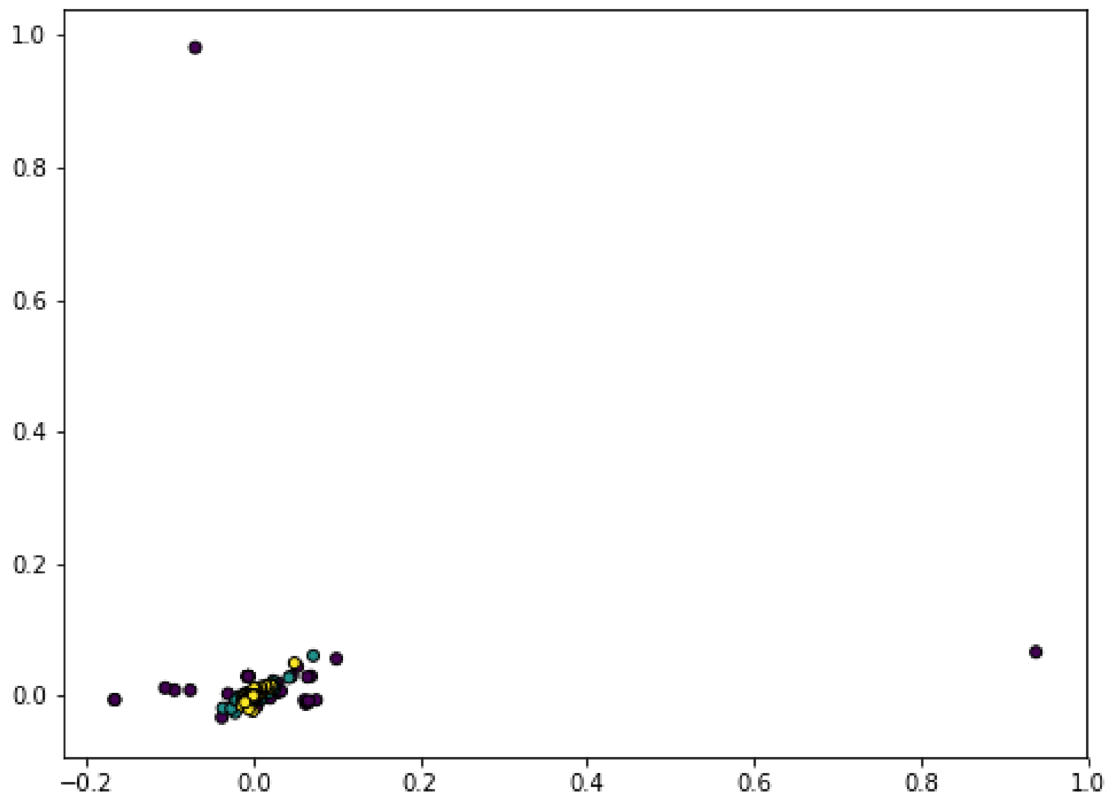
Using rbf kernel:

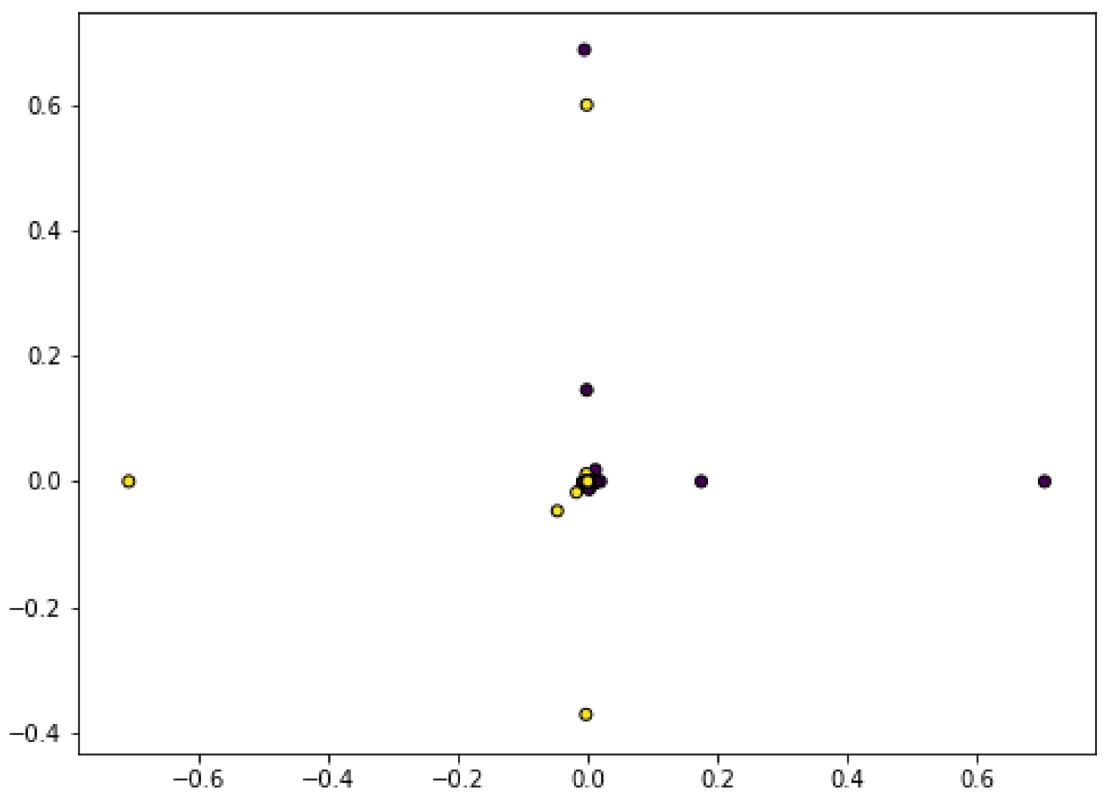
```
In [9]: sigma_array = [0.09,0.01,0.1,0.3,0.5,1,2]

        for sg in sigma_array:
                K2_SIGMA = sg
                A_new = kpca(A, gaussian_kernel)
                plt.figure(fig, figsize=(8, 6))
                plt.clf()
                plt.scatter(A_new[:, 0], A_new[:, 1], c=Y,s=25, edgecolor='k')
                plt.show()
                fig += 1
```

```
C:\Users\Muaz\Anaconda3\lib\site-packages\ipykernel_launcher.py:23: ComplexWarning: Casting co
```

