Programming Assignment 6: SVM

1. Group member

   a) Hao Yang, USCID: 5284267375. (Contributions: All)

2. Part 1: Implementation

   1. Programming language: Python 3.7

   2. Libraries used: numpy, matplotlib.pyplot, cvxopt

   3. Algorithm: SVM (and with kernel function).

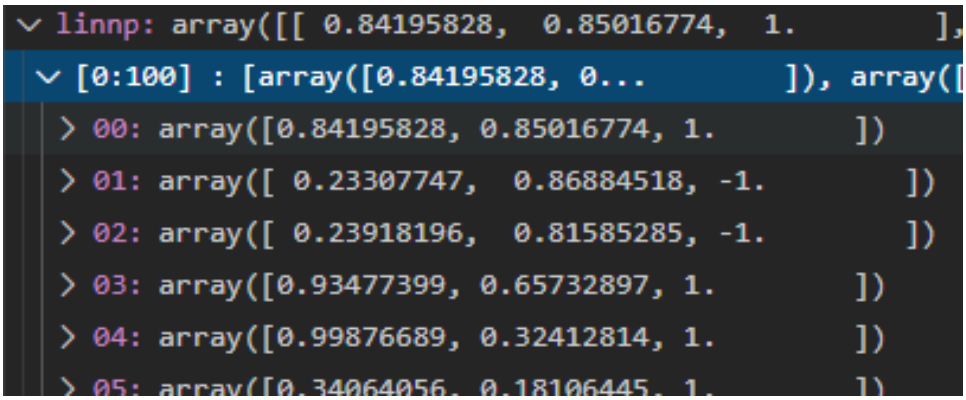
   4. Script description:

      1. Data processing: read 2 txt files row by row, record content in list.

```
linsep_data = []
with open("linsep.txt",'r') as file:
    line = file.readline().strip('\n').split(',')
    while line != ['']:
        line = list(map(float, line))
        linsep_data.append(line)
        line = file.readline().strip('\n').split(',')
linnp = np.array(linsep_data)

nonlin_data = []
with open("nonlinsep.txt",'r') as file:
    line = file.readline().strip('\n').split(',')
    while line != ['']:
        line = list(map(float, line))
        nonlin_data.append(line)
        line = file.readline().strip('\n').split(',')
nonnp = np.array(nonlin_data)
```

      Get the 100x3 matrix of data:



2. SVM without kernel function:

The function has 2 parameters, data is the data for processing, kernel is a Boolean type value to determine if use kernel function.

```
def SVM(data,kernel=False):
```

Objective function:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j$$

$$\text{s.t.} \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \,,$$

$$\alpha_i \geqslant 0 \,, \quad i = 1, 2, \ldots, m \,.$$

For using cvxopt.solver to solve this quadratic problem, change this into standard form:

$$\begin{aligned} \min_{x} \quad & \tfrac{1}{2} x^\top P x + q^\top x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

Step 1: compute all parameters needed for cvxopt.solver:

```
xx = x*x.T
if kernel==True:
    #xx = xx + 1
    xx = np.multiply(xx,xx)

yy = y*y.T
P = matrix(np.multiply(xx,yy))# P
q = matrix(np.mat([-1 for i in range(100)]).T,tc='d')    # q
GG = -1 * np.eye(100)
G = matrix(GG)                              # G
h = matrix([0 for i in range(100)],tc='d')     # h
A = matrix(y.T)                             # A
b = matrix([0],tc='d')                                   # b
```

Step 2: use cvxopt.solver.qp(P,q,G,h,A,b) to solve this quadratic problem,

and get the result of αs value:

```
result = solvers.qp(P,q,G,h,A,b)
arf = np.mat(result['x'])
```

αs value:

```
arf: array([1.16382858e-09, 2.95760572e-09,
[0:100] : [1.1638285823267093e-09, 2.95760
  00: 1.1638285823267093e-09
  01: 2.9576057208000305e-09
  02: 4.320742177542237e-09
  03: 6.229506504269129e-10
  04: 3.9651494672660533e-10
  05: 3.3085766278638857e-09
  06: 9.197179433631477e-10
  07: 2.296266155117199e-09
  08: 4.688750415285276e-10
  09: 9.067042131113468e-10
  10: 2.999572599038016e-09
  11: 7.271098890715295e-10
```

We can see most of it is nearly 1e-9 ~ 1e-10, we can see them as 0, and some of them is greater, which means they are for support vector:

```
26: 2.3916281291064953e-09
27: 33.738751924716944
28: 2.830795901558446e-09
```

Step 3: Based on the αs value, compute the weight and bias of the classified equation:

$$f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b$$
$$= \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x} + b \ .$$

```
if kernel == False:
    ya = np.multiply(y,arf)
    w = np.multiply(ya,x).sum(axis=0)          # w



    o = np.argwhere(arf == max(arf)).tolist()[0][0]
    b = 1/y[o] - w*x[o,:].T                    #b

    sv = np.argwhere(arf > 1)[:,0].tolist()

    """ arf = np.array(arf).reshape(100,)
    top2 = np.argsort(arf)[::-1]
    arf_sort = np.sort(arf)[::-1] """

    return w.tolist()[0],b.tolist()[0][0],sv
```

