MA4270 Computational Exercise

Wang Yanhao

A0113742N

Tut01

This project use Python to implement all the algorithms.
At the start of the program, we need to import the following libraries:

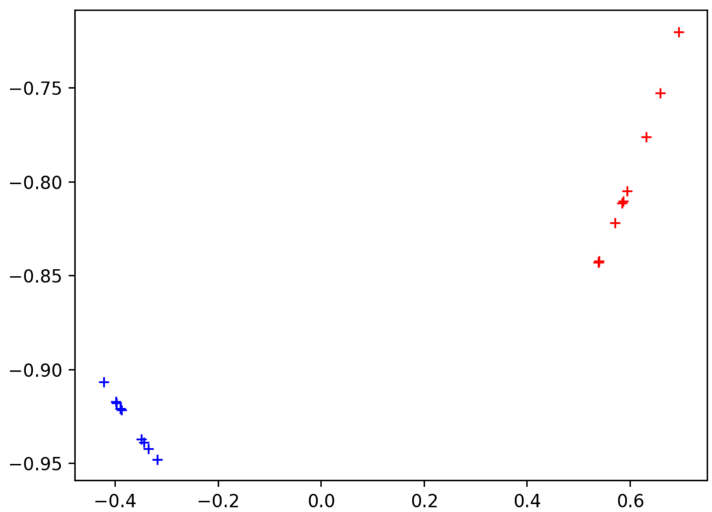
```
import csv
import math
import numpy
import matplotlib.pyplot as plt
# check following link on how to use cvxopt:
# https://courses.csail.mit.edu/6.867/wiki/images/a/a7/Qp-cvxopt.pdf
from cvxopt import matrix
from cvxopt import solvers

```

Problem1: Perceptron

We will read the input file and preprocess the data before performing any tasks:

```
# main
reader = csv.reader(open("Problem1.csv", "rb"), delimiter=",")
data = numpy.array(list(reader))
y = data[:, len(data[0]) - 1]
y = y.astype(int)
x = numpy.delete(data, len(data[0])-1, axis=1)
x = x.astype(float)
```

1. Plot: as shown from the graph, the dataset is linearly separable.



Code:

```
def part1(x, y):
    x1 = []
    x2 = []
    for i in range(len(x)):
        if y[i] == 1:
            x1.append(x[i, :])
        else:
            x2.append(x[i, :])
    x1 = numpy.array(x1)
    x2 = numpy.array(x2)
    plt.plot(x1[:,0], x1[:,1], 'r+')
    plt.plot(x2[:,0], x2[:,1], 'b+')
    plt.show()
```

2. Using quadratic programming solver, we can find normalised optimal theta and corresponding gamma using primal SVM without offset and slack variables:

```
def part2(x, y):
    # QP to solve primal form for SVM
    # convert to cxvopt matrices
    P = matrix(numpy.eye(2), tc='d')
    q = matrix(numpy.zeros(2), tc='d')
    G = []
    for i in range(len(y)):
        G.append([-1 * y[i] * x[i, 0], -1 * y[i] * x[i, 1]])
    G = matrix(numpy.array(G), tc='d')
    h = matrix(-1 * numpy.ones(len(y)), tc='d')
    sol = solvers.qp(P,q,G,h)

    theta = numpy.array([sol['x'][0], sol['x'][1]])
    theta /= numpy.linalg.norm(theta)
    min_gamma = get_min_gamma(x, y, theta)
    print "is optimal status : " + str(sol["status"])
    print "optimal theta is : " + str(theta)
    print "corrsponding minimum gamma is : " + str(min_gamma)
```

And the result is:

```
Optimal solution found.
is optimal status : optimal
optimal theta is : [ 0.99257345  0.12164684]
corrsponding minimum gamma is : 0.431484596179
```

