STATS 415 - Homework 1

Zixuan Zhu

September 2018

1. **Locally weighted linear regression**

   (a) As we can see, $y_{n*1} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$, $X_{n*d} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix}$, $W = \begin{bmatrix} w_1 & & \\ & \ddots & \\ & & w_n \end{bmatrix}$, $\beta_{d*1} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_d \end{bmatrix}$.

   Then we have: $y - X\beta = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix} \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_d \end{bmatrix} = \begin{bmatrix} y_1 - x_1^T\beta \\ \vdots \\ y_n - x_n^T\beta \end{bmatrix} = \begin{bmatrix} y_1 - \beta^T x_1 \\ \vdots \\ y_n - \beta^T x_n \end{bmatrix}$

   Thus,

$$\begin{aligned} J(\beta) &= \frac{1}{2}(y - X\beta)^T W (y - X\beta) \\ &= \frac{1}{2} \begin{bmatrix} y_1 - \beta^T x_1 \\ \vdots \\ y_n - \beta^T x_n \end{bmatrix}^T \begin{bmatrix} w_1 & & \\ & \ddots & \\ & & w_n \end{bmatrix} \begin{bmatrix} y_1 - \beta^T x_1 \\ \vdots \\ y_n - \beta^T x_n \end{bmatrix} \\ &= \frac{1}{2} \begin{bmatrix} w_1(y_1 - \beta^T x_1) \\ \vdots \\ w_n(y_n - \beta^T x_n) \end{bmatrix}^T \begin{bmatrix} y_1 - \beta^T x_1 \\ \vdots \\ y_n - \beta^T x_n \end{bmatrix} \\ &= w_1(y_1 - \beta^T x_1)^2 + \cdots + w_n(y_n - \beta^T x_n)^2 \\ &= \frac{1}{2}\sum_{i=1}^{n} w_i (y_i - \beta^T x_i)^2 \end{aligned}$$

   (b) Because W is a diagonal matrix, so $W^T = W$.

$$\begin{aligned} \nabla J &= \frac{1}{2}[(-X)^T (W^T + W)(y - X\beta)] \\ &= X^T W (X\beta - y) \end{aligned}$$

   To make $\nabla J = 0$, we need to make $X^T W(X\beta - y) = 0$, which means $\beta = (X^T W X)^{-1} X^T W y$.

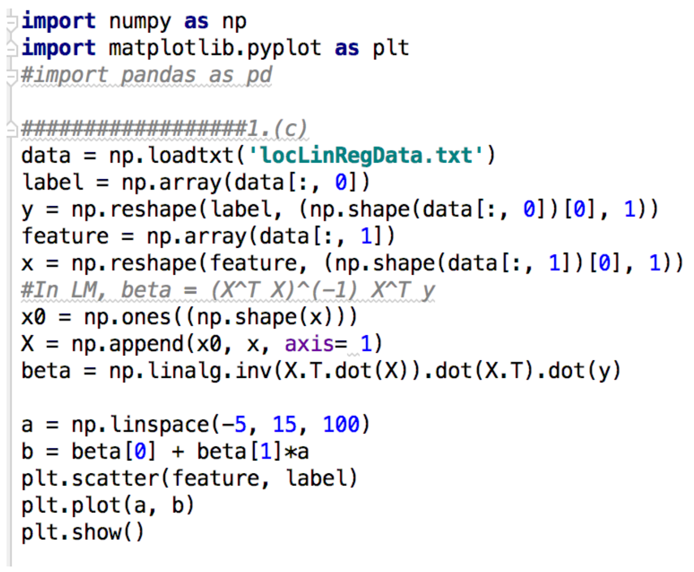
   (c) Since $y_i \mid x_i \sim (\beta^T x_i, \sigma_i^2)$, then we have $P(y_i|x_i) = \frac{1}{\sqrt{2\pi}\sigma_i} exp(-\frac{1}{2\sigma^2}(y_i - \beta^T x_i)^2$. So $L(\beta) = \prod_{i=1}^{n} P(x_i)P(y_i|x_i)$. And since $P(x_i)$ is not related to β, so we can ignore this term when doing MLE.