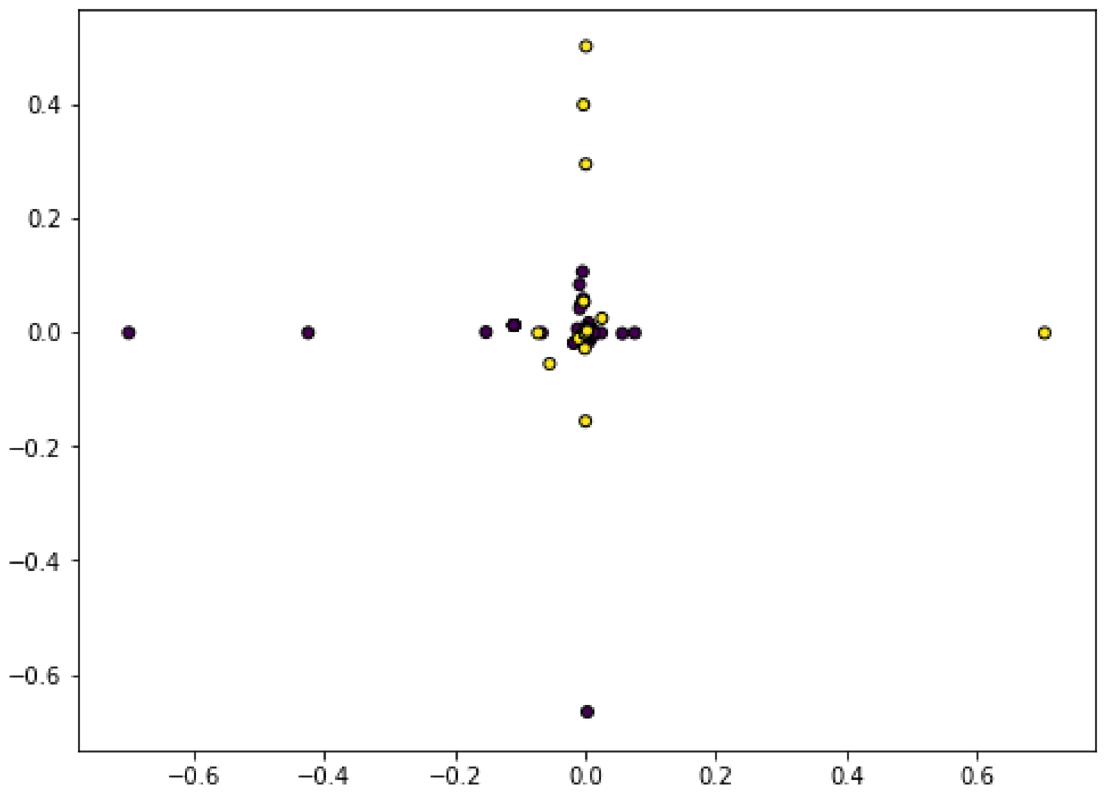


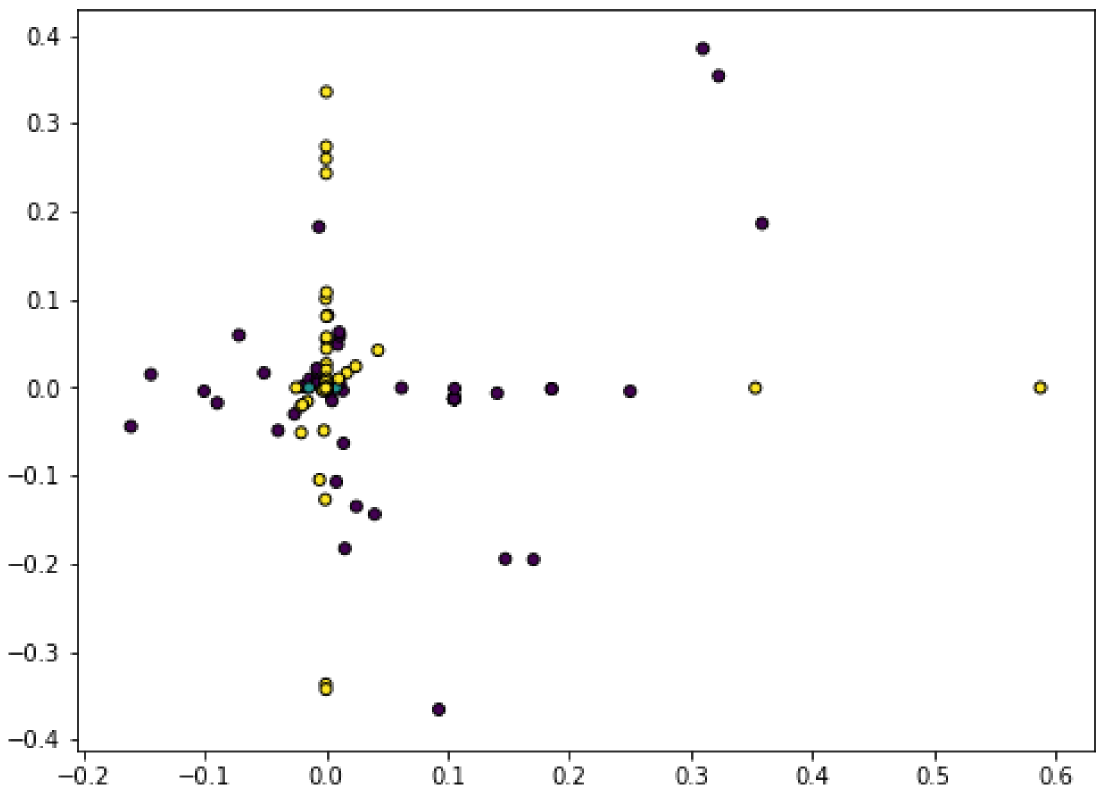

0.10
0.05
0.00
−0.05
−0.10
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

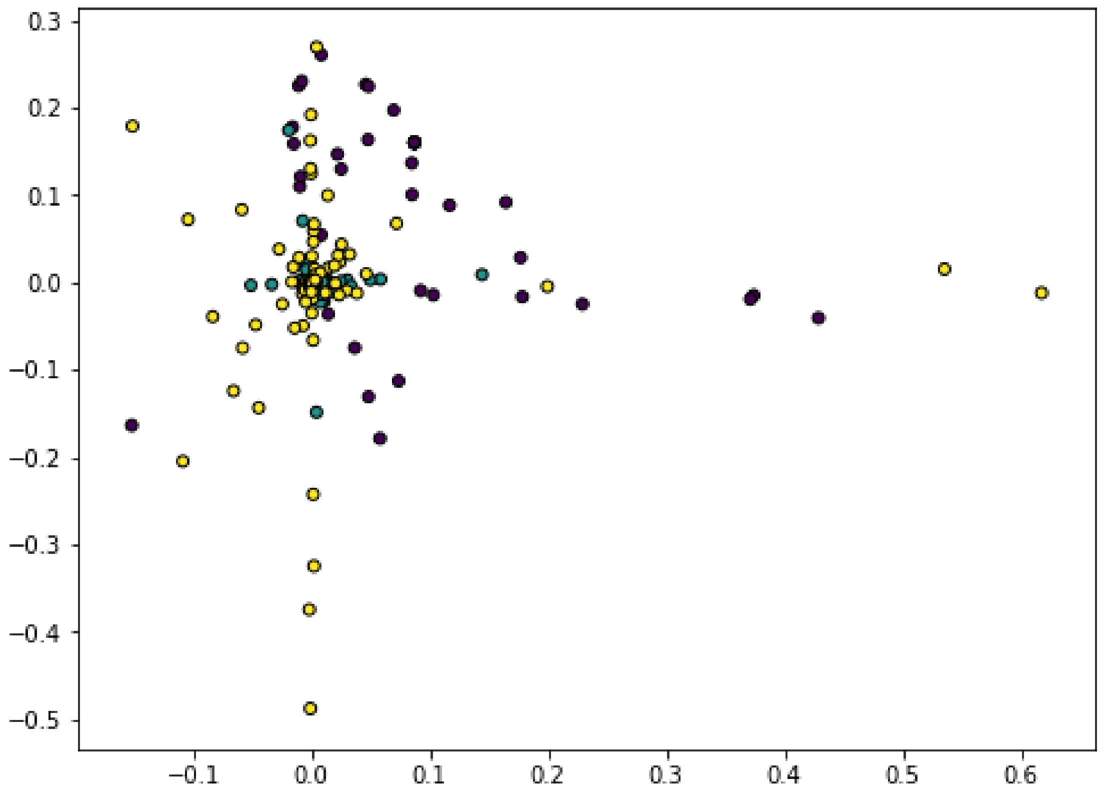

1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0



0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6


0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6

Kernelized Kmeans

November 4, 2018

1 Kmeans

Given N unlabeled examples and a number of desired partitions K, the goal in kmeans is to group the examples into K homogeneous partitions The most common algorithm uses an iterative refinement technique, given C an initial set of k centroids, the algorithm proceeds by alternating between two steps: - Assignment step: Assign each observation to the cluster whose mean has the least distance, this is intuitively the nearest centroid - Update step: Calculate the new means to be the centroids of the observations in the new clusters.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.cluster import SpectralClustering

        fig = 1

        def kmeans(X, K, maxIters = 10, plot_progress = None):
                global fig
                size = len(X)
                centroids_indexes = np.random.choice(np.arange(size), K)
