Notebook

February 23, 2022

1 Support Vector Machines

[195]:
```
import numpy as np
import pickle as pkl
from scipy import optimize
from scipy.linalg import cho_factor, cho_solve
import matplotlib.pyplot as plt
from utils import plotClassification, plotRegression, plot_multiple_images,␣
 ↪generateRings, scatter_label_points, loadMNIST
```

1.1 Loading the data

The file 'classification_datasets' contains 3 small classification datasets:

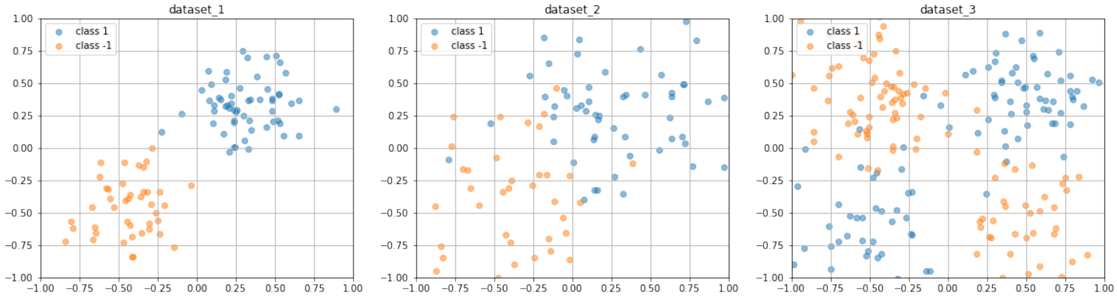
```
- dataset_1: mixture of two well separated gaussians
- dataset_2: mixture of two gaussians that are not separeted
- dataset_3: XOR dataset that is non-linearly separable.
```

Each dataset is a hierarchical dictionary with the following structure:

```
    dataset = {'train': {'x': data, 'y':label}
                'test': {'x': data, 'y':label}
              }
```

The data x is an N by 2 matrix, while the label y is a vector of size N.

[185]:
```
file = open('datasets/classification_datasets', 'rb')
datasets = pkl.load(file)
file.close()
fig, ax = plt.subplots(1,3, figsize=(20, 5))
for i, (name, dataset) in enumerate(datasets.items()):

    plotClassification(dataset['train']['x'], dataset['train']['y'], ax=ax[i])
    ax[i].set_title(name)
```



1.2 III- Kernel SVC

1.2.1 1- Implementing the Gaussian Kernel

Implement the method 'kernel' of the class RBF below, which takes as input two data matrices X and Y of size $N \times d$ and $M \times d$ and returns a gramm matrix G of shape $N \times M$ whose components are $k(x_i, y_j) = \exp(-\|x_i - y_i\|^2/(2\sigma^2))$. (The fastest solution does not use any for loop!)

[186]:
```
class RBF:
    def __init__(self, sigma = 1.):
        self.sigma = sigma  ## the variance of the kernel

    def kernel(self,X,Y):
        d = X.shape[1]
        M = Y.shape[0]
        N = X.shape[0]

        self.X = X
        self.Y = Y
        G = np.zeros((X.shape[0], Y.shape[0]))

        G[:,:] = [[np.exp(-(1/2*self.sigma)*np.linalg.norm((X[i,:]- Y[j,:]) ,␣
↪2)) for j in range(M)]
                  for i in range(N)]

        return G


class Linear:

    def kernel(self,X,Y):
        #d = X.shape[1]
        M = Y.shape[0]
        N = X.shape[0]

        self.X = X
```

```
        self.Y = Y
        G = np.zeros((X.shape[0], Y.shape[0]))

        G[:,:] = np.dot(X,Y.T)
        return G
```

1.2.2 2- Implementing the classifier

Implement the methods 'fit' and 'separating_function' of the class KernelSVC below to learn the Kernel Support Vector Classifier.