3. Code for standard perceptron algorithm:

```python
def get_min_gamma(x, y, theta):
    min_gamma = float("inf")
    for i in range(len(x)):
        min_gamma = min(min_gamma, y[i] * numpy.dot(theta, x[i, :]))
    return min_gamma

def perceptron(x, y, theta_0, s_index=0):
    theta = theta_0
    index = s_index
    counter = 0
    tot_updates = 0
    max_iterations = 1000000
    while counter < len(x):
        # check if classified correctly
        gamma = y[index] * numpy.dot(theta, x[index, :])
        if gamma <= 0:
            counter = 0
            tot_updates += 1
            theta += numpy.dot(int(y[index]), x[index, :])
            if (tot_updates > max_iterations):
                raise Exception('Maximum number of iterations exceed!')
        else:
            counter += 1
        index = (index + 1) % len(x)
    # normalise theta before return
    theta /= numpy.linalg.norm(theta)
    min_gamma = get_min_gamma(x, y, theta)
    return [theta, min_gamma, tot_updates]
```

A. Using zero vector as the starting point:

```python
def part3(x, y):
    # part a
    print "\nsubtask 3 part (a):"
    result = perceptron(x, y, numpy.zeros(len(x[0])))
    print "number of iterations: " + str(result[2])
    print "converged solution theta: " + str(result[0])
    print "corresponding minimum gamma: " + str(result[1])
```

And the result:

```
subtask 3 part (a):
number of iterations: 2
converged solution theta: [ 0.98365688  0.18005314]
corresponding minimum gamma: 0.377454883747
```

B. Run perceptron 10 times with different starting points:

```
# part b
print "\nsubtask 3 part (b):"
for i in range(10):
    theta_0 = numpy.array([numpy.random.rand(), numpy.random.rand()])
    print "starting point: " + str(theta_0)
    print perceptron(x, y, theta_0)

```

And the result:

For each of the result, the corresponding value represents:

normalised theta   corresponding gamma   number of iterations

```
subtask 3 part (b):
starting point: [ 0.62716575  0.58684248]
[array([ 0.99180762, -0.12774054]), 0.19484386544643881, 1]
starting point: [ 0.49213671  0.48666704]
[array([ 0.97435096, -0.22503378]), 0.097057982812294885, 1]
starting point: [ 0.28530049  0.46775625]
[array([ 0.95740413, -0.28875134]), 0.031261226389209307, 1]
starting point: [ 0.46017331  0.79743867]
[array([ 0.99919253,  0.04017823]), 0.35636813604394341, 1]
starting point: [ 0.87318812  0.48990007]
[array([ 0.87211626,  0.48929871]), 0.056771431745041623, 0]
starting point: [ 0.44190921  0.62229196]
[array([ 0.99307917, -0.11744687]), 0.20500637210602318, 1]
starting point: [ 0.19085655  0.64217914]
[array([ 0.99168022, -0.12872583]), 0.19386931683409739, 1]
starting point: [ 0.03659156  0.43842097]
[array([ 0.85888624,  0.5121664 ]), 0.030377553991586 37, 2]
starting point: [ 0.87229227  0.36794529]
[array([ 0.92138421,  0.38865297]), 0.16811588730146892, 0]
starting point: [ 0.28205161  0.08731426]
[array([ 0.95527386,  0.29572258]), 0.26468300722379312, 0]
```

From the above result, we observe that with different theta_0, we obtain different value of theta, gamma and number of iteration.

C. Prove upper bound for number of iterations required to successfully classify the dataset in terms of the starting point

D. Code to generate random point from unit circle:

```
def sample_unit_circle(npoints=1, ndim=2):
    vec = numpy.random.randn(ndim, npoints)
    vec = numpy.array([vec[0][0], vec[1][0]])
    return vec / numpy.linalg.norm(vec)

