I. Data Structure:

In implementing the support vector machine, I store the data into two python lists: one for the dataset 'linsip' and the other for 'nonlinsip'. They both have 100 elements with each element containing a single data point's 2D coordinates and its label:

```
In [3]: #Check our dataset:
        data

Out[3]: [[0.8419582822517558, 0.8501677437535663, 1.0],
         [0.23307746968273768, 0.8688451798911642, -1.0],
         [0.23918195824280708, 0.8158528536107376, -1.0],
         [0.9347739856327857, 0.6573289672157173, 1.0],
         [0.9987668869294112, 0.32412814215540975, 1.0],
         [0.3406405565820768, 0.1810644470444791, 1.0],
         [0.7256286049793105, 0.5075491271252338, 1.0],
         [0.6579669580938098, 0.7249592233990867, 1.0],
         [0.9741429172229583, 0.4724817472271823, 1.0],
         [0.6964835591575733, 0.4432087334183863, 1.0],
         [0.5591983734448611, 0.7037231355121139, 1.0],
         [0.6427350428227832, 0.20288368241020427, 1.0],
         [0.8761901105630542, 0.0708735322827797, 1.0],
         [0.7907550377804496, 0.4353234451342274, 1.0],
         [0.5124699958132691, 0.2750169688080156, 1.0],
         [0.756985175663958, 0.3940145993280094, 1.0],
         [0.7453415590888238, 0.4909175717419657, 1.0],
         [0.4107754534295499, 0.23457670796198404, 1.0],
         [0.37331149235116445, 0.13326311871430152, 1.0],
         [0.14301641813492094, 0.8531307906946035, -1.0],
         [0.051148602411527744, 0.8688355965822792, -1.0],
         [0.743192906140769, 0.6482934760317383, 1.0],
         [0.24989810202115104, 0.15693916581171696, 1.0],
         [0.6562836735929861, 0.7781237235377999, 1.0],
         [0.4829667161520854, 0.2565830728633888, 1.0],
         [0.2787257192329964, 0.2355277703641776, 1.0],
         [0.4090987808220684, 0.24822535270420076, 1.0],
         [0.24979413895365565, 0.18230306153352005, 1.0],
```

List for linsip.txt

```
In [3]: #Check our dataset:
        data

Out[3]: [[13.1470327418812, -19.8118230981223, 1.0],
         [10.2459271690412, 7.95373492160003, 1.0],
         [-20.5676950682991, 8.68464483573845, 1.0],
         [-13.5846563546538, 21.9403650399311, 1.0],
         [-14.2312187356579, 8.57661163030343, 1.0],
         [-8.47422847210542, 5.15621612964772, -1.0],
         [-15.6471972804416, 3.32039056055535, 1.0],
         [1.31699546839155, -5.38274816498792, -1.0],
         [10.4216324733359, 23.4527917054507, 1.0],
         [-11.6462129369006, -0.87217731265774, 1.0],
         [12.7478093058878, 0.199130319585805, 1.0],
         [-24.6824190933852, 21.1659453450988, 1.0],
         [-10.8605753317432, -23.245879585815, 1.0],
         [-22.4777991603284, 6.69516467703666, 1.0],
         [-10.0283331743503, 11.0935451070112, 1.0],
         [22.1938456843331, -20.8159949753654, 1.0],
         [9.40739889650235, -15.8014102533983, 1.0],
         [7.46390710222666, -21.9407849609038, 1.0],
         [-2.7471552037829, -8.47244873015782, -1.0],
         [-22.6381465946831, 6.10417436761104, 1.0],
         [21.9265386985187, 15.5313758383215, 1.0],
         [12.092887819127, -11.6528367117279, 1.0],
         [3.28969027044028, -14.1585453568545, 1.0],
```

List for nonlinsip.txt

Meanwhile, I create a list—'lambdas' to store the Lagrange multipliers λ_i in the form of the sympy variable:

```
In [5]: lambdas = []
        for i in range(len(data)):
            globals()['lambda'+str(i+1)] = sp.symbols('lambda{}'.format(i+1))
            lambdas.append(globals()['lambda'+str(i+1)])

In [6]: lambdas

Out[6]: [lambda1,
         lambda2,
         lambda3,
         lambda4,
         lambda5,
         lambda6,
         lambda7,
         lambda8,
         lambda9,
         lambda10,
         lambda11,
         lambda12,
         lambda13,
         lambda14,
         lambda15,
         lambda16,
         lambda17,
         lambda18,
         lambda19,
```

The list 'lambdas' is of great importance. It will participate in the construction of the Lagrange Duality functions, KKT conditions, and the optimization computation using the scipy package. Each of the 'linsep' and 'nonlinsep' dataset creates 100 λ_i with each λ_i corresponding to one data point.