[192]:
```
class KernelSVC:

    def __init__(self, C, kernel, epsilon = 1e-3):
        self.type = 'non-linear'
        self.C = C
        self.kernel = kernel
        self.alpha = None
        self.support = None
        self.epsilon = epsilon
        self.norm_f = None


    def fit(self, X, y):
        N = len(y)
        self.X = X
        self.y = y
        self.Y = np.diag(self.y)
        self.K = self.kernel(self.X, self.X)
        #K = self.kernel(X,X)

        # Lagrange dual problem
        def loss(alpha):
            return -np.sum(alpha) +1/2*alpha.T.dot(np.diag(y)).dot(self.
↪kernel(X,X)).dot(np.diag(y)).dot(alpha)

        # Partial derivate of Ld on alpha
        def grad_loss(alpha):
            return -np.ones(N) + np.diag(y).dot(self.kernel(X,X)).dot(np.
↪diag(y)).dot(alpha)


        # Constraints on alpha of the shape :
        # -  d - C*alpha  = 0
        # -  b - A*alpha >= 0

        A = np.vstack((-np.eye(N), np.eye(N)))
```

```
        b = np.concatenate((np.zeros(N), self.C * np.ones(N)))

        fun_eq = lambda alpha: np.dot(self.y,alpha)
        jac_eq = lambda alpha: self.y
        fun_ineq = lambda alpha: b - np.dot(A, alpha)
        jac_ineq = lambda alpha: -A


        constraints = ({'type': 'eq',  'fun': fun_eq, 'jac': jac_eq},
                       {'type': 'ineq',
                        'fun': fun_ineq ,
                        'jac': jac_ineq})

        optRes = optimize.minimize(fun = lambda alpha: loss(alpha),
                                   x0 = np.ones(N),
                                   method = 'SLSQP',
                                   jac = lambda alpha: grad_loss(alpha),
                                   constraints = constraints)
        self.alpha = optRes.x

        ## Assign the required attributes
        sv = (self.alpha > self.epsilon)*(self.alpha < self.C)
        supportIndices = np.arange(len(self.alpha))[sv]
        self.support = X[supportIndices] #'''-------------------- A matrix with␣
↪each row corresponding to a support vector ------------------'''

        alpha_sv = self.alpha[sv]
        sv = np.argwhere(sv == True)
        y_sv = y[sv]

        # Bias value/intercept
        self.b = 0*1.0;
        for i in range(len(alpha_sv)):

            self.b += y_sv[i] - np.sum(alpha_sv * y_sv[:,0] * self.
↪kernel(X,X)[sv,supportIndices[i]])

        self.b /= len(alpha_sv)

        # '''------------------------RKHS norm of the function f␣
↪------------------------------'''
        self.norm_f = self.alpha.T.dot(self.Y).dot(self.kernel(X,X)).dot(self.
↪Y).dot(self.alpha)


    ### Implementation of the separting function $f$
    def separating_function(self,x):
```

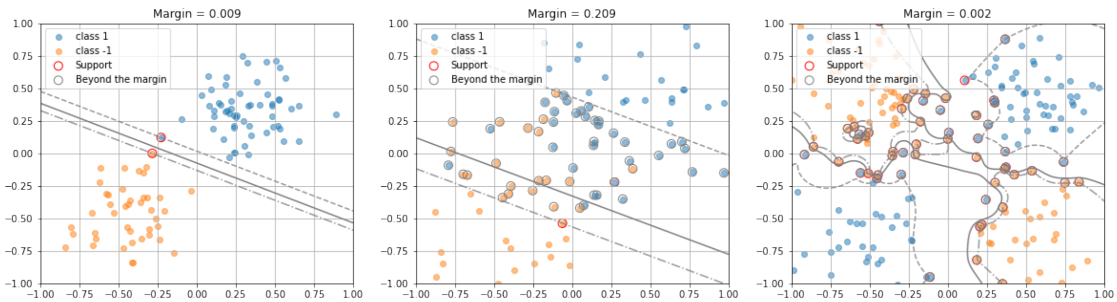
```
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        N, d = x.shape
        sv = (self.alpha > self.epsilon)*(self.alpha < C)
        out = np.ones(N)
        for i in range(N):
            x_i = x[i]
            for j in range(np.sum(sv)):
                out[i]+=self.alpha[sv][j]*self.y[sv][j]*self.kernel(x_i.
 ↪reshape(-1,1).T,self.support[j].reshape(-1,1).T)
        return out

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        d = self.separating_function(X)
        return 2 * (d+self.b> 0) - 1
```