```

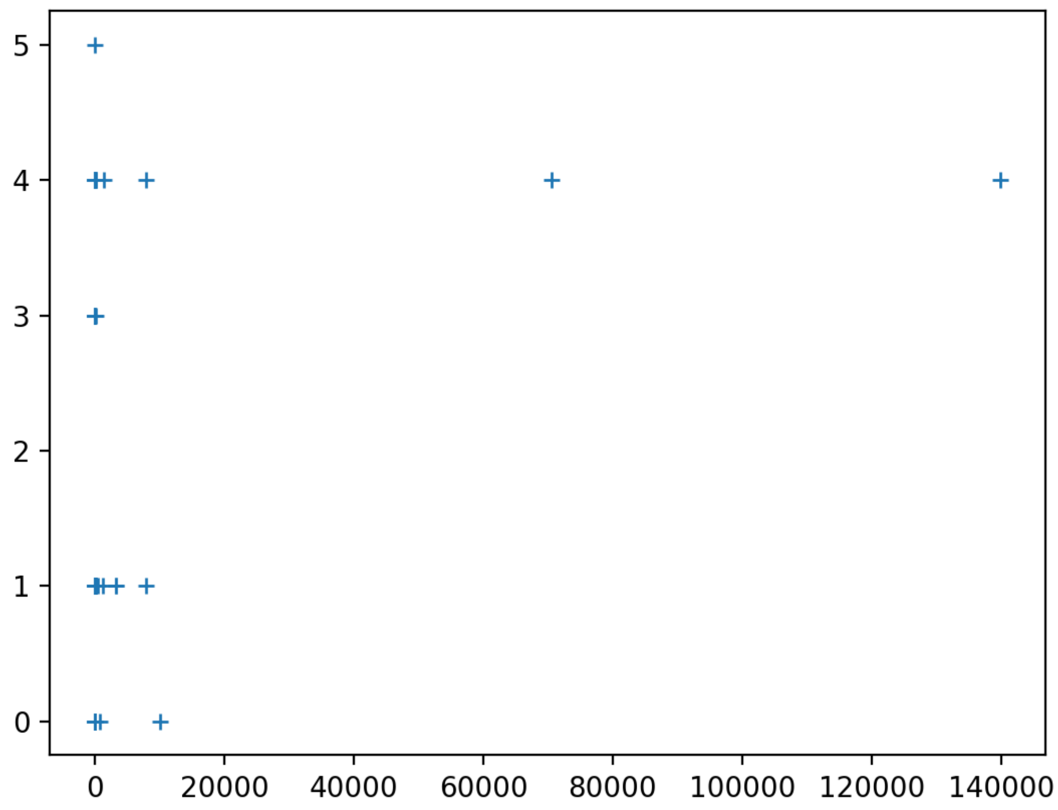
Code to compute the upper bound:

```
def compute_bound(theta_opt, theta_0, gamma):
    a = numpy.dot(theta_opt, theta_0)
    first_bound = -1.0 * a / gamma
    second_bound = (1 - 2*a*gamma +
            math.sqrt( math.pow(2*a*gamma-1, 2) - 4*gamma*gamma * (a*a - 1) )
        ) / (2*gamma*gamma)
    return max(first_bound, second_bound)

```

Code that run perceptron 10,000 times with random starting points, and plot the results of the first 100 runs by comparing the number of iterations required and corresponding theoretical bound:

```
    # part d
    print "\nsubtask 3 part (d):"
    tot_iteration = 0.0
    tot_gamma = 0.0
    # variables to assist plotting
    counter = 0
    upper_bound_lst = []
    iteration_lst = []
    for i in range(10000):
        theta_0 = sample_unit_circle()
        result = perceptron(x, y, theta_0)
        tot_iteration += result[2]
        tot_gamma += result[1]
        if counter <= 100:
            counter += 1
            upper_bound = compute_bound(result[0], theta_0, result[1])
            upper_bound_lst.append(upper_bound)
            iteration_lst.append(result[2])

    avg_iteration = tot_iteration / 10000
    avg_gamma = tot_gamma / 10000
    print("average number of iteration is :" + str(avg_iteration))
    print("average gamma is :" + str(avg_gamma))

    plt.plot(upper_bound_lst, iteration_lst, '+')
    # plt.plot(numpy.array(upper_bound_lst) - numpy.array(iteration_lst), '+')
    plt.show()

```

Resulting plot:



The graph above have x-axis being the theoretical upper bound k, and the y-axis being the actual number of iterations required to classify the dataset.