Let the maximum likelihood of β is $\hat{\beta}_{MLE}$,then:

$$\begin{aligned}
\hat{\beta}_{MLE} &= \arg\max_{\beta} log(L(\beta)) = \arg\min_{\beta} -log(L(\beta)) \\
&= \arg\min_{\beta} \Sigma_{i=1}^{n}[-logP(y_i|x_i) - logP(x_i)] \\
&= \arg\min_{\beta} \Sigma_{i=1}^{n} - logP(y_i|x_i) \\
&= \arg\min_{\beta} \Sigma_{i=1}^{n}[log(\sqrt{2\pi}\sigma_i) + \frac{1}{2\sigma_i^2}(y_i - \beta^T x_i)^2] \\
&= \arg\min_{\beta} \Sigma_{i=1}^{n} \frac{1}{2\sigma_i^2}(y_i - \beta^T x_i)^2 \\
&= \arg\min_{\beta} \frac{1}{2}\Sigma_{i=1}^{n} \frac{1}{\sigma_i^2}(y_i - \beta^T x_i)^2
\end{aligned}$$

Compared this maximum likelihood with the form of $J(\beta)$, we can regard $\frac{1}{\sigma_i^2}$ as w_i, so maximize $\hat{\beta}_{MLE}$ is actually equivalent as minimize $J(\beta)$ with $w_i = \frac{1}{\sigma_i^2}$. So the maximum likelihood of β is equivalent to solving a weight least squares problem with problem (b)'s conclusion.

(d) We do this problem in python, and the code is as follows:



```python
import numpy as np
import matplotlib.pyplot as plt
#import pandas as pd

##################1.(c)
data = np.loadtxt('locLinRegData.txt')
label = np.array(data[:, 0])
y = np.reshape(label, (np.shape(data[:, 0])[0], 1))
feature = np.array(data[:, 1])
x = np.reshape(feature, (np.shape(data[:, 1])[0], 1))
#In LM, beta = (X^T X)^(-1) X^T y
x0 = np.ones((np.shape(x)))
X = np.append(x0, x, axis= 1)
beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

a = np.linspace(-5, 15, 100)
b = beta[0] + beta[1]*a
plt.scatter(feature, label)
plt.plot(a, b)
plt.show()
```

Figure 1: Python code

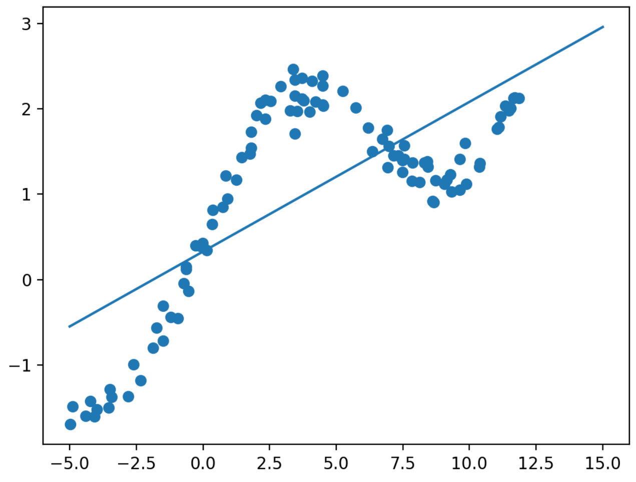
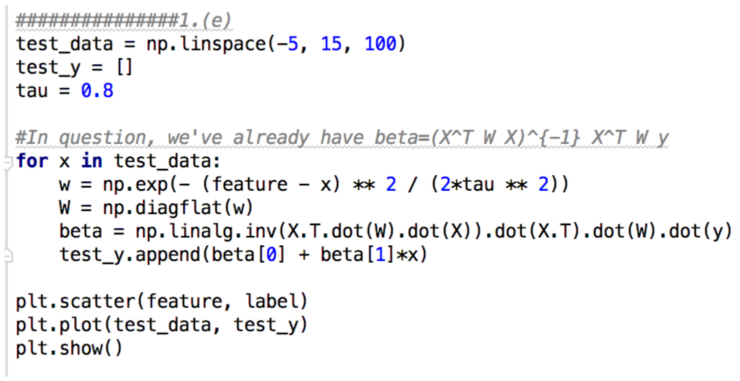