[196]:

```
fig, ax = plt.subplots(1,3, figsize=(20, 5))
C = 10000.
kernel = Linear().kernel
model = KernelSVC(C=C, kernel=kernel)
train_dataset = datasets['dataset_1']['train']
model.fit(train_dataset['x'], train_dataset['y'])
plotClassification(train_dataset['x'], train_dataset['y'], model,␣
 ↪label='Training', ax = ax[0])

C = 10.
model = KernelSVC(C=C, kernel=kernel)
train_dataset = datasets['dataset_2']['train']
model.fit(train_dataset['x'], train_dataset['y'])
plotClassification(train_dataset['x'], train_dataset['y'], model,␣
 ↪label='Training', ax = ax[1])


sigma = 1.5
C=100.
kernel = RBF(sigma).kernel
model = KernelSVC(C=C, kernel=kernel)
train_dataset = datasets['dataset_3']['train']
model.fit(train_dataset['x'], train_dataset['y'])
plotClassification(train_dataset['x'], train_dataset['y'], model,␣
 ↪label='Training', ax=ax[2])

plt.savefig('SVM Classification.pdf')
```

```
Number of support vectors = 2
Number of support vectors = 3
```

```
Number of support vectors = 57
```



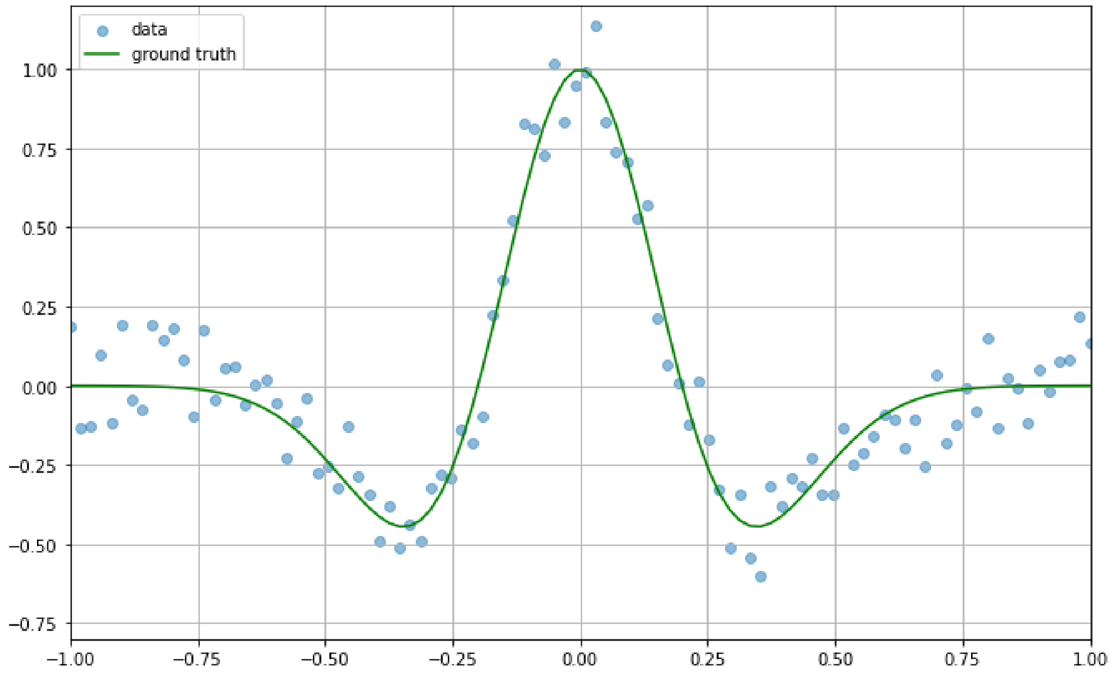
1.2.3 2- Fitting the classifier

Run the code block below to fit the classifier and report its output.

2 Kernel Regression

2.1 Loading the data

[22]:
```
file = open('datasets/regression_datasets', 'rb')
datasets = pkl.load(file)
file.close()
train_set = datasets['dataset_1']['train']
train_set = datasets['dataset_1']['test']
plotRegression(train_set['x'], train_set['y'],Y_clean= train_set['y_clean'])
```



2.2 Kernel Support Vector Regression

2.2.1 1- Implementing the regressor

Implement the method 'fit' of the classes KernelSVR below to perform Kernel Support Vector Regression.