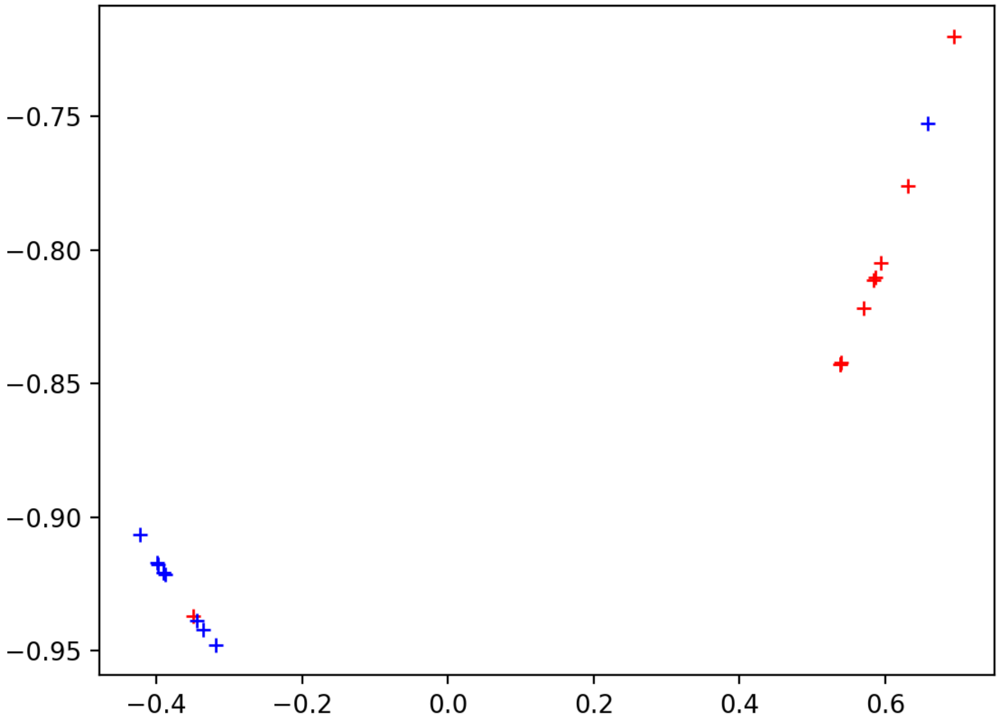
As we can see, the actual number of iterations have very small value although some of the theoretical upper bound can be large.

And the statistics after 10,000 runs:

```
subtask 3 part (d):
average number of iteration is :1.8079
average gamma is :0.24231992989
```

4. After changing the label of the first sample to -1 and label of the third sample to +1

Perform same plotting as part 1:



The new dataset is clearly not linearly separable, because if we draw two lines, each connecting the points with the same label, we will have those lines intersecting each other, meaning the convex hull for the two sets has intersection. Therefore, the two sets are not linearly separable.

After performing the same code as part 2, we failed to get a feasible solution:

```
Terminated (singular KKT matrix).
is optimal status : unknown
optimal theta is : [ 0.27486026 -0.96148418]
corrsponding minimum gamma is : -0.90450548623
```

After rerun the perceptron algorithm, we will exceed the maximum number of iterations, indicating the standard perceptron algorithm also does not converge.

