# Exercises03-Solutions

February 21, 2015

```
In [1]: import os
        import sys
        import glob
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        %matplotlib inline
        plt.style.use('ggplot')
        %precision 4

Out[1]: u'%.4f'
```

**Exercise 1 (10 pts)**. Avodiing catastrophic cancellation.
The tail of the standard logistic distributon is given by $1 - F(t) = 1 - (1 + e^{-t})^{-1}$.

- Define a function `f1` to calculate the tail probability of the logistic distribution using the formula given above
- Use `sympy` to find the exact value of the tail distribution (using the same symbolic formula) to 20 decimal digits
- Calculate the *relative error* of `f1` when $t = 25$ (The relative error is given by `abs(exact - approximate)/exact`)
- Rewrite the expression for the tail of the logistic distribution using simple algebra so that there is no risk of cancellation, and write a function `f2` using this formula. Calculate the *relative error* of `f2` when $t = 25$.
- How much more accurate is `f2` compared with `f1` in terms of the relative error?

```
In [2]: # Your code here

        def f1(t):
            """Calculates tail probabilty of the logistic distribution."""
            return 1 - 1.0/(1 + np.exp(-t))

        def fsymb(t, n=20):
            """Exact value to n decimal digits using symbolic algebra."""
            from sympy import exp
            return (1 - 1/(1 + exp(-t))).evalf(n=n)

        def f2(t):
            """Calculates tail probabilty of the logistic distribution - no cancellation."""
            return 1/(1 + np.exp(t))

        r1 = abs(fsymb(25) - f1(25))/fsymb(25)
        r2 = abs(fsymb(25) - f2(25))/fsymb(25)
        print "Relative error of f1:\t%.16f" % r1
        print "Relative error of f2\t%.16f" % r2
        print "f2 improvieemnt over f1\t%g" % (r1/r2)
```

```
Relative error of f1:        0.0000041759147666
Relative error of f2        0.0000000000000001
f2 improvieemnt over f1        3.66247e+10
```

**Exercise 2 (10 pts)**. Ill-conditioned linear problems.

You are given a $n \times p$ design matrix $X$ and a $p$-vector of observations $y$ and asked to find the coefficients $\beta$ that solve the linear equations $X\beta = y$.

```
X = np.load('x.npy')
y = np.load('y.npy')
```

The solution $\beta$ can also be loaded as

```
beta = np.load('b.npy')
```

- Write a formulat that could solve the system of linear equations in terms of $X$ and $y$ Write a function `f1` that takes arguments $X$ and $y$ and returns $\beta$ using this.
- How could you code this formula using `np.linalg.solve` tht does not require inverting a matrix? Write a function `f2` that takes arguments $X$ and $y$ and returns $\beta$ using this.
- Note that carefully designed algorithms *can* solve this ill-conditioned problem, which is why you should always use library functions for linear algebra rather than write your own.

  ```
  np.linalg.lstsq(x, y)[0]
  ```

