1 Background

1.1 Line Fitting

In the previous session we discussed the gradient method to fit a line, we also saw how using a Newton-based algorithm is much more efficient. Here is an implementation of the problem we discussed last session using num numpy's version of BFGS

```
T = np.array([14.2, 16.4, 11.9, 15.2, 18.5, 22.1, 19.4, 25.1, 23.4, 18.1, 22.6, 17.2])
S = np.array([215, 325, 185, 332, 406, 522, 412, 614, 544, 421, 445, 408])
def cost(x):

    a, b, c = x[0:3]
    f = (a * T + b * S + c)**2 / (a**2 + b**2)

    return f.sum()

def gradcost(x):

    a, b, c = x[0:3]

    dfda = 2 * T * (a * T + b * S + c) / (a**2 + b**2) \
    - 2 * a * (a * T + b * S + c)**2 / (a**2 + b**2)**2

    dfdb = 2 * S * (a * T + b * S + c) / (a**2 + b**2) \
    - 2 * b * (a * T + b * S + c)**2 / (a**2 + b**2)**2

    dfdc = 2 * (a * T + b * S + c) / (a**2 + b**2)

    return np.array([dfda.sum(), dfdb.sum(), dfdc.sum()])

x0 = np.array([3, -1, 0.])
res = minimize(cost, x0, method='BFGS', jac=gradcost, options={'disp': True})
print res.x
```

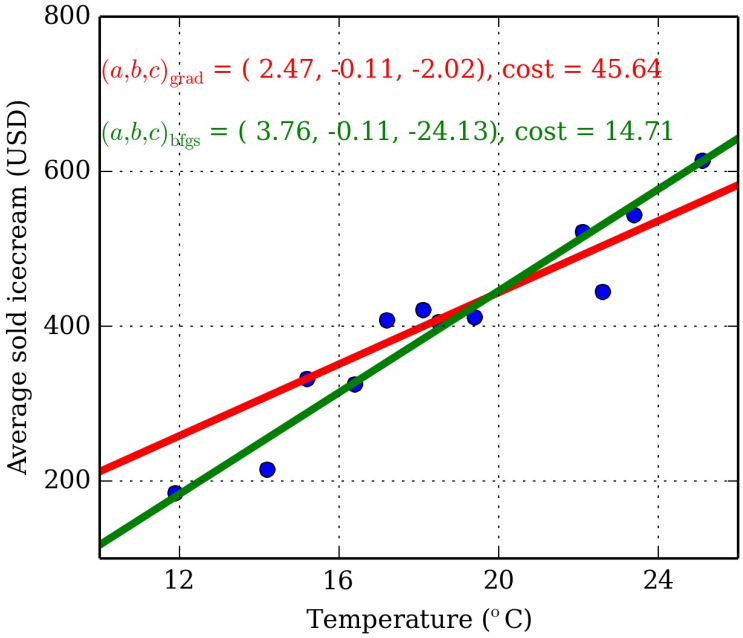
Which yields the output

```
Optimization terminated successfully.
         Current function value: 14.707936
         Iterations: 10
         Function evaluations: 14
         Gradient evaluations: 14
[  3.76326673  -0.11468162 -24.12921274]
```

There are two important things to realize here,

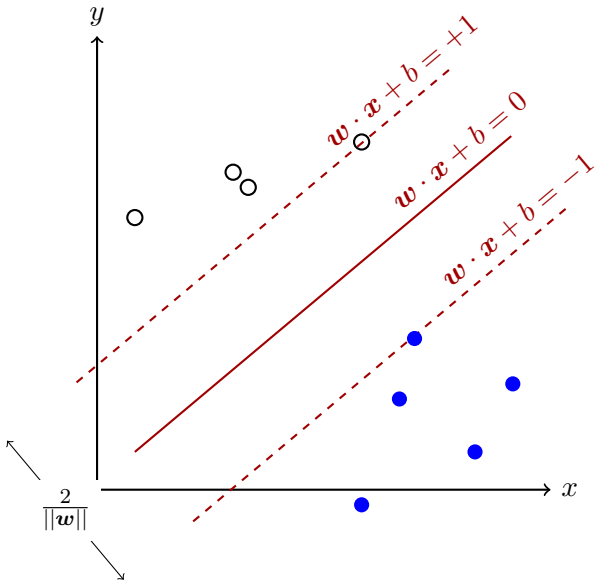
1. The final value of the cost function is 14.71 for the BFGS method, and it is 45.63 for the gradient descent method. That means that the second solution is preferred over the first one
2. BFGS reaches this value in 14 iterations, while the gradient method takes 500000!

The figure shows the solutions obtained using both methods,



2 Support Vector Machine

In this session we will use a technique known as support vector machine designed to classify features. Imagine we have a dataset of the form $D = \{(x_i, y_i), c_i\}_{i=0}^{n-1}$, where the values c are categoric variables: True/False, 1/0, +1/-1, Yes/No, ... The idea behind SVM is to predict to which class c a given point (x, y) belongs to based on the information collected in D. For simplicity we will assume that there exists a line $\boldsymbol{w} \cdot x + b = 0$ that divides the data into two distinct sets



It is possible to show that finding the best fitting line that separates the points D in two distinct groups is equivalent to finding the maximum of the function $1/||\boldsymbol{w}||$ subject to the conditions $\boldsymbol{w} \cdot \boldsymbol{x} + b \geq 1$ for

points of the first class (c = True, 1, +1, Yes, ...) and $\boldsymbol{w} \cdot \boldsymbol{x} + b \leq -1$ for points of the second class (c = False, 0, -1, No, ...). For the sake of convenience let us use the label $c = \{-1, 1\}$ in which case the problem is the equivalent to