Figure 2: Plot

(e) We do this problem in python, and the code is as follows:



```
###############1.(e)
test_data = np.linspace(-5, 15, 100)
test_y = []
tau = 0.8

#In question, we've already have beta=(X^T W X)^{-1} X^T W y
for x in test_data:
    w = np.exp(- (feature - x) ** 2 / (2*tau ** 2))
    W = np.diagflat(w)
    beta = np.linalg.inv(X.T.dot(W).dot(X)).dot(X.T).dot(W).dot(y)
    test_y.append(beta[0] + beta[1]*x)

plt.scatter(feature, label)
plt.plot(test_data, test_y)
plt.show()
```

Figure 3: Python code

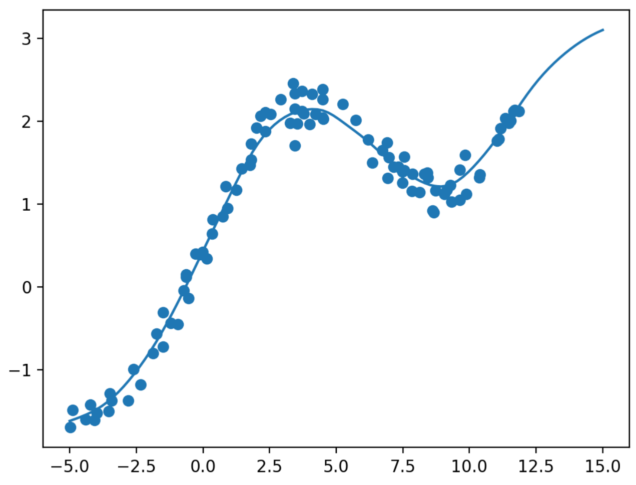
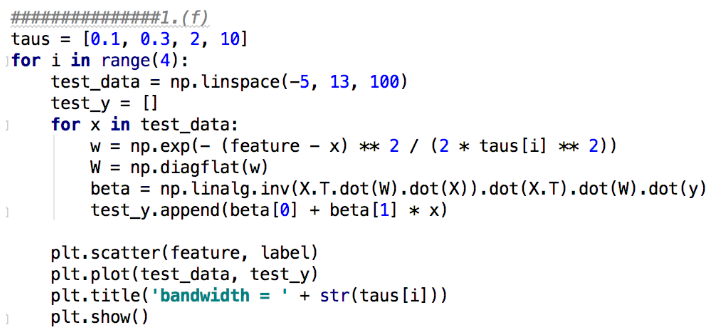


Figure 4: Plot

(f) We do this problem in python, and the code is as follows:



```
###############1.(f)
taus = [0.1, 0.3, 2, 10]
for i in range(4):
    test_data = np.linspace(-5, 13, 100)
    test_y = []
    for x in test_data:
        w = np.exp(- (feature - x) ** 2 / (2 * taus[i] ** 2))
        W = np.diagflat(w)
        beta = np.linalg.inv(X.T.dot(W).dot(X)).dot(X.T).dot(W).dot(y)
        test_y.append(beta[0] + beta[1] * x)

    plt.scatter(feature, label)
    plt.plot(test_data, test_y)
    plt.title('bandwidth = ' + str(taus[i]))
    plt.show()
```

