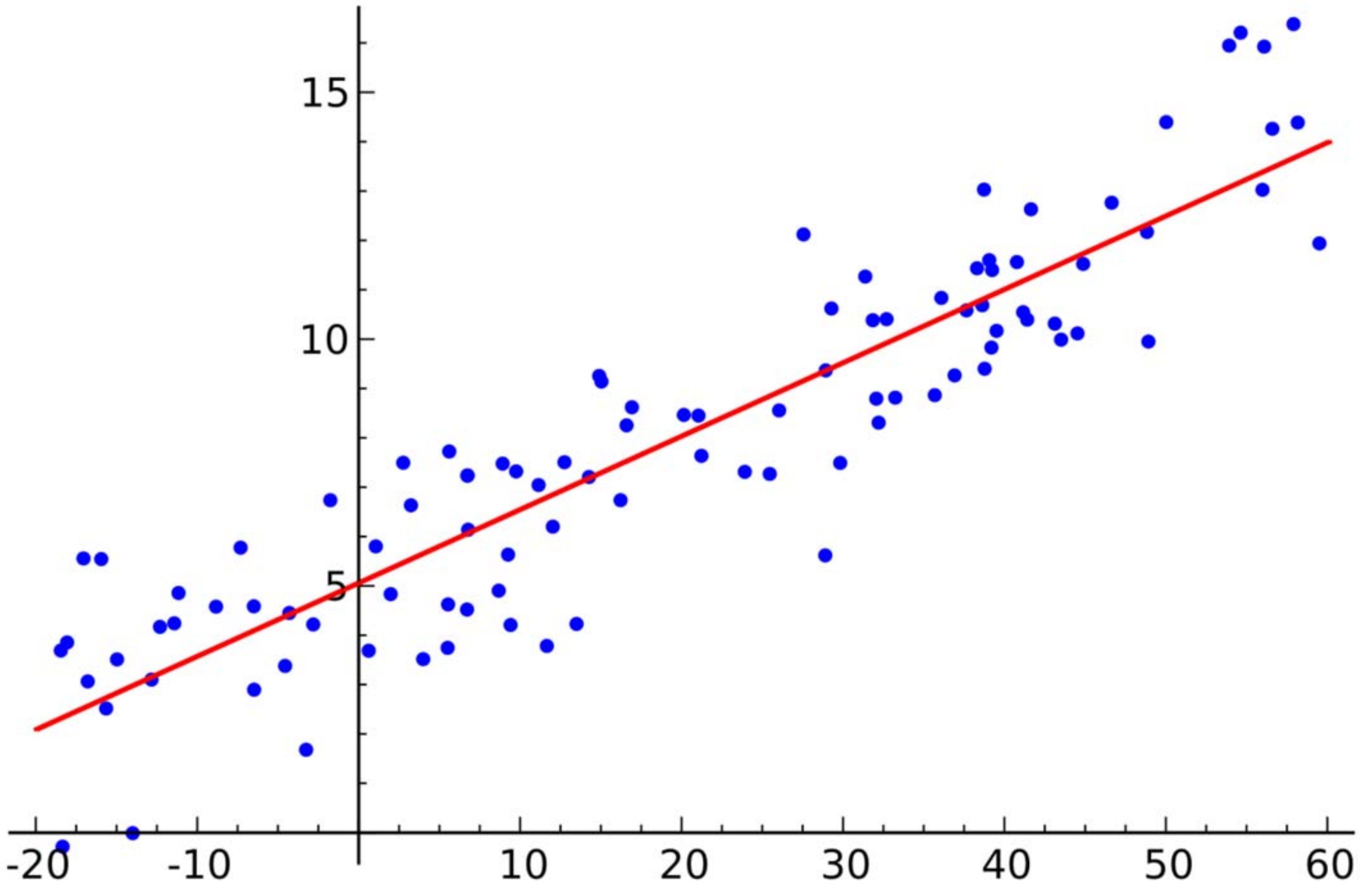
Iteratively Reweighted Least Square Method with python

Minchul Kim April 12, 2017



Iteratively Reweigthed Least Square

This approach is used when you want to maximize an objective function that involves a parameter that has a complex form that is not differente with.

Personally I have seen it used in logistic regression,Gaussian Process, and Multivariate Linear regression in Repetitive measurement statistics.