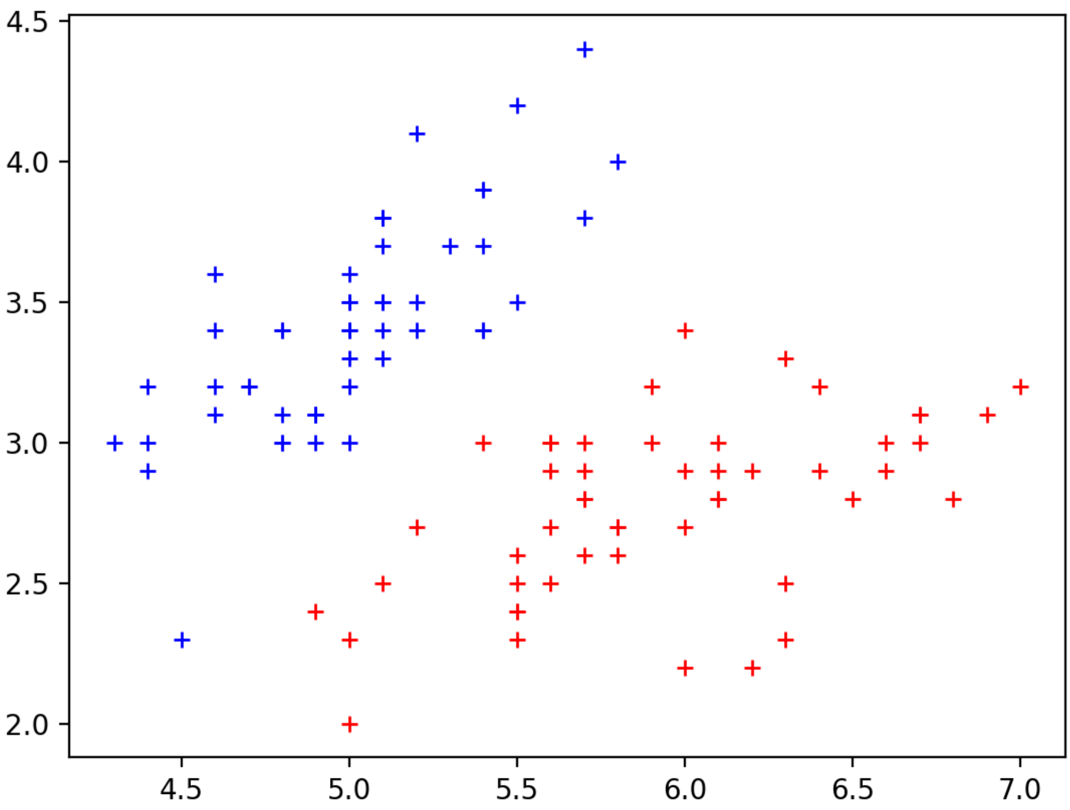
```
Traceback (most recent call last):
  File "p1.py", line 154, in <module>
    part4(x, y)
  File "p1.py", line 140, in part4
    perceptron(x, y, numpy.zeros(len(x[0])))
  File "p1.py", line 30, in perceptron
    raise Exception('Maximum number of iterations exceed!')
Exception: Maximum number of iterations exceed!
```

Problem 2: SVM

Similar to Problem 1, we will read the input file and preprocess the data before performing any tasks:

```
# main
reader = csv.reader(open("iris1.csv", "rb"), delimiter=",")
data = numpy.array(list(reader))
y = data[:, len(data[0]) - 1]
y = y.astype(int)
x = numpy.delete(data, len(data[0])-1, axis=1)
x = x.astype(float)
```

1.
   A. We can use the same function as Problem 1 part 1 to generate the following plot



Clearly, this dataset is linearly separable.

B. Solve primal with offset

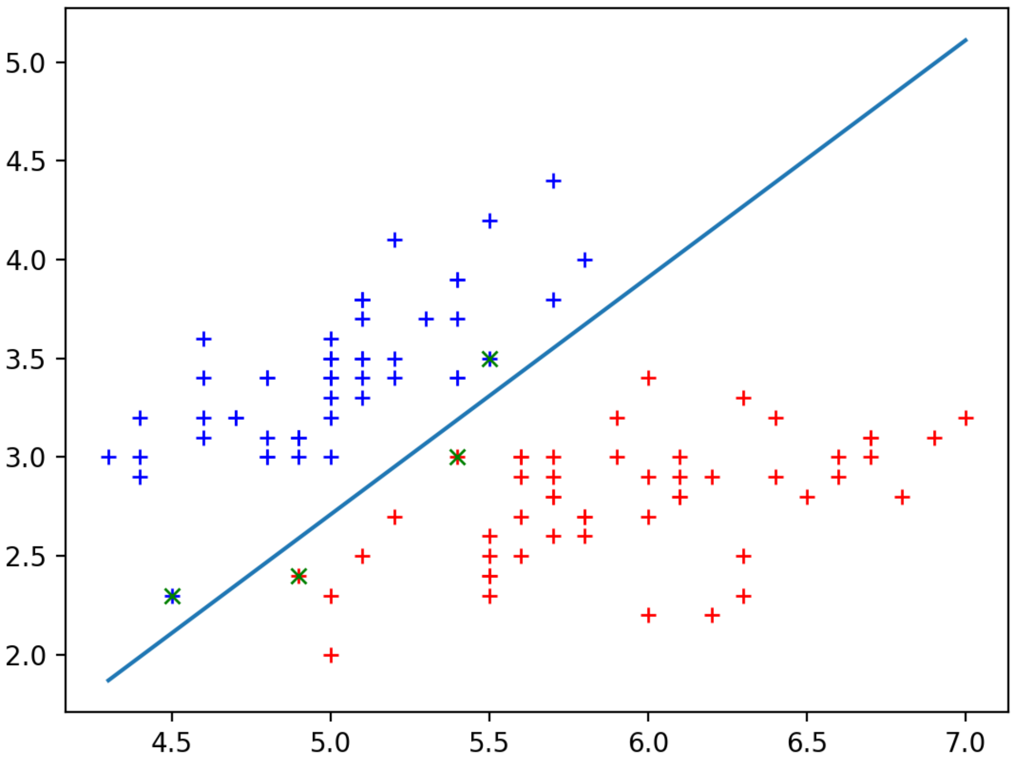
Using quadratic programming solver, but with a different set of matrices, we can obtain the desired result:

```
def part1_bcde(x, y):
    # QP to solve primal form for SVM -- convert to cxvopt matrices
    P = numpy.eye(3)
    P[2][2] = 0
    P = matrix(P, tc='d')
    q = matrix(numpy.zeros(3), tc='d')
    G = []
    for i in range(len(y)):
        G.append([-1 * y[i] * x[i, 0], -1 * y[i] * x[i, 1], -1 * y[i]])
    G = matrix(numpy.array(G), tc='d')
    h = matrix(-1 * numpy.ones(len(y)), tc='d')
    sol = solvers.qp(P, q, G, h)
```