                centroids = X[centroids_indexes,:]

                for i in range(maxIters):
                        # cluster Assignment step
                        C = np.array([np.argmin([np.dot(x_i-y_k, x_i-y_k) for y_k in centroids
                        # Move centroids
                        centroids = [X[C == k].mean(axis = 0) for k in range(K)]
                        if plot_progress:
                                fig += 1
                                plt.figure(fig, figsize=(8,6))
                                plt.clf()
                                plt.scatter(X[:,0], X[:,1], c=C, s=25, edgecolor='k')
                                plt.show()
                return np.array(centroids) , C
```

1.0.1 Kernel Kmeans

Kmeans is kernelized by replace the distance/similarity computations between an example x_n and a centroid $_k$ by the kernelized versions. for example:

$d(x_n,{}_k) = ||(x_n)({}_k)||$
replaced by:
$||(x_n) - ({}_k)||^2 = ||(x_n)||^2 + ||({}_k)||^2 - 2(x_n)^T({}_k) = K(x_n, x_n) + K({}_k,{}_k) - 2K(x_n,{}_k)$
where $K(a, b)$ denotes the kernel function and is its (implicit) feature map

1.0.2 Implementation

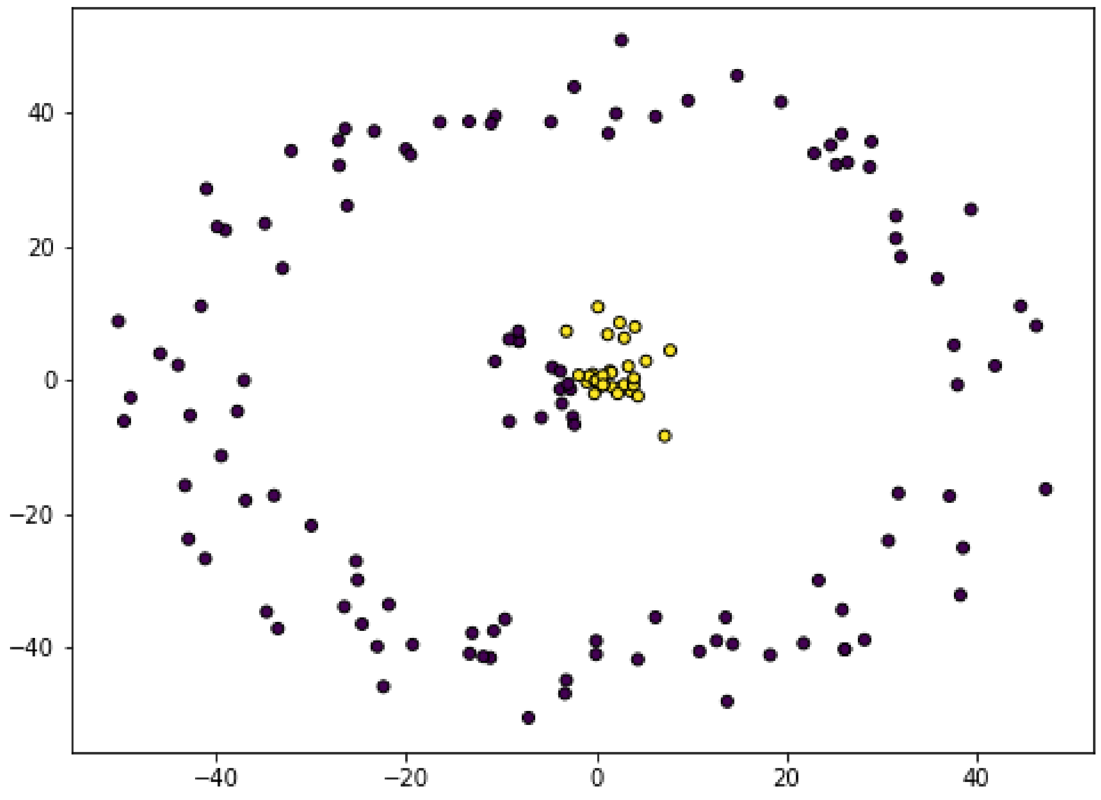
```
In [2]: from math import sqrt

