Simple linear regression lab

03/10/2019

Aims

- Do a simple linear regression on the Olympic 100~m data in Python
- Compare least square solution with gradient descent solution

Tasks

- Download the data (‘olympic100m.txt’) from the Moodle page
- Plot Olympic year against winning time
- Plot the loss function in 1D and 2D
- Test the provided gradient descent code with your own code for loss and gradient
- Using the expressions derived provided in the supplement slides to compute w_0 and w_1.
- Create a new plot that includes the data and the function defined by w_0 and w_1
- Make a prediction at 2012

Step 1: We start by loading the Olympic 100m men's data

In [1]:

```
import numpy as np

data = np.loadtxt('olympic100m.txt', delimiter=',') # load olympic data
x =  # make x a column vector
t =  # make t a column vector
```

Setp 2: Plot the data

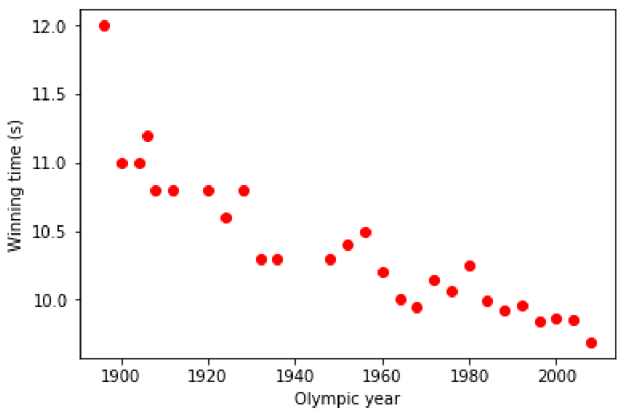
It's useful to start with a plot

In [3]:

```
%matplotlib inline
import pylab as plt

# plot x (x-axis) vs t (y-axis) with matplotlib's plot function: plt.plot
```

Out[3]:

```
Text(0,0.5,'Winning time (s)')
```



Step 3: Plot the averaged squared loss

Let's plot the loss function in 1D and 2D (w_0 and w_1)

Recall that the average squared loss was given by:

$$L = \frac{1}{N}\sum_{n=1}^{N}(t_n - w_0 - w_1x_n)^2$$

$L$ is a function of $w_0$ and $w_1$. All x_n and t_n are given.

Step 3.1: Generate candidates of $w_0$ and $w_1$ for plotting

In [7]:

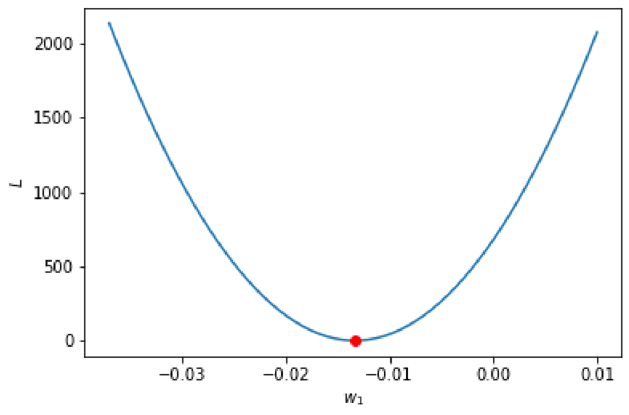
```
num_candidates =  # number of candidates, e.g. 100
w0_candidates =  # generate a numpy array of possible w0 values e.g. -10 to 82 with np.linspace
w1_candidates =  # generate a numpy array of possible w1 values e.g. -0.037 to 0.01
```

Step 3.2: Plot the average squared loss in 1D ($w_1$)

The average loss $L$ has both $w_0$ and $w_1$ as variables. In order to plot $L$, we have to fix one variable. For example, to plot $L$ vs $w_1$, we fix $w_0 = 36.4164559025$. See how the plot change when you change w0.

In [8]:

```
L =  # preallocating a vector for L e.g. np.zeros with num_candidates
for j in range(num_candidates): # For loop to evaluate L at every w1_candidates with
                                 # w0 = 36.4164559025
    L[j] =  # Wrtie code to compute the average squared loss.
            # The "np.mean" function could be helpful
            # Make sure only w1_candidates is changing while w0 is fixed at 36.4164559025

# plot w1_candidates (x-axis) vs L (y-axis) here
plt.plot( , , 'ro') # also plot the point w1 = -0.013330885711, L = 0.05
plt.xlabel('$w_1$')
plt.ylabel('$L$')
```

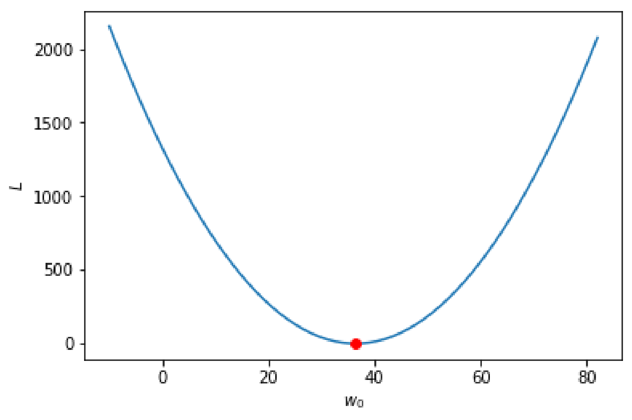


Step 3.3: Plot the average squared loss in 1D ($w_0$)

Let's plot the average squared loss in $w_0$, this time fix $w_1 = -0.013330885711$.

In [10]:

```
L =  # preallocating a vector for L e.g. np.zeros again with num_candidates
for j in range(num_candidates): # For loop to evaluate L at every w0_candidates
                                 # with w1 = -0.013330885711
    L[j] =   # Wrtie code to compute the average squared loss.
             # The "np.mean" function could be helpful
             # Make sure only w0_candidates is changing while w1 is fixed at -0.013330885711


# plot w0_candidates (x-axis) vs L (y-axis) here
plt.plot( , , 'ro') # also plot the point w0 = 36.4164559025, L = 0.05
plt.xlabel('$w_0$')
plt.ylabel('$L$')
```

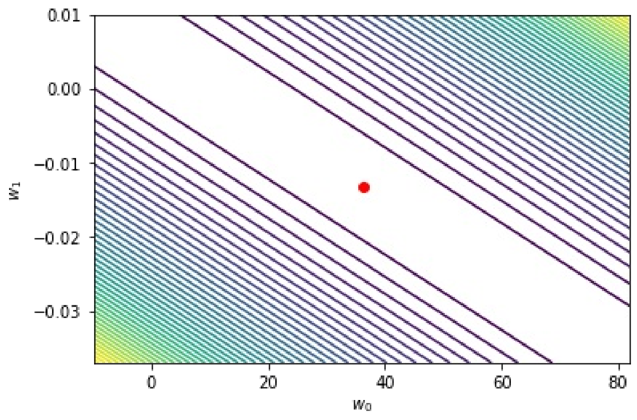


Step 3.4: Plot the loss function using the contour plot

Now, we let $w_0$ and $w_1$ change at the same time, and make the contour plot

In [11]:

```
L =  # Prelocate the loss. We are going to have num_candidates times num_candidates of them.
    #You can use np.zeros again, this time L has to be a num_candidates by num_candidates matrix.

# Two nested for loops
for i in range(num_candidates): # changing index of w0_candidates
    for j in range(num_candidates): # changing index of w1_candidates
        L[i,j] = # Wrtie code to compute the average squared loss.
                 # Can you compute the average squared loss without np.mean ?
                 # This time make sure both w0_candidates and w1_candidates are both changing

X, Y = np.meshgrid(w0_candidates, w1_candidates) # Make the x and y coordinates for contour plot
plt.contour(X, Y, L, 50) # change the number 50 to see what happens
plt.plot(, ,'ro') # plot the point w0= 36.4164559025, w1=-0.013330885711
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
```



Step 4: Gradient Descent

Repeat until convergence {

$$w_i = w_i - \alpha \frac{\partial L(w_0, w_1)}{\partial w_i}$$

}

It will be easier if we enclose the code to compute the gradient and the loss function in functions

Step 4.1 Define your own functions for the average squared loss and its gradient

In [14]:

```
def loss(x, t, w0, w1): # define the loss function
```

```
    L = # the average squared loss function
    return(L)

def gradient(x, t, w0, w1): # define the gradient function
    g0 = # partial derivative with respect to w0. This should be just one number.
    g1 = # partial derivative with respect to w1. This should be just one number.
    g = np.array([g0, g1])
    return(g)
```

Step 4.2: Run the following cell with your own functions for loss and gradient, it should only takes about a few seconds. If it gets too long, debug your functions

Our Olympic data turns out to be a quite challenging dataset for gradient descent. Try change value of `alpha` and `precision` in the following cell to see what happens.

In [15]:

```
alpha = 2e-7 # learning rate
precision = 1e-6 # convergence criterion
w_old = np.zeros(2) # Intial old guess
w_new = np.array([36, 0.1]) # Atural starting point
w_list, l_list = [w_old], [loss(x, t, w_new[0], w_new[1] )] # two lists to store w0, w1 and loss
rate_modifier = np.array([1e6, 1]) # modified rate due the difference in scale between w0 and w1

while sum(abs(w_new - w_old)) > precision: # check convergence
    w_old = w_new # update parameters
    g = gradient(x, t, w_old[0], w_old[1]) # compute gradient at w_old
    w_new = w_old - alpha*rate_modifier * g # update parameters
    w_list.append(w_new) # store w
    l_list.append(loss(x, t, w_new[0], w_new[1])) # store loss

print "Minimum loss occurs at: ", w_new
print "Minimum loss is:", float(loss(x, t, w_new[0], w_new[1]))
print "Gradient:", gradient(x, t, w_new[0], w_new[1])
print "Number of steps:", len(l_list)
```

```
Minimum loss occurs at:  [  3.64066449e+01  -1.33258618e-02]
Minimum loss is: 0.050307141413
Gradient: [ -4.99649022e-06   2.55854012e-03]
Number of steps: 83052
```

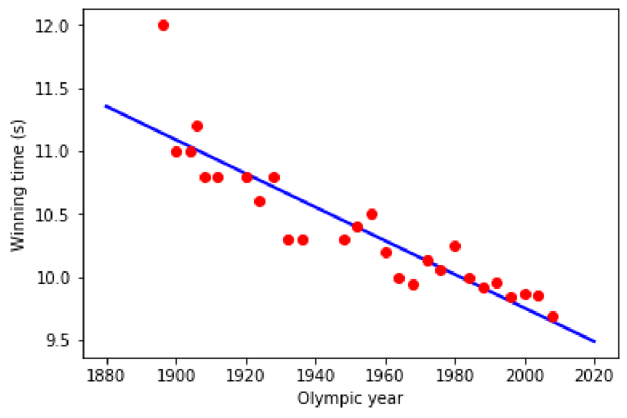
Step 4.3: Plot the resulting fitted line and data

In [28]:

```
x_test =  # generate new x to plot the fitted line. Note better not to use the original x !
f_test = # compute the corresponding prediction by the fitted model
# plot the fitted line
# plot data
plt.xlabel('Olympic year')
plt.ylabel('Winning time (s)')
```

Out[28]:

```
Text(0,0.5,'Winning time (s)')
```



Step 4.4: Inspect gradient descent

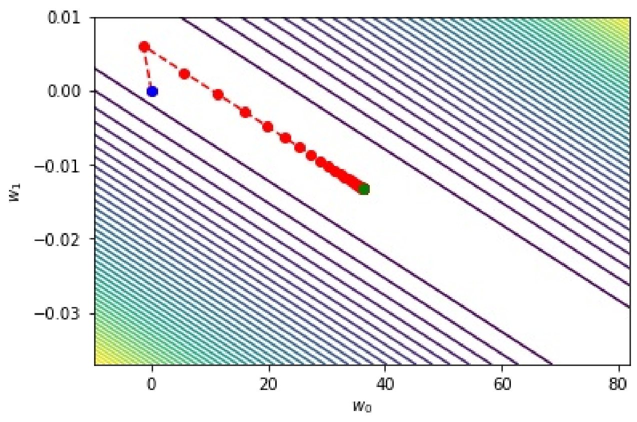
Use the `w_list` and `l_list` to visualise the trace of w_0 and w_1 during gradient descent

In [24]:

```
w_list = np.asanyarray(w_list) # convert the parameter list to a numpy array

plt.plot(w_list[::2000,0], w_list[::2000,1], "ro--") # only showing a subsample of the points
plt.plot(w_list[0,0], w_list[0,1], "bo") # plot the 1st point
plt.plot(w_list[-1,0], w_list[-1,1], "go") # plot the final point
plt.contour(X, Y, L, 50) # contour again
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
```

Out[24]:

```
Text(0,0.5,'$w_1$')
```



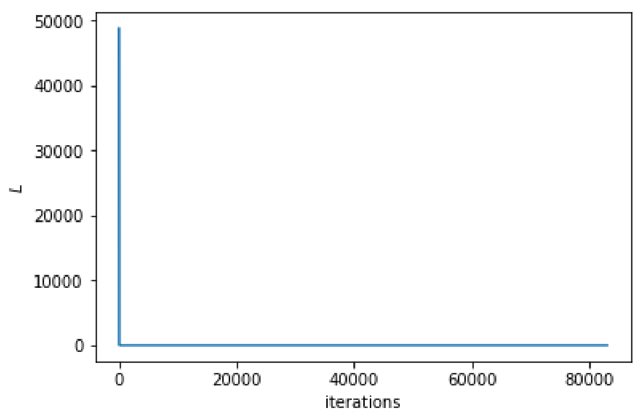
Step 4.5: Plot the learning curve: the trace of the average squared loss

In [20]:

```
plt.plot(l_list) # plot l_list
plt.xlabel('iterations')
plt.ylabel('$L$')
```

Out[20]:

```
Text(0,0.5,'$L$')
```



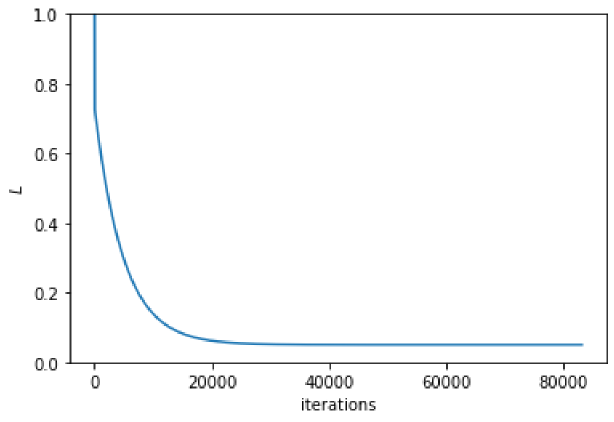
Step 4.6: Plot the learning curve: a zoom in look

In [25]:

```
plt.plot(l_list) # zoom in
plt.ylim((0,1))
plt.xlabel('iterations')
plt.ylabel('$L$')
```

Out[25]:

```
Text(0,0.5,'$L$')
```



Step 5: Least square solution

Solving

$$\frac{\partial L(w_0, w_1)}{\partial w_0} = 0, \quad \frac{\partial L(w_0, w_1)}{\partial w_1} = 0$$

, the average loss is minimised:

$$w_1 = \frac{\overline{xt} - \overline{x}\,\overline{t}}{\overline{xx} - \overline{x}^2}$$

and

$$w_0 = \overline{t} - w_1\overline{x}$$

where $\overline{z} = \frac{1}{N}\sum_{n=1}^{N} z_n$.