Basic Idea: Weighted Least Square

Consider the n given data,

$$(y_i, X_i) \in (\mathbb{R}^1, \mathbb{R}^p)$$

and

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 - X_1 - - \\ 1 - X_2 - - \\ 1\dots\dots\dots\dots\dots \\ 1 - X_n - - \end{bmatrix} \begin{bmatrix} \beta_1 \\ \dots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \dots \\ \epsilon_n \end{bmatrix}$$

is the model.

In Ordinary Least square model, it was assumed that

$$\begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \dots \\ \epsilon_n \end{bmatrix} \sim N\left(\begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \begin{bmatrix} \sigma^2 & 0 & 0 & 0 \\ 0 & \sigma^2 & 0 & 0 \\ 0 & 0 & \sigma^2 & 0 \\ 0 & 0 & 0 & \sigma^2 \end{bmatrix}\right)$$

But now in Weigthed Least Square, we are giving a structure to the error.

$$\begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \dots \\ \epsilon_n \end{bmatrix} \sim N\left(\begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_n^2 \end{bmatrix}\right)$$

where σ_i may not be same. (Of course we can give the structure other than a diagnoal form. but it is not the scope of this article. Instead of WLS, it is called Generalized Least Square)

So the question is how to solve for β when ϵ has a different σ_i^2 ?

getting β in OLS

In OLS, where $var(\epsilon) = \sigma^2 I$, We are finding β which maximizes the likelihood of Y. The likelihood is given as

$$L(\beta) = \frac{1}{\sqrt{2\pi}}\sigma^{-2p}\exp\left[-\frac{1}{2}(\mathbf{Y} - \mathbf{X}\beta)^T[\sigma^2\mathbf{I}]^{-1}(\mathbf{Y} - \mathbf{X}\beta)\right]$$

- (In using this likelihood, I am assuming that $\epsilon \sim MVN$, but it can be constructed without this normal assumption. But I will just forgo the more rigorous derivation.)

Since only the exponential part depends on the β, we can try to minimize the term inside the exp without the negative term.

$$\hat{\beta} = \arg\min_{\beta}(\mathbf{Y} - \mathbf{X}\beta)^T[\sigma^2\mathbf{I}]^{-1}(\mathbf{Y} - \mathbf{X}\beta) = \arg\min_{\beta} SS(\beta)$$
$$\begin{aligned} SS(\beta) &= (\mathbf{Y} - \mathbf{X}\beta)^T[\sigma^2\mathbf{I}]^{-1}(\mathbf{Y} - \mathbf{X}\beta) \\ &= \sigma^{-2}[\mathbf{Y}^T\mathbf{Y} - 2\mathbf{Y}^T(\mathbf{X}\beta) + (\mathbf{X}\beta)^T(\mathbf{X}\beta)] \end{aligned}$$

As you can see below, when you are taking the derivative of SS with respect to β and setting it equal to 0, σ^2 cancels out and we do not need to know σ^2 when we are getting β

$$\begin{aligned} \frac{dSS}{d\beta} &= \sigma^{-2}[-2\mathbf{X}^T\mathbf{Y} + 2(\mathbf{X}^T\mathbf{X})\beta] = 0 \\ &= [-2\mathbf{X}^T\mathbf{Y} + 2(\mathbf{X}^T\mathbf{X})\beta] = 0 \\ \hat{\beta} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \end{aligned}$$

getting β in WLS

In WLS, where variance of ϵ is$var(\epsilon) = V \neq \sigma^2 I$.

$$V = \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_n^2 \end{bmatrix}$$
$$\hat{\beta} = \arg\min_{\beta}(\mathbf{Y} - \mathbf{X}\beta)^T V^{-1}(\mathbf{Y} - \mathbf{X}\beta) = \arg\min_{\beta} WSS(\beta)$$
$$\begin{aligned} WSS(\beta) &= (\mathbf{Y} - \mathbf{X}\beta)^T V^{-1}(\mathbf{Y} - \mathbf{X}\beta) \\ &= \mathbf{Y}^T V^{-1}\mathbf{Y} - 2\mathbf{Y}^T V^{-1}(\mathbf{X}\beta) + (\mathbf{X}\beta)^T V^{-1}(\mathbf{X}\beta) \end{aligned}$$