        K2_SIGMA = sqrt( 1.0/ (2.0*0.3) )
        K3_alpha = 2
        K3_constatnt = 1
        def polynomial_kernel(X, Y):
                return (X.T.dot(Y) + 1) ** 2

        def gaussian_kernel(X, Y):
                return np.exp( (-1 * (np.linalg.norm(X - Y) ** 2)) / (2 * (K2_SIGMA ** 2)) )

        def rbf_kernel(v1, v2, sigma=1.0):
            return np.exp((-1 * np.linalg.norm(v1 - v2) ** 2) / (2 * sigma ** 2))

        def sigmoid_kernel(X, Y):
                return np.tanh(K3_alpha*X.T.dot(Y) + K3_constatnt)

        def kkmeans_kernel(x,y,kernel):
                return kernel(x,x) + kernel(y,y) - 2*kernel(x,y)

        def kkmeans(X, K,kernel, maxIters=10 ,plot_progress = None):
                global fig
                size = len(X)
                centroids_indexes = np.random.choice(np.arange(size), K)
                centroids = X[centroids_indexes,:]

                for i in range(maxIters):
                        # cluster Assignment step
                        C = np.zeros(len(X))
                        for x_i in range(len(X)):
                                idx = 0
                                current_kernel_distance = kkmeans_kernel(X[x_i],centroids[0],ke
                                for y_k in range(len(centroids)):
                                        temp = kkmeans_kernel(X[x_i],centroids[y_k],kernel)
                                        if temp < current_kernel_distance:
                                                current_kernel_distance = temp
                                                idx = y_k
                                C[x_i] = idx

                        centroids = [X[C == k].mean(axis = 0) for k in range(K)]
                print(centroids)
                return np.array(centroids) , C
```

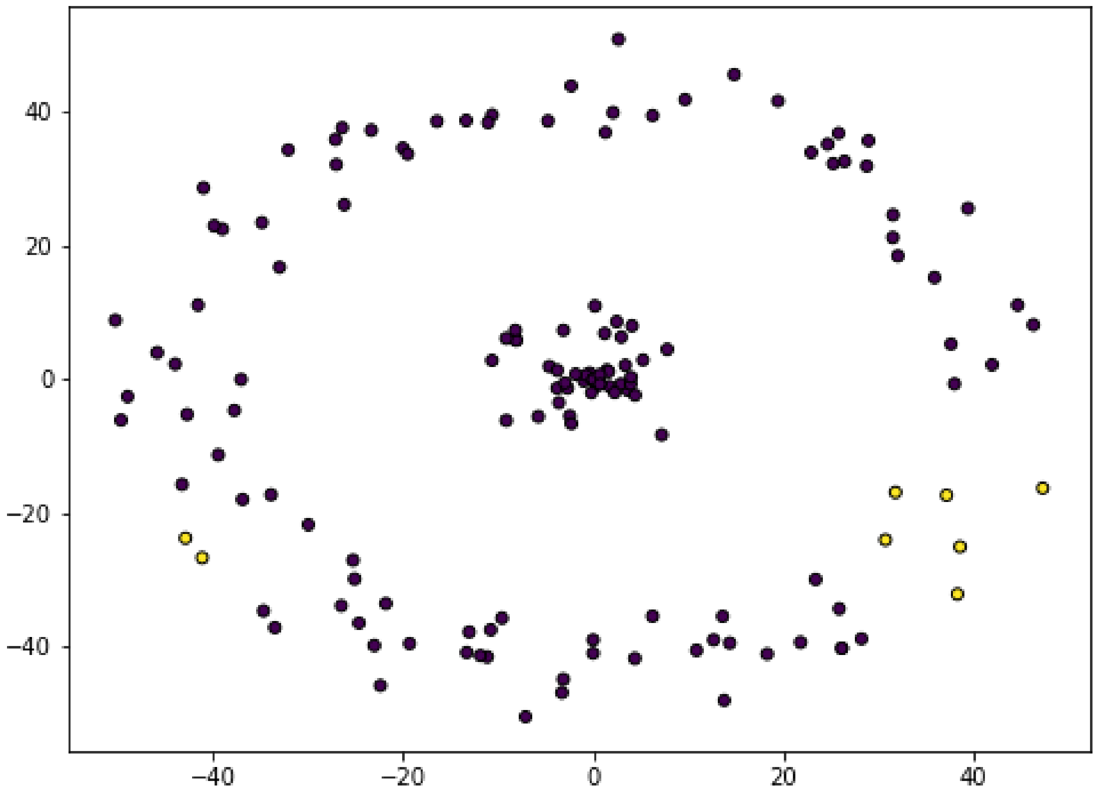
1.0.3 Tests

comparing our implementation of kernel kmeans with the one in sklearn

```
In [3]: if __name__ == "__main__":
                data = np.loadtxt("../data_k2.data")
                centroids, C = kkmeans(data, 2, gaussian_kernel ,maxIters=50, plot_progress=Tru
                fig += 1
                plt.figure(fig, figsize=(8,6))
                plt.clf()
                plt.scatter(data[:,0], data[:,1], c=C, s=25, edgecolor='k')
                plt.show()

                clustering = SpectralClustering(n_clusters=2 ,affinity = "rbf", gamma=0.3, coe
                fig += 1
                plt.figure(fig, figsize=(8,6))
                plt.clf()
                plt.scatter(data[:,0], data[:,1], c=clustering.labels_, s=25, edgecolor='k')
                plt.show()
```

```
[array([-3.90218148, -2.8254477 ]), array([1.59390657, 1.21866111])]
```





40
20
0
−20
−40
−40
−20
0
20
40

Kernelized Logistic Regression

November 3, 2018

1 Logistic Regression

Given a set of inputs X, we want to assign them to one of two possible categories (0 or 1). Logistic regression models the probability that each input belongs to a particular category.

1.0.1 Hypothesis

Logistic regression uses the sigmoid function that gives probability output between 0 and 1 for all values of X.
$h(\theta, X) = sigmoid(\theta^T X) = \frac{1}{1+e^{-\theta^T X}}$

1.0.2 Loss function

in logistic regression we learn the weights θ and we want to find the best values for them. To start we pick random values and we need a way to measure how well the algorithm performs using those random weights. That measure is computed using the loss function, defined as:
$J(\theta) = \frac{1}{m}(-y^T \log(h) - (1-y)^T \log(1-h))$

1.0.3 Gradient descent

The goal in logistic regression is to minimize the loss function by increasing/decreasing the weights θ fitting them with the data, to do that we use the derivative of the loss function with respect to each weight (i.e. the gradient). It tells us how loss would change if we modified the parameters. Then we update the weights by substracting to them the derivative times the learning rate α.
$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T (h - y)$
$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$

1.0.4 Prediction

to predict on new data we just need to check if the sigmoid probability is greater than 0.5 to predict class 1 otherwise 0.

1.0.5 Implementation