Step 5.1: Write code to compute $\overline{x}, \overline{t}, \overline{x^2}$ and $\overline{xt}$

In [22]:

```
xbar =
tbar =
xxbar =
xtbar =
print(xbar)
print(tbar)
print(xxbar)
print(xtbar)
```

```
1952.37037037
10.3896296296
3812975.55556
20268.0681481
```

Step 5.2: Write code to compute w_0 and w_1

In [23]:

```
w1 =
w0 =
print(w0)
print(w1)
```

```
36.4164559025
-0.013330885711
```

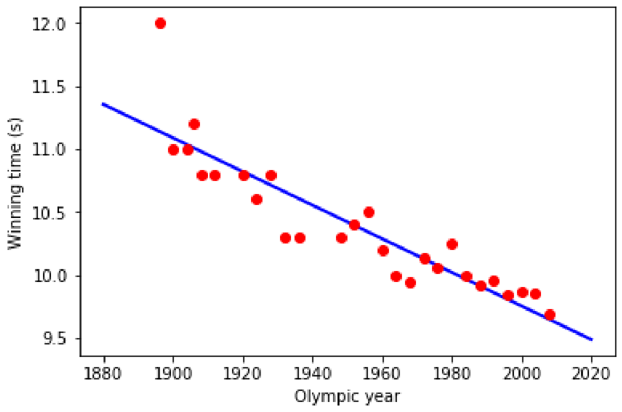
Step 5.3: Plotting the fitted line and data

In [29]:

```
x_test =  # generate new x to plot the fitted line. Note better not to use the original x !
f_test =  # compute the corresponding target variables
# plot the fitted line
# plot data
plt.xlabel('Olympic year')
plt.ylabel('Winning time (s)')
```

Out[29]:

```
Text(0,0.5,'Winning time (s)')
```



Step 6: Predictions

We can now compute the prediction at 2012:

In [32]:

```
win_2012_least_square = # make a prediction with the least square solution
win_2012_gradient_descent =  # make a prediction with the gradient descent solution
print(win_2012_least_square)
print(win_2012_gradient_descent)
```

```
9.59471385205
9.59501092799
```

Step 7: Optional task

Let's simulate some data using the following model

$$t_n = w_0 + w_1 x_n + w_2 x_n^2$$

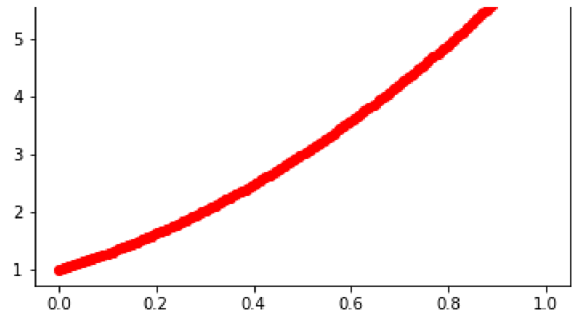
Step 7.1: Generate simulated data according to the model with $w_0 = 1$, $w_1 = 2.5$, **and** $w_2 = 3$

In [39]:

```
x = # generate x
t = # generate the corresponding t with above model
# plot x and t
```

Out[39]:

```
[<matplotlib.lines.Line2D at 0x10f250d90>]
```



Now, assume that we didn't know $w_0 = 1$, $w_1 = 2.5$, and $w_2 = 3$. We only have the data, x and t

Step 7.2: Write your own functions for the average squared loss and gradient for this model

In [42]:

```
def new_loss(x, t, w0, w1, w2): # define average squared loss function
    L =
    return(L)

def new_gradient(x, t, w0, w1, w2): # define the gradient

    g0 =  # partial derivative with respect to w0
    g1 =  # partial derivative with respect to w1
    g2 =  # partial derivative with respect to w2
    g = np.array([g0, g1, g2])
    return(g)
```

Step 7.3: Run the gradient descent code

In [46]:

```
alpha = 1e-2
precision = 1e-6
w_old = np.zeros(3)
w_new = np.array([-3, -0.1, -1])
w_list, l_list = [w_old], [loss(x, t, w_new[0], w_new[1] )]
rate_modifier = np.array([1, 1, 1])

while sum(abs(w_new - w_old)) > precision:
    w_old = w_new
    g = new_gradient(x, t, w_old[0], w_old[1], w_old[2])
    w_new = w_old - alpha*rate_modifier * g
    w_list.append(w_new)
    l_list.append(new_loss(x, t, w_new[0], w_new[1], w_new[2]))

print "Minimum loss occurs at: ", w_new
print "Minimum loss is:", float(new_loss(x, t, w_new[0], w_new[1], w_new[2]))
print "Gradient:", new_gradient(x, t, w_new[0], w_new[1], w_new[2])
print "Number of steps:", len(l_list)
```

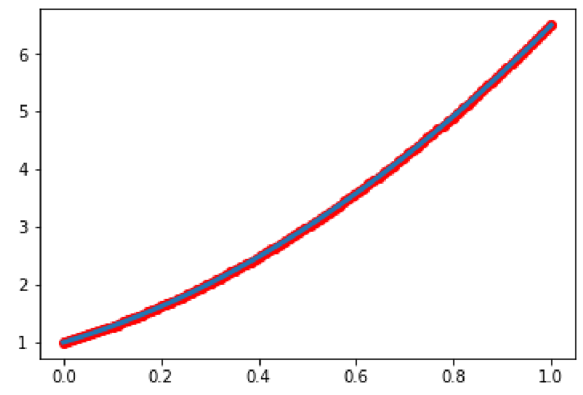
```
Minimum loss occurs at:  [ 0.99851817  2.50834951  2.99194944]
Minimum loss is: 3.82261619455e-07
Gradient: [ -8.28611112e-06   4.66889992e-05  -4.50173070e-05]
Number of steps: 85630
```

In [52]:

```
x_test =  # generate new x to plot the fitted line. Note better not to use the original x !
f_test =  # compute the corresponding target variables
# plot the fitted line
# plot data
```

Out[52]:

```
[<matplotlib.lines.Line2D at 0x10cd6cad0>]
```



Step 7.4 Can you derive the least square solution for this model?

Week 3: Vector/Matrix form linear regression lab

10/10/2018

Aims

- Practice general linear regression with polynomial and RBF on the Olympic 100m data
- Choose the order of polynomials with cross validation on some simulated data

Tasks

- Rescale our data
- Write functions to construct the design matrix, X, with polynomials and RBFs
- Computing least square solutions in both
- Solve the same problem with more sophicated gradient descent (code provided)
- Create partitions over the data of perform C-fold cross validation
- Check to cross validation results

Step 1: Again, we start by loading the Olympic 100m men's data

In [1]:

```
import numpy as np
%matplotlib inline
import pylab as plt

data = np.loadtxt('olympic100m.txt', delimiter=',') # make sure olympic100m.txt is in the right
folder
x = data[:,0][:,None] # make x a matrix
t = data[:,1][:,None] # make t a column vector
```

Setp 2: Vector/Matrix form least square solution

Setp 2.1 Rescale x

We rescale x to make it small. Doing so will stablise the computatoin, otherwise it quickly becomes unfeasible to fit polynomials over ~2000. Let's test the following two options:

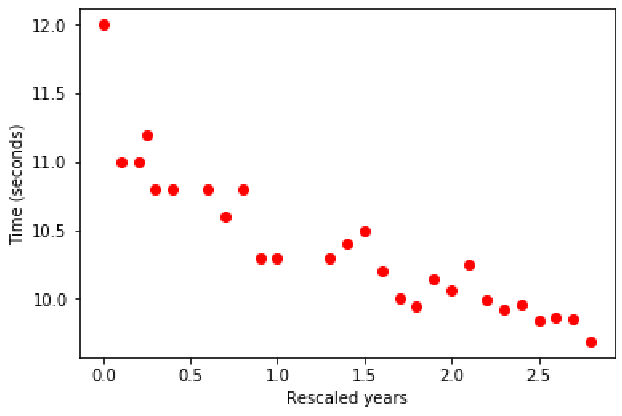
- Option 1: `(x-1896)/40`
- Option 2: `(x-np.mean(x))/np.std(x)`

In [578]:

```
x = ...# Test both options
...# plot the rescaled data
```

Out[578]:

```
Text(0,0.5,'Time (seconds)')
```



Step 2.2: Check the effect of the rescaling on contour plot for simple linear model $w_0 + w_1 x$

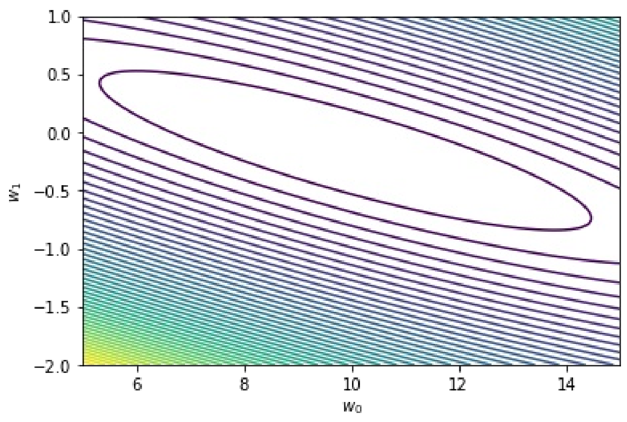
The rescaling shows the previously diffcult to see elliptical contours. For both rescaling options, you can use 5 to 15 for w_0, and -2 to 1 for w_1

In [579]:

```
num_candidates = ...# number of candidates
w0_candidates = ...# generate a numpy array of possible w0 values e.g. 5 to 15
w1_candidates = ...# generate a numpy array of possible w1 values e.g. -2 to 1
L = np.zeros( shape = (num_candidates,num_candidates) ) # Prelocate the loss. We are going to have
num_candidates times num_candidates of them

...# Write two nested for loops to compute L

plt.contour(w0_candidates, w1_candidates, L, 50) # A different way to plot contour without using me
shgrid
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
```

Out[579]:

```
Text(0,0.5,'$w_1$')
```



Step 2.2 Write your own function to construct the design matrix with polynomials

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^K \\ 1 & x_2 & x_2 & \dots & x_2^K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^K \end{bmatrix}$$

In [580]:

```
def polynomial (x, maxorder): # The np.hstack function can be very helpful
    ... # Write you own code here
    return(X)
```

Step 2.3 Construct the design matrix with a predefined maximum polynomial order, say 9

In [581]:

```
maxorder = 9
X_poly = polynomial (x, maxorder)
X_poly.shape
```

Out[581]:

```
(27, 10)
```

Step 2.4: Write your code to compute the least square solution

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

In [4]:

```
def least_square(X, t): # np.linalg.solve, np.linalg.dot
    ... # write your code here
    return( w )

w_poly =  least_square(X_poly, t)
w_poly
```

Out[4]:

```
array([[  11.98032055],
       [ -15.41168737],
       [  86.98553266],
       [-240.58030702],
       [ 363.14740079],
       [-322.95573921],
       [ 174.02670068],
       [ -55.8618961 ],
       [   9.82801402],
       [  -0.72952352]])
```

Step 2.5: Plot the fitted line

To make model predictions, we need transform any x_{new} with the basis function:

$$\mathbf{x}_{new} = \begin{bmatrix} h_0(x_{new}) \\ \vdots \\ h_K(x_{new}) \end{bmatrix}, \quad t_{new} = \mathbf{x}_{new}^T\hat{\mathbf{w}}$$

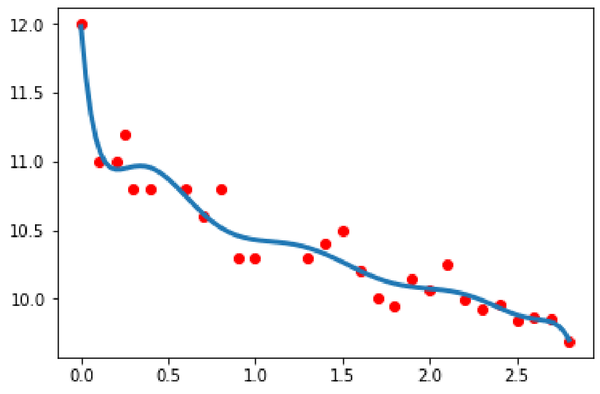
You need to construct a new design matrix for model prediction, e.g. `polynomial(x_test, maxorder)`

In [583]:

```
plt.plot(x, t, "ro")
x_test = ... # generate a separated set of x for plotting, min(x) to max(x) could be a good choice
f_test = ... # compute the corresponding prediction by the fitted model
plt.plot(x_test, f_test, linewidth=3)
```

Out[583]:

```
[<matplotlib.lines.Line2D at 0x1306faf10>]
```



Step 2.6: Now we perform general linear regression with RBF. Write you own function to construct the design matrix with RBF

$$h_k(x) = \exp(-\frac{(x - \text{center}[k])^2}{2\text{width}})$$

In [584]:

```
def rbf (x, center, width):
    ... # write your code here, again the np.hstack function can be very helpful
    return(X)
```

Step 2.7 Construct the design matrix with x itself as the center parameter

Start with `width = 10` and test different values. The result should be a `(27,27)` matrix

In [585]:

```
center = x
width = 10
X_rbf = rbf(x, center, width)
```

Step 2.8 Compute the least square solution with the previouly defined function

In [586]:

```
w_rbf = least_square(X_rbf,t)
w_rbf
```

Out[586]:

```
array([[  692209.91976258],
       [  299825.08348191],
       [ -364653.89636356],
       [ -288870.4928943 ],
       [-1521302.56428617],
       [ -166662.65353032],
       [  535254.33162671],
       [ 1467581.42051947],
       [  622080.53300001],
       [-1675528.41538212],
       [  591914.03644835],
       [ -400652.00714203],
       [ 1778320.47400029],
       [-1141530.00177157],
       [  229822.95468855],
       [ -495300.09431947],
       [ -456727.1449735 ],
       [  111779.047544  ],
       [-1342657.01184276],
       [  522613.00879356],
       [   -5052.79579146],
       [ 1377965.06600793],
       [  239847.14090297],
       [  337063.728521  ],
       [-1020432.54982108],
       [   67050.36860499],
       [    6133.73883264]])
```

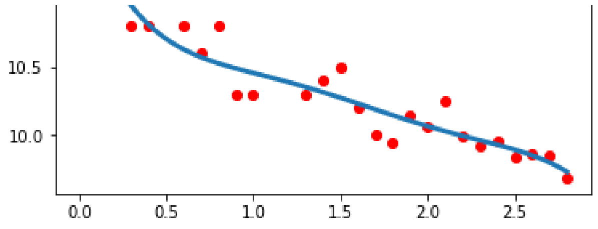
Step 2.9 Plot the fitted line

In [587]:

```
plt.plot(x, t, "ro")
x_test = ... # generate a separated set of x for plotting, min(x) to max(x) could be a good choice
f_test = ... # compute the corresponding prediction by the fitted model
plt.plot(x_test, f_test, linewidth=3)
```

Out[587]:

```
[<matplotlib.lines.Line2D at 0x130302950>]
```



Step 3: Instead of using the least square solution, we test gradient descent in general linear regression setting

- Average squared loss: $L(\mathbf{w}) = \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w})$
- Gradient: $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = -\frac{2}{N}(\mathbf{X}^T\mathbf{t} - \mathbf{X}^T\mathbf{X}\mathbf{w})$

Step 3.1 Define your own functions for the average squared loss and its gradient

Note the extra code to change the shape properities of `w` and `g`. These are made to fit the requirement in scipy.optimize.minimize. It requires the graident (`g`) to be the shape of `(d,)`, `d` being of the dimension of `w` and `g`.

In [588]:

```
def loss(w, X, t): # define the loss function
    w = w[:,None]
    L = np.mean( (t-np.dot(X, w))**2 ) # the average squared loss function
    return(L)

def gradient(w, X, t): # define the gradient function
    w = w[:,None]
    g = -2.0/len(t) * ( np.dot( np.transpose(X), t ) - np.dot(np.transpose(X), np.dot(X, w)) )
    return(g[:,0])
```

Step 3.2 This cell checks if your gradient function is correct by compare it with numerical approximation

In [593]:

```
w0 = np.ones((X_rbf.shape[1], 1))[:,0]

eps    = 1e-4 # step size
mygrad = gradient(w0, X_rbf, t)
fdgrad = np.zeros(w0.shape)
for d in range(len(w0)): # pertub each dimension in term
    mask = np.zeros(w0.shape) # a binary mask that only allows selected dimension to change
    mask[d]    = 1
    fdgrad[d] = (loss(w0 + eps*mask, X_rbf, t) - loss(w0 - eps*mask, X_rbf, t))/(2*eps) # numerical
approximation with the definition of gradient

print("MYGRAD: ", mygrad) # my gradient output
print("FDGRAD: ", fdgrad) # numerical gradient
print("Error: ", np.linalg.norm(mygrad-fdgrad)/np.linalg.norm(mygrad+fdgrad) ) # error
```

```
('MYGRAD: ', array([ 25.76727511,  26.1033838 ,  26.41932734,  26.56947464,
        26.7142716 ,  26.98743141,  27.4655223 ,  27.66915795,
        27.84842412,  28.00282802,  28.13194315,  28.36432437,
        28.38940744,  28.38812136,  28.36046748,  28.30652048,
        28.22642797,  28.12040983,  27.98875724,  27.8318313 ,
        27.65006144,  27.44394342,  27.21403711,  26.96096398,
        26.68540425,  26.3880939 ,  26.06982141]))
('FDGRAD: ', array([ 25.76727511,  26.1033838 ,  26.41932734,  26.56947464,
        26.7142716 ,  26.98743141,  27.4655223 ,  27.66915795,
        27.84842412,  28.00282802,  28.13194315,  28.36432437,
        28.38940745,  28.38812136,  28.36046748,  28.30652048,
        28.22642797,  28.12040983,  27.98875724,  27.8318313 ,
        27.65006144,  27.44394342,  27.21403711,  26.96096398,
        26.68540425,  26.3880939 ,  26.06982141]))
('Error: ', 3.3030581185676388e-12)
```

Step 3.3: We run an advanced gradient descent method, BFGS, which automatically determine the learning rate.