Figure 5: Python code

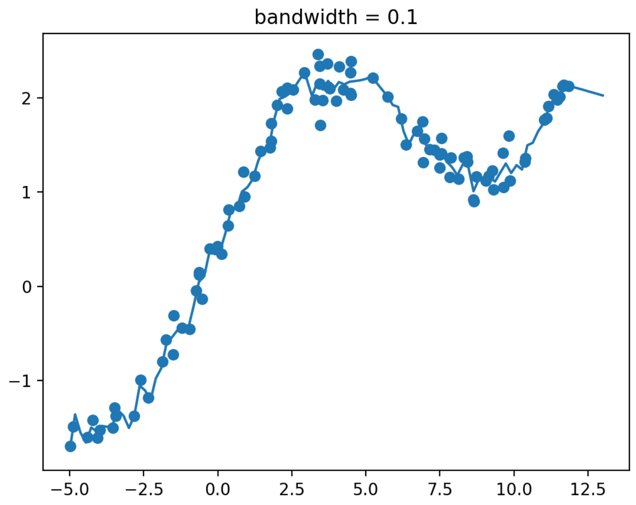


Figure 6: Plot 1

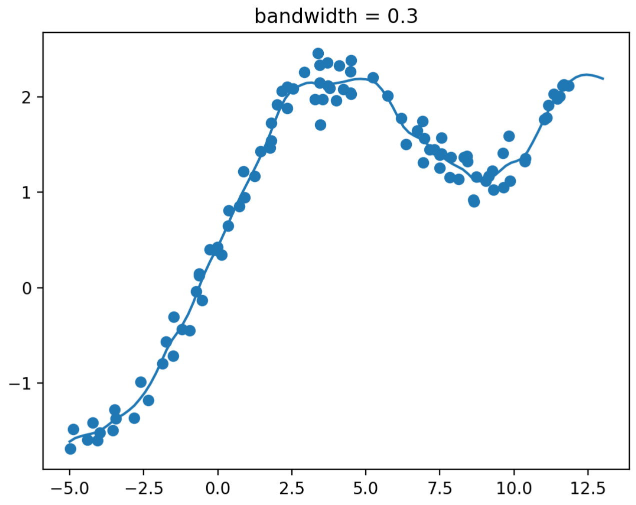


Figure 7: Plot 2

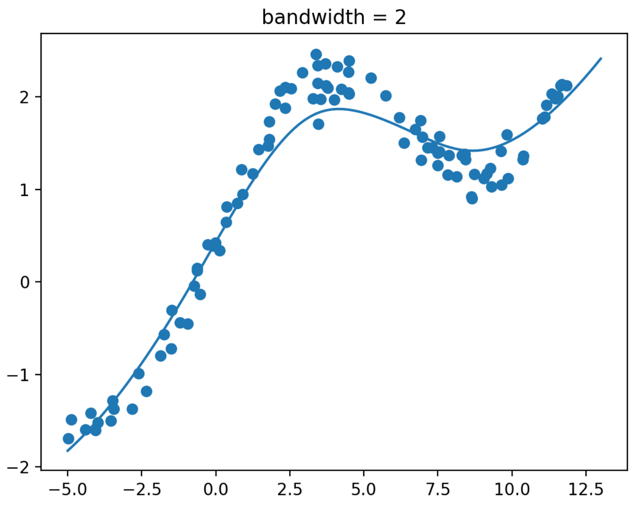


Figure 8: Plot 3

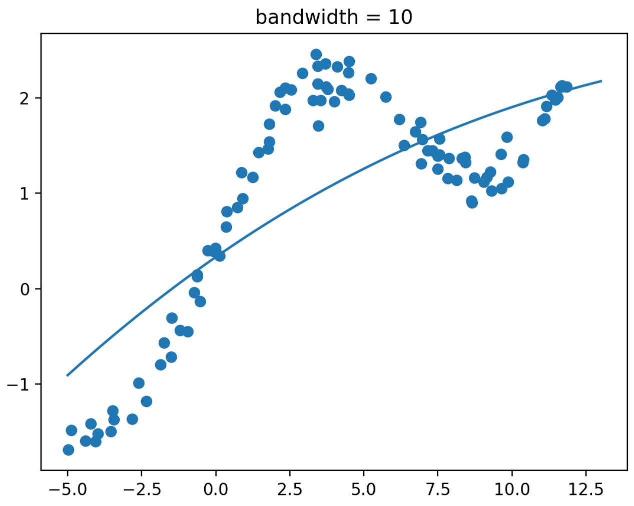
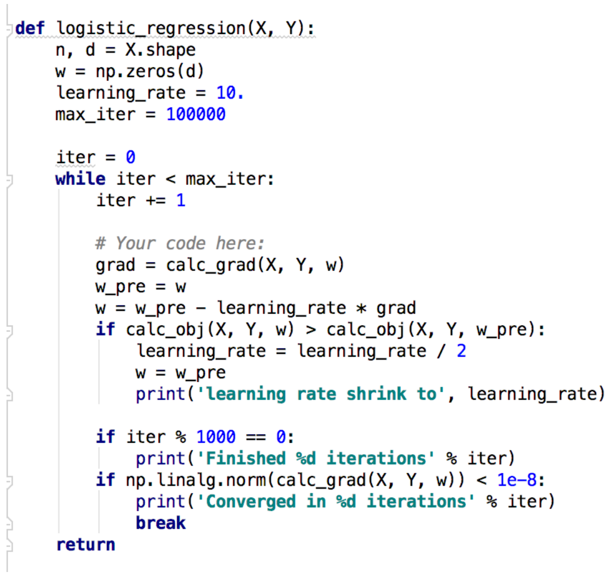


Figure 9: Plot 4

So we can see that, as τ goes bigger, the fitted regression line goes smoother, then the probability of overfitting reduces.

2. **Debugging logistic regression**

   (a) First, we implement the python code:



```
def logistic_regression(X, Y):
    n, d = X.shape
    w = np.zeros(d)
    learning_rate = 10.
    max_iter = 100000

    iter = 0
    while iter < max_iter:
        iter += 1

        # Your code here:
        grad = calc_grad(X, Y, w)
        w_pre = w
        w = w_pre - learning_rate * grad
        if calc_obj(X, Y, w) > calc_obj(X, Y, w_pre):
            learning_rate = learning_rate / 2
            w = w_pre
            print('learning rate shrink to', learning_rate)

        if iter % 1000 == 0:
            print('Finished %d iterations' % iter)
        if np.linalg.norm(calc_grad(X, Y, w)) < 1e-8:
            print('Converged in %d iterations' % iter)
            break
    return
```

Figure 10: Implement of the code

Then,the output for A is:

==== Training model on data set A ====
learning rate shrink to 5.0
Finished 1000 iterations
Finished 2000 iterations
Finished 3000 iterations
Finished 4000 iterations
⋮
Finished 21000 iterations
Finished 22000 iterations
Finished 23000 iterations
Finished 24000 iterations
Converged in 24199 iterations

The output for B is:

==== Training model on data set B ====
Finished 1000 iterations
Finished 2000 iterations
Finished 3000 iterations
Finished 4000 iterations
⋮
Finished 96000 iterations
Finished 97000 iterations
Finished 98000 iterations
Finished 99000 iterations
Finished 100000 iterations

So we can see that for A, it converges, and for B, it doesn't work since the iterations never converges.