```
In [1]: import numpy as np
        from time import time

        def sigmoid(z):
            return 1 / (1 + np.exp(-z))

        def loss(h, y):
            return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

        # append new feature with value of ones (represents the bias for theta zero)
        def add_ones_feature(X):
            _X = np.ones((X.shape[0], X.shape[1] + 1))
            _X[:, 1:] = X
            return _X

        def fit(X, y, learn_rate=0.01, num_iter=100000):
            X = add_ones_feature(X)
            m = X.shape[0]
            d = X.shape[1]
            theta = np.zeros(d)

            for iteration in range(num_iter):
                h = sigmoid(np.dot(X, theta))
                gradient = np.dot(X.T, (h - y)) / m
                theta -= learn_rate * gradient

                if iteration % (num_iter // 10) == 0:
                    print(f'loss: {loss(h, y)}')

            return theta

        def predict(X, theta, threshold=0.5):
            X = add_ones_feature(X)
            return sigmoid(np.dot(X, theta)) >= threshold
```

1.0.6 Stochastic gradient decend

In stochastic gradient decend we calculate the gradient depending on one example at a time, the algorithm is altered as follows

```
In [2]: def fit_stochastic(X, y, learn_rate=0.01, num_iter=100000):
            X = add_ones_feature(X)
            m = X.shape[0]
            d = X.shape[1]
            theta = np.zeros(d)

            # reduce the number of iteration according to size of examples
```

```
        num_iter = num_iter // m
        for iteration in range(num_iter):
            for i in range(m): # loop through examples
                h = sigmoid(np.dot(X, theta))
                # calcualate the gradient depending on example i
                gradient = np.dot(X[i].T, (h[i] - y[i])) / m
                # decend
                theta -= learn_rate * gradient

            if iteration % (num_iter // 10) == 0:
                print(f'loss: {loss(h, y)}')

        return theta
```

1.0.7 Kernelized Logistic Regression

Logistic regression can be kernelized in the formula of the gradient as follows:

$Kmat = K(X_{train}, X_{train})$

$\frac{\partial J(\theta)}{\partial\theta} = \frac{1}{m} Kmat^T(h - y)$

$\theta = \theta - \alpha\frac{\partial J(\theta)}{\partial\theta}$

$h(\theta, X) = sigmoid(\theta^T K(X, X_{train})) = \frac{1}{1+e^{-\theta^T K(X, X_{train})}}$

```
In [3]: def rbf_kernel(v1, v2, sigma=1.0):
            return np.exp((-1 * np.linalg.norm(v1 - v2) ** 2) / (2 * sigma ** 2))

        def polynomial_kernel(v1, v2, c=1, d=2):
            return (v1.T.dot(v2) + c) ** d

        def sigmoid_kernel(v1, v2, alpha=2, c=1):
            return np.tanh(alpha * v1.T.dot(v2) + c)

        def compute_kmat(X, kernel):
            m = X.shape[0]
            kmat = np.zeros((m, m))
            for i in range(m):
                for j in range(m//2 + 1):
                    kmat[i, j] = kmat[j, i] = kernel(X[i], X[j])
            return kmat

        def compute_kmat2(X, train, kernel):
            m = X.shape[0]
            d = train.shape[0]
            kmat = np.zeros((m, d))
            for i in range(m):
                for j in range(d):
                    kmat[i, j] = kernel(X[i], train[j])
            return kmat
```

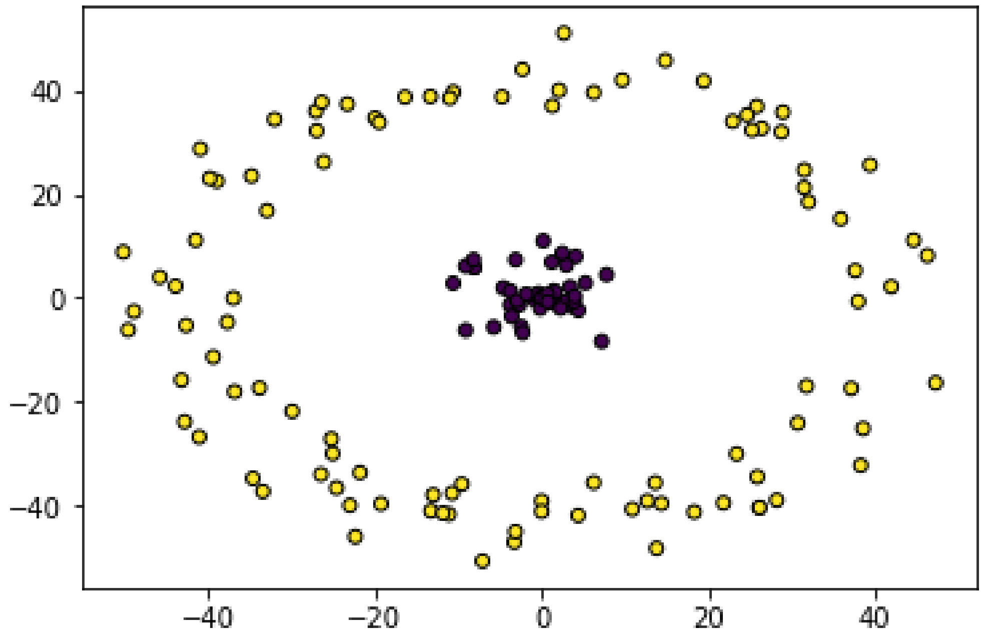
```
def k_fit(X, y, learn_rate=0.01, num_iter=100000, kernel=rbf_kernel):
    kmat = compute_kmat(X, kernel)
    kmat = add_ones_feature(kmat)
    m = kmat.shape[0]
    d = kmat.shape[1]
    theta = np.zeros(d)

    for iteration in range(num_iter):
        h = sigmoid(np.dot(kmat, theta))
        gradient = np.dot(kmat.T, (h - y)) / m
        theta -= learn_rate * gradient

        if iteration % (num_iter // 10) == 0:
            print(f'loss: {loss(h, y)}')