SciPy's `minimize` has already implemented a range of different methods

In [595]:

```
import scipy.optimize as opt

res = opt.minimize(loss, w0, args=(X_rbf, t), method='BFGS', jac=gradient,
                   options={'gtol': 1e-7, 'disp': True}) # google scipy minimize for more
information
res.x # solution
```

```
Optimization terminated successfully.
         Current function value: 0.030630
         Iterations: 48
         Function evaluations: 53
         Gradient evaluations: 53
```

Out[595]:

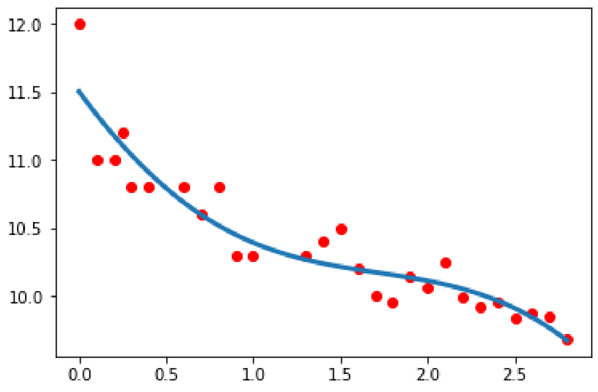
```
array([ 262.59547062,  152.11220565,   59.54065923,   19.77493508,
        -15.76871558,  -74.62054094, -146.89916173, -162.63629414,
       -166.51481814, -159.97545703, -144.54964009,  -61.31843512,
        -26.95268372,    7.81239462,   41.22623502,   71.55905497,
         97.11812649,  116.26364078,  127.42400131,  129.11040034,
        119.9305278 ,   98.60128641,   63.96038565,   14.9767092 ,
        -49.2406375 , -129.43468482, -226.19441778])
```

In [596]:

```
plt.plot(x, t, "ro")
x_test = ... # generate a separated set of x for plotting, min(x) to max(x) could be a good choice
f_test = ... # compute the corresponding prediction by the fitted model
plt.plot(x_test, f_test, linewidth=3)
```

Out[596]:

```
[<matplotlib.lines.Line2D at 0x1305fae50>]
```



The loss at the least square solution is still lower

In [598]:

```
loss(w_rbf[:,0], X_rbf, t) < loss(res.x, X_rbf, t)
```

Out[598]:

```
True
```

Step 5: Cross-validation to determine the order of polynomials </h3>

In this part, we perform a cross-validation to try and choose the polynomial order for some synthetic data.

The loss (in our case, this is equal to the squared error) on the training data will always decrease as we make the model more complex, and therefore cannot be used to select how complex the model ought to be. In some cases we may be lucky enough to have an independent test set for model validation but often we will not. In a cross-validation procedure (CV) we split the data into K folds, and train K different models, each with a different data fold removed. This removed fold is used for testing and the performance is averaged over the K different folds.

This process is illustrated in the following figure:

Step 5.1: We start by generating two datasets - one on which we will perform the CV, and a second independent testing set.

The true model is $f(x; \mathbf{w}) = x - x^2 + 5x^3$

In [672]:

```
np.random.seed(1) # fix random seed such that every time we get the same random numbers

N = 200 # total number of data points
x = 10*np.random.rand(N,1) - 5 # generate random x
t = 5*x**3 - x**2 + x + 200*np.random.randn(N,1) # generate t according to the true model with
additive noise

N_test = 100 # total number of independent testing data points
x_test = np.linspace(-5,5,N_test)[:,None] # generate independent testing x
t_test = 5*x_test**3 - x_test**2 + x_test + 200*np.random.randn(N_test,1) # generate independent te
sting t with noise
```

Step 5.2: Cross validation settings

In [673]:

```
max_order = 10 # the maximum polynomial order to be tested
num_folds = 10 # number of folds, C in the slides

# This block of code is used to generate coordinates for the folds. There are mulitple way of doin
g it.
sizes = np.tile(np.floor(N/num_folds),(1,num_folds))
sizes[-1] = sizes[-1] + N - sizes.sum()
c_sizes = np.hstack((0,np.cumsum(sizes)))
c_sizes =c_sizes.astype(int)
print(c_sizes)
```

```
[   0  20  40  60  80 100 120 140 160 180 200]
```

Step 5.3: Perform cross validation

In [674]:

```
cv_loss = ... # preallocate cross validation losses, you need num_folds times max_order+1 of them
ind_loss = ... # preallocate independent testing losses, you need num_folds times max_order+1 of t
hem
train_loss = ... # preallocate training losses, you need num_folds times max_order+1 of them

for k in range(max_order+1): # iterate over all orders

    X = ...  # construct the design matrix over all training data using order k
    X_test = ... # construct the design matrix over independent testing data using order k


    for fold in range(K): # iterate over all folds

        X_train = np.delete(X,np.arange(c_sizes[fold],c_sizes[fold+1],1),0) # select N-C training x

        t_train = np.delete(t,np.arange(c_sizes[fold],c_sizes[fold+1],1),0) # select N-C training t

        w = ... # compute least square solution


        X_fold = X[c_sizes[fold]:c_sizes[fold+1],:] # select C testing x
        t_fold = t[c_sizes[fold]:c_sizes[fold+1]] # select C testing t
        train_loss[fold,k] = ... # record training loss at k in the current fold
        cv_loss[fold,k] = ... # record cross validation loss at k in the current fold
        ind_loss[fold,k] = ... # record independent testing loss at k in the current fold
```

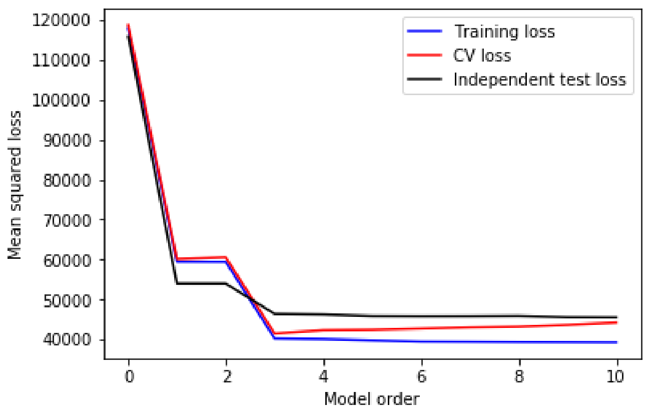
Step 5.4: Plot results

In [669]:

```
order = np.arange(max_order+1) # all tested orders
plt.plot(order,train_loss.mean(axis=0),'b-',label="Training loss") # avearage training losses
plt.plot(order,cv_loss.mean(axis=0),'r-',label="CV loss") # avearage CV losses
plt.plot(order,ind_loss.mean(axis=0),'k',label="Independent test loss") # avearage independent
testing losses
plt.legend()
plt.xlabel('Model order')
plt.ylabel('Mean squared loss')
```

Out[669]:

```
Text(0,0.5,'Mean squared loss')
```



Week 4: Random variables

17/10/2018

Aim

- Practice `numpy` random number generators.
- Practice simulating noisy data from simple linear model

Task

- Generate normally distribution random variables
- Use histogram to inspect emprical distribution of random variables
- Plot confidence bonds for a simple linear model

Step 1: Set up parameters for a Gaussian distribution

In [113]:

```
import numpy as np
import pylab as plt
%matplotlib inline

np.random.seed(1) # fix random seed such that every time we get the same random numbers

mu = ...# mean, try 1
sigma = ...# standard deviation, try 0.5
```

Step 2: Generate Gaussian random number using `numpy.random.randn` with μ and $sigma$ in the previous cell

Details of how to use `numpy.random.randn`

https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randn.html

In [114]:

```
n = ...# number of data points
x = ...# Code to generate the random variable. np.random.randn
    # generates random numbers from the standard normal distribution
    # N(0,1). For random samples from N(\mu, \sigma^2),
    # use: sigma * np.random.randn(...) + mu
```

Setp 3: Plot the histogram of the generated random numbers

Details of how to use `matplotlib.pyplot.hist`

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html

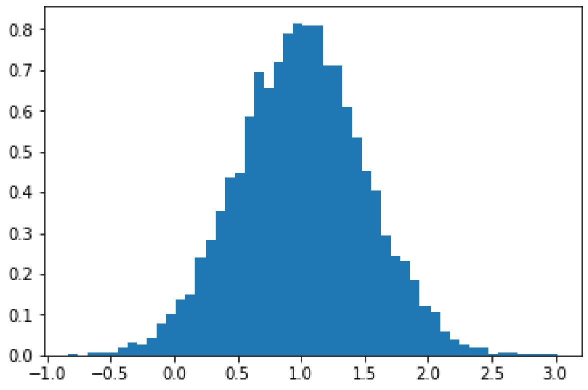
You can experiment to see how the histogram changes when you use different `n`. Try $100, 1000, 10000, 100000$.

In [115]:

```
...# bins is the number bins the histogram
# uses to compute counts. If density is true, the counts
# will compute normalised counts to approximate
# probability density function.
```

Out[115]:

```
(array([ 0.00130153,  0.        ,  0.0052061 ,  0.00650763,  0.0052061 ,
         0.01691984,  0.03123662,  0.02733204,  0.04425188,  0.07679003,
         0.10151902,  0.13796175,  0.14837396,  0.23948077,  0.28243113,
         0.351412  ,  0.43601118,  0.44642339,  0.58698819,  0.69371332,
         0.65466754,  0.71844231,  0.78872471,  0.81475523,  0.80954912,
         0.80954912,  0.71193468,  0.71193468,  0.61041566,  0.5323241 ,
         0.45032797,  0.40477456,  0.29414486,  0.24468687,  0.23037009,
         0.18221363,  0.11974038,  0.10672513,  0.05726714,  0.03904578,
         0.02603052,  0.01691984,  0.01691984,  0.00390458,  0.00780916,
         0.0052061 ,  0.00260305,  0.00130153,  0.00130153,  0.00260305]),
 array([-0.82822005, -0.75138716, -0.67455427, -0.59772138, -0.52088848,
        -0.44405559, -0.3672227 , -0.29038981, -0.21355692, -0.13672403,
        -0.05989114,  0.01694176,  0.09377465,  0.17060754,  0.24744043,
         0.32427332,  0.40110621,  0.4779391 ,  0.554772  ,  0.63160489,
         0.70843778,  0.78527067,  0.86210356,  0.93893645,  1.01576934,
         1.09260224,  1.16943513,  1.24626802,  1.32310091,  1.3999338 ,
         1.47676669,  1.55359958,  1.63043248,  1.70726537,  1.78409826,
         1.86093115,  1.93776404,  2.01459693,  2.09142983,  2.16826272,
         2.24509561,  2.3219285 ,  2.39876139,  2.47559428,  2.55242717,
         2.62926007,  2.70609296,  2.78292585,  2.85975874,  2.93659163,
         3.01342452]),
 <a list of 50 Patch objects>)
```



Step 4: Write you own function to compute the analytical probability density function (PDF) of an univariate Gaussian distribution

The univariate Gaussian PDF is given by:

$$p(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$$

This is a probability density function of x with μ and σ^2 as its parameters.

In [116]:

```python
def gaussian1d_pdf(x, mu, sigma):
    ... # write you own code: you can use np.pi for \pi and np.exp for
        # the exponential function
```

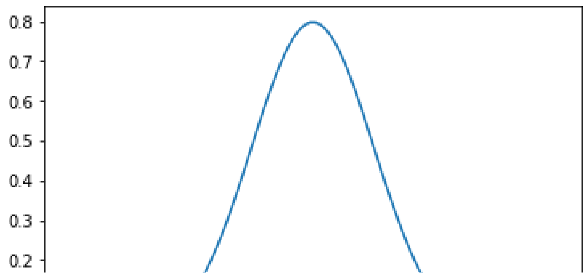
Step 5: Plot the Gaussian PDF

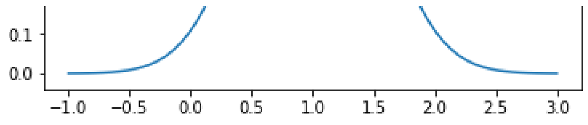
In [117]:

```python
x_candidates = ... # use np.linspace to generate possible x for the plot
y = ... # compute the PDF with your own function
... # Plot the PDF
```

Out[117]:

```
[<matplotlib.lines.Line2D at 0x1165edc90>]
```





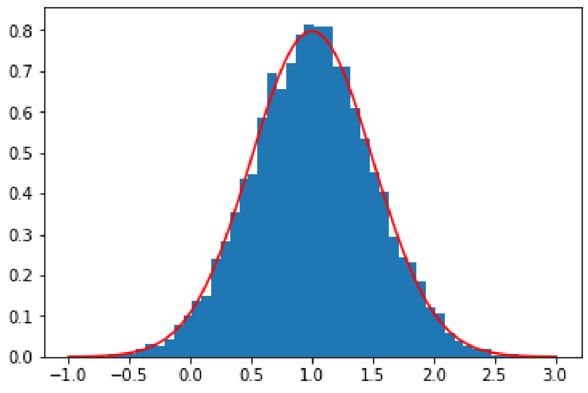
Step 6: We can overlay the analytical Gaussian PDF and the emprical histogram.