[189]:
```python
class KernelSVR:

    def __init__(self, C, kernel, eta= 1e-2, epsilon = 1e-3):
        self.C = C
        self.kernel = kernel
        self.alpha = None # Vector of size 2*N
        self.support = None
        self.support1 = None
        self.support2 = None
        self.eta = eta
        self.epsilon = epsilon
        self.eps = 0.
        self.type='svr'
        self.sv = None
    def fit(self, X, y):
        #### You might define here any variable needed for the rest of the code
        N = len(y)

        self.X = X
        self.y = y
        # Lagrange dual problem
        def loss(alpha):
            alpha_plus = alpha[0:N]
            alpha_moins = alpha[N:2*N]
            return 1/2*(alpha_plus + alpha_moins).T.dot(self.kernel(X,X)).
↪dot(alpha_plus + alpha_moins) -(alpha_plus - alpha_moins).T.dot(y - self.
↪eta*np.ones(N))
        #'''-------------dual loss ------------------ '''

        # Partial derivate of Ld on alpha
        def grad_loss(alpha):
            alpha_plus = alpha[0:N]
            alpha_moins = alpha[N:2*N]
            grad_plus = -(y - self.eta*np.ones(N)) +self.kernel(X,X).
↪dot(alpha_plus + alpha_moins)
            grad_moins = (y - self.eta*np.ones(N)) +self.kernel(X,X).
↪dot(alpha_plus + alpha_moins)
            return np.concatenate((grad_plus,grad_moins))
```

```
        # Constraints on alpha of the shape :
        # -  d - C*alpha  = 0
        # -  b - A*alpha >= 0

        A = np.vstack((-np.eye(2*N), np.eye(2*N)))
        b = np.concatenate((np.zeros(2*N), C * np.ones(2*N)))

        Cmat = np.concatenate((np.ones(N), -np.ones(N)))
        fun_eq = lambda alpha:  np.dot(alpha,Cmat)
        # '''----------------function defining the equality␣
↪constraint------------------'''
        jac_eq = lambda alpha:   Cmat  #'''----------------jacobian wrt alpha␣
↪of the  equality constraint------------------'''
        fun_ineq = lambda alpha: b - np.dot(A, alpha)  #␣
↪'''---------------function defining the inequality␣
↪constraint-------------------'''
        jac_ineq = lambda alpha:  -A # '''---------------jacobian wrt alpha of␣
↪the  inequality constraint-------------------'''

        constraints = ({'type': 'eq',  'fun': fun_eq, 'jac': jac_eq},
                       {'type': 'ineq', 'fun': fun_ineq , 'jac': jac_ineq})

        optRes = optimize.minimize(fun=lambda alpha: loss(alpha),
                                   x0= self.C*np.ones(2*N),
                                   method='SLSQP',
                                   jac=lambda alpha: grad_loss(alpha),
                                   constraints=constraints,
                                   tol=1e-7)
        self.alpha = optRes.x
        ## Assign the required attributes
        sv = (self.alpha > self.epsilon)*(self.alpha < self.C)
        self.sv = sv
        sv1 = sv[0:N]
        sv2 = sv[N:2*N]
        sv = sv1|sv2
        supportIndices = np.arange(N)[sv]
        self.support = X[supportIndices] #'''------------------- A matrix with␣
↪each row corresponding to a support vector ------------------'''
        y_sv = y[supportIndices]
        self.support = [item for sublist in self.support for item in sublist]
        self.support = np.column_stack((self.support,y_sv))

        self.support1 = X[np.arange(N)[sv1]]
        self.support2 = X[np.arange(N)[sv2]]
```