When we are taking the derivative of WSS with respect to β and setting it equal to 0, V does not cancels out and we need to know V.

$$\begin{aligned} \frac{dWSS}{d\beta} &= -2\mathbf{X}^T V^{-1}\mathbf{Y} + 2(\mathbf{X}^T V^{-1}\mathbf{X})\beta = 0 \\ \hat{\beta} &= (\mathbf{X}^T V^{-1}\mathbf{X})^{-1}\mathbf{X}^T V^{-1}\mathbf{Y} \end{aligned}$$

getting V in WLS

Without the OLS assumption, we need to estimate V as well. How would you estimate this weights? Introduction to Linear Regression Analysis by Douglas C. Montgomery, Elizabeth A. Peck, G. Geoffrey Vining. explains few of them.

1. Technically you would have to know V in advance.
2. Just use $\hat{\sigma_i^2} = \hat{\sigma^2} x_i$ where $\hat{\sigma^2}$ is the sample covariance in the OLS.

I personally think

$$V \propto diag(\mathbf{Y} - \mathbf{X}\hat{\beta})(\mathbf{Y} - \mathbf{X}\hat{\beta})^T$$

is an estimator that in line with the idea of iteratively reweighted least square. Read on below. (I have not tested...it would be very unstable)

Note that V only needs to be known upto a proportion if our interest is only in finding β, as the proportionality terms in V are canceled out in $\hat{\beta}$

Iteratively Reweighted Least Square

In the WLS, the unknowns V and β arise together. And the basic idea of Iteratively Reweigthed Least Square is that you are going to update one given the other. The key is finding out the the right formula for

1. V that depends on β
2. β that depends on V

Then either one would be initialized and updated iteratively. For the simple WLS setup,

$$\hat{\beta} = (\mathbf{X}^T V^{-1} \mathbf{X})^{-1} \mathbf{X}^T V^{-1} \mathbf{Y}$$
$$\hat{V} \propto diag(\mathbf{Y} - \mathbf{X}\hat{\beta})(\mathbf{Y} - \mathbf{X}\hat{\beta})^T$$

is an example.

Using IRLS to solve Logistic Regression

Model specification: $\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$ where $y_i \in \{0, 1\}$.

$$X = \begin{bmatrix} 1 - X_1 - - \\ 1 - X_2 - - \\ 1 \dots\dots\dots\dots\dots \\ 1 - X_n - - \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \dots \\ \beta_p \end{bmatrix} \text{ where } X_i \in \mathbb{R}^p$$

We assume that y_i are generated from a bernoulli distribution with a probability p_i that depends on X_i

$$y_i \sim ber(p_i) = p^{y_i}(1 - p_i)^{1-y_i}$$

We would like to find an estimator of β such that

$$\log \frac{p_i}{1 - p_i} = \beta^T X_i + \epsilon_i$$
$$<=> p_i = \frac{e^{\beta^T X_i + \epsilon_i}}{1 + e^{\beta^T X_i + \epsilon_i}}$$

One justification for coming up with the model is we want to map the outcome of the $\beta^T X_i + \epsilon_i$ to be between 0 and 1

In a matrix form, we could write,

$$\log \frac{p}{1 - p} = X\beta + \epsilon$$
$$<=> \begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_n \end{bmatrix} = p = \frac{e^{\beta^T X + \epsilon}}{1 + e^{\beta^T X + \epsilon}}$$

where $\begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_n \end{bmatrix} \in \mathbb{R}^n$

As you can see, $p(X_i)$ takes X_i as an input and maps the input to a number between 0 and 1.

framing Logistic as WLS problem.

We can think of this as a problem of weighted least square where

Logistic Setting
$\log \frac{p}{1-p} = X\beta$

$p = \sigma(X\beta)$

And $Y \sim binom(1, p)$

How do we find such β?

As done in OLS and WLS, we fit the model using the independent X and observed Y.

But here it is important to note that Y is either 0 or 1. And when Y is mapped by the logistic function $\log \frac{y}{1-y}$, then it is either $+\infty$ or $-\infty$

work around : taylor expansion approximation

This is the key to framing the logistic regression into the WLS setting.

We substitute $\log \frac{y}{1-y}$ with the taylor expansion of it.