In [118]:

```
... # plot the histogram of previous generated x
... # plot the Gaussian PDF
```

Out[118]:

```
[<matplotlib.lines.Line2D at 0x1161f6890>]
```



Step 7: Back to our model, generate noisy data

$$t_n = w_0 + w_1 x + \epsilon_n, \quad \epsilon_n \sim >(\mu, \sigma^2)$$

Step 7.1: Generate x, this time as a uniformly distributed random variable `numpy.random.rand`

Details of `numpy.random.rand`

https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.rand.html

In [119]:

```
n = ... # number of data points
x = ... # generate n uniformly distributed random numbers
```

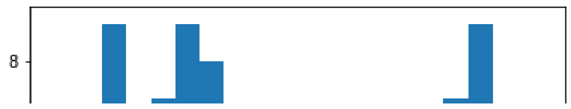
Step 7.2: Check the histogram

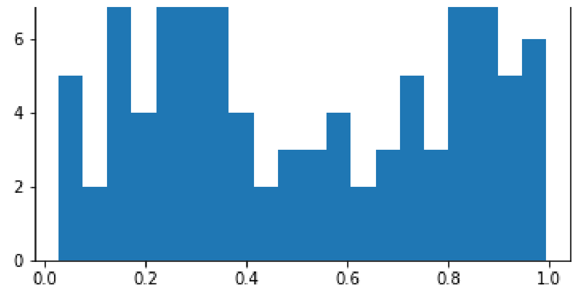
In [120]:

```
... # bins set at 20
```

Out[120]:

```
(array([ 5.,  2.,  9.,  4.,  7.,  9.,  8.,  4.,  2.,  3.,  3.,  4.,  2.,
         3.,  5.,  3.,  7.,  9.,  5.,  6.]),
 array([ 0.02648081,  0.07493029,  0.12337977,  0.17182925,  0.22027873,
         0.26872822,  0.3171777 ,  0.36562718,  0.41407666,  0.46252614,
         0.51097563,  0.55942511,  0.60787459,  0.65632407,  0.70477355,
         0.75322303,  0.80167252,  0.850122  ,  0.89857148,  0.94702096,
         0.99547044]),
 <a list of 20 Patch objects>)
```





Step 7.3: This time, we encapsulate the simple linear model into a function

In [121]:

```
def my_model(x, w):
    ... # write your own code for simple linear model
```

Step 7.4: Setup parameters

In [122]:

```
w = ... # a numpy array for w0 and w1
sigma = ... # standard deviation of the noise
```

Step 7.5 Generate noisy data from the model

In [123]:

```
y = ... # Use the same method in step 2 to generate the noisy data
```

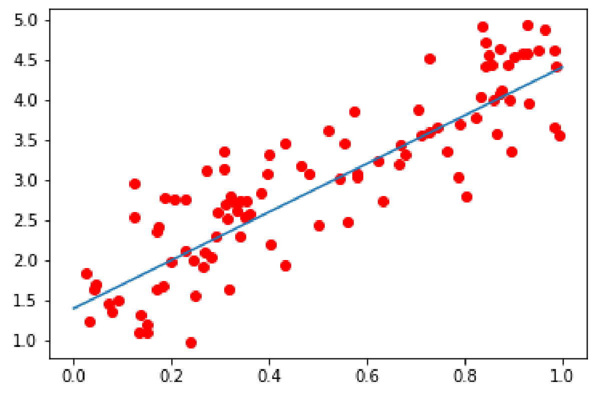
Step 7.6 Plot the noisy data and the true model

In [124]:

```
... # you know what to do here :)
```

Out[124]:

```
[<matplotlib.lines.Line2D at 0x1168a0a10>]
```



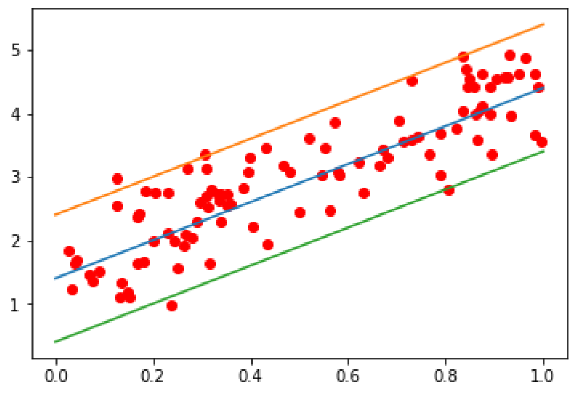
Step 7.6 Plot the noisy data, the true model, and the upper and lower confidence bounds

In [125]:

```
... # upper (~95%) confidence bound: my_model(x_axis, w) + 2*sigma
... # lower (~95%) confidence bound: my_model(x_axis, w) - 2*sigma
```

Out[125]:

```
[<matplotlib.lines.Line2D at 0x116a1cd10>]
```



Week 5

- #### Aims
  - Practice maximum likelihood with olympic data
  - Practice expectation and numerical approaximation
  - Practice predictive variance on simulated data
- #### Tasks
  - Write your own functions to compute maximum likelihood estimates of w and σ^2
  - Test optimal log-joint likelihood against different polynomial orders
  - Compare empirical and exact expectation
  - Visualise predictive variance and simulate possible models

Step 1: Load olympic data and rescale

In [3]:

```
import numpy as np
import pylab as plt
%matplotlib inline

data = np.loadtxt('olympic100m.txt',delimiter=',')
x = ... # again, make x a column vector
t = ... # again, make t a column vector
x = ... # rescale x
```

Step 2: Write your function for the log of Gaussian PDF and construct polynomial design matrix

$$\log > (x;\mu,\sigma^2) = \log(\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{1}{2\sigma^2}(x-\mu)^2)) = -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}(x-\mu)^2$$

In [4]:

```
def loggausspdf(x,mu,sigma2): # your own function to compute log of gaussian pdf
    ...


def polynomial(x, max_order): # your own function to construct polynomial design matrix
    ...
```

Step 3: Write your function for maximum likelihood estimator for w and σ^2

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t},\ \ \hat{\sigma^2} = \frac{1}{N}(\mathbf{t}-\mathbf{X}\hat{\mathbf{w}})^T(\mathbf{t}-\mathbf{X}\hat{\mathbf{w}})$$

In [5]:

```
def max_like_w(X, t): # your own function to compute maximum likelihood estimate of w
    ...

def max_like_sigma2(X, t, w): # your own function to compute maximum likelihood estimate of
sigma^2
    ...
```

Step 4: Test the joint likelihood against order of polynomails

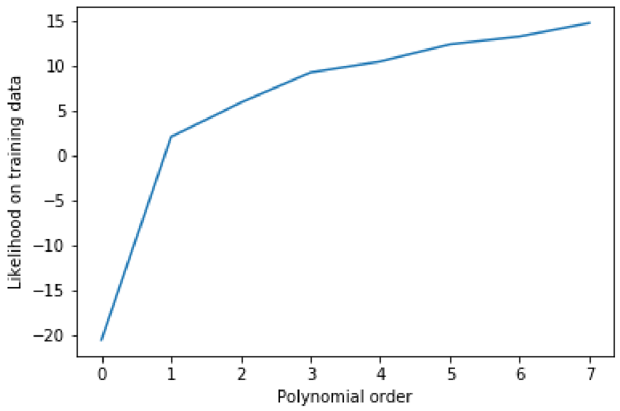
- log joint likelihood: $\sum_{n=1}^{N}\log p(t_n|x_n,w,\sigma^2) = \sum_{n=1}^{N}\log > (t_n;\mathbf{x}_n^T\mathbf{w},\sigma^2)$
- log joint likelihood at $\hat{\mathbf{w}}$ and $\hat{\sigma^2}$: $\sum_{n=1}^{N}\log > (t_n;\mathbf{x}_n^T\hat{\mathbf{w}},\hat{\sigma^2})$

In [6]:

```
max_order = 7 # maximum order
L = np.zeros((max_order+1,1)) # preallocate log joint likelihoods
for i in np.arange(max_order+1): # for loop over all polynomial order
    X = ... # construct polynomial design matrix
    w_ml = ... # compute maximum likelihood estimate of w
    sigma2_ml = ... # compute maximum likelihood estimate of sigma^2
    L[i] = ... # compute the log-joint likelihood at the maximum likelihood estimates

... # plot L vs polynomial orders
plt.xlabel('Polynomial order')
plt.ylabel('Likelihood on training data')
```

Out[6]:

```
Text(0,0.5,'Likelihood on training data')
```



It can be seen from the plot that the likelihood increases as the polynomial order increases.

Step 5: Sample-based approximations to expectations

We will try and compute the expected value of y^2 where $y \sim U(0,1)$ i.e.

$$E_{p(y)}(y^2) \approx \frac{1}{N}\sum_{n=1}^{N} y_n^2, y_n \sim U(0,1)$$

Analytically, we can compute the value as:

$$E_{p(y)}(y^2) = \int_0^1 y^2 p(y)\, dy = \int_0^1 y^2\, dy = \left[\frac{1}{3}y^3\right]_0^1 = \frac{1}{3}$$

To compute a sample based approximation, we will draw samples from $U(0,1)$, square them, and compute the average.

In [7]:

```
np.random.seed(1)
N = ...; # maximum number of samples, e.g. 10000
u = ... # generate random numbers from uniform distribution (0,1)
expected_val = np.zeros((N-1,1)) # preallocate emprical approximation with different N
for i in np.arange(N-1): # for loop over different number of samples
    expected_val[i] = ... # compute empirical approximation to the expectation
```

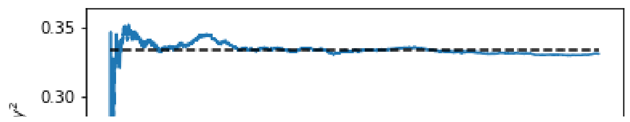
Step 6: Plot empirical approximation aganist number of samples

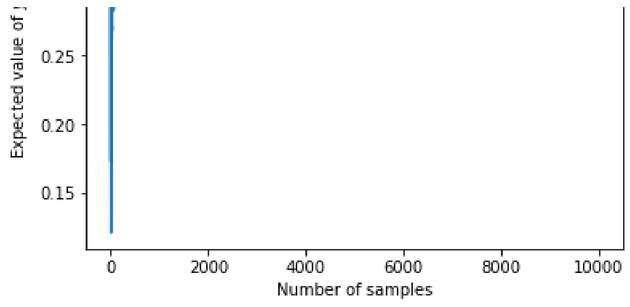
In [9]:

```
...# plot the empirical approximations vs the number of samples
... # plot the truth
plt.xlabel('Number of samples')
plt.ylabel('Expected value of $y^2$')
```

Out[9]:

```
Text(0,0.5,'Expected value of $y^2$')
```





Step 7: Predictive variance example, step up

In this example, we look at the predictive variance. Recall that, for maximum likelihood estimators

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}, \quad \hat{\sigma}^2 = \frac{1}{N}(\mathbf{t} - \mathbf{X}\hat{\mathbf{w}})^T(\mathbf{t} - \mathbf{X}\hat{\mathbf{w}})$$

we have mean prediction:

$$t_{new} = \hat{\mathbf{w}}^T\mathbf{x}_{new}$$

and variance:

$$\text{var}(t_{new}) = \hat{\sigma}^2\mathbf{x}_{new}^T(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{x}_{new}$$

We'll start by generating some synthetic data from a third order polynomial with data between 0 and 2 missing:

details of the `sort` function https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.sort.html

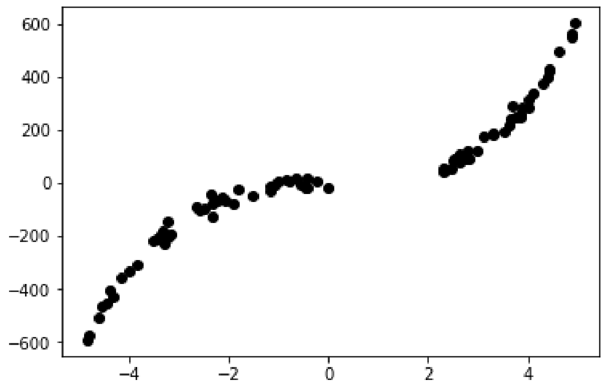
details of the `delete` function, useful for remove data points between 0 and 2 https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.delete.html

In [10]:

```
N = ... # number of data points, e.g. 100
x = np.sort((10*np.random.rand(N,1)-5),axis=0) # uniform number between -5 and 5
t = ... # true model, w0=0, w1=1, w2=-1, w3=5
noise_var = ... # true sigma^2 = 300
t = ... # add noise
pos = ((x>0)*(x<2)).nonzero()[0] # code to remove data points between 0 and 2
x = np.delete(x,pos)[:,None]
t = np.delete(t,pos)[:,None]
plt.plot(x,t,'ko') # plot the resulting data
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x11137c050>]
```



Step 8: Compute the predictive variance for different polynomial orders

We now fit the models and plot predictive mean and variances (as error bars) based on the equations above. Note that the `diag` function lets us extract the diagonal of a matrix which allows us to compute the variances for all test points in one operation.

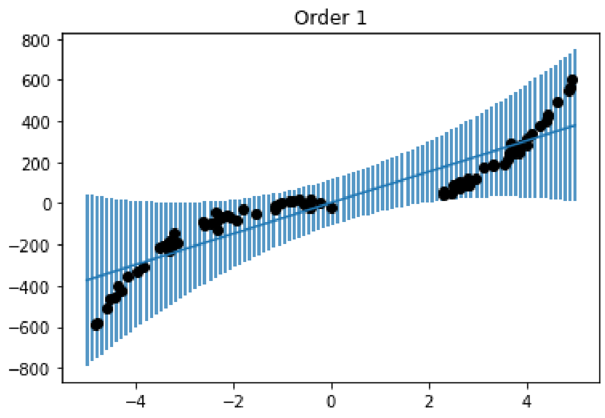
$$var\{t_{new,1}\}, \ldots, var\{t_{new,M}\} = diag\left(\hat{\sigma}^2\mathbf{X}_{new}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}_{new}^T\right)$$

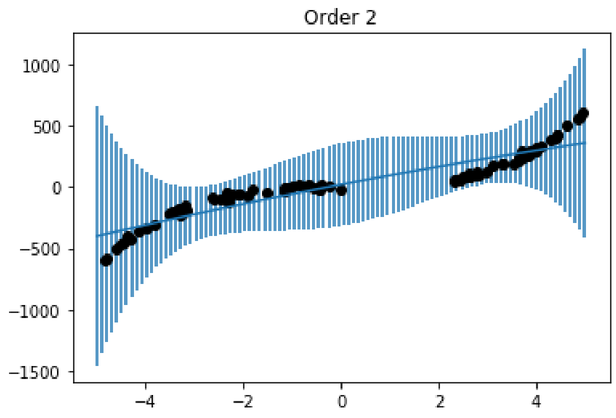
details of the `diag` function https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.diag.html details of `errorbar` function https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html

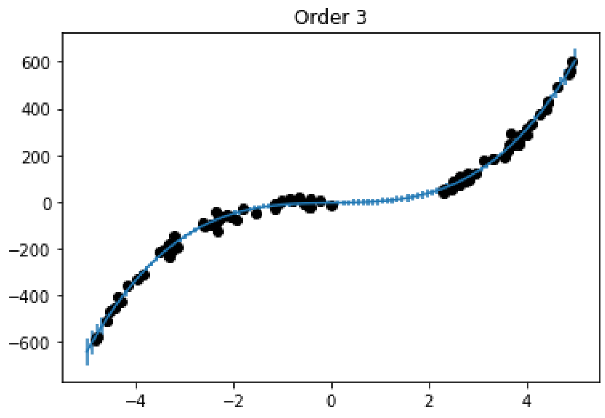
details of the `flatten` function, useful for argument for `errorbar` function https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.ndarray.flatten.html

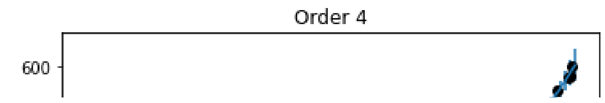
In [11]:

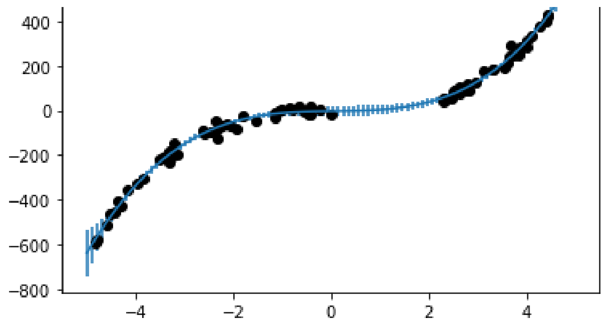
```
testx = ... # new x to predict. e.g -5 to 5, better to be a column vector
orders = ... # array of possible orders
for i in orders:
    X = ... # construct polynoimals on training data
    testX = ... # construct polynoimals on new data
    w = ... # compute maximum likelihood estimate of w
    sigma2 = ... # compute maximum likelihood estimate of sigma2
    test_mean = ... # expected prediction
    test_var = ... # prediction variance
    plt.figure()
    ... # plot training data
    plt.errorbar(testx.flatten(), test_mean.flatten(), yerr=test_var.flatten()) # plot error bar
    plt.title("Order " + str(i))
```