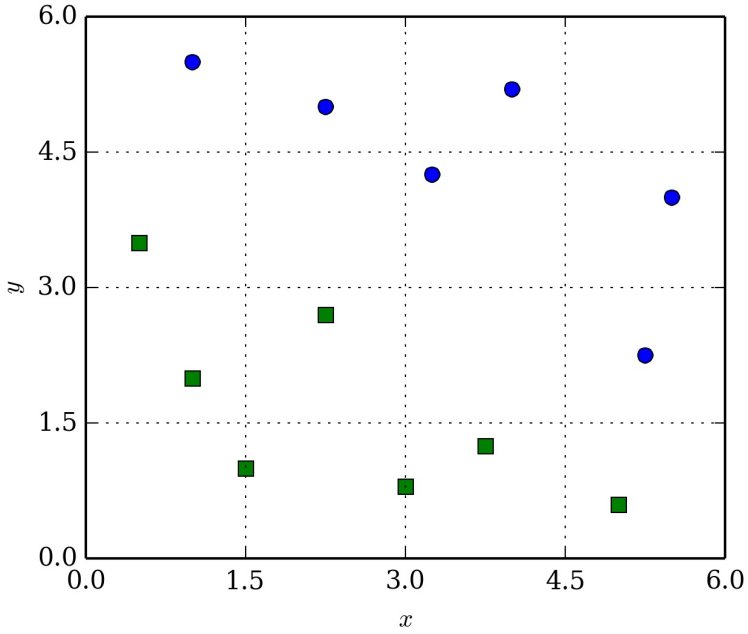
$$\begin{array}{ll} \text{minimize} & ||\boldsymbol{w}||^2 \\ \text{subject to} & c_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 \qquad \text{for} \quad i = 0, \cdots, n-1 \end{array} \tag{1}$$

3 Problems

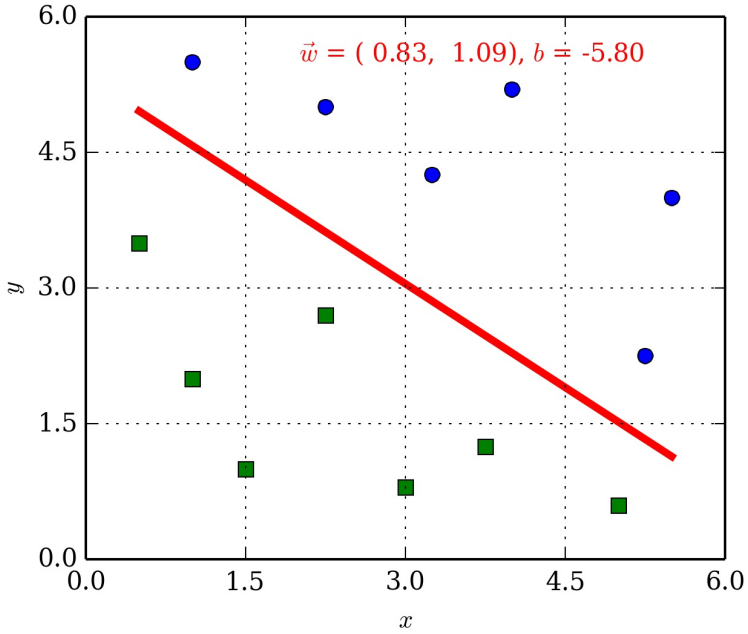
(P1) Consider the set

x	y	c
1.00	5.50	+1
2.25	5.00	+1
3.25	4.25	+1
4.00	5.20	+1
5.25	2.25	+1
5.50	4.00	+1
0.50	3.50	-1
1.00	2.00	-1
1.50	1.00	-1
2.25	2.70	-1
3.00	0.80	-1
3.75	1.25	-1
5.00	0.60	-1

Make a plot of x vs y and color according to the classes



(P2) Use the python implementation of the SLSQP algorithm to find the values of $\boldsymbol{w}$ and b that best fit the data



4 Solutions

(P1)
```
x = np.array([1.0, 2.25, 3.25, 4.0, 5.25, 5.5, 0.5, 1., 1.5, 2.25, 3.0, 3.75, 5.0])
y = np.array([5.5, 5., 4.25, 5.2, 2.25, 4.0, 3.5, 2.0, 1.0, 2.7, 0.8, 1.25, 0.6])
c = np.array([1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1])
plt.plot(x[c == 1], y[c == 1], 'ro')
plt.plot(x[c == -1], y[c == -1], 'gs')
```

(P2)
```
z = np.array(zip(x, y))
n = len(z[0])

def func(x):

    w = x[1 : ]
    b = x[0]
    return (w**2).sum()

def gradfunc(x):

    w = x[1 : ]
    b = x[0]

    d = np.zeros_like(x)
    d[0] = 0
    d[1 : ] = 2 * w

    return d

x0 = np.ones(n + 1)
cons = []
for i in range(len(z)):
    cons.append({'type': 'ineq',
                 'fun': lambda x, i = i: c[i] * (np.dot(x[1 : ], z[i]) + x[0]) - 1,
                 'jac': lambda x, i = i: np.append(c[i], c[i] * z[i])})

res = minimize(func, x0, jac=gradfunc, method='SLSQP',
      constraints=cons, options={'disp': True})
print res.x
```