(b) The reason why B is not converge is that B is a linearly separable set. This means that we have infinitely many hyperplane for B, i.e. the loss function can't reach its minimum. For convinience, we plot dataset A and B to see what a linearly separable set means:

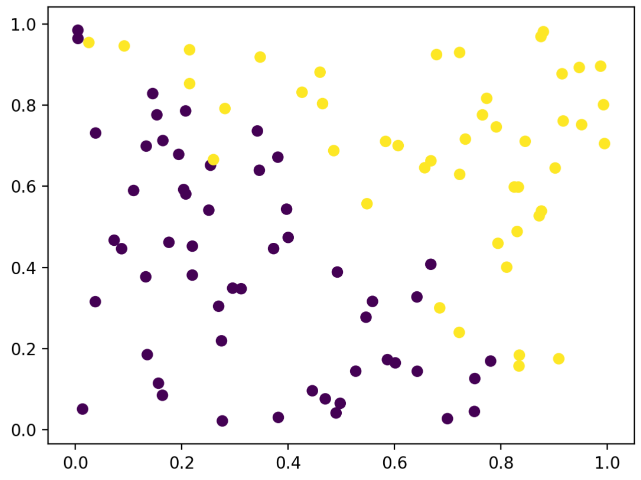


Figure 11: Scatter plot of set A

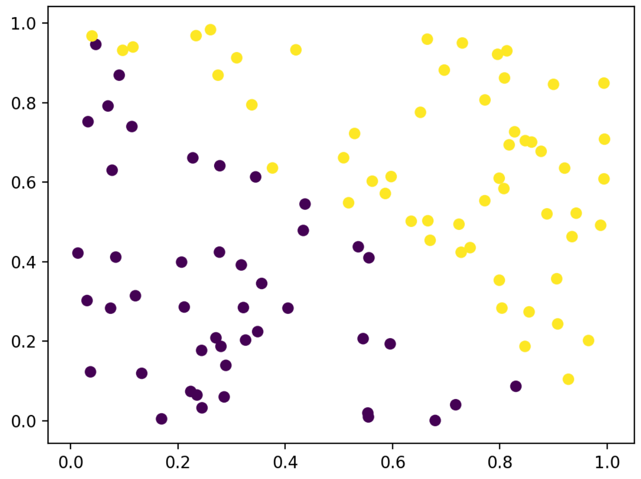
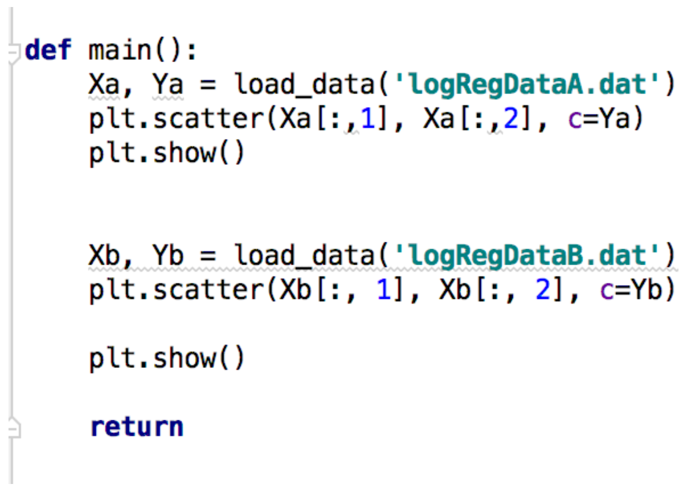