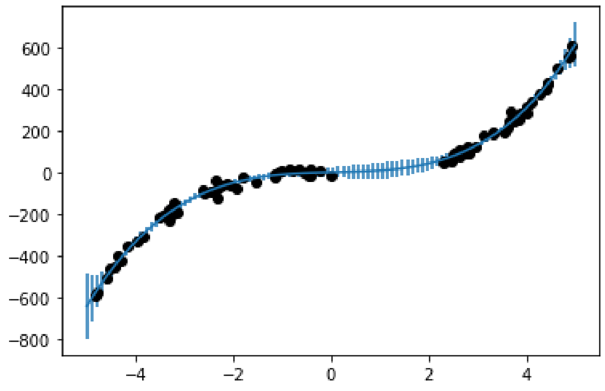




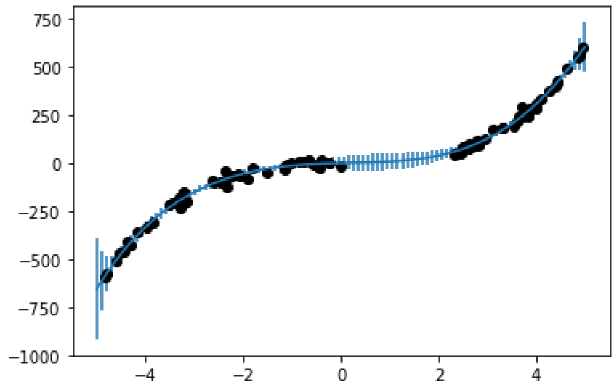




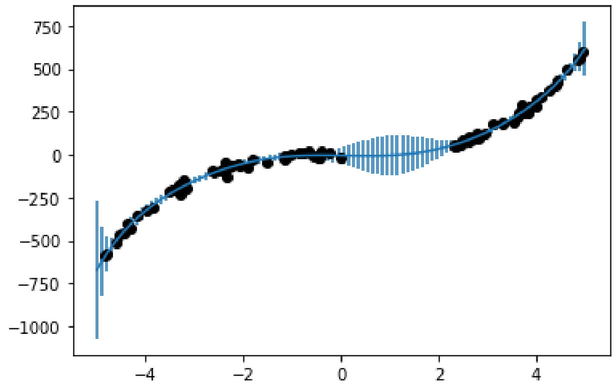
Order 5



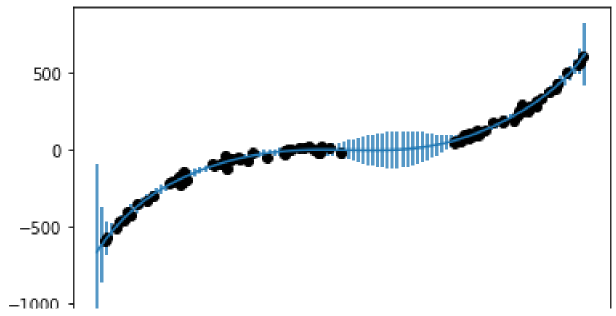
Order 6

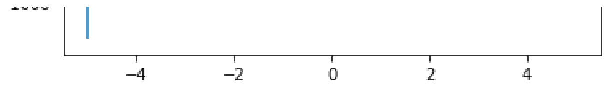


Order 7



Order 8





Step 9 (Optional): Sample models from $> (\hat{\mathbf{w}}, cov\{\mathbf{w}\})$

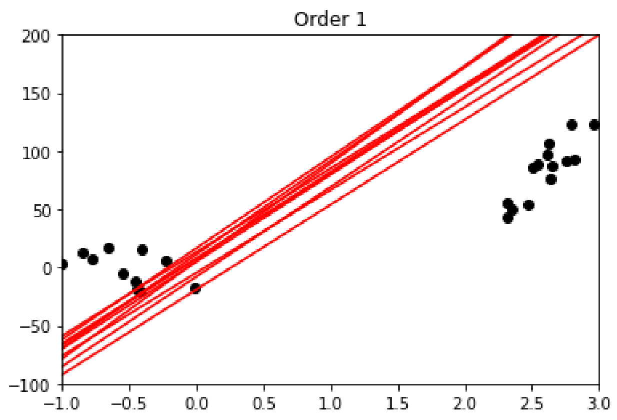
$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$
$$cov\{\mathbf{w}\} = \hat{\sigma^2}(\mathbf{X}^T\mathbf{X})^{-1}$$

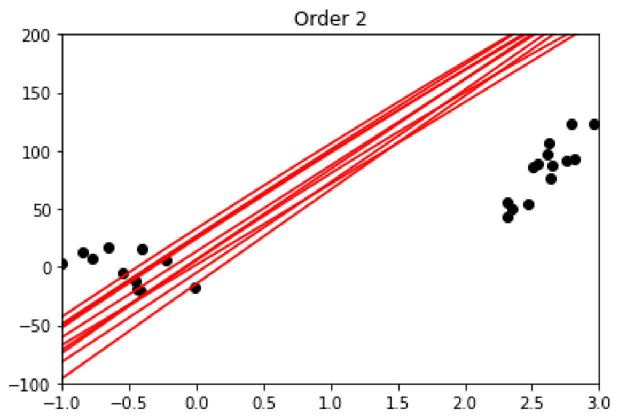
We notice that the variance decreases but then starts increasing again as the model order increases. This is due to the increased flexibility of the higher order models, particularly in regions where there is no data. The following plots show possible models - as the order increases it's clear that the models exhibit greater variability within the areas lacking data.

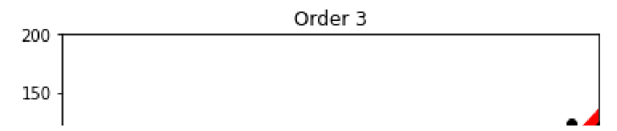
You can use `numpy`'s `multivariate_normal` function to generate samples from a multivariate normal distribution https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.multivariate_normal.html

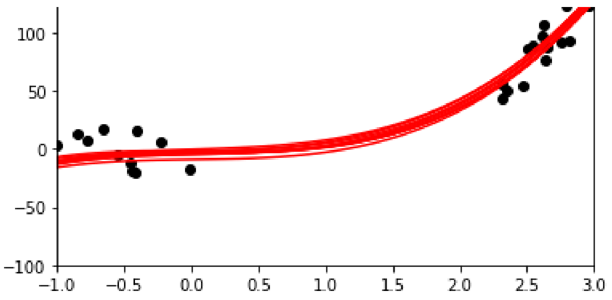
In [14]:

```
orders = ... # array of possible orders
for i in orders:
    X = ... # construct polynoimals on training data
    testX = ... # construct polynoimals on new data
    w = ... # compute maximum likelihood estimate of w
    sigma2 = ... # compute maximum likelihood estimate of sigma2
    covw = ... # covariance of w
    sampw = np.random.multivariate_normal(w.flatten(),covw,10) # generate 10 samples of w
    testt = np.dot(testX,sampw.T) # 10 model predictions based on sampw
    plt.figure()
    plt.plot(x,t,'ko')
    plt.plot(testx, testt,'r')
    plt.xlim([-1,3])
    plt.ylim([-100,200])
    plt.title('Order ' + str(i))
```

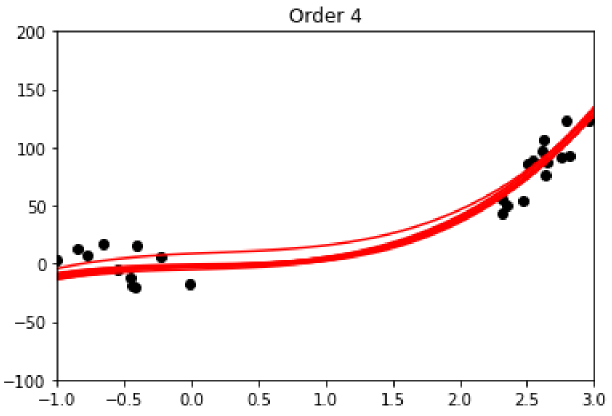




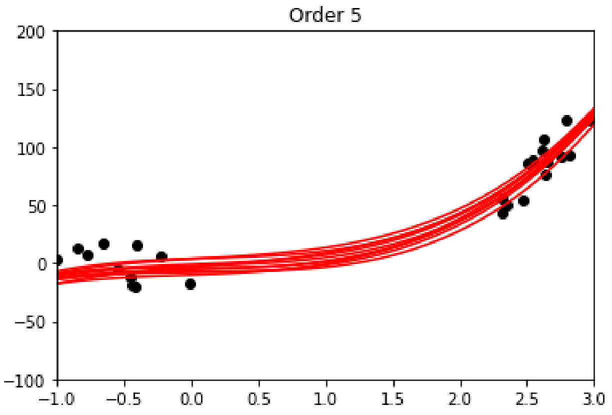




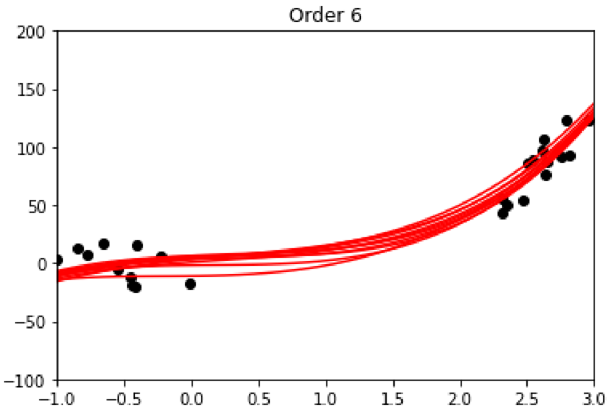
100
50
0
−50
−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



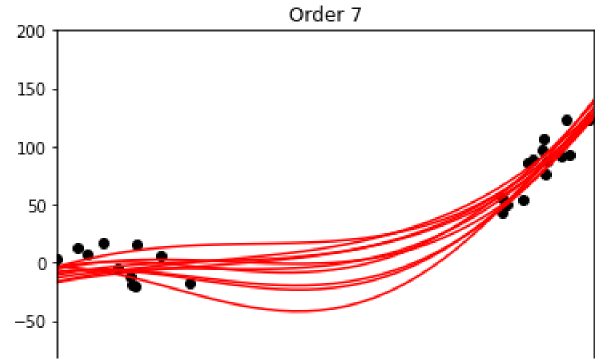
Order 4
200
150
100
50
0
−50
−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



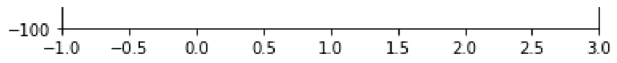
Order 5
200
150
100
50
0
−50
−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



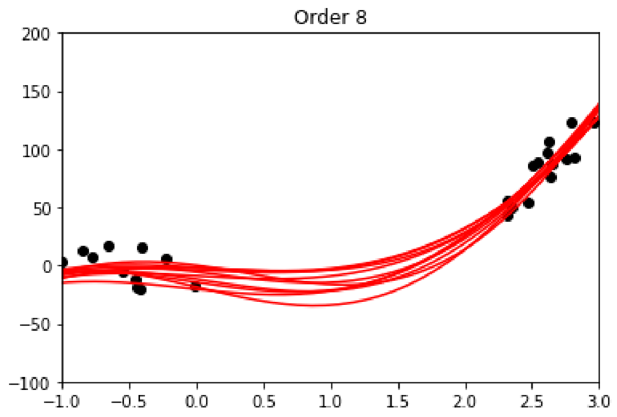
Order 6
200
150
100
50
0
−50
−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



Order 7
200
150
100
50
0
−50



−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



Order 8
200
150
100
50
0
−50
−100
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Week 6: Bayesian linear regression lab

- #### Aims
  - To implement Bayesian inference over the parameters of the linear model for the Olympic data.
  - To get more experience in the use of Bayesian models for regression, in particular predictions.
  - Practise model selection with marginal likelihood

Task 1: Bayesian treatment of the Olympic regression problem

In this task, we will perform a Bayesian treatment of the Olympic regression problem.

Step 1: We start by loading the data and rescaling it to aid with numerics.

In [161]:

```
import numpy as np
import pylab as plt
%matplotlib inline
np.random.seed(1)


... # load olympic data
... # make x a colmun vector
... # make t a colmun vector
x = (x - x[0])/4.0 # rescale x
```

Step 2: Step up prior, $p(\mathbf{w})$

We'll define a Gaussian prior over $\mathbf{w}$, with mean $\mathbf{0}$ and covariance $\begin{bmatrix} 100 & 0 \\ 0 & 5 \end{bmatrix}$. We'll also fix $\sigma^2 = 2$.

In [162]:

```
prior_mean = ... # vector of mean
prior_cov = ... # covariance matrix
sig_sq = ... # variance of the additive noise
```

Step 3: Let's see what this prior means by sampling some $\mathbf{w}$ vectors from it and plotting the models (polynomial order = 1)

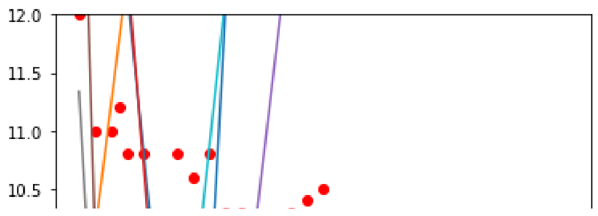
Use `numpy`'s `multivariate_normal` to generate samples from a multivariate Gaussian https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.multivariate_normal.html

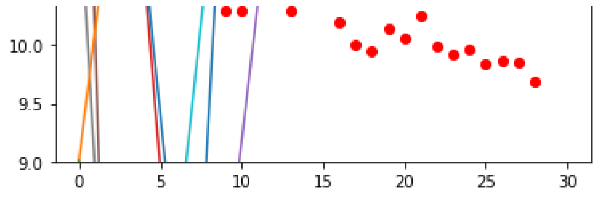
In [163]:

```
w_samp = ... # draw 20 samples from the prior
plt.figure()
... # plot data
... # generate new x for plotting the sampled model, e.g. you need construct design matrix for any
new x, you just need 2 data points
... # plot the lines
plt.ylim([9,12])
```

Out[163]:

```
(9, 12)
```





Step 4: Draw a contour plot of the prior:

The multivariate Gaussian pdf is given by:

$$p(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\}$$

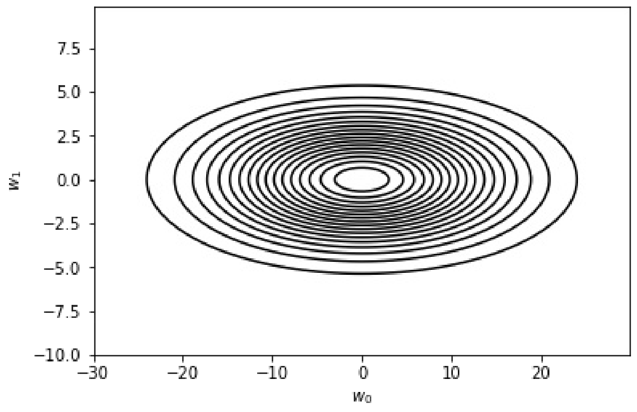
You can use `numpy`'s `det` function to compute the matrix determinant https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.linalg.det.html

In [164]:

```
def gaussian2d(mu,sigma,xvals,yvals): # A function to plot 2d Gaussian pdf,
                                       # here you can setup an input for each dimension.
                                       # This way can help us to use meshgrid to aviod a double for
oop.
    ...
    return


xp = ... # select a suitable range for x
yp = ... # select a suitable range for y
Xp,Yp = np.meshgrid(xp,yp) # you can use meshgrid instead of a for loop
Z = gaussian2d(prior_mean,prior_cov,Xp,Yp) # run your function this way to avoid for loop
CS = plt.contour(Xp,Yp,Z,20,colors='k')
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
```

Out[164]:

```
Text(0,0.5,'$w_1$')
```



Step 5: Compute the posterior and draw samples from it

First let's write functions to construct polynomial design matrix, and to compute posterior mean and covariance

$$\boldsymbol{\Sigma} = \left(\frac{1}{\sigma^2}\mathbf{X}^T\mathbf{X} + \mathbf{S}^{-1}\right)^{-1}$$

$$\boldsymbol{\mu} = \frac{1}{\sigma^2}\boldsymbol{\Sigma}\mathbf{X}^T\mathbf{t}$$

where $\mathbf{S}$ is the covariance matrix of the prior $p(\mathbf{w})$

In [165]:

```
def polynomial(x, max_order): # your own function to construct polynomial design matrix
    ...
```

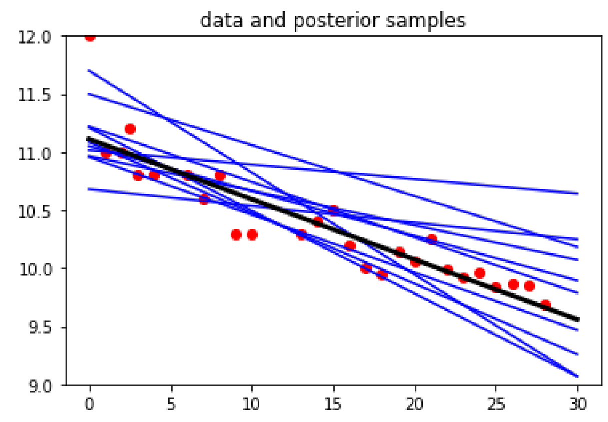
```
def compute_post_cov(X, prior_cov, sig_sq): # your own function to compute posterior mean
    ...

def compute_post_mean(post_cov, sig_sq, X, t): # your own function to compute posterior covariance
    ...
```