- What happens if you try to solve for $\beta$ using `f1` or `f2`? Remove the column of $X$ that is making the matrix singluar and find the $p-1$ vector $b$ using `f2`.
- Note that the solution differs from that given by `np.linalg.lstsq`. This arises because the relevant condition number for `f2` is actually for the matrix $X^T X$ while the condition number of `lstsq` is for the matrix $X$. Why is the condition so high even after removing the column that makes the matrix singular?

```
In [3]: X = np.load('x.npy')
        y = np.load('y.npy')
        beta = np.load('b.npy')

        # Your code here

        def f1(X, y):
            """Direct translation of normal equations to code."""
            return np.dot(np.linalg.inv(np.dot(X.T, X)), np.dot(X.T, y))

        def f2(X, y):
            """Solving normal equations wihtout matrix inversion."""
            return np.solve(np.dot(x.T, x), np.dot(x.T, np.dot(x, b)))

        %precision 2
        print "X = "
        print X
        # counting from 0 (so column 5 is the last column)
        # we can see that column 5 is a multiple of column 3
```

```
        # so one approach is to simply remove this (dependent) column

        print "True solution\t\t", beta
        print "Library function\t", np.linalg.lstsq(X, y)[0]
        print "Using f2\t\t", f1(X[:, :5], y)

        # Condition number is still high because column 1 is on a much
        # larger scale than all the other columns
```

```
X =
[[  5.00e+00   4.82e+14   9.00e+00   5.00e+00   0.00e+00   5.00e+01]
 [  1.00e+00   4.21e+14   6.00e+00   9.00e+00   2.00e+00   9.00e+01]
 [  5.00e+00   1.20e+14   4.00e+00   2.00e+00   4.00e+00   2.00e+01]
 [  7.00e+00   5.42e+14   1.00e+00   7.00e+00   0.00e+00   7.00e+01]
 [  9.00e+00   5.42e+14   7.00e+00   6.00e+00   9.00e+00   6.00e+01]
 [  0.00e+00   6.02e+13   8.00e+00   8.00e+00   3.00e+00   8.00e+01]
 [  8.00e+00   4.21e+14   3.00e+00   6.00e+00   5.00e+00   6.00e+01]
 [  9.00e+00   1.81e+14   4.00e+00   8.00e+00   1.00e+00   8.00e+01]
 [  0.00e+00   1.81e+14   9.00e+00   2.00e+00   0.00e+00   2.00e+01]
 [  9.00e+00   1.20e+14   7.00e+00   7.00e+00   9.00e+00   7.00e+01]]
True solution                [ 0.47  0.1   0.9   0.12  0.52  0.08]
Library function        [ 0.47  0.1   0.9   0.01  0.52  0.09]
Using f2                [ 0.47  0.1   0.9   0.95  0.53]
```