Let $g(y) = \log \frac{y}{1-y}$.

$$g(y) \approx g(\sigma(X\beta)) + (y - \sigma(X\beta))g'(\sigma(X\beta)) := g^*(y)$$

We chose to taylor expand around $\sigma(X\beta)$ because $\sigma(X\beta)$ is a probability that should be close to y. Note that this derivative is not with respect to β but a function parameter p. So ther is no chain rule involved.

Now, we see that $g^*(y)$ has an expectation = $X\beta$ and variance = $g'(X\beta)^2 var(y)$.

Since $y \sim ber(p)$, $var(y) = p(1-p)$

The model is

$$\log\frac{Y}{1-Y} \approx g^*(Y) = \beta^T X + \epsilon$$

Iteratively update

By the framework of WLS,

$$\hat{\beta} = (\mathbf{X}^T V^{-1}\mathbf{X})^{-1}\mathbf{X}^T V^{-1} g^*(Y)$$

$$V = \begin{bmatrix} g'(\sigma(X_1\beta))^2 p_1(1-p_1) & 0 & 0 & 0 \\ 0 & g'(\sigma(X_2\beta))^2 p_2(1-p_2) & 0 & 0 \\ 0 & 0 & g'(\sigma(X_i\beta))^2 p_i(1-p_i) & 0 \\ 0 & 0 & 0 & g'(\sigma(X_n\beta))^2 p_n \end{bmatrix}$$

Note

$$p_i = \frac{e^{\beta^T X_{i+\epsilon_i}}}{1+e^{\beta^T X_{i+\epsilon_i}}}$$

Then we could say

$$\hat{\beta^{(t+1)}} = (\mathbf{X}^T V^{-1(t)}\mathbf{X})^{-1}\mathbf{X}^T V^{-1(t)}[g(\sigma(X\beta^{(t)})) + (y - \sigma(X\beta^{(t)}))g'(\sigma(X\beta^{(t)}))]$$

$$\hat{V}$$

$$= \begin{bmatrix} g'(\sigma(X_1\hat{\beta}^{(t)}))^2 \frac{e^{\hat{\beta}^{T(t)} X_i}}{1+e^{\hat{\beta}^{T(i)} X_i}}(1 - \frac{e^{\hat{\beta}^{T(t)} X_i}}{1+e^{\hat{\beta}^{T(t)} X_i}}) & 0 & 0 \\ 0 & g'(\sigma(X_2\hat{\beta}))^2 p_2(1-p_2) & 0 \\ 0 & 0 & g'(\sigma(X_i\hat{\beta}))^2 p_i(1-p_i) \\ 0 & 0 & 0 & g' \end{bmatrix}$$

Comparison with Newton Raphson Approach

Usually when IRLS is explained in the logistic regression setting, often times, the derivation starts with the Newton Raptson method (Machine Learing A Probabilistic Perspective by Murphy pg250.). What is important to note is that heuristically it is the same either way.

Often times, when automatic gradient is not used, the iterative algorithm would be written as V

$$V^{-1} = diag(p-(1-p))$$

And it is because

$$\frac{dg}{gp} = \frac{1}{p(1-p)}$$

and

$$V = \left[\begin{matrix} g'(\sigma(X_1\beta))^2p_1(1-p_1) & 0 & 0 & 0 \\ 0 & g'(\sigma(X_2\beta))^2p_2(1-p_2) & 0 & 0 \\ 0 & 0 & g'(\sigma(X_i\beta))^2p_i(1-p_i) & 0 \\ 0 & 0 & 0 & g'(\sigma(X_n\beta))^2p_n \end{matrix}\right.$$

$$= \begin{bmatrix} \frac{1}{p_1(1-p_1)} & 0 & 0 & 0 \\ 0 & \frac{1}{p_2(1-p_2)} & 0 & 0 \\ 0 & 0 & \frac{1}{p_i(1-p_i)} & 0 \\ 0 & 0 & 0 & \frac{1}{p_n(1-p_n)} \end{bmatrix}$$

Detailed Derivation of the Newton Raphson Method

I would like to show the equivalence of the WLS and Newton method.