And the output:

```
Optimal solution found.
is optimal status : optimal
optimal theta_0 is : -17.3157894733
optimal theta is : [ 6.31578947 -5.26315789]
optimal objective function value is : 33.7950138486
```

C. Plot with the solid line being the classification boundary, and points marked by * being the support vectors :



Code to generate above plot where prep_graph(x, y) function is used to generate graph for part A

```python
def prep_graph(x, y):
    x1 = []
    x2 = []
    for i in range(len(x)):
        if y[i] == 1:
            x1.append(x[i, :])
        else:
            x2.append(x[i, :])
    x1 = numpy.array(x1)
    x2 = numpy.array(x2)
    plt.plot(x1[:,0], x1[:,1], 'r+')
    plt.plot(x2[:,0], x2[:,1], 'b+')
```

```python
# part c
prep_graph(x, y)    # original points
# find line
min_x1 = min(x[:, 0])
min_x2 = (0 - theta[0] * min_x1 - theta_0) / float(theta[1])
max_x1 = max(x[:, 0])
max_x2 = (0 - theta[0] * max_x1 - theta_0) / float(theta[1])
plt.plot(numpy.array([min_x1, max_x1]), numpy.array([min_x2, max_x2]))
# compute support vectors
dist_matrix = y * (numpy.dot(x, theta) + theta_0)
min_dist = min(dist_matrix)
support_vectors_x1 = []
support_vectors_x2 = []
for i in range(len(y)):
    if (dist_matrix[i] <= min_dist + 1.0 / 1000000):    # to prevent floating number error
        support_vectors_x1.append(x[i, 0])
        support_vectors_x2.append(x[i, 1])
plt.plot(support_vectors_x1, support_vectors_x2, 'gx')
plt.show()
```

D. Solve dual

Since the quadratic programming solver already solve dual for us, we can straight away apply the result:

```python
# part d
indices = []
values = []
for i in range(len(sol['z'])):
    if sol['z'][i] > 1.0/1000000:
        indices.append(i)
        values.append(sol['z'][i])
print "number of non-zero entries : " + str(len(indices))
print "indices of non-zero entries (starting from 0 index) : " + str(indices)
print "and corresponding values : " + str(values)
print "dual optimal objective function value is : " + str(sol['dual objective'])
```

And the results are:

```
number of non-zero entries : 4
indices of non-zero entries (starting from 0 index) : [36, 41, 57, 84]
and corresponding values : [20.855103478792383, 12.939910347561224, 6.48923
8982032194, 27.30577486567907]
dual optimal objective function value is : 33.795013812
```

We observe that the dual optimal objective value is the same as the primal one.

Meanwhile, the primal will take a shorter time to solve since the number of points is significantly larger than number of dimensions, resulting a smaller dimension matrix to compute for the primal case when compared with the dual case.