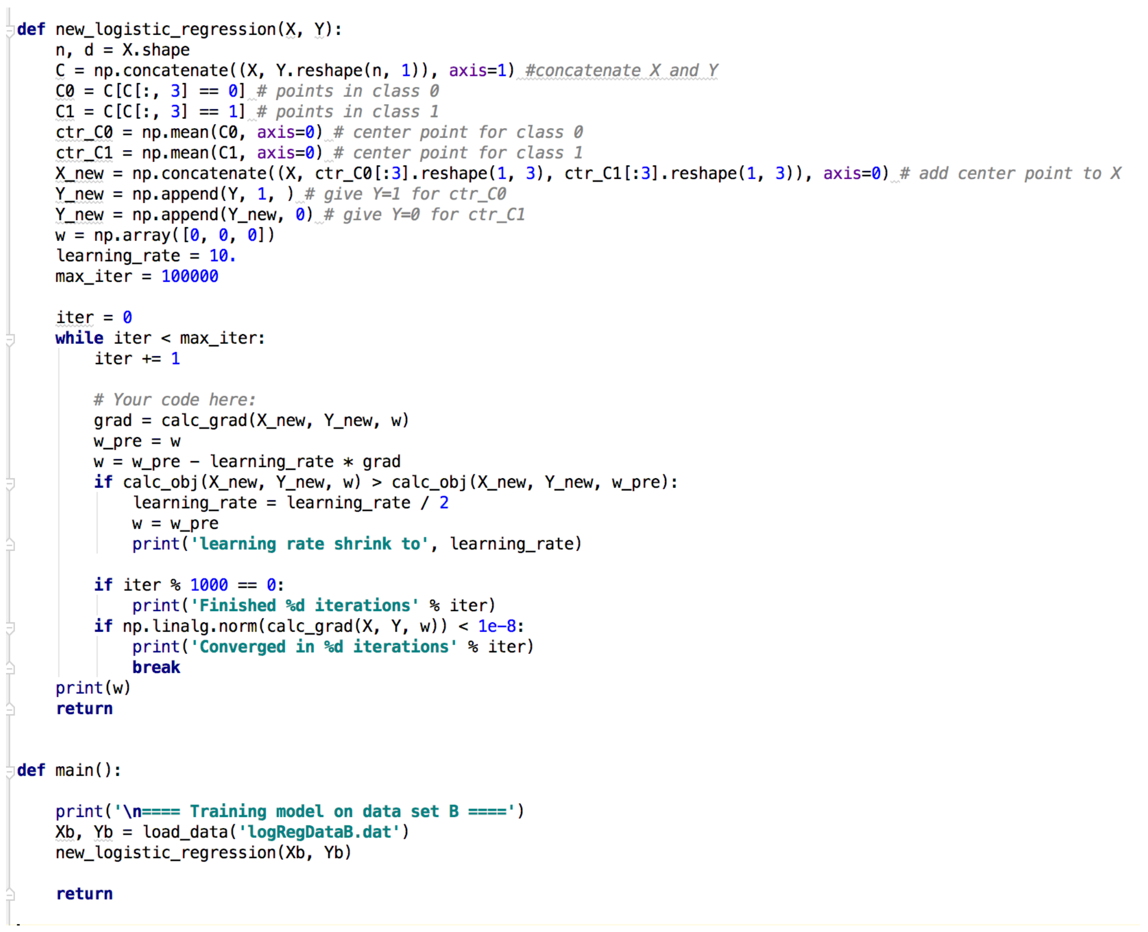
Figure 12: Scatter plot of set B



```
def main():
    Xa, Ya = load_data('logRegDataA.dat')
    plt.scatter(Xa[:,1], Xa[:,2], c=Ya)
    plt.show()


    Xb, Yb = load_data('logRegDataB.dat')
    plt.scatter(Xb[:, 1], Xb[:, 2], c=Yb)

    plt.show()

    return
```