$$\begin{aligned} NLL(\beta) &= -\sum_{i=1}^{n} \log\left[p_i^{y_i}(1-p_i)^{1-y_i}\right] \\ &= -\sum_{i=1}^{n} y_i \log p_i + (1-y_i)log[1-p_i] \\ &= -\sum_{i=1}^{n} y_i \log\left(\frac{p_i}{1-p_i}\right) + \log[1-p_i] \\ &= -\sum_{i=1}^{n} y_i(B^TX_i) + \log\left[\frac{1}{1+e^{B^TX_i}}\right] \\ &= -\sum_{i=1}^{n} y_i(B^TX_i) - \log\left[1+e^{B^TX_i}\right] \end{aligned}$$

We differentiate the negative log likelihood.

$$\begin{aligned}\frac{df}{d\beta} &= -\sum_{i=1}^{n} y_i X_i - \frac{1}{1+e^{\beta^T X_i}} e^{\beta^T X_i} \\ &= -\sum_{i=1}^{n} y_i X_i - \sigma(\beta^T X_i) X_i \qquad \text{(p by 1 vector)} \\ &= -\sum_{i=1}^{n} y_i X_i - p_i X_i \\ &= \sum_{i=1}^{n} (p_i - y_i) X_i \\ &= X^T (P - Y) \\ &:= g(\beta) \qquad \text{p is a function of } \beta.\end{aligned}$$

where $\sigma()$ is a sigmoid function.

We also take the second derivative of the negative log likelihood function to get the Hessian Matrix.

$$\begin{aligned}\frac{dg(\beta)}{d\beta} &= \frac{d}{d\beta}\left[-\sum_{i=1}^{n} y_i X_i - \sigma(\beta X_i) X_i\right]_{p\times 1} \\ &= -\sum_{i=1}^{n} -\sigma(\beta X_i)(1-\sigma(\beta X_i)) X_i X_i^T \\ &= X^T W^{-1} X \\ &:= H\end{aligned}$$

where

$$W^{-1} = \begin{bmatrix} \sigma(\beta X_1)(1-\sigma(\beta X_1)) & 0 & 0 & 0 \\ 0 & \sigma(\beta X_2)(1-\sigma(\beta X_2)) & 0 & 0 \\ 0 & 0 & \sigma(\beta X_i)(1-\sigma(\beta X_i)) & 0 \\ 0 & 0 & 0 & \sigma(\beta X_n)(1-\sigma(\beta \end{bmatrix}$$

The Newton Raphson method is done by updating with the following rule:

$$\begin{aligned}\beta_{k+1} = \beta_k &= \left(\frac{d^2 NLL}{d\beta}\right)^{-1} \frac{d}{d\beta} NLL \\ &= \beta_k = H^{-1} g \\ &= \beta_k - (X^T W^{-1} X)^{-1} X^T (\sigma(X\beta) = Y) \\ &= (X^T W^{-1} X)^{-1} X^T W^{-1} (X\beta_k + W(Y - \sigma(X\beta_k)))\end{aligned}$$

And you will see that it is exactly the same as in the WLS approach.

how to take derivatives using Theano

In [1]:

```
import numpy as np
import pandas as pd
from theano import function, shared
from theano import tensor as TT
import theano
import theano.sandbox.rng_mrg
```

```
Using gpu device 0: GeForce GTX 1070 (CNMeM is disabled, cuDNN not available)
```

It is important to note that when you are taking derivaties using Theano, the function you are taking the derivative of must have a scalar output. Below are the examples of taking a derivative when the output in a scalar.

In [2]:

```
sharedX = (lambda X, name: shared(np.asarray(X, dtype=theano.config.floatX), name=name))

def dotprod(x):
    return (theano.tensor.dot(x.T, x))

def g(p):
    return TT.log(p / (1-p))

temp = sharedX([1,2],"X")

print "Dot product: ", dotprod(temp).eval()
print "gradient at (1,2): ", TT.grad(dotprod(temp), temp).eval()

prob = sharedX(0.4,"Y")
print "g(p)", g(prob).eval()
print "gradient at g(0.4): ", TT.grad(g(prob), prob).eval()
```

```
Dot product:  5.0
gradient at (1,2):  [ 2.  4.]
g(p) -0.405465185642
gradient at g(0.4):  4.16666650772
```

taking derivative when the output must be a vector.

There are times when the output must be a vector. For example, $g'(p)$ in the above WLS setup, since p is a vector, its gradient is also a vector. In such a case, we need to evaluate the gradient in a loop and aggregate the result back into a vector.