Then return the value of weights, bias, and index of support vectors.

```
return w.tolist()[0],b.tolist()[0][0],sv
```

Step 4: plot the result:

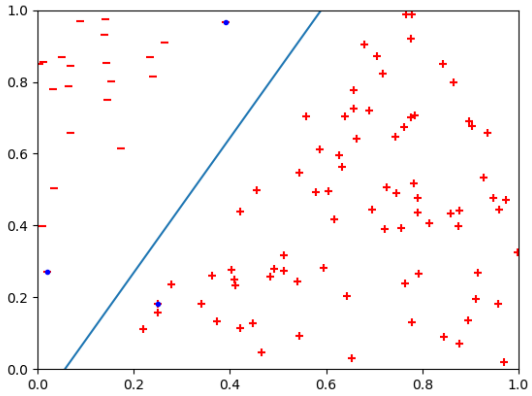
```
lin_po = []
lin_ne = []
for row in linsep_data:
    if row[2] == 1:
        lin_po.append(row[0:2])
    else:
        lin_ne.append(row[0:2])
lin_po = np.array(lin_po)
lin_ne = np.array(lin_ne)


plt.scatter(lin_po[:,0],lin_po[:,1], c='r',marker='+')
plt.scatter(lin_ne[:,0],lin_ne[:,1], c='r',marker='_')

svp = linnp[sv]
plt.scatter(svp[:,0],svp[:,1],c='b',marker='.')

X = np.linspace(0,1)
Y = -w[0]/w[1] * X + b
plt.plot(X,Y)

plt.xlim(0,1)
plt.ylim(0,1)
plt.show()
```



3. SVM with kernel function:

Same as SVM without kernel function, but set parameter of kernel as True:

```
sv = SVM(nonnp,kernel=True)
```

So, when solving the quadratic problem, our objective function becomes to:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

$$\text{s.t.} \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \ ,$$

$$\alpha_i \geqslant 0 \ , \quad i = 1, 2, \ldots, m \ .$$

In this assignment, we choose polynomial function of degree of 2 as the kernel function:

$\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j)^d$ and d = 2.

So, the P matrix in solver changes:

```
if kernel==True:
    #xx = xx + 1
    xx = np.multiply(xx,xx)
```

Then, same as part 2, we solve quadratic problem with cvxopt.solver.qp, then record the result of αs:

```
50: 3.1508852508531136e-13
51: 0.006619827376107213
52: 9.028772322548912e-13
```

I found that because I didn't standardize the data, so the αs become like 1000 times smaller. But we can still easily recognize the larger αs.

So, I pick αs who larger than 1e-5 as the non-zero αs which correspond support vectors, and return the list of index of support vectors.

```
else:
    """ arf = np.array(arf).reshape(100,)
    arf_sort = np.sort(arf)[::-1] """
    sv = np.argwhere(arf > 1e-5)[:,0].tolist()
    return sv
```

5. Optimizations
    1. For using solver.qp, need to compute matrix P. I compute it through matrix multiplication rather than computing it element by element, which can improve speed a lot.
    2. Combine SVM with and without kernel function in one function, for easier testing and using.
6. Challenges faced
    1. When loading cvxopt library at first, it always announces can't found module. After google the reason, found that the packages installed by pip and .whl file are incompatible. So, I reinstall the numpy library.
7. Result
    1. Run code below:

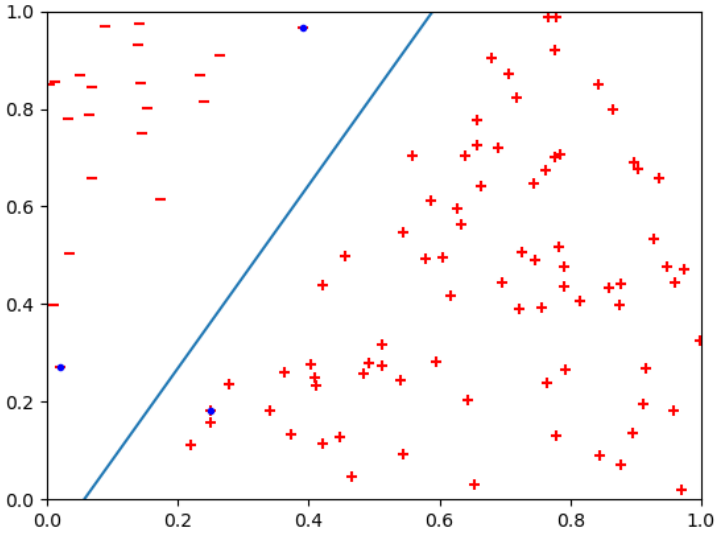
```
w,b,sv = SVM(linnp)
sv2 = SVM(nonnp,kernel=True)
```

```
print('the weight is %s and bias is %f' % (w,b))
print('The equation of the classification line:')
print('Y=%fX%f' % (-w[0]/w[1],b))
print('Support vectors:\nindex:'+str(sv)+'\ncoordinates:')
print(linnp[sv][:,0:2])
```