Figure 13: Python code for scatter plot

(c) I found there are actually some methods to solve with the linearly separable set, our main idea is to artificially fuzz up the boundary line without leading much bias. So we change the *def logistic regression(X, Y)* part to *def new logistic regression(X,Y)* part with other parts remain the same. The change part is as follows:



```python
def new_logistic_regression(X, Y):
    n, d = X.shape
    C = np.concatenate((X, Y.reshape(n, 1)), axis=1) #concatenate X and Y
    C0 = C[C[:, 3] == 0] # points in class 0
    C1 = C[C[:, 3] == 1] # points in class 1
    ctr_C0 = np.mean(C0, axis=0) # center point for class 0
    ctr_C1 = np.mean(C1, axis=0) # center point for class 1
    X_new = np.concatenate((X, ctr_C0[:3].reshape(1, 3), ctr_C1[:3].reshape(1, 3)), axis=0) # add center point to X
    Y_new = np.append(Y, 1, ) # give Y=1 for ctr_C0
    Y_new = np.append(Y_new, 0) # give Y=0 for ctr_C1
    w = np.array([0, 0, 0])
    learning_rate = 10.
    max_iter = 100000

    iter = 0
    while iter < max_iter:
        iter += 1

        # Your code here:
        grad = calc_grad(X_new, Y_new, w)
        w_pre = w
        w = w_pre - learning_rate * grad
        if calc_obj(X_new, Y_new, w) > calc_obj(X_new, Y_new, w_pre):
            learning_rate = learning_rate / 2
            w = w_pre
            print('learning rate shrink to', learning_rate)

        if iter % 1000 == 0:
            print('Finished %d iterations' % iter)
        if np.linalg.norm(calc_grad(X, Y, w)) < 1e-8:
            print('Converged in %d iterations' % iter)
            break
    print(w)
    return


def main():

    print('\n==== Training model on data set B ====')
    Xb, Yb = load_data('logRegDataB.dat')
    new_logistic_regression(Xb, Yb)

    return
```

Figure 14: Python code for changed part

After we change into the new logistic regression, we can see the output as follows:

```
==== Training model on data set B ====
learning rate shrink to 5.0
Finished 1000 iterations
Finished 2000 iterations
Finished 3000 iterations
Finished 4000 iterations
Finished 5000 iterations
Finished 6000 iterations
Finished 7000 iterations
Finished 8000 iterations
learning rate shrink to 2.5
learning rate shrink to 1.25
learning rate shrink to 0.625
⋮
learning rate shrink to 5.960464477539062e-07
learning rate shrink to 2.980232238769531e-07
```

This means, after about 8000 iterations, the learning rate shrink to 0, so we can point out the value of w, which is $[-12.8358969813.4415464812.66874279]$. And this should be the w at the minimum point of the convex function.

3. **Poisson regression**

   (a) We can rewrite $p(y;\lambda)$ as follows:

$$\begin{aligned} p(y;\lambda) &= \frac{e^{-\lambda}\lambda^y}{y!} \\ &= \frac{1}{y!}exp(ylog(\lambda) - \lambda) \end{aligned}$$

   Here, we compare it with the density forms of exponential family : $p(y;\theta) = h(y)exp(\theta^T T(y) - a(\theta))$, we have:

$$\begin{aligned} h(y) &= \frac{1}{y!} \\ \theta &= log(\lambda) \\ T(y) &= y \\ a(\theta) &= \lambda = e^\theta \end{aligned}$$

   So the Poisson model is an exponential family.

   (b)

$$\begin{aligned} l(\beta) = -logL(\beta) &= \Sigma_{i=1}^n[\lambda - y_i log(\lambda) + log(y_i!)] \\ &\propto \Sigma_{i=1}^n[h(\beta^T x_i) - y_i log(h(\beta^T x_i))] \end{aligned}$$

   So $l(\beta) = \Sigma_{i=1}^n[h(\beta^T x_i) - y_i log(h(\beta^T x_i))]$ is the maximum likelihood estimator of the Poisson model.
   The gradient descent is $\nabla l(\beta) = \Sigma_{i=1}^n[x_i h'(\beta^T x_i) - y_i \frac{x_i h'(\beta^T x_i)}{h(\beta^T x_i)}]$
   So we require $\beta^{(0)} in R^d$ is the starting value, then repeat $\beta^{(t+1)} \longleftarrow \beta^{(t)} - \eta_k \nabla l(\beta^{(t)})$ until converge.