**Exercise 3 (10 pts)**. Importance of using efficient algorihtms.

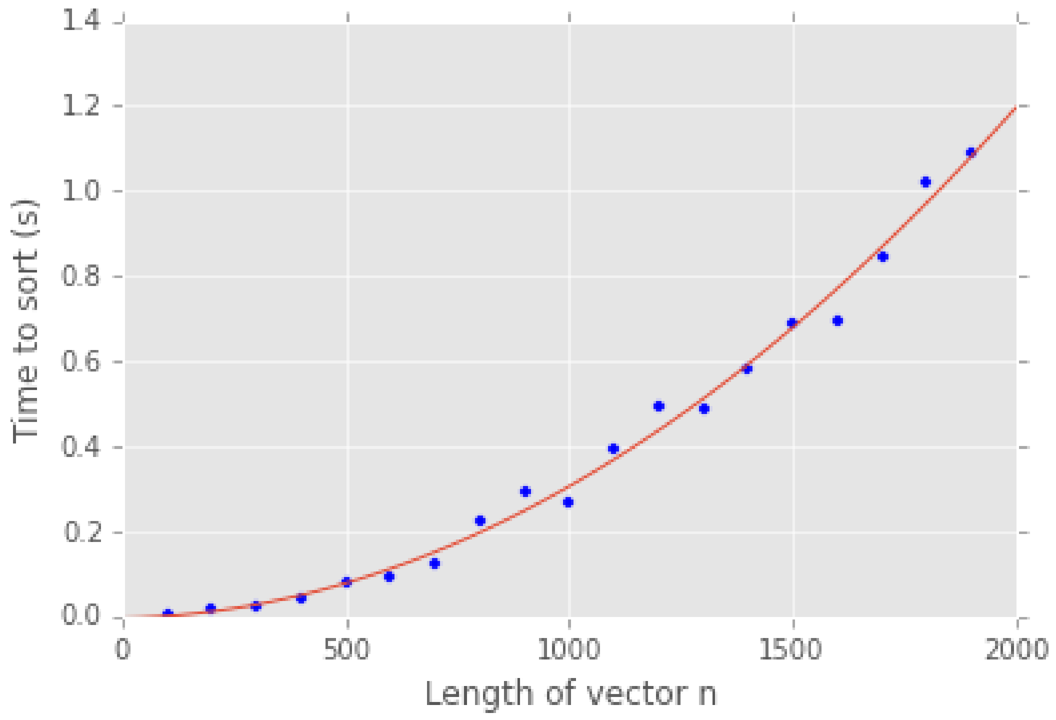
- Implement bubble sort
- Calculate its big $\mathcal{O}$ algorithmic complexity
- Time the performance of bubble sort on random uniform deivate vectors of sizes `range(100, 2000, 100)` using `time.time()` from the standard library
- Use `scipy.optimize.curve_fit` to fit an appropirate function to the collection of (size, execution time) data points. Extrapolate how long it would take to sort a ranodm vector of size 1,000,000. Now time how long it takes for the system sort to sort a ranodm vector of size 1,000,000.
- Plot the fits together with the data points uisng `matplotlib.pyplot` functions.

```
In [24]: # Your code here

         def bubble(xs):
             """Bubble sort."""
             for i in range(len(xs)):
                 for j in range(i, len(xs)):
                     if xs[i] > xs[j]:
                         xs[i], xs[j] = xs[j], xs[i]
             return xs

         import time

         ns = range(100, 2000, 100)
         bubble_times = []

         for n in ns:
             xs = np.random.random(n)
             start = time.time()
             bubble(xs)
             bubble_times.append(time.time() - start)
```

```python
def func(x, a, b, c):
    """Quadratic function for O(n^2) complexity."""
    return a*x**2 + b*x + c

from scipy.optimize import curve_fit

plt.scatter(ns, bubble_times, c='blue')
popt, pcov = curve_fit(func, ns, bubble_times)
a, b, c = popt

print "Predicted time to sort 1,000,000 items = %.2f seconds" % func(1000000, a, b, c)

x = np.random.random(1000000)
start = time.time()
x.sort()
elapsed = time.time() - start
print "Time for system sort to sort 1,000,000 items = %.2f seconds" % elapsed

xp = np.linspace(0, 2000)
plt.plot(xp, func(xp, a, b, c))
plt.xlim([0, 2000]);
plt.ylim([0, plt.ylim()[1]])
plt.xlabel('Length of vector n')
plt.ylabel('Time to sort (s)');
```

```
Predicted time to sort 1,000,000 items = 290278.71 seconds
Time for system sort to sort 1,000,000 items = 0.14 seconds
```

```
Out[24]: <matplotlib.text.Text at 0x115cd9750>
```



**Exercise 4 (20 pts)**. One of the goals of the course it that you will be able to implement novel algorihtms from the literature.

- Implement the mean-shift algorithm in 1D as described here.
  - Use the following function signature

    `def mean_shift(xs, m, kernel, max_iters=100, tol=1e-6):`
  - xs is the data set, m is the starting location, and kernel is a kernel function
  - tol is the difference in $||x||$ across iterations
- Use the following kernels with bandwidth $h$ (a default value of 1.0 will work fine)
  - Flat - return 1 if $||x|| < h$ and 0 otherwise
  - Gaussian