    return theta

def k_fit_stochastic(X, y, learn_rate=0.01, num_iter=100000, kernel=rbf_kernel):
    kmat = compute_kmat(X, kernel)
    kmat = add_ones_feature(kmat)
    m = kmat.shape[0]
    d = kmat.shape[1]
    theta = np.zeros(d)
    num_iter = num_iter // m
    for iteration in range(num_iter):
        for i in range(m):
            h = sigmoid(np.dot(kmat, theta))
            gradient = np.dot(kmat[i].T, (h[i] - y[i]))
            theta -= learn_rate * gradient

        if iteration % (num_iter // 10) == 0:
            print(f'loss: {loss(h, y)}')

    return theta

def k_predict(X, train, theta, threshold=0.5, kernel=rbf_kernel):
    kmat = compute_kmat2(X, train, kernel)
    kmat = add_ones_feature(kmat)
    return sigmoid(np.dot(kmat, theta)) >= threshold
```

1.0.8 Performance and accuracy tests

for testing we use a non linearly seperable data as follows: since the data is non linearly seperable, we gain better accuracy using the kernel method but more execution time since for calculating the kernel matrix *kmat* and learning the weights θ of size m in the new feature space

```
In [6]: import matplotlib.pyplot as plt
        data = np.loadtxt('../data2.data')
        X = data[:,:2]
        y = data[:,2]
        plt.scatter(X[:,0], X[:,1], c=y , s=25, edgecolor='k')
```

```
Out[6]: <matplotlib.collections.PathCollection at 0x7f0ffa911e48>
```



```
In [5]: if __name__ == '__main__':
            data = np.genfromtxt('../data2.data', delimiter=' ')
            np.random.shuffle(data) # shuffle the examples
            train_size = len(data) * 60 // 100 # split into train/test
            X_train = data[:train_size,:2]
            y_train = data[:train_size,2]
            X_test = data[train_size:,:2]
            y_test = data[train_size:,2]

            print("logistic regression")
            start = time()
            theta = fit(X_train, y_train)
            p = predict(X_test, theta)
            print(f'accuracy: {int((p == y_test).mean() * 100)}%')
            print(f'execution time: {time() - start} sec')

            print("\nkernelized logistic regression")
            start = time()
            theta = k_fit(X_train, y_train)
            p = k_predict(X_test, X_train, theta)
            print(f'accuracy: {int((p == y_test).mean() * 100)}%')
            print(f'execution time: {time() - start} sec')

            print("\nlogistic regression (stochastic GD)")
            start = time()
            theta = fit_stochastic(X_train, y_train)
            p = predict(X_test, theta)
            print(f'accuracy: {int((p == y_test).mean() * 100)}%')
            print(f'execution time: {time() - start} sec')

            print("\nkernelized logistic regression (stochastic GD)")
            start = time()
            theta = k_fit_stochastic(X_train, y_train)
            p = k_predict(X_test, X_train, theta)
            print(f'accuracy: {int((p == y_test).mean() * 100)}%')
            print(f'execution time: {time() - start} sec')
```

```
logistic regression
loss: 0.6931471805599454
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
accuracy: 63%
execution time: 3.0068016052246094 sec

kernelized logistic regression
loss: 0.6931471805599454
loss: 0.3557513106520843
loss: 0.3096036916085649
loss: 0.28634628609401674
loss: 0.2722585317916767
loss: 0.2627847679921432
loss: 0.25591848284368135
loss: 0.25065718472643855
loss: 0.24645382637279642
loss: 0.24298765662718153
accuracy: 90%
execution time: 5.7770607471466064 sec

logistic regression (stochastic GD)
```

```
loss: 0.6940528211005405
loss: 0.664342294738323
loss: 0.6463898368070594
loss: 0.6359956563432866
loss: 0.6299235146407501
loss: 0.626345621706906
loss: 0.6242220036959032
loss: 0.622954423544964
loss: 0.6221948321324273
loss: 0.6217386409496074
loss: 0.6214645316844505
accuracy: 63%
execution time: 2.7731752395629883 sec

kernelized logistic regression (stochastic GD)
loss: 0.6581890258247205
loss: 0.3550137394351634
loss: 0.3092470475276124
loss: 0.2861286940092524
loss: 0.2721096464247646
loss: 0.26267413155629904
loss: 0.2558309386026887
loss: 0.2505845471955586
loss: 0.2463913830660107
loss: 0.24293254250062946
loss: 0.2400092685665866
accuracy: 90%
execution time: 3.9141945838928223 sec
```

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)

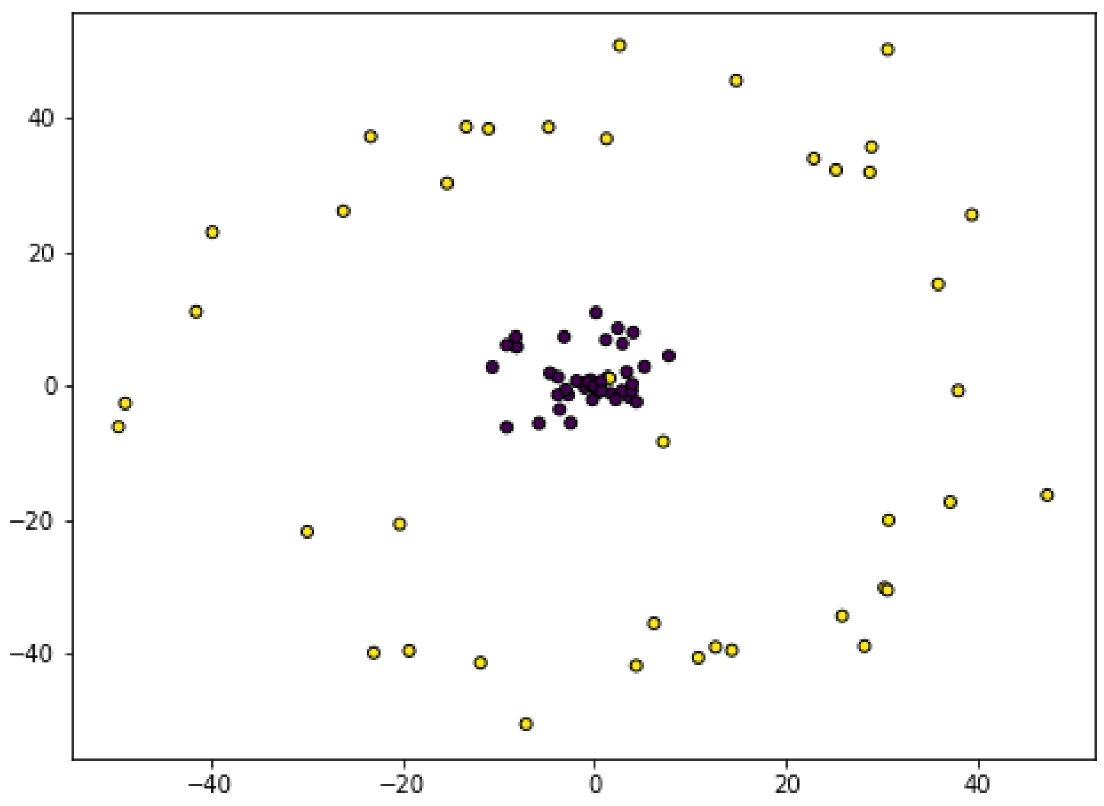
1.2 SVDD with Kernels:

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
```