In theano, loops are carried out by the special operation called scan. This function provides an interface to do loops in theano.

scan function takes

1. function: that takes the input that needs to be updated and output that has the updated value. It could also output a dictionary.
2. outputs_info: a list of dictionary. The list contains a dictionary in the following form; {inital: variable_name}
3. sequences: a list that contains variables that are going to be used element-wise.
4. non_sequences: a list that contains variables that are going to be used as a whole.

scan function outputs

1. list of updated variables. It returns all history.
2. updates dictionary. It must be used when compiling the theano function when scan function draws random stream samples.

There are two ways of doing it.

In [3]:

```
Prob = sharedX([0.4,0.5],"Y")

# first way
J, updates = theano.scan(lambda i, Prob : TT.grad(g(Prob[i]), Prob),
sequences=TT.arange(Prob.shape[0]), non_sequences=Prob)
print J.eval()

# second way
x = TT.dvector('x')
y = x ** 2
J, updates = theano.scan(lambda i, y, x : TT.grad(y[i], x), sequences=TT.arange(y.shape[0]),
non_sequences=[y, x])
f = theano.function([x], J, updates=updates)
print f([4, 4])
```

```
[[ 4.16666651  0.        ]
 [ 0.          4.        ]]
[[ 8.  0.]
 [ 0.  8.]]
```

Note that you would need to take the diagonal elements only.

In [4]:

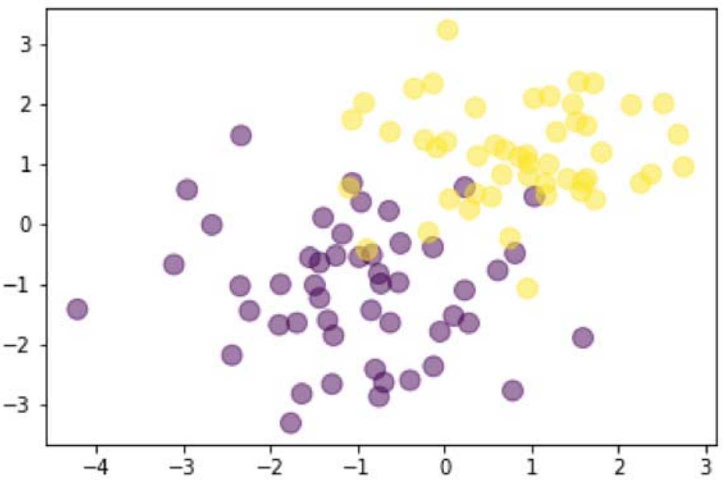
```
Prob = sharedX([0.4,0.5],"Y")

J, updates = theano.scan(lambda i, Prob : TT.grad(g(Prob[i]), Prob),
sequences=TT.arange(Prob.shape[0]), non_sequences=Prob)
print J.diagonal().eval()
```

```
[ 4.16666651  4.        ]
```

Example Data

In [13]:

```
# data generation part of the code is from the lazy programmer logistic regression lecture.

import numpy as np
import matplotlib.pyplot as plt

N = 100
D = 2

X = np.random.randn(N,D)

# center the first 50 points at (-1,-1)
X[:50,:] = X[:50,:] - np.ones((50,D))

# center the last 50 points at (1, 1)
X[50:,:] = X[50:,:] + np.ones((50,D))

# labels: first 50 are 0, last 50 are 1
T = np.array([0]*50 + [1]*50)

# add a column of ones
ones = np.array([[1]*N]).T
Xb = np.concatenate((ones, X), axis=1)

# randomly initialize the weights
w = np.zeros(D + 1)

plt.scatter(X[:,0], X[:,1], c=T, s=100, alpha=0.5)

plt.show()
```