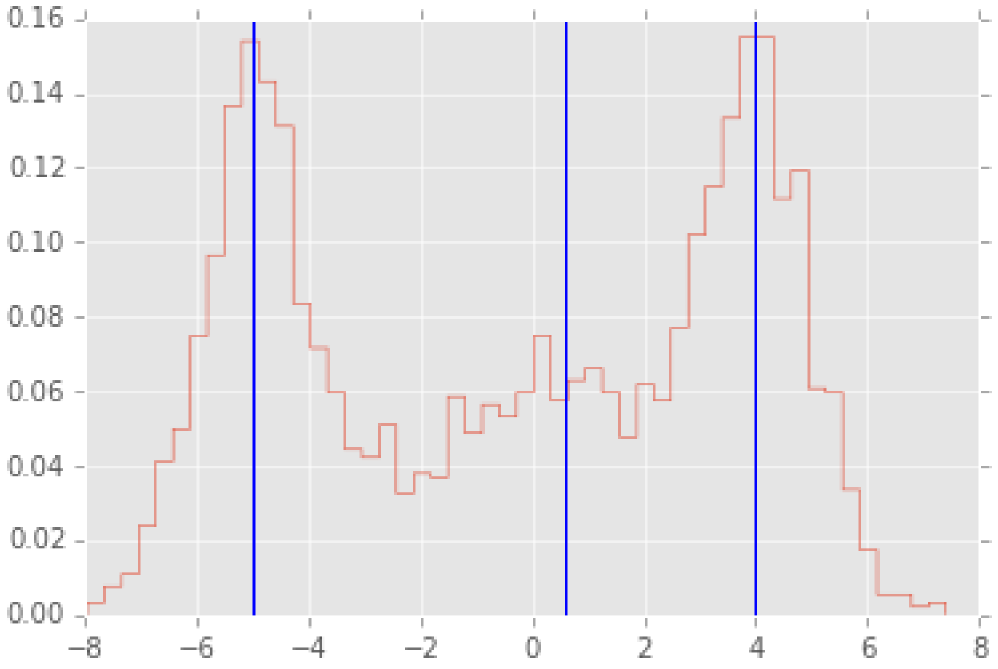
$$\frac{1}{\sqrt{2\pi h}} e^{\frac{-||x||^2}{h^2}}$$

  - Note that $||x||$ is the norm of the data point being evaluated minus the current value of $x$
- Use both kernels to find all 3 modes of the data set in `x1d.npy`
- Modify the algorihtm abd/or kernels so that it now works in an arbitrary number of dimensions.
- Uset both kernels to find all 3 modes of the data set in `x2d.npy`
- Plot the path of successive intermeidate solutions of the mean-shift algorithm starting from `x0 = (-4, 10)` until it converges onto a mode in the 2D data for each kernel. Superimposet the path on top of a contour plot of the data density.

```
In [101]: # Your code here

          def gaussian_kernel(xs, x, h=1.0):
              """Gaussian kernel for a shifting window centerd at x."""

              X = xs - x
              try:
                  d = xs.shape[1]
              except:
                  d = 1
              k = np.array([(2*np.pi*h**d)**-0.5*np.exp(-(np.dot(_.T, _)/h)**2) for _ in X])
              if d != 1:
                  k = k[:, np.newaxis]
              return k

          def flat_kernel(xs, x, h=1.0):
              """Flat kenrel for a shifting window centerd at x."""

              X = xs - x
              try:
                  d = xs.shape[1]
              except:
                  d = 1
              k = np.array([1 if np.dot(_.T, _) < h else 0 for _ in X])
              if d != 1:
                  k = k[:, np.newaxis]
              return k
          def mean_shift(xs, x, kernel, max_iters=100, tol=1e-6, trace=False):
              """Finds the local mode using mean shift algorithm."""

              record = []

              for i in range(max_iters):
                  if trace:
                      record.append(x)
                  m = (kernel(xs, x)*xs).sum(axis=0)/kernel(xs, x).sum(axis=0) - x
                  if np.sum(m**2) < tol:
                      break
                  x += m
              return i, x, np.array(record)
```

```
In [110]: x1 = np.load('x1d.npy')

          # choose kernel to evaluate
          kernel = flat_kernel
          # kernel = gaussian_kernel

          i1, m1, path = mean_shift(x1, 1, kernel)
          print i1, m1

          i2, m2, path = mean_shift(x1, -7, kernel)
          print i2, m2

          i3, m3, path = mean_shift(x1, 7 ,kernel)
          print i3, m3

          xp = np.linspace(0, 1.0, 100)
          plt.hist(x1, 50, histtype='step', normed=True);
          plt.axvline(m1, c='blue')
          plt.axvline(m2, c='blue')
          plt.axvline(m3, c='blue');
```

```
15 0.575467435455
16 -4.99502467965
16 3.97936417657
```