In [166]:

```
X = ... # construct design matrix, order = 1
post_cov = ... # compute posterior mean
post_mean = ... # compute posterior covariance

plt.figure()
plt.title('data and posterior samples')
w_samp = ... # draw 10 samples from the posterior
plt.plot(x,t,'ro')
... # plot the sampled lines, only need 2 points to plot a straight line
... # plot the posterior mean prediction, only need 2 points to plot a straight line
plt.ylim([9,12])
```

Out[166]:

```
(9, 12)
```



Task 2: We'll now look at predictions

Step 1: Functions for posterior prediction

$$p(t_{new}|\mathbf{X}, \mathbf{t}, \mathbf{x}_{new}, \sigma^2) = >(\mathbf{x}_{new}^T \boldsymbol{\mu}, \sigma^2 + \mathbf{x}_{new}^T \boldsymbol{\Sigma} \mathbf{x}_{new})$$

In [167]:

```
order = 1
test_x = ... # generate some test data
testX = ...

pred_mean = ... # compute predictive mean
pred_var = ... # compute predictive variance
```

Step 2: Plot error bars

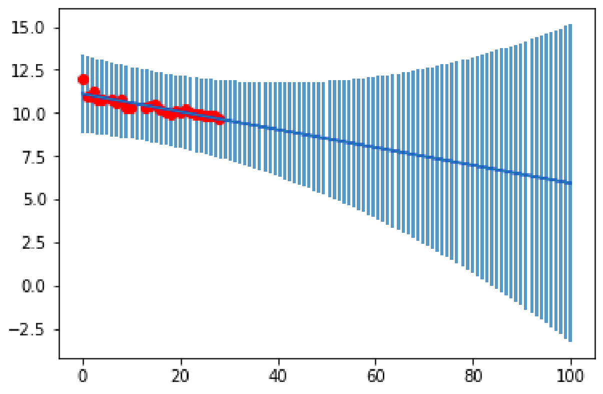
Use the `errorbar` function from `matplotlib` https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html

In [168]:

```
... # plot data
... # plot mean prediction
... # plot error bars
```

Out[168]:

```
<ErrorbarContainer object of 3 artists>
```



Task 3: Model selection with the marginal likelihood

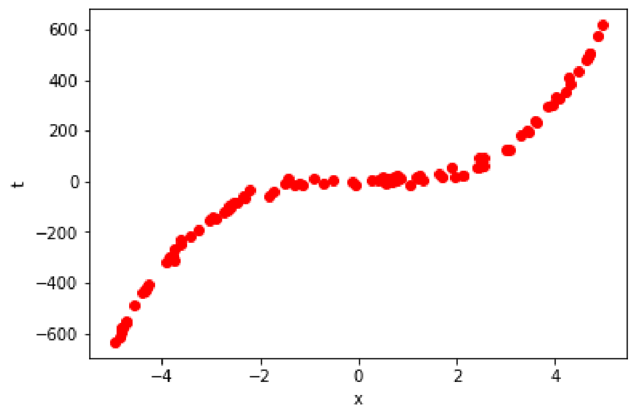
Step 1: Simulate data

In [169]:

```
N = ... # number of data points
x = ... # uniform number between -5 and 5
t = ... # true model # true model, w0=0, w1=1, w2=-1, w3=5
noise_var = ... # true sigma^2 = 150
t = ... # add noise
plt.plot(x,t,'ro')
plt.xlabel('x')
plt.ylabel('t')
```

Out[169]:

```
Text(0,0.5,'t')
```



Step 2: Write your function to compute the mean and covariance of the marginal likelihood

$$p(\mathbf{t}|\mathbf{X}, \mathbf{t}, \sigma^2, \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) = >(\mathbf{X}\boldsymbol{\mu}_0, \sigma^2\mathbf{I} + \mathbf{X}\boldsymbol{\Sigma}_0\mathbf{X}^T)$$

where

$$\boldsymbol{\mu}_0 = \mathbf{0}, \boldsymbol{\Sigma}_0 = \mathbf{I}$$

You can use `numpy`'s `eye` function to construct the identity matrix https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.eye.html

In [170]:

```
def compute_marg_mean(X, prior_mean): # your function to compute the mean of the marginal
likelihood
    ...

def compute_marg_cov(X, sig_sq, prior_cov): # your function to compute the covariance of the
marginal likelihood
    ...
```

```
def log_multivariate_gaussian_pdf(x, mean, cov): # your function to compute the log of
multivariate gaussian pdf,
                                                 # make sure the function returns a scalar (not arr
y)
    ...
```

Step 3: Computing the marginal likelihood for different polynomial orders, this will be much simpler than cross-validation

In [171]:

```
max_order = ... # max order
log_marg_like = [] # A list to collection log_marg_like
for i in ...: # for loop over all possible polynomial orders
    prior_mean = ... # setup prior mean, note the dimension should change with polynomial orders
    prior_cov = ... # setup prior covariance, note the dimension should change with polynomial
orders
    X = ... # construct polynomial design matrix
    marg_cov = ... # compute the mean of the marginal likelihood
    marg_mean = ... # compute the covariance of the marginal likelihood
    this_marg = ... # compute the log of Gaussian pdf
    log_marg_like.append(this_marg)
```

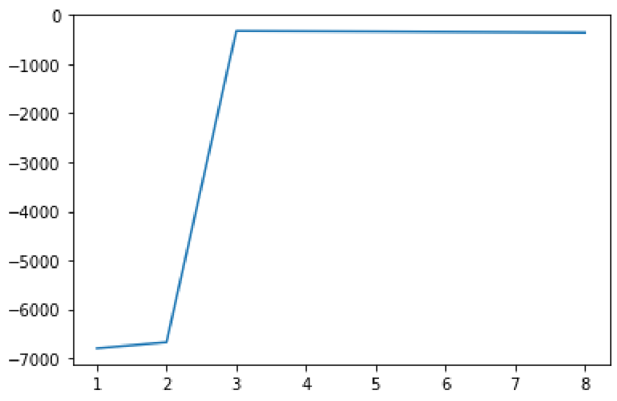
Step 4 Plot log marginal likelihood vs polynomial order

In [172]:

```
log_marg_like = np.array(log_marg_like) # convert list to array
... # plot log_marg_like vs polynomial order
```

Out[172]:

```
[<matplotlib.lines.Line2D at 0x10e249c90>]
```



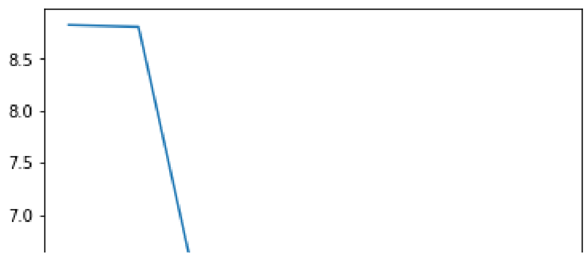
A bit diffcult to see, let's change it to a different scale, say `np.log( -log_marg_like )`

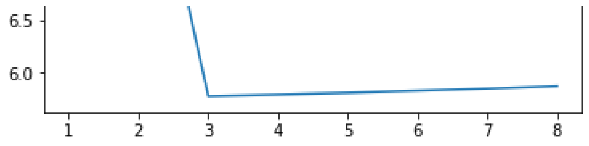
In [173]:

```
... # plot the rescaled log marginal likelihood
```

Out[173]:

```
[<matplotlib.lines.Line2D at 0x10f290f10>]
```





```
In [ ]:
```

Week 7: KNN, Naive Bayes Classifier, and ROC/AUC analysis

- #### Aims:
  - ##### Implement a KNN classifier
  - ##### Implement a Naive Bayes classifier
  - ##### Compare the two classifiers with ROC and AUC

In [318]:

```
import numpy as np
import pylab as plt
%matplotlib inline
```

Task 1: Implement a KNN classifier

Step 1: Load classification data

Download `trainx.csv` and `testx.csv` from Moodle. In trainx.csv, each row corresponds to an instance. The first two columns are the values for two features and the third is the class label. The same format is used in `testx.csv`. Load these datasets into python (numpy.loadtxt) and create an X matrix consisting of the first two columns and a t vector as the last one. Do the same for the test data so you have four objects: $\mathbf{X}$, $\mathbf{X}_{test}$, $\mathbf{t}$ and $\mathbf{t}_{test}$.

In [319]:

```
traindata = np.loadtxt('trainx.csv',delimiter=',')
testdata = np.loadtxt('testx.csv',delimiter=',')
... # X, training data
...  # t, training label
...   # X_test,
...   # t_test
```

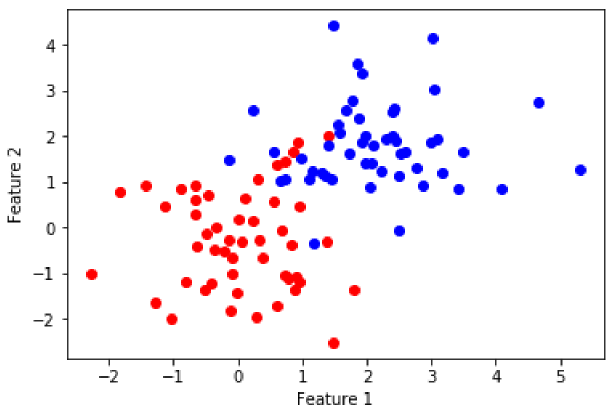
Plot the training data by class

In [320]:

```
... # plot one class in red
... # plot one class in blue
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
```

Out[320]:

```
Text(0,0.5,'Feature 2')
```



Step 2: Write a KNN function for a single test example (a row)

Implement a KNN function that takes a single test example and a value of K and returns a classification. Your function should find the K closest (see below) training points to the test point and return the majority class amongst these training points.

K closest (see below) training points to the test point and return the majority class amongst these training points.

If your training data is in a numpy array with 100 rows and 2 columns, then the distance between a test point and the ith row is given by:

```
sq_diff = (test_row - trainx[i,:])**2
dist = np.sqrt(sq_diff.sum())
```

where test*row is a row of $\mathbf{X}*{test}$. The first line creates a new vector which holds the squared difference of the two pairs of values. The second line takes the sum of these differences and then takes the square root. This is computing the Euclidean distance. Other distance metrics could also be used.

The `zip`, `sorted` and `numpy`'s `unique` can be helpful for finding the nearest neighbours.

Make sure your function returns both the predicted class and predicted score. For KNN, the score can be the percentage of votes for each class.

In [321]:

```
def knn_classifier(trainX, traint, test_data, K=3):
    ... # Step 1: computing distances from the testing data to all training data
    ... # Step 2: sort distance
    ... # Step 3: select K number of neareast neighbours
    ... # Step 4: count votes

    prediction = {}
    prediction["predicted_class"] = ... # Winning class
    prediction["predicted_score"] = ... # percentage of votes for class 0 and 1

    return(prediction)
```

Step 3: Test with a test point

In [322]:

```
test_index = 152
c = knn_classifier(trainX, traint, testX[test_index,:], K=13)
print c
```

```
{'predicted_score': array([ 0.30769231,  0.69230769]), 'predicted_class': 1.0}
```

Step 4: Plot Accuracy vs K

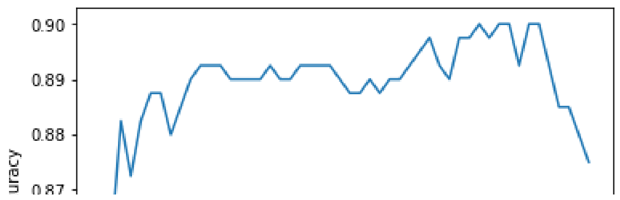
Test your function with a range of possible `K`, and compare accuracies on all test data. Plot the results.

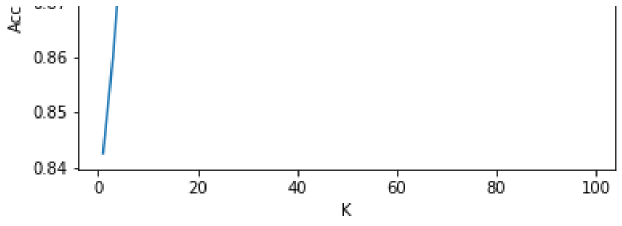
In [323]:

```
Kvals = np.arange(1,100,2)
accuracy = [] # create a list to store accuracy
for k in Kvals: # for loop over possible Ks
    ...
    for i,row in enumerate(testX):  # for loop over test data
        ...

plt.figure()
plt.plot(Kvals,accuracy)
plt.xlabel("K")
plt.ylabel("Accuracy")
```

Out[323]:

```
Text(0,0.5,'Accuracy')
```





Task 2: Implement a Naive Bayes classifier

Step 1: Write your function to train a Gaussian Naive Bayes classifier

We will use Gaussian distributions for each class. For each class, we fit a Gaussian to each dimension (by compute the mean and variance). The prior for each class will be the proportion of training data in that class.

In [324]:

```
def naive_bayes_training(trainX, traint): # Your function to training a Gaussian Naive Bayes
classifier.
                                           # Here is an example, you can do it anyway you like.
    parameters = {}
    for cl in range(2): # for loop over all classes
        ... # select all training data in class cl
        class_pars = {}
        class_pars['mean'] = ... # mean
        class_pars['vars'] = ... # variance
        class_pars['prior'] = ... # prior
        parameters[cl] = class_pars
    return(parameters)

parameters = naive_bayes_training(trainX, traint)
```

Step 2: Write your function to make prediction on testX with the trained Gaussian Naive Bayes classifier

Computing the likelihood for each class and multiplying by the prior, and normalise. Make sure you function also returns the probability of assigning the test data to both classes.

In [325]:

```
def naive_bayes_prediction(parameters, testX): # Your function to make prediction with Gaussian Na
ive Bayes
    for cl in parameters: # for loop over classes
        ...
        for i,m in enumerate(parameters[cl]['mean']): # for loop over features
            ...
    prediction = {}
    prediction['predicted_class'] = ...
    prediction['predicted_score'] = ...
    return(prediction)
```

Step 3: Make predictions

Loop through the test points,

In [326]:

```
predictions_nb = np.zeros((400, 3))
for j,tx in enumerate(testX):
    naive_bayes_results = naive_bayes_prediction(parameters, tx)
    predictions_nb[j, 0] = naive_bayes_results['predicted_class']
    predictions_nb[j, 1:] = naive_bayes_results['predicted_score']
```

Compute the accuracy of the classifier

In [327]:

```
accuracy = (predictions_nb[:,0] == testt).mean()
```

```
print (accuracy)
```

```
0.89
```

Task 3: Comparing classifiers with ROC and AUC

Step 1: Plot the ROC curve for your Gaussian Naive Bayes classifier.