Implementing IRLS algorithm

This will be implemented as it is in the book, Machine Learning by Kevin Murphy

In [14]:

```
from theano.tensor.nlinalg import matrix_inverse
def sigmoid(mu):
    return TT.exp(mu) / (1 + TT.exp(mu))

X = sharedX(Xb, name = "X")
beta_temp = np.random.randn(D + 1)
beta_temp[0] = np.log(np.mean(T) / (1.0 - np.mean(T)))
beta = sharedX(beta_temp, name = "beta")
Y = sharedX(T, name = "Y")


for i in range(10):
    # updating Weight matrix
    phat = sigmoid(TT.dot(X,beta))
    wi = phat * (1-phat)
    Winv = TT.nlinalg.diag(wi)

    #updating beta
    z = TT.dot(X,beta) + (Y - phat) / wi
    beta = matrix_inverse((X.T).dot(Winv).dot(X)).dot(X.T).dot(Winv).dot(z)
    print "Theano beta: " , beta.eval()
```

```
Theano beta:  [-0.10418594  1.34005606  1.89331162]
Theano beta:  [-0.1516296   1.62159991  2.19636106]
Theano beta:  [-0.15961677  1.71281934  2.29786062]
Theano beta:  [-0.15973835  1.72007632  2.30620003]
Theano beta:  [-0.15973656  1.720119    2.30625033]
Theano beta:  [-0.15973644  1.720119    2.3062501 ]
Theano beta:  [-0.15973657  1.72011888  2.3062501 ]
Theano beta:  [-0.15973648  1.72011888  2.3062501 ]
Theano beta:  [-0.15973659  1.72011864  2.30624986]
Theano beta:  [-0.15973653  1.72011876  2.30624986]
```

In [15]:

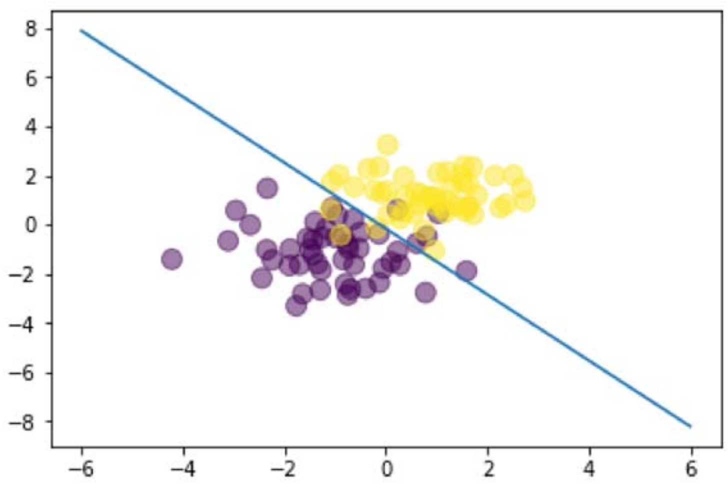
```
# numpy version
def np_sigmoid(mu):
    return np.exp(mu) / (1 + np.exp(mu))

for i in range(10):
    # updating Weight matrix
    npphat = np_sigmoid(Xb.dot(beta_temp))
    np_wi = npphat * (1- npphat)
    npWinv = np.diag(np_wi)

    #updating beta
    np_z = Xb.dot(beta_temp) + (T - npphat) / np_wi
    beta_temp = np.linalg.inv((Xb.T).dot(npWinv).dot(Xb)).dot(Xb.T).dot(npWinv).dot(np_z)
    print "numpy beta: ", beta_temp
```

```
numpy beta:  [-0.10418589  1.34005626  1.89331192]
numpy beta:  [-0.15162975  1.62160006  2.19636107]
numpy beta:  [-0.15961676  1.71281896  2.29786066]
numpy beta:  [-0.15973833  1.72007652  2.30620006]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
numpy beta:  [-0.15973655  1.72011878  2.30624994]
```

In [16]:

```
plt.scatter(Xb[:,1], Xb[:,2], c=T, s=100, alpha=0.5)

x_axis = np.linspace(-6, 6, 100)
beta_value = beta.eval()
print beta_value
y_axis = beta_value[0] + x_axis*(-beta_value[2] / beta_value[1])
plt.plot(x_axis, y_axis)
plt.show()
```

```
[-0.15973653  1.72011876  2.30624986]
```