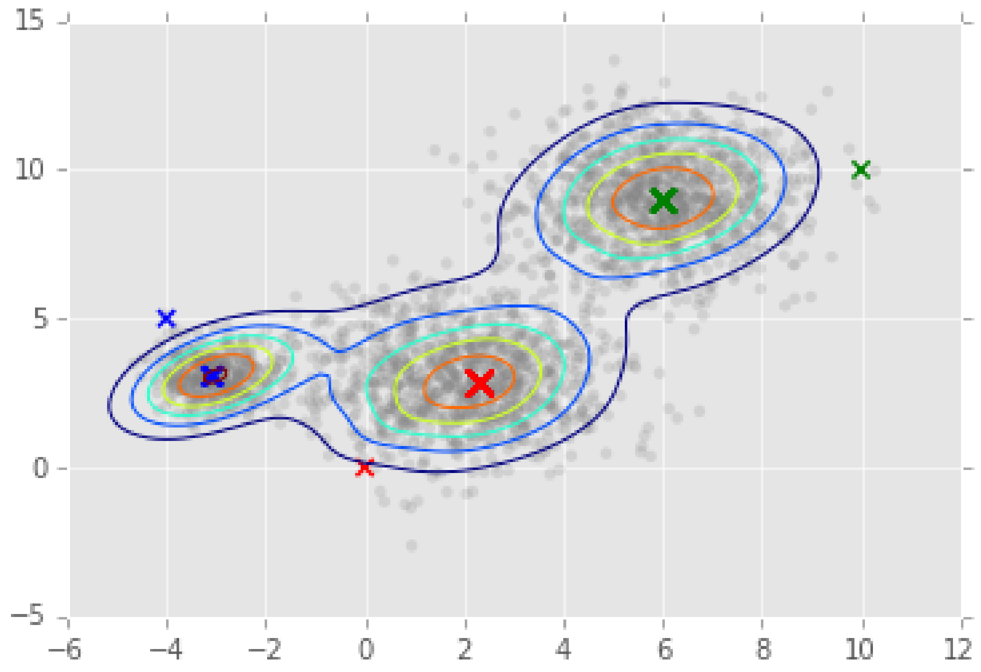


```
In [123]: x2 = np.load('x2d.npy')

          # choose kernel to evaluate
          # kernel = flat_kernel
          kernel = gaussian_kernel

          i1, m1, path1 = mean_shift(x2, [0,0], kernel, trace=True)
          print i1, m1

          i2, m2, path2 = mean_shift(x2, [-4,5], kernel, trace=True)
          print i2, m2

          i3, m3, path3 = mean_shift(x2, [10,10] ,kernel, trace=True)
          print i3, m3
```

```
59 [ 2.318177222  2.825508637]
12 [-3.0704798654  3.0567861169]
42 [ 6.0225434115  8.9506699557]
```

```
In [155]: import scipy.stats as stats

          # size of marekr at starting position
          base = 40

          # set up for estimating density using gaussian_kde
          xmin, xmax = -6, 12
          ymin,ymax = -5, 15
          X, Y = np.mgrid[xmin:xmax:100j, ymin:ymax:100j]
          positions = np.vstack([X.ravel(), Y.ravel()])
```

```
kde = stats.gaussian_kde(x2.T)
Z = np.reshape(kde(positions).T, X.shape)

plt.contour(X, Y, Z)
# plot data in background
plt.scatter(x2[:, 0], x2[:, 1], c='grey', alpha=0.2, edgecolors='none')

# path from [0,0]
plt.scatter(path1[:, 0], path1[:, 1], s=np.arange(base, base+len(path1)),
            c='none', edgecolors='red', marker='x', linewidth=1.5)

# path from [-4,5]
plt.scatter(path2[:, 0], path2[:, 1], s=np.arange(base, base+len(path2)),
            c='none', edgecolors='blue', marker='x', linewidth=1.5)

# path from [10,10]
plt.scatter(path3[:, 0], path3[:, 1], s=np.arange(base, base+len(path3)),
            c='none', edgecolors='green',marker='x', linewidth=1.5)
plt.axis([xmin, xmax, ymin, ymax]);
```



```
In [ ]:
```