In this task you can use the `roc_curve` and `roc_auc_score` function from `sklearn`

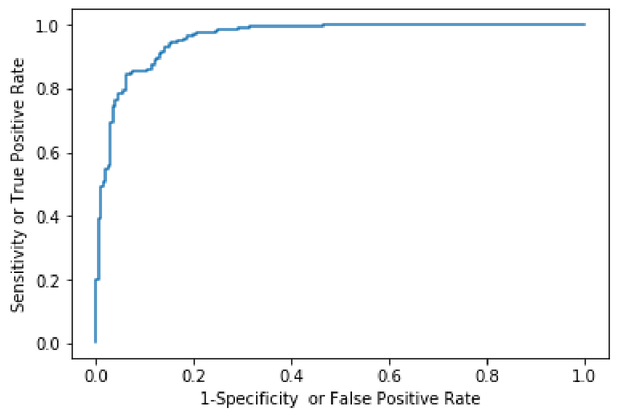
Details of `roc_curve` http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html Details of `roc_auc_score` http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

In [328]:

```
from sklearn.metrics import roc_curve, roc_auc_score
... # compute x and y coordinates of the ROC curve for Gaussian Naive Bayes using roc_curve
plt.xlabel("1-Specificity  or False Positive Rate")
plt.ylabel("Sensitivity or True Positive Rate")
```

Out[328]:

```
Text(0,0.5,'Sensitivity or True Positive Rate')
```



Step 2: Make predictions with KNN (K = 3)

In [329]:

```
predictions_knn = np.zeros((400, 3))
for j,tx in enumerate(testX):
    knn_results = knn_classifier(trainX, traint, tx, K = 3)
    predictions_knn[j, 0] = knn_results['predicted_class']
    predictions_knn[j, 1:] = knn_results['predicted_score']
```

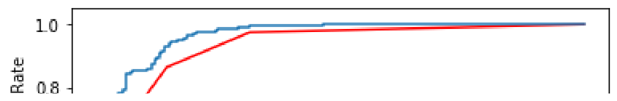
Step 3: Overlay the two ROC curves

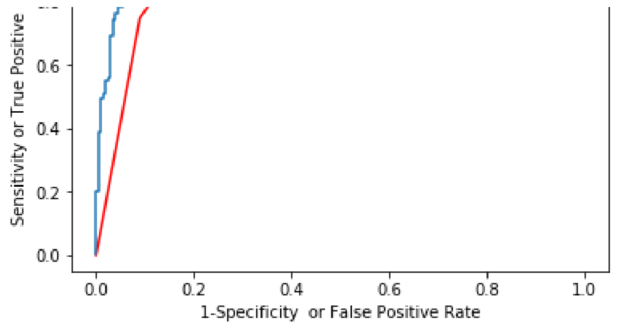
In [330]:

```
... # compute x and y coordinates of the ROC curve for KNN using roc_curve, and overlay the two RO
C curves
plt.xlabel("1-Specificity  or False Positive Rate")
plt.ylabel("Sensitivity or True Positive Rate")
```

Out[330]:

```
Text(0,0.5,'Sensitivity or True Positive Rate')
```





Step 4: Compute the AUC for the two classifiers

AUCs range between 0.5 and 1. Higher AUC indicates better classifier

In [331]:

```
auc_knn = ... # AUC for KNN
auc_nb = ... # AUC for Gaussian Naive Bayes
print(auc_knn)
print(auc_nb)
```

```
0.911
0.95985
```

In [ ]:

Week 8: Bayesian Logistic Regression

14/11/2018

- #### Aims:
  - ##### Practice training Bayesian logistic regression classifier with MAP
  - ##### Practice training Bayesian logistic regression classifier with Laplace approximation
  - ##### Practice training Bayesian logistic regression classifier with Metropolis-Hastings

In [1]:

```
import numpy as np
import pylab as plt
%matplotlib inline
```

Task 1: Bayesian logistic regression classifier with MAP

Step 1: Load data

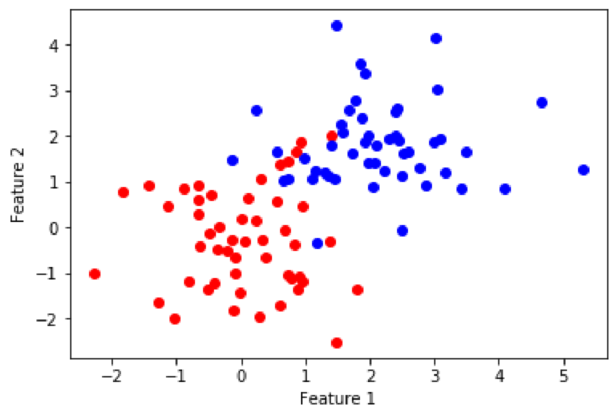
We use the same `trainx.csv` again.

In [2]:

```
traindata = np.loadtxt('trainx.csv',delimiter=',')
trainx = ...# X, training data
traint = ...# t, training label, make it a column vector

plt.figure()
...# plot the data
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
```

Out[2]:

```
Text(0,0.5,'Feature 2')
```



Step 2: Make a design matrix, just as we did in linear regression

We transform our features with polynomial basis function of order 1.

$$\mathbf{x}_n = \begin{bmatrix} x_{n,1} \\ x_{n,2} \\ 1 \end{bmatrix}$$

In [3]:

```
trainX = ...
```

Step 3: MAP solution

Make sure you understand the following equations and their roles:

Prior $p(\mathbf{w}) = >(\mathbf{0}, \frac{1}{2s^2}\mathbf{I})$:

$$\log p(\mathbf{w}) \propto -\frac{1}{2s^2}\mathbf{w}^T\mathbf{w}$$

Likelihood per data point:

$$\log p(t_n|\mathbf{x}_n, \mathbf{w}) = t_n \log(\sigma(\mathbf{w}^T\mathbf{x}_n)) + (1-t_n)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_n))$$

where $\sigma(\cdot) = \frac{1}{1+\exp(-\cdot)}$

Joint Likelihood:

$$\log p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \sum_{n=1}^{N} t_n \log(\sigma(\mathbf{w}^T\mathbf{x}_n)) + (1-t_n)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_n))$$

Joint Likelihood $\times$ Prior (Objective function):

$$\log g(\mathbf{w}; \mathbf{t}, \mathbf{X}, s^2) \propto -\frac{1}{2s^2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N} t_n \log(\sigma(\mathbf{w}^T\mathbf{x}_n)) + (1-t_n)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_n))$$

Gradient of Joint Likelihood: $\times$ Prior:

$$\frac{\partial \log g(\mathbf{w}; \mathbf{t}, \mathbf{X}, s^2)}{\partial \mathbf{w}} = -\frac{1}{s^2}\mathbf{w} + \mathbf{X}^T(\mathbf{t} - \boldsymbol{\sigma})$$

where

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix}, \qquad \boldsymbol{\sigma} = \begin{bmatrix} \sigma(\mathbf{w}^T\mathbf{x}_1) \\ \sigma(\mathbf{w}^T\mathbf{x}_2) \\ \vdots \\ \sigma(\mathbf{w}^T\mathbf{x}_N) \end{bmatrix}$$

Write your own functions for sigmoid, the objective function for Bayesian logistic regression, and its gradient

We are going to use a minimizer to get the MAP solution, so make sure your objective function and its gradient return the negative of the original one.

In [4]:

```
def sigmoid(x): # sigmoid function
    ...

def BLR_objective(w, t, X, s): # The negative objective function for Bayesian logistic regression
    w = w[:,None]
    ...


def BLR_gradient(w, t, X, s): # The gradient of the negative objective function for Bayesian
logistic regression
    w = w[:,None]
    ...
```

Step 4: Test your objective and gradient before using the minimizer

In [5]:

```
s = 1 # set s^2 = 1
w0 = np.ones((trainX.shape[1], )) # make sure w0's shape is (d,)
eps    = 1e-4 # step size
mygrad = BLR_gradient(w0, traint, trainX, s)
fdgrad = np.zeros(w0.shape)
for d in range(len(w0)): # pertub each dimension in term
    mask = np.zeros(w0.shape) # a binary mask that only allows selected dimension to change
    mask[d]   = 1
    fdgrad[d] = (BLR_objective(w0 + eps*mask, traint, trainX, s) -
                 BLR_objective(w0 - eps*mask, traint, trainX, s))/(2*eps) # definition of gradient

print("MYGRAD: ", mygrad) # my gradient output
print("FDGRAD: ", fdgrad) # numerical gradient
print("Error: ", np.linalg.norm(mygrad-fdgrad)/np.linalg.norm(mygrad+fdgrad) ) # error
```

```
('MYGRAD: ', array([  7.42532166,   0.85610828,  31.98456474]))
('FDGRAD: ', array([  7.42532165,   0.85610827,  31.98456474]))
('Error: ', 2.1128009861731074e-10)
```

Step 5: Use `scipy`'s `minimize` to estimate $\mathbf{w}_{MAP}$

$$\mathbf{w}_{MAP} = \underset{\mathbf{w}}{\operatorname{argmax}} \log g(\mathbf{w}; \mathbf{t}, \mathbf{X}, s^2)$$

In [6]:

```
import scipy.optimize as opt
w0 = np.zeros((trainX.shape[1],)) # set starting point
res = opt.minimize(BLR_objective, w0, args=(traint, trainX, s), method='BFGS',
                   jac=BLR_gradient, options={'gtol': 1e-7, 'disp': True})
w_map = res.x [:,None]
print("MAP solution: ", w_map)
```

```
Optimization terminated successfully.
         Current function value: 24.135640
         Iterations: 12
         Function evaluations: 14
         Gradient evaluations: 14
('MAP solution: ', array([[ 1.44883475],
       [ 1.16575571],
       [-2.15198994]]))
```

Step 6: Plot the decision boundaries

In [7]:

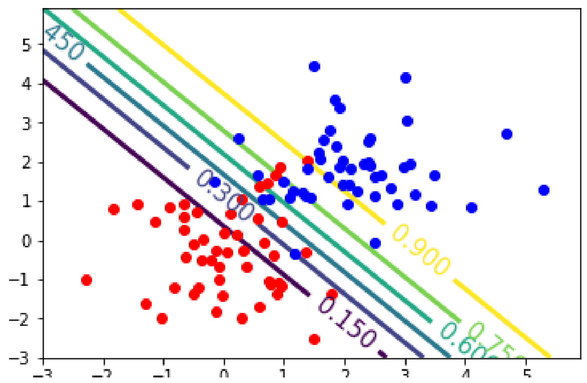
```
def BLR_prediction_map(w, x): # prediction with MAP solution
    ...

xvals = np.arange(-3,6,0.1)
Ngrid = len(xvals)
gridpred = np.zeros((Ngrid,Ngrid))
for i in range(len(xvals)):
    for j in range(len(xvals)):
        pp = np.hstack((xvals[i],xvals[j], 1))[:,None]
        gridpred[i][j] = BLR_prediction_map(w_map, pp)
```

In [8]:

```
plt.figure()
pos0 = np.where(traint==0)[0]
pos1 = np.where(traint==1)[0]
plt.plot(trainx[pos0,0],trainx[pos0,1],'ro')
plt.plot(trainx[pos1,0],trainx[pos1,1],'bo')
A = plt.contour(xvals,xvals,gridpred.T,linewidths=3)
plt.clabel(A, inline=1, fontsize=15)
```

Out[8]:

```
<a list of 6 text.Text objects>
```



Task 2: Laplace approximation

With Laplace approximation we approximate the posterior distribution by a Gaussian distribution:

$$p(\mathbf{w}|\mathbf{t},\mathbf{X},s) \approx > (\boldsymbol{\mu},\boldsymbol{\Sigma})$$

where

$$\boldsymbol{\mu} = \mathbf{w}_{MAP}, \quad \boldsymbol{\Sigma}^{-1} = -\frac{\partial^2 \log g(\mathbf{w};\mathbf{t},\mathbf{X},s^2)}{\partial\mathbf{w}\partial\mathbf{w}^T}\bigg|_{\mathbf{w}=\mathbf{w}_{MAP}} = -\frac{1}{s^2}\mathbf{I} - \mathbf{X}^T d\boldsymbol{\sigma}\mathbf{X}$$

where $d\boldsymbol{\sigma}$ is a diagonal matrix with elements on the diagonal equals to the following

$$\text{diag}(d\boldsymbol{\sigma}) = [\sigma(\mathbf{w}_{MAP}^T\mathbf{x}_1)(1-\sigma(\mathbf{w}_{MAP}^T\mathbf{x}_1)), \sigma(\mathbf{w}_{MAP}^T\mathbf{x}_2)(1-\sigma(\mathbf{w}_{MAP}^T\mathbf{x}_2)), \ldots, \sigma(\mathbf{w}_{MAP}^T\mathbf{x}_N)(1-\sigma(\mathbf{w}_{MAP}^T\mathbf{x}_N))]$$

User `numpy`'s `diag` function https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.diag.html

Step 2: Compute the mean and covariance matrix of $>(\boldsymbol{\mu},\boldsymbol{\Sigma})$

In [9]:

```
lap_mean = w_map

def compute_lap_cov(w, X, s): # your function to compute the covariance matrix of the Gaussian
distribution in Laplace approximation
    w = w[:,0]
    ...

lap_cov =  compute_lap_cov(w_map, trainX, s)

print(lap_cov)
```

```
[[ 0.13501075 -0.02320111 -0.09023716]
 [-0.02320111  0.09957826 -0.04752084]
 [-0.09023716 -0.04752084  0.20773268]]
```

Step 2: Generate lots of samples from $>(\boldsymbol{\mu},\boldsymbol{\Sigma})$

In [10]:

```
np.random.seed(1)
n_samps = 1000
w_samps = ... # draw 1000 samples from N(mu, sigma)
w_samps.shape
```

Out[10]:

```
(1000, 3)
```

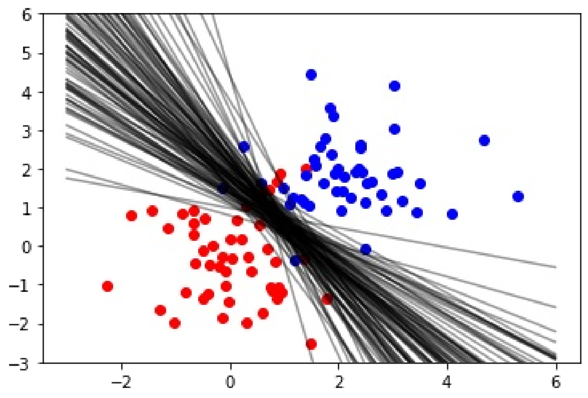
Step 3: Plot some of the sampled decision boundaries

In [11]:

```
plt.figure()
pos0 = np.where(traint==0)[0]
pos1 = np.where(traint==1)[0]
plt.plot(trainx[pos0,0],trainx[pos0,1],'ro')
plt.plot(trainx[pos1,0],trainx[pos1,1],'bo')

xlims = np.array([-3,6])
for i in range(100):
    this_w = w_samps[i,:]
    ylims = ... # Decision boundry at w^T*x = 0
    plt.plot(xlims,ylims,'k',alpha=0.4)

plt.ylim((-3,6))
```

Out[11]:

```
(-3, 6)
```



Step 4: Average over them to create the contours

In [12]:

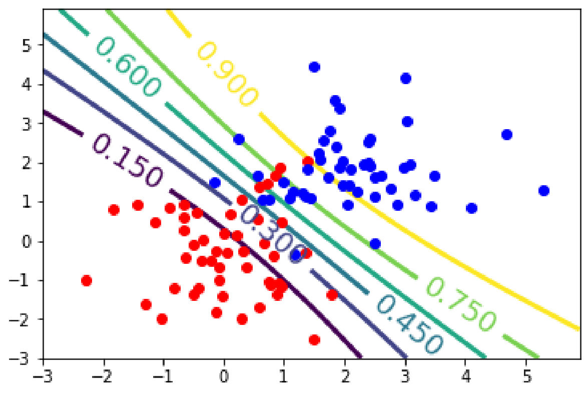
```
def BLR_prediction_lap(w_samps, x): # prediction with samples from the Laplace approximation
    ...

gridpred = np.zeros((Ngrid,Ngrid))
for i in range(len(xvals)):
    for j in range(len(xvals)):
        pp = np.hstack((xvals[i],xvals[j],1))[:,None]
        gridpred[i][j] = BLR_prediction_lap(w_samps, pp)
```

In [13]:

```
plt.figure()
pos0 = np.where(traint==0)[0]
pos1 = np.where(traint==1)[0]
plt.plot(trainx[pos0,0],trainx[pos0,1],'ro')
plt.plot(trainx[pos1,0],trainx[pos1,1],'bo')
A = plt.contour(xvals,xvals,gridpred.T,linewidths=3)
plt.clabel(A, inline=1, fontsize=18)
```

Out[13]:

```
<a list of 6 text.Text objects>
```



Task 3: Metropolis-Hastings sampling

Step 1: Implement the MH algorithm

In [14]:

```
np.random.seed(1)

w = np.zeros_like(w_map) # starting point
n_samps = 10000 # number of samples
w_samps = np.zeros((n_samps,w.shape[0])) # preallocate samples
w_samps = np.zeros((n_samps,w.shape[0])) # preallocate samples
old_like = ...# This time the orginal objective function not the negative one

for i in range(n_samps):

    # Propose a sample
    w_new =...# add standard normally distributed noise to w
    new_like =... # compute the orginal objective at w_new

    # Accept/Reject proposed sample
    if ...:
        w = w_new
        old_like = new_like
    w_samps[i,:] = w.T
```