Get the result of weights and biases, and support vectors:

```
the weight is [7.250056295527885, -3.8618892417741426] and bias is -0.106987
The equation of the classification line:
Y=1.877334X-0.106987
Support vectors:
index:[27, 83, 87]
coordinates:
[[0.24979414 0.18230306]
 [0.3917889  0.96675591]
 [0.02066458 0.27003158]]
```

Plot the data points and equation:



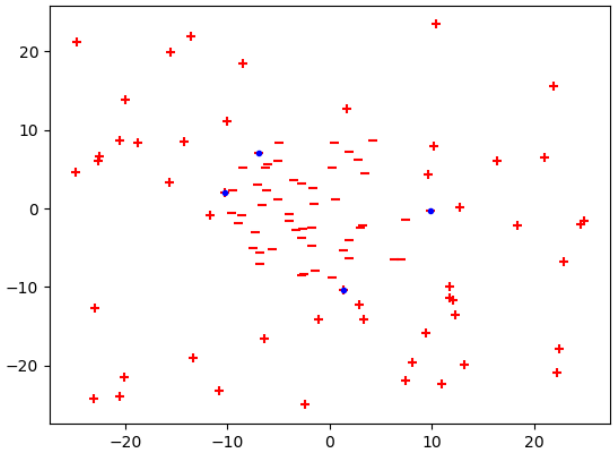
2. With kernel function:

$$\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j)^d$$ with d = 2

```
print('Support vectors for non seperable data:\nindex:'+str(sv2)+'\ncoordinates:')
print(nonnp[sv2][:,0:2])
```

Result:

```
Support vectors for non seperable data:
index:[36, 51, 59, 95]
coordinates:
[[-10.260969     2.07391791]
 [  1.3393313  -10.29098822]
 [ -6.90647562   7.14833849]
 [  9.90143538  -0.31483149]]
```



Can see the support vector is the point closest to margin.

3. Part 2: Software familiarization

    1. Library function: sklearn.svm.SVC

    2. How to use:

        1. same preprocessing for dataset, then define the model, set kernel as ‘linear’ for linear separable data, and train model by using clf.fit

```
clf = SVC(kernel='linear',C=1e4)
#clf = SVC(kernel='poly',degree=2,C=1e4)
clf.fit(x1,y1)

w = clf.coef_.tolist()[0]
b = clf.intercept_.tolist()[0]
```

then get the result of weight and bias:

```
> w: [7.248370686419742, -3.8609917821617756]
```

```
b: -0.10703977170718931
```

And index of support vector:

```
> support_: array([83, 87, 27])
> support_vectors_: array([[0.3917889 , 0.96675591],
```

        2. For polynomial kernel:

```
#clf = SVC(kernel='linear',C=1e4)
clf = SVC(kernel='poly',degree=2,C=1e4)
clf.fit(x2,y2)

""" w = clf.coef_.tolist()[0]
b = clf.intercept_.tolist()[0] """

w = clf.dual_coef_.tolist()[0]
b = clf.intercept_.tolist()[0]
```

Use ‘poly’ for the kernel parameter, and degree=2 for setting polynomial degree.

Result:

```
> w: [-825.2010252333962, -1447.9878541284113, 1830.718644205017, 442.4702351567914]
```

```
b: -18.92806835910506
```

```
> support_: array([59, 95, 36, 51])
> support_vectors_: array([[ -6.90647562,   7.14833849],
```

3. Comparison:

1. Difference:

The function in sklearn have more parameters can set than mine. For example, it can many different kernels like ‘linear’, ‘poly’, ‘rbf’ and so on. And it can set soft margin with different penalty. What’s more, it can use probability way to do prediction. These are what my implementation doesn’t have.

2. Results:

1. In linear separable case:

Sklearn:

```
> w: [7.248370686419742, -3.8609917821617756]
```

```
b: -0.10703977170718931
```

```
> support_: array([83, 87, 27])
> support_vectors_: array([[0.3917889 , 0.96675591],
```

My implement:

```
the weight is [7.250056295527885, -3.8618892417741426] and bias is -0.106987
```

```
Support vectors:
index:[27, 83, 87]
```

Totally same result.

2. In non-linear separable case:

Sklearn:

```
> support_: array([59, 95, 36, 51])
> support_vectors_: array([[ -6.90647562,   7.14833849],
```

My implement:

```
Support vectors for non seperable data:
index:[36, 51, 59, 95]
```

Totally same result.

4. Part 3: Applications

SVM can be seen as a more powerful classifier than normal classification algorithm such as PLA. So, it can be used in many real problems which need to classify something. For example, we can record patients' data like age, gender and number of disasters they have, to implement classification to see whether they are likely to get coronavirus, so that we can provide test to these group of people first. And it has better performance when new data come in.