E. Code to compute the two required sum:

```python
# part e
first_sum = [0, 0]
for i in range(len(indices)):
    first_sum += values[i] * y[indices[i]] * x[indices[i], :]
print "the first sum is : " + str(first_sum)

j = numpy.random.randint(len(indices))
second_sum = 0
for i in range(len(y)):
    second_sum += sol['z'][i] * y[i] * numpy.dot(x[i, :], numpy.transpose(x[indices[j], :]))
second_sum = y[indices[j]] - second_sum
print "the second sum is : " + str(second_sum)
```

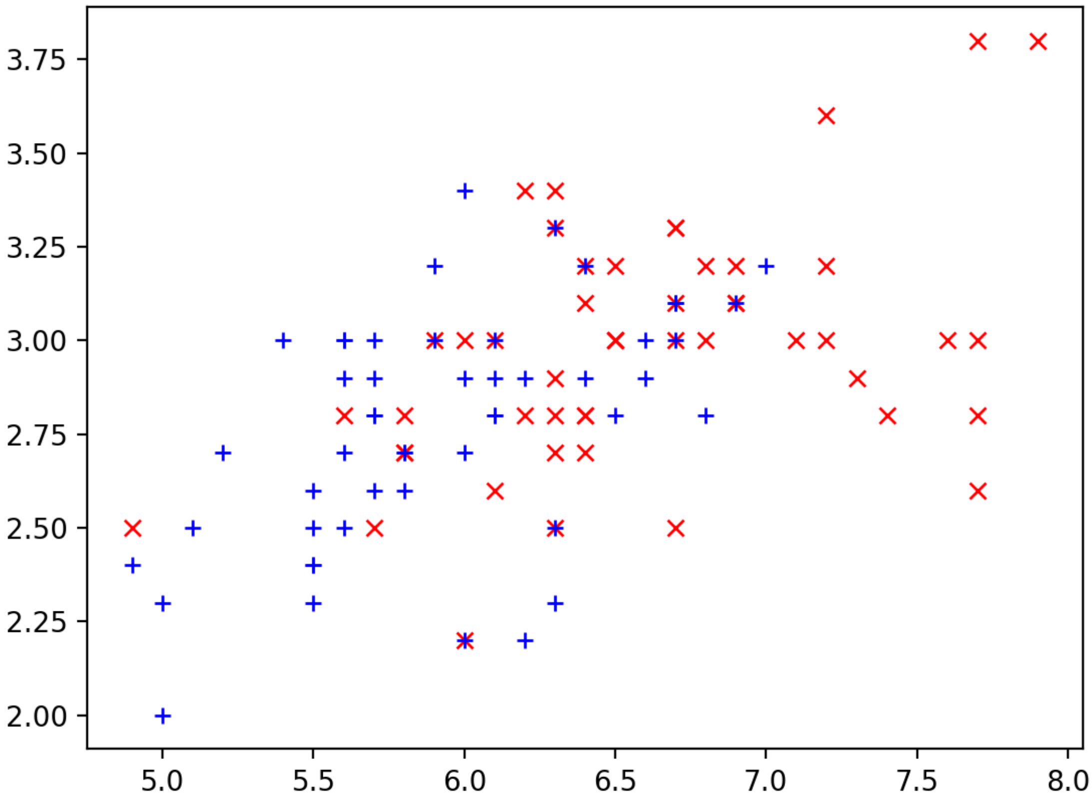
And the result:

```
the first sum is : [ 6.31578959 -5.26315782]
the second sum is : -17.3157894879
```

We observe that the first sum $\sum_{i=1}^{n} \alpha_i y_i x_i$ has the same value as the optimal theta, and the second sum $y_j - \sum_{i=1}^{n} \alpha_i y_i x_i^T x_j.$ has the same value as the optimal theta_0.

2.

A. Plot first two dimensions of the dataset



This set of points are clearly not linearly separable because if we draw the convex hull for the two sets, there will be intersections.

Code for plotting the above graph is similar as before:

```python
def part_2(x, y):
    x1 = []
    x2 = []
    for i in range(len(x)):
        if y[i] == 1:
            x1.append(x[i, :])
        else:
            x2.append(x[i, :])
    x1 = numpy.array(x1)
    x2 = numpy.array(x2)
    plt.plot(x1[:,0], x1[:,1], 'rx')
    plt.plot(x2[:,0], x2[:,1], 'b+')
    plt.show()
```