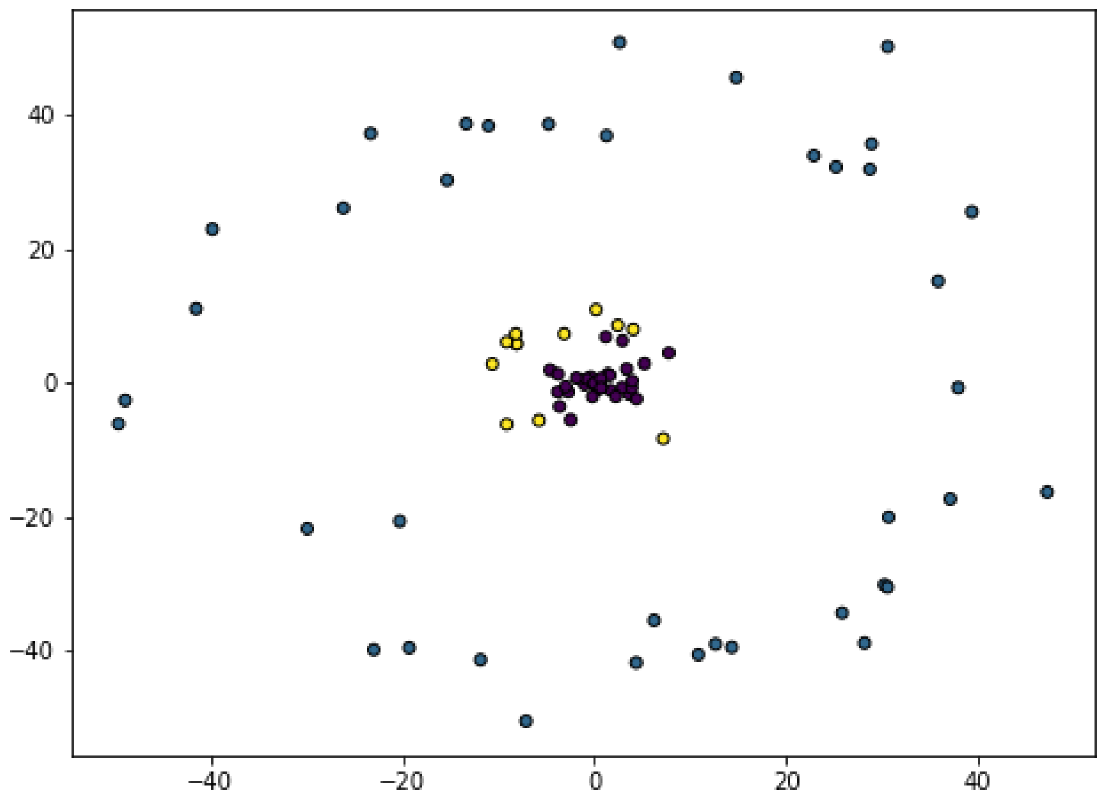
Note that for each $x_i$ we have three possibilities: $\alpha_i = C$, $x_i$ is an outlier
$0 < \alpha_i < C$, $x_i $ is a suport vector
$\alpha_i = 0$, $x_i $ is not a support vector or an outlier

```
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
```