(*To TA/professor: I'm sorry this homework is half-completed, I struggle to find the way to do the multivariable optimization for calculating λ_i. In the following, I can only list the processes and the concepts of coding without final results; as for detailed problems and the approaches I've tried, I discuss them in III. Challenges.)

II. Coding

(a) SVM for linear separable data:

1. Import the packages and read in the dataset 'linsep.txt'.
2. Store the 'linsep.txt' in the form of list.
3. Create a list 'lambdas' to store λ_i
4. Construct the duality Lagrange function:

$$max_{\lambda} min_{w,b} L(w,b,\lambda) = \frac{1}{2} w^T w + \sum_{i=1}^{100} \lambda_i \left(1 - y_i (w^T \cdot x_i + b)\right).$$

After computing the gradients of weights and bias, we can rewrite the Lagrange function to:

$$\min \ \frac{1}{2} \sum_{i=1}^{100} \sum_{j=1}^{100} \lambda_i \lambda_j y_i y_j x_i^T x_j - \sum_{i=1}^{100} \lambda_i$$

This is our target of optimization.

In [7]:
```
#Constructing the duality lagrange function:
sigma_ll_yy_xx = 0
for i in range(len(data)):
    for j in range(len(lambdas)):
        sigma_ll_yy_xx += lambdas[i] * lambdas[j] * data[i][2] * data[j][2] * np.dot(data[i][0:2],data[j][0:2])

sigma_lambda = 0

for lambda_i in lambdas:
    sigma_lambda += lambda_i

#This equation is our minimization target:
lagrange = (1/2)*sigma_ll_yy_xx-(sigma_lambda)
```

In [ ]:
```
#Write the duality lagrange function into the general python function form
def lagrange_min(x):
    zipped = zip(lambdas,x)
    return lagrange.subs(zipped)
```

5. Define the functions for the KKT condition:
   (1) $\sum_{i=1}^{100} \lambda_i y_i = 0$:
   (2) all $\lambda_i \geq 0$

In [67]:
```
#KKT:
#Sum of lambdai x yi must be zero:
def sigma_lambda_y(x):
    sigma_lambda_y = 0
    for i in range(len(lambdas)):
        sigma_lambda_y  += lambdas[i]*data[i][2]-0
    zipped = zip(lambdas,x)
    return sigma_lambda_y.subs(zipped)
```

In [78]:
```
#KKT:
#All lambda value must be no less than zero:
lamb_positive = []
for i in range(len(lambdas)):
    globals()['bound'+str(i+1)]= (0,np.inf)
    lamb_positive.append(globals()['bound'+str(i+1)])

lamb_positive = tuple(lamb_positive)
```

6. Solving the optimization problem using *scipy.optimization* (incomplete)

In [1]:
```
#Minimize the Lagrangian function:
x0=[2]*100
solution_lambda = minimize(lagrange_min,x0,tol=1e-8,method='COBYLA',bounds=lamb_positive,constraints=[{'type':'eq','fun':sigma_la
```

In [ ]:
```
#Second attempt
solution_lambda2 = minimize(lagrange_min,x0,tol=1e-8,method='nelder-mead')
```

In [ ]:
```
#Third attempt, using differential_evolution
solution_lambda3 = differential_evolution(lagrange,bounds=lamb_positive)
```

7. Find the equation of fittest margin line, using

$$W = \sum_{i=1}^{100} \lambda_i y_i x_i$$

$$b = \ y_i - \sum_{i=1}^{100} \lambda_i y_i \, x_i^T x_j$$

(b) SVM with Kernel method for non-linear separable data:

1. Import the packages and read in the dataset 'linsep.txt'.
2. Store the 'linsep.txt' in the form of list.
3. Transform 2D data points to z-D space using polynomial transformation:

$$\Phi(x) = (1, x_1, x_2, x_1x_1, x_1x_2, x_2x_2, x_2x_1)$$

In [8]:
```
#polynomial transformation
data_transform = []
for i in range(len(data)):
    new_space = [1,data[i][0],data[i][1],data[i][0]**2,data[i][0]*data[i][1],data[i][1]**2,data[i][1]*data[i][0]]
    data_transform.append(new_space)
```