In [17]:

```
# theano automatic gradient version

from theano.tensor.nlinalg import matrix_inverse
def sigmoid(mu):
    return TT.exp(mu) / (1 + TT.exp(mu))

def g(p):
    return TT.log(p / (1-p))



X = sharedX(Xb, name = "X")
beta_temp = np.random.randn(D + 1)
beta_temp[0] = np.log(np.mean(T) / (1.0 - np.mean(T)))
beta = sharedX(beta_temp, name = "beta")
Y = sharedX(T, name = "Y")



for i in range(10):
    # updating Weight matrix
    phat = sigmoid(TT.dot(X,beta))
    ggrad_mat, updates = theano.scan(lambda i, phat : TT.grad(g(phat)[i], phat),
                                     sequences=TT.arange(phat.shape[0]), non_sequences=
[phat])
    ggrad = ggrad_mat.diagonal()
    W = TT.nlinalg.diag( (ggrad * ggrad) * phat * (1-phat) )
    Winv = matrix_inverse(W)

    #updating beta
    wi = phat * (1-phat)
    z = g(phat) + (Y - phat) * ggrad
    beta = matrix_inverse((X.T).dot(Winv).dot(X)).dot(X.T).dot(Winv).dot(z)
    print "Theano beta automatic gradient: " , beta.eval()
```

```
Theano beta automatic gradient:  [-0.04001372  0.77342385  1.16920805]
Theano beta automatic gradient:  [-0.10400032  1.18300915  1.66813147]
Theano beta automatic gradient:  [-0.14864211  1.52157211  2.07248306]
Theano beta automatic gradient:  [-0.15981655  1.6900928   2.27066755]
Theano beta automatic gradient:  [-0.15977888  1.71938908  2.30537224]
Theano beta automatic gradient:  [-0.15973657  1.72011828  2.30624914]
Theano beta automatic gradient:  [-0.15973654  1.72011876  2.3062501 ]
Theano beta automatic gradient:  [-0.15973654  1.72011876  2.30624986]
Theano beta automatic gradient:  [-0.15973648  1.72011852  2.30624986]
Theano beta automatic gradient:  [-0.15973648  1.72011876  2.3062501 ]
```

Running Iteration

Because it takes time to move data from cpu and gpu back and forth, theano provides an efficient way to do loops.

In [25]:

```
def one_step_update(beta, W, X, Y):

    # updating Weight matrix
    phat = sigmoid(TT.dot(X,beta))
    ggrad_mat, updates = theano.scan(lambda i, phat : TT.grad(g(phat)[i], phat),
                                     sequences=TT.arange(phat.shape[0]), non_sequences=
[phat])
    ggrad = ggrad_mat.diagonal()
    W = TT.nlinalg.diag( (ggrad * ggrad) * phat * (1-phat) )
    new_Winv = matrix_inverse(W)

    #updating beta
    wi = phat * (1-phat)
    z = g(phat) + (Y - phat) * ggrad
    new_Beta = matrix_inverse((X.T).dot(Winv).dot(X)).dot(X.T).dot(Winv).dot(z)

    return [new_Beta, new_Winv], {}



# theano automatic gradient version

from theano.tensor.nlinalg import matrix_inverse
def sigmoid(mu):
    return TT.exp(mu) / (1 + TT.exp(mu))

def g(p):
    return TT.log(p / (1-p))



X = sharedX(Xb, name = "X")
beta_temp = np.random.randn(D + 1)
beta_temp[0] = np.log(np.mean(T) / (1.0 - np.mean(T)))
beta = sharedX(beta_temp, name = "beta")
Y = sharedX(T, name = "Y")
W = sharedX(np.zeros((len(T),len(T))), "W")
n_steps = 10



(all_Beta, all_V), scan_updates = theano.scan(
    one_step_update,
    outputs_info= [
        dict(initial = beta),
        dict(initial = W),
    ],
    non_sequences = [X,Y],
    n_steps= n_steps -1
    )
```

In [27]:

```
all_Beta.eval()
```

Out[27]:

```
array([[-0.89910471,  1.05932784,  3.010221  ],
       [ 0.57522774,  2.11588073,  1.84953523],
       [-0.40036905,  1.42199016,  1.92405152],
       [-0.16519095,  1.68244576,  2.12460279],
       [-0.16087598,  1.72427213,  2.22886395],
       [-0.16220255,  1.72420025,  2.27026844],
       [-0.16102445,  1.72240114,  2.28904366],
       [-0.16038504,  1.72123504,  2.29797554],
       [-0.16004953,  1.72065783,  2.30227208]], dtype=float32)
```

Home (/) Machinle Learning (../ML/index.html)
Statistics (../statistics/index.html) Others (../others/index.html)
Resources (../resources/index.html)