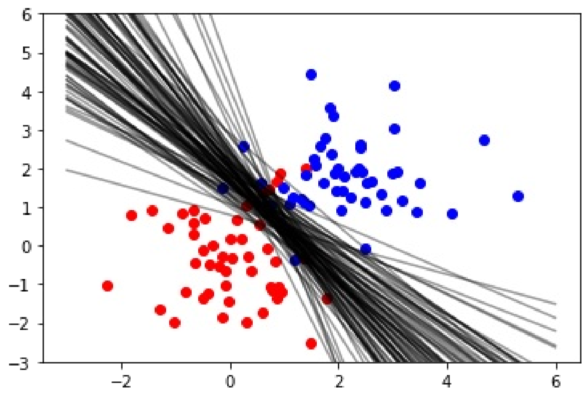
Step 2: Plot some of the sampled decision boundaries

In [18]:

```
plt.figure()
pos0 = np.where(traint==0)[0]
pos1 = np.where(traint==1)[0]
plt.plot(trainx[pos0,0],trainx[pos0,1],'ro')
plt.plot(trainx[pos1,0],trainx[pos1,1],'bo')
xlims = np.array([-3,6])
for i in range(100):
    this_w = w_samps[2+np.random.randint(n_samps),:]
    ylims = ... # Decision boundry at w^T*x = 0
    plt.plot(xlims,ylims,'k',alpha=0.4)

plt.ylim((-3,6))
```

Out[18]:

```
(-3, 6)
```



Step 3: Average over them to create the contours

In [19]:

```
def BLR_prediction_mh(w_samps, x): # The function should be the same with prediction with Laplace
approximation
    ...

gridpred = np.zeros((Ngrid,Ngrid))
for i in range(len(xvals)):
    for j in range(len(xvals)):
        pp = np.hstack((xvals[i],xvals[j],1))[:,None]
        gridpred[i][j] = BLR_prediction_mh(w_samps, pp)
```

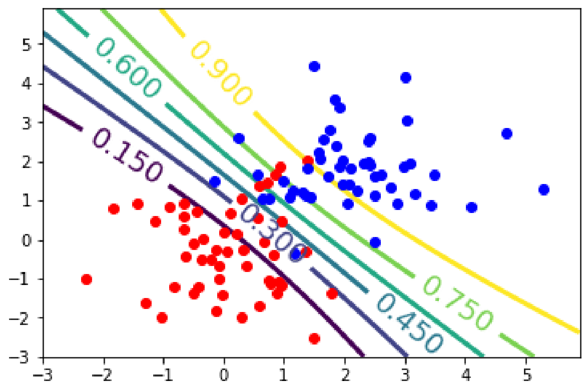
In [20]:

```
plt.figure()
pos0 = np.where(traint==0)[0]
```

```
pos1 = np.where(trainl==1)[0]
plt.plot(trainx[pos0,0],trainx[pos0,1],'ro')
plt.plot(trainx[pos1,0],trainx[pos1,1],'bo')
A = plt.contour(xvals,xvals,gridpred.T,linewidths=3)
plt.clabel(A, inline=1, fontsize=18)
```

Out[20]:

```
<a list of 6 text.Text objects>
```



In [ ]:

Week 9: Support Vector Machines and Kernels

Aims

To experiment with an SVM classifier, gaining an understanding of the effects of varying the different parameters.

Tasks

- ##### Download simplesvm.py from Moodle. This is an implementation of an SVM using the SMO algorithm (https://en.wikipedia.org/wiki/Sequential_minimal_optimization). Place this in the same directory as all this notebook.
- ##### Run the code and observe the predictive contours obtained. Experiment with varying C, as well as the kernel type (linear or rbf) and observe how the predictive contours change.
- ##### In the first cell there is some code commented out that adds some noise to the labels (i.e. changes some classes). Uncomment this and run the notebook again. You should now see large effects from varying C. Note that if you make C too large now, it might take the optimiser a long time to converge.

To avoid additional dependencies, this uses a very inefficient SVM optimiser!

In [11]:

```
import numpy as np
import pylab as plt
import simplesvm
%matplotlib inline
%load_ext autoreload
%autoreload 2

# Generate some data
trainx = np.vstack((np.random.randn(50,2),np.random.randn(50,2)+4))
traint = np.hstack((-1*np.ones(50,),np.ones(50,)))[:,None]


# Uncomment this if you want to add noise (last bit of exercise)
# for i in range(3):
#     pos = np.random.randint(50)
#     if traint[pos] == -1:
#         traint[pos] = 1
#     else:
#         traint[pos] = -1
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

The following creates the SVM object. You can vary the kernel type (linear or rbf), the kpar (only makes a difference for rbf) and C

In [12]:

```
svm = simplesvm.SimpleSVM(trainx,traint,kernel='rbf',kpar = 0.5,C=100.0)
```

Now call the optimiser to train the SVM

In [13]:

```
svm.smo_optimise()
```

The following block creates a grid of data points and then evaluates the SVM function at each point. We can then draw contours.

In [14]:

```
xvals = np.arange(-3,6,0.1)
Ngrid = len(xvals)
gridpred = np.zeros((Ngrid,Ngrid))
for i in range(len(xvals)):
    for j in range(len(xvals)):
        pp = np.hstack((xvals[i],xvals[j]))
        pp = np.hstack((xvals[i],xvals[j]))
        gridpred[i][j] = svm.test_predict(pp)
```

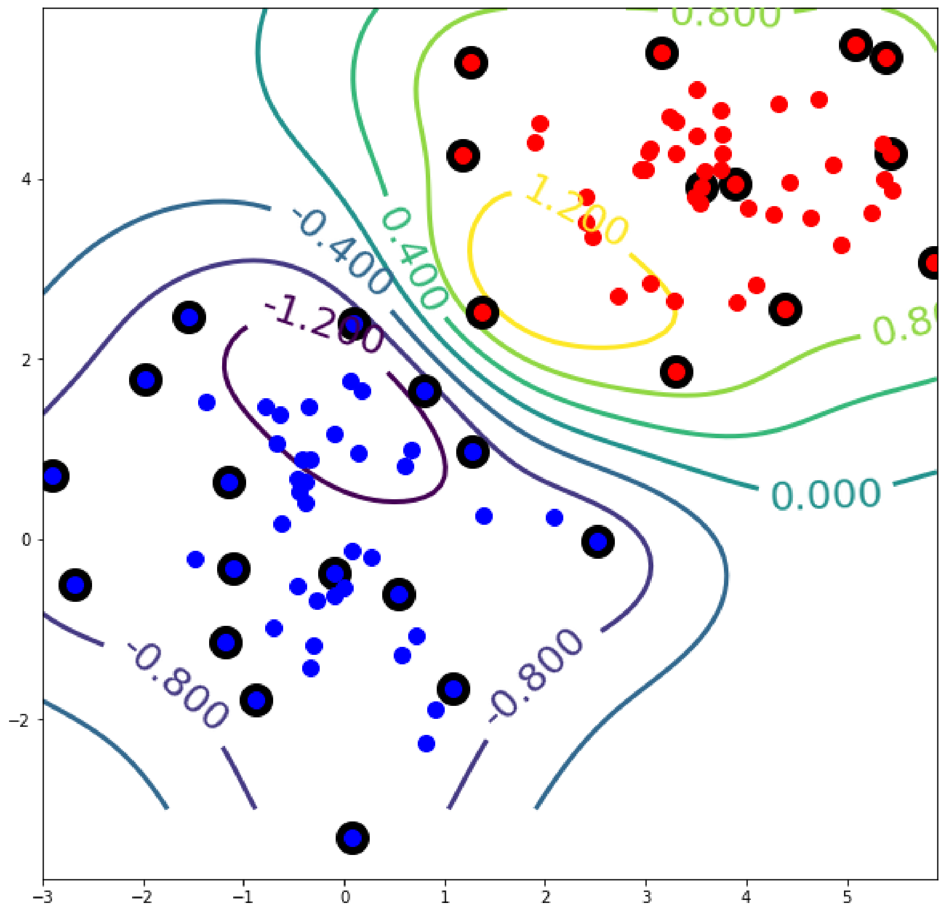
Plot the data and the contours. The support vectors are highlighted.

In [15]:

```
plt.figure(figsize=(10,10))
c0 = np.where(traint==-1)[0]
c1 = np.where(traint==1)[0]
sv = np.where(svm.alpha>1e-6)[0]

plt.plot(trainx[sv,0],trainx[sv,1],'ko',markersize=20)
plt.plot(trainx[c0,0],trainx[c0,1],'bo',markersize=10)
plt.plot(trainx[c1,0],trainx[c1,1],'ro',markersize=10)
A = plt.contour(xvals,xvals,gridpred.T,linewidths=3)
plt.clabel(A, inline=1, fontsize=25)
```

Out[15]:

```
<a list of 9 text.Text objects>
```



Look at the alpha values

In [17]:

```
print np.hstack((traint,svm.alpha))
```

```
[[-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.72692479]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.24353429]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.23002905]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          1.66250168]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.20515286]
 [-1.          0.        ]
 [-1.          0.09232975]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.89173393]
 [-1.          0.00255958]
 [-1.          0.0295245 ]
 [-1.          0.55963724]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.8140894 ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.45434889]
 [-1.          0.        ]
 [-1.          0.        ]
 [-1.          0.48347954]
 [-1.          0.42801444]
 [-1.          0.04362808]
 [-1.          0.53117796]
 [-1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.17334537]
 [ 1.          0.17206408]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          2.27379327]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.3214505 ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.30274326]
 [ 1.          0.82342018]
 [ 1.          0.        ]
 [ 1.          0.0289077 ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.81411521]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.70614466]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.88676676]
 [ 1.          0.79012495]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.10584085]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]
 [ 1.          0.        ]]
```

Week 10 K-means lab

Aim

- #### To implement the K-means clustering algorithm.

Tasks and instructions

- #### The first couple of cells generate a dataset with clear cluster structure
- #### Stored the data is in a 60x2 matrix, X
- #### After the data is plotted, the various things needed by K-means are created: the number of clusters is stored in a variable `K`, and `z` is defined as an $N \times K$ matrix that will store the current cluster memberships of the N points (z_{nk} = 1 if object n is in class k).
- #### Your task is to write the next cell which should run the K-means iterations. Comments have been provided to help.
- #### Once written, try experimenting with different values of K, or change the data to see how the algorithm behaves.

In [1]:

```
import numpy as np
import pylab as plt
%matplotlib inline
```

Geneate some data with three clear clusters

In [2]:

```
np.random.seed(1234)
cluster_means = [[0,0],[4,4],[-4,4]]
n_data = 40 # Number in each cluster
X = np.empty(shape=(0,2))
for i,m in enumerate(cluster_means):
    X = np.vstack((X,np.random.randn(n_data,2) + np.tile(m,(n_data,1))))
```

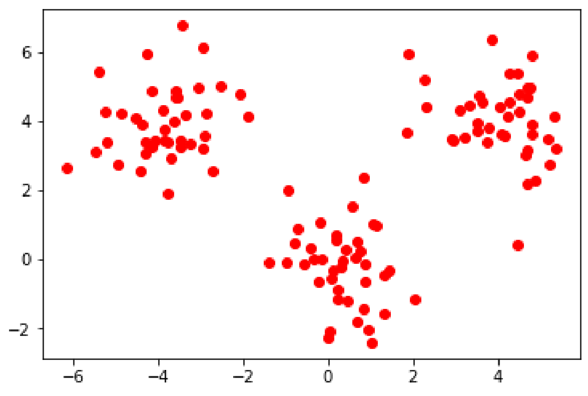
Plot the data

In [3]:

```
plt.figure()
plt.plot(X[:,0],X[:,1],'ro')
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x1053ef050>]
```



Initialise the things needed for k-means

In [4]:

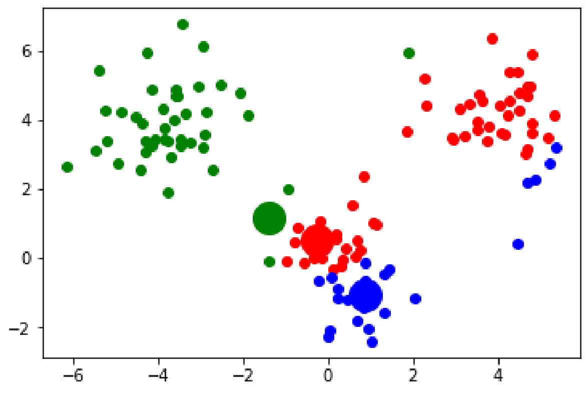
```
# set K
```

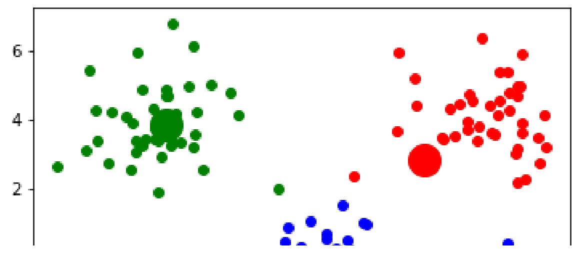
```
K = 3
# Initialise the means
mu = np.random.randn(K,2)
# Set maximum number of iterations
max_its = 20
N = len(X)
z = np.zeros((N,K))
oldz = np.ones((N,K)) # just to make sure it is different from z in iteration 1

# Colours for plotting - if you set K bigger than the
# length of this, it'll crash
cols = ['ro','bo','go','yo','mo','ko']
```

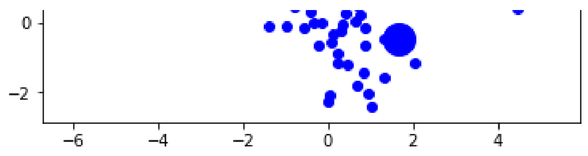
Run the algorithm - plotting the state at each iteration

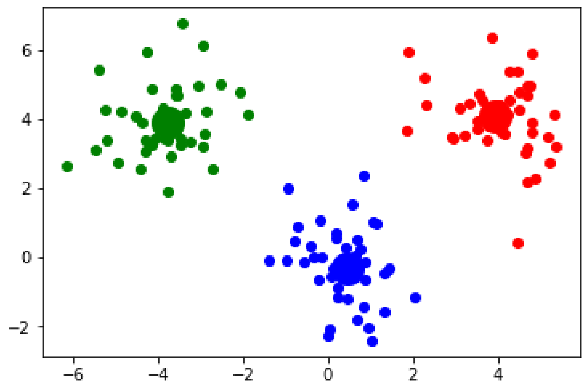
In [5]:

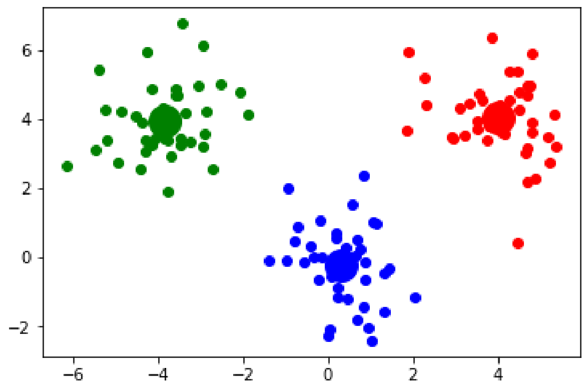
```
for it in range(max_its):


    # Assign each point to its closest mean
    # assignments are stored in z


    # ADD CODE HERE

    # Plot the status of the algorithm
    # The data are coloured according to the memberships in z
    # and the means are plotted in the same colours with larger symbols
    plt.figure()
    for k in range(K):
        plt.plot(X[z[:,k]==1,0],X[z[:,k]==1,1],cols[k])
        plt.plot(mu[k,0],mu[k,1],cols[k],markersize=20)


    # Check if anything has changes
    changes = (np.abs(z - oldz)).sum()
    if changes == 0:
        break
    # Update the means

    # ADD CODE HERE


    # Make a deep copy of z...
    oldz = np.copy(z)
```











In [6]:

```
it
```

Out[6]:

```
3
```

In [ ]:

```

```