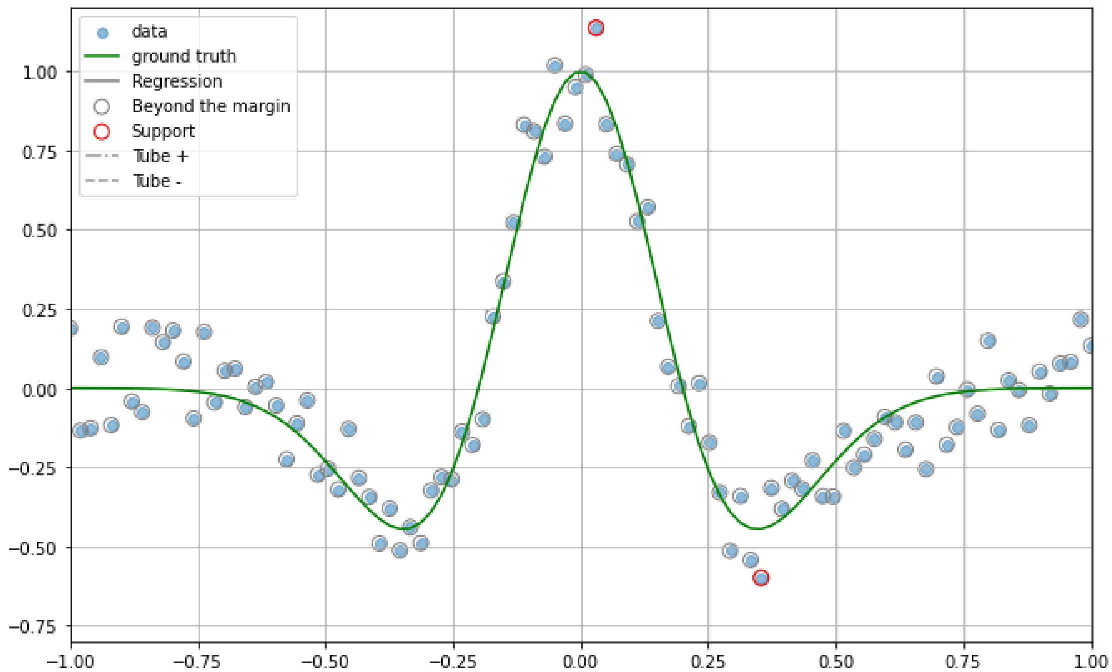
```
        alpha1_sv = self.alpha[0:N][sv1]
        sv1 = np.argwhere(sv1==True)
        sv1_y = y[sv1]

        margin_pos = 0*1.0;
        for i in range(len(alpha1_sv)):
            margin_pos += sv1_y[i] - np.sum(alpha1_sv * sv1_y[:,0] * self.
↪kernel(X,X)[sv1,np.arange(N)[sv1][i]]) - self.eta
        margin_pos /= len(alpha1_sv)

        alpha2_sv = self.alpha[N:2*N][sv2]
        sv2 = np.argwhere(sv2==True)
        sv2_y = y[sv2]

        margin_neg = 0*1.0;
        for i in range(len(alpha2_sv)):
            margin_neg += self.eta + sv2_y[i] - np.sum(alpha2_sv * sv2_y[:,0] *␣
↪self.kernel(X,X)[sv2,np.arange(N)[sv2][i]])
        margin_neg /= len(alpha2_sv)

        self.b = 0.5*(margin_pos + margin_neg)    #''' -----------------offset␣
↪of the regressor ------------------ '''

    ### Implementation of the separting function $f$
    def regression_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        N, d = x.shape
        out = np.ones(N)
        sv1 = self.sv[0:N]
        sv2 = self.sv[N:2*N]
        alpha1 = self.alpha[0:N]
        alpha2 = self.alpha[N:2*N]
        for i in range(N):
            x_i = x[i]
            for j in range(np.sum(sv1)):
                out[i]+=alpha1[sv1][j]*self.y[sv1][j]*(self.kernel(x_i.
↪reshape(-1,1),self.support1[j].reshape(-1,1)))
            for j in range(np.sum(sv2)):
                out[i]+=alpha2[sv2][j]*self.y[sv2][j]*(self.kernel(x_i.
↪reshape(-1,1).T,self.support2[j].reshape(-1,1)))
        return out

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        return self.regression_function(X)+self.b
```

2.2.2 2- Fitting the regressor

Run the code block below to fit the regressor and report its output.

[197]:
```
sigma = 0.2
C = 10.
kernel = RBF(sigma).kernel
model = KernelSVR(C,kernel, eta= .1, epsilon = 1e-6)
model.fit(train_set['x'].reshape(-1,1),train_set['y'])
plotRegression(train_set['x'], train_set['y'], Y_clean= train_set['y_clean'],␣
↪model=model, label='Train')
plt.savefig('Kernel Regression.pdf')
```

```
Number of support vectors = 2
```



[ ]:
```

```