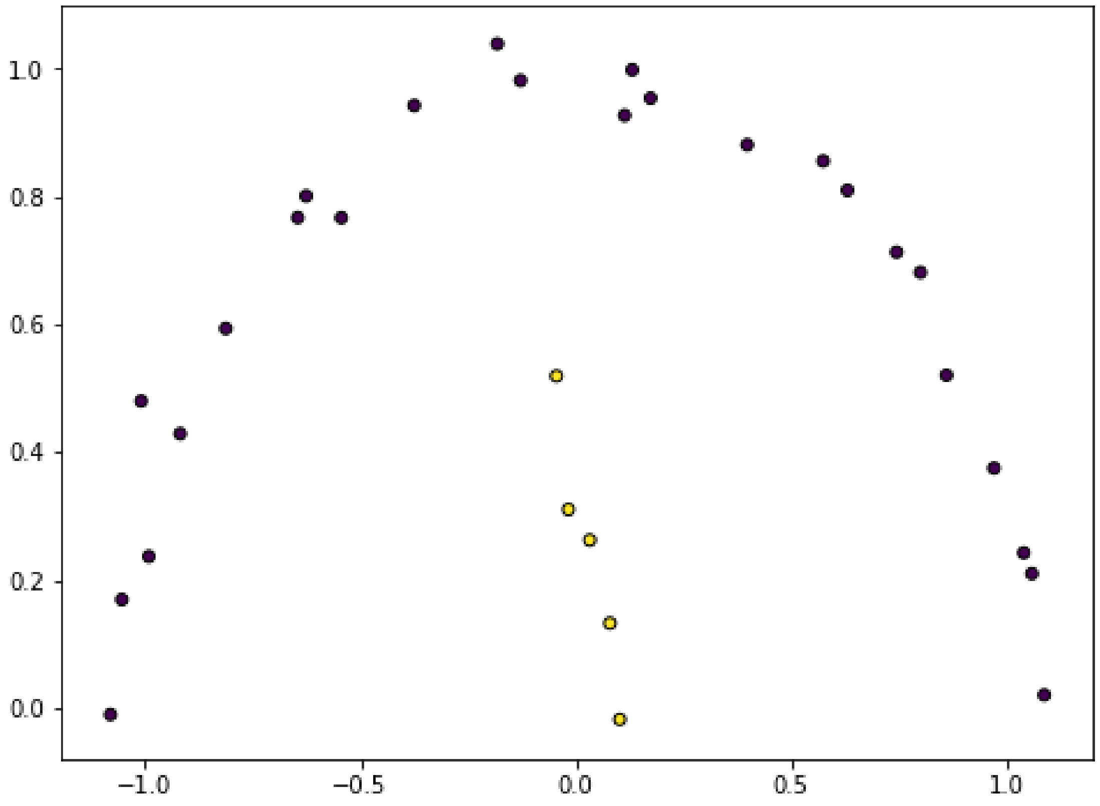
```python
m = X.shape[1] # number of features
# function to optimize
def L(a,*args):
    ret = 0
    for i in range(len(a)):
        ret += a[i]*kernel_arr[i,i]
    for i in range(len(a)):
        for j in range(len(a)):
            ret -= a[i]*a[j]*kernel_arr[i,j]
    return -1 * ret
# we add -1 because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

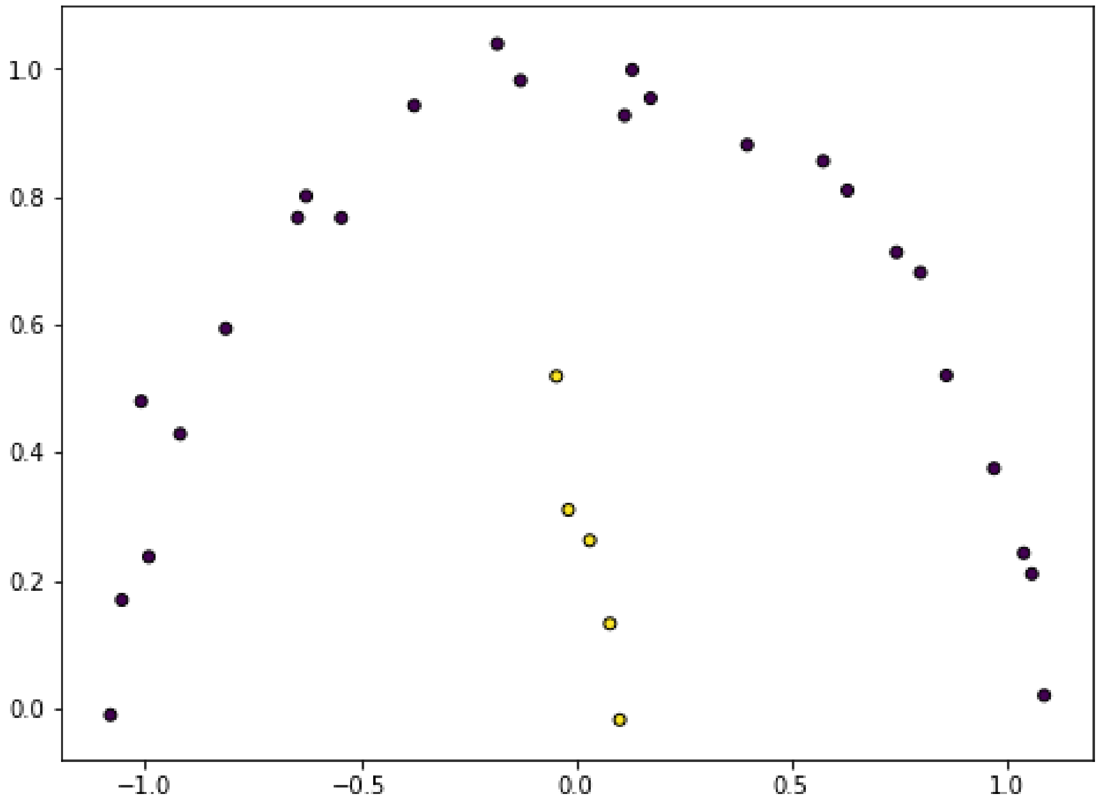
```
In [11]: from sklearn.datasets import make_moons

         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
```

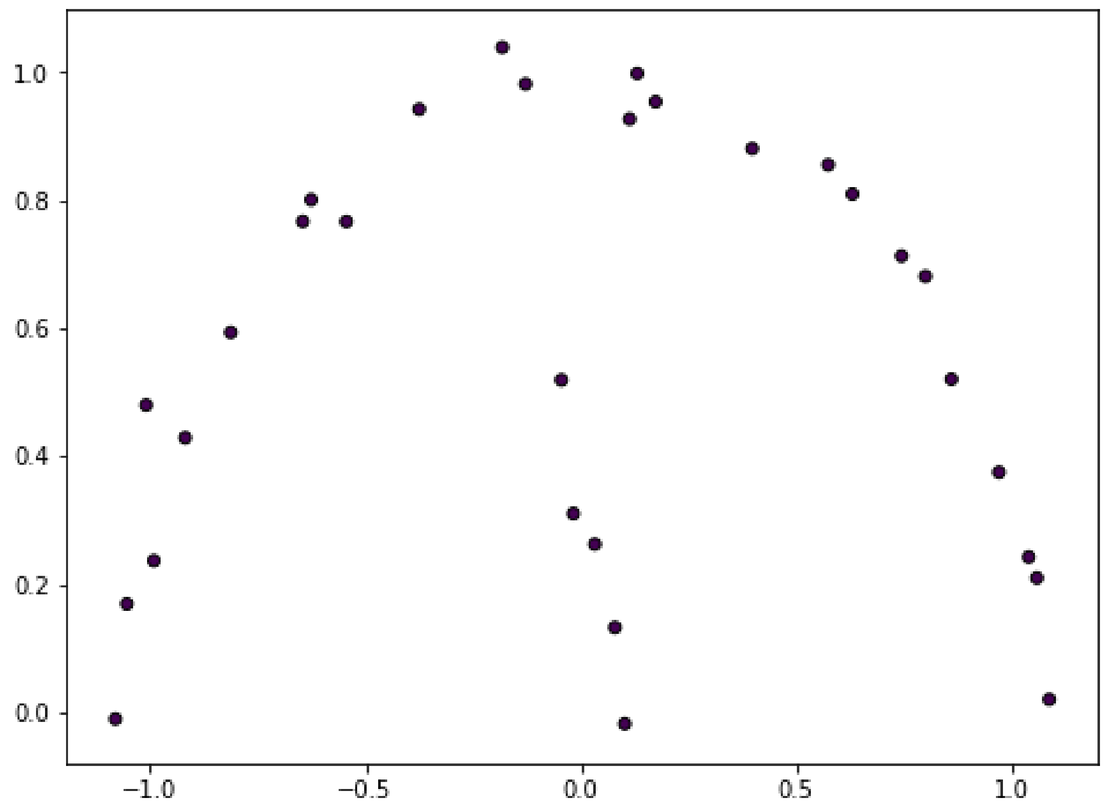
```
from scipy.spatial.distance import pdist, squareform

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