In [6]:
```
data_transform[0:10]
```

Out[6]:
```
[[1,
  13.1470327418812,
  -19.8118230981223,
  172.8444699160963,
  -260.4666869473721,
  392.5083344712923,
  -260.4666869473721],
 [1,
  10.2459271690412,
  7.95373492160003,
  104.97902355329663,
  81.49338872857354,
  63.261899203079835,
  81.49338872857354],
 [1,
  -20.5676950682991,
  8.68464483573845,
  423.03008042253504,
  -178.62312675794695,
  75.42305592291852,
  -178.62312675794695],
 [1,
  -13.5846563546538,
  21.9403650399311,
  184.54288827403587,
```

4. Define the kernel function for polynomial transformation:

$$k(x_i, x_j) = 1 + x_i^T \cdot x_j + (x_i^T \cdot x_j)^2$$

```
In [ ]: #polynomial kernel
        def polynomial_kernel(xi,xj):
            k = 1 + np.dot(xi,xj) + (np.dot(xi,xj))**2
            return k
```

5. Create a list 'lambdas' to store λ_i.
6. Construct the duality Lagrange function using kernel trick:

$$\min \frac{1}{2}\sum_{i=1}^{100}\sum_{j=1}^{100}\lambda_i\lambda_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{100}\lambda_i$$

```
In [ ]: sigma_ll_yy_xx = 0
        for i in range(len(data)):
            for j in range(len(lambdas)):
                sigma_ll_yy_xx += lambdas[i] * lambdas[j] * data[i][2] * data[j][2] * polynomial_kernel(data[i][0:2],data[j][0:2])
```

```
In [ ]: def lagrange_min(x):
            sigma_lambda = 0
            for lambda_i in lambdas:
                sigma_lambda += lambda_i

            lagrange = (sigma_lambda)-(1/2)*sigma_ll_yy_xx

            zipped = zip(lambdas,x)

            return lagrange.subs(zipped)
```

7. Define functions for the KKT condition.
8. Solve the optimization problem(incomplete):

```
In [ ]: #Minimize the Lagrangian function:
        solution_lambda = minimize(lagrange_min,x0,tol=1e-8,bounds=lamb_positive,constraints=[{'type':'eq','fun':sigma_lambda_y}])
```

```
In [ ]: #Second attempt
        solution_lambda2 = minimize(lagrange_min,x0,tol=1e-8,method='nelder-mead')
```

```
In [ ]: #Third attempt, using differential_evolution
        solution_lambda3 = differential_evolution(lagrange_min,bounds=lamb_positive)
```

9. Find the equation of fittest margin line, using

$$W = \sum_{i=1}^{100}\lambda_i y_i x_i$$

$$b = y_i - \sum_{i=1}^{100}\lambda_i y_i\, k(x_i, x_j)$$

III. Challenges

The primary struggle upon me is solving optimization problem for the Lagrange function. Speaking further, when I solve the minimum of the target function:

$$\min \frac{1}{2}\sum_{i=1}^{100}\sum_{j=1}^{100}\lambda_i\lambda_j y_i y_j x_i^T x_j - \sum_{i=1}^{100}\lambda_i$$

, its mathematic nature, as well as the KKT constraints, both narrow down the scope of usable

python commends; and give us a close but no cigar outcome.

First, the targeted function and the constraints are non-smooth functions; and of the commands I've tried: *scipy.optimization* and *sympy.diff*, unfortunately, non-smoothness becomes the cause of many problems. In *scipy.optimization*, the smoothness of the targeted function is the prerequisite for most optimization method, which includes 'COBYLA' method and 'SLSQP' method. When this prerequisite is unattainable, the *scipy.optimization.minimize* function will be a malfunctioning one and always return the initial guess values. Considering this, I attempt to make the targeted function smooth using log transformation and the Savitzky-Golay filter; however, they both return the λ_i that only have mathematic meaning but no meaning about the SVM, that is, not forming the proper classification plane.

So, the next step I take is to circumvent the glitch of non-smoothness with two approaches. The first one is using another method in *scipy.minimize*, like 'nelder-mead' method; the second one is trying to find the global minima, as opposed to local minima, of the targeted function using *scipy.differential_evolution* or *sympy.diff*. Nonetheless, when I apply these approaches, a new obstacle is born: the constraints function. Either in the method like 'nelder-mead' or *scipy.differential_evolution*, they both demand a non-constraint optimization problem, which makes the outcomes fail to meet the KKT conditions.

In sum, this homework reminds me how broad the room I can improve my coding and mathematical thinking abilities. Particularly, I fall short of an ability to active-mindedly formulate a new coding approach when an old one fails to meet the expectation; not to mention when the failure is mixed with a complex mathematic concept. I hope I will refine this downside of me by learning more computer languages.