EECS 545 Homework 1

Yirui Gao

Jan 25

Contents

Question 1. Logsistic regression

(a) For logistic regression, we can derive the Hessian H for the log-likelihood function by continuing the derivation in the lecture slide for the gradient of the log-likelihood. Since we know

$$\nabla_{\mathbf{w}}\ell(\mathbf{w}) = \sum_{n=1}^{N}(y^{(n)} - \sigma^{(n)})\phi(\mathbf{x}^{(n)}),$$

where $\sigma^{(n)}$ is the sigmoid function with the weights $\mathbf{w}$. Notice that the derivate for this sigmoid function is:

$$\sigma'(s) = \sigma(s)(1 - \sigma(s)).$$

Therefore, the second derivative of the log-likelihood function can be found:

$$\begin{aligned}\nabla^2_{\mathbf{w}}\ell(\mathbf{w}) &= \sum_{n=1}^{N} -\phi(\mathbf{x}^{(n)})\sigma^{(n)}(1-\sigma^{(n)})\phi(\mathbf{x}^{(n)}) \\ &= -\sum_{n=1}^{N}\mathbf{x}^{(n)}\sigma^{(n)}(1-\sigma^{(n)})\mathbf{x}^{(n)\mathbf{T}} \\ &= -\sum_{n=1}^{N}\sigma^{(n)}(1-\sigma^{(n)})\mathbf{x}^{(n)}\mathbf{x}^{(n)\mathbf{T}}.\end{aligned}$$

So the Hessian matrix for the log-likelihood function can be represented as:

$$H = -\mathbf{X}^T diag\ (\sigma^{(n)}(1-\sigma^{(n)}))\mathbf{X},$$

where $diag(\sigma^{(n)}(1-\sigma^{(n)}))$ is a $n \times n$ diagonal matrix with the value $\sigma^{(n)}(1-\sigma^{(n)})$ falling at the n-th position on the diagonal line and other positions remain 0.

Then, to show that Hessian of the log-likelihood function is negative semi-definite, we can show:

$$\begin{aligned}\mathbf{z}^T H\mathbf{z} &= -\mathbf{z}^T\mathbf{X}^T diag\ (\sigma^{(n)}(1-\sigma^{(n)}))\mathbf{X}\mathbf{z} \\ &= -(\mathbf{X}\mathbf{z})^T diag\ (\sigma^{(n)}(1-\sigma^{(n)}))(\mathbf{X}\mathbf{z}) \\ &= -\sum_{n=1}^{N}\sigma^{(n)}(1-\sigma^{(n)})(\mathbf{x}^{(n)}\mathbf{z})(\mathbf{x}^{(n)}\mathbf{z})^{\mathbf{T}} \\ &= -\sum_{n=1}^{N}\sigma^{(n)}(1-\sigma^{(n)})(\mathbf{x}^{(n)}\mathbf{z})^2,\end{aligned}$$

since the squared term $(\mathbf{x}^{(n)}\mathbf{z})^2$ must be non-negative, and for the sigmoid function, we know that the derivative can never go below 0, so the overall sum of product must be non-negative, so plus the minus sign ahead, the overall term is non-positive, $\mathbf{z}^T H\mathbf{z} \leq 0$, so the Hessian is negative semi-definite.

(b) First implement functions for calculating log-likelihoods and hessians:

```
import numpy as np
import math

def sigmoid(X):
    return 1/(1 + np.exp(-X))

def log_likelihood(X, w, y):
    epsilon = 1e-5
    p = sigmoid(X.dot(w))
    return np.sum(y * np.log(p) + (1 - y) * np.log(1 - p))

def compute_gradient(X, w, y):
    grad = -X.T.dot(y - sigmoid(X.dot(w)))
    return grad

def hessian(X, w, y):
    m = sigmoid(X.dot(w))*(1-sigmoid(X.dot(w))).reshape(1, X.shape[0])
    h = -X.T @ np.diag(np.diag(m)) @ X
    return h
```

The Newton's method is defined in this way:

```
def newton(X, w, y, num_iter = 100):
    print("***************Newton's method********************")
    l = 0
    l_new = 1
    epsilon = 1e-10
    i = 1
    while abs(l - l_new) > epsilon and i <= num_iter:
        l = l_new
        grad = compute_gradient(X, w, y)
        w = w + np.linalg.inv(hessian(X, w, y)).dot(grad)
        l_new = log_likelihood(X, w, y)
        print("This is Iteration {} and the log likelihood is {}".format(i, l_new))
        i += 1
    print("The final weight is {}".format(w))
    return w

X = np.load('data/q1x.npy')
y = np.reshape(np.load('data/q1y.npy'), (X.shape[0], 1))
N = X.shape[0]
X = np.concatenate((np.ones((N, 1)), X), axis=1)
w = np.zeros((X.shape[1], 1))
w = newton(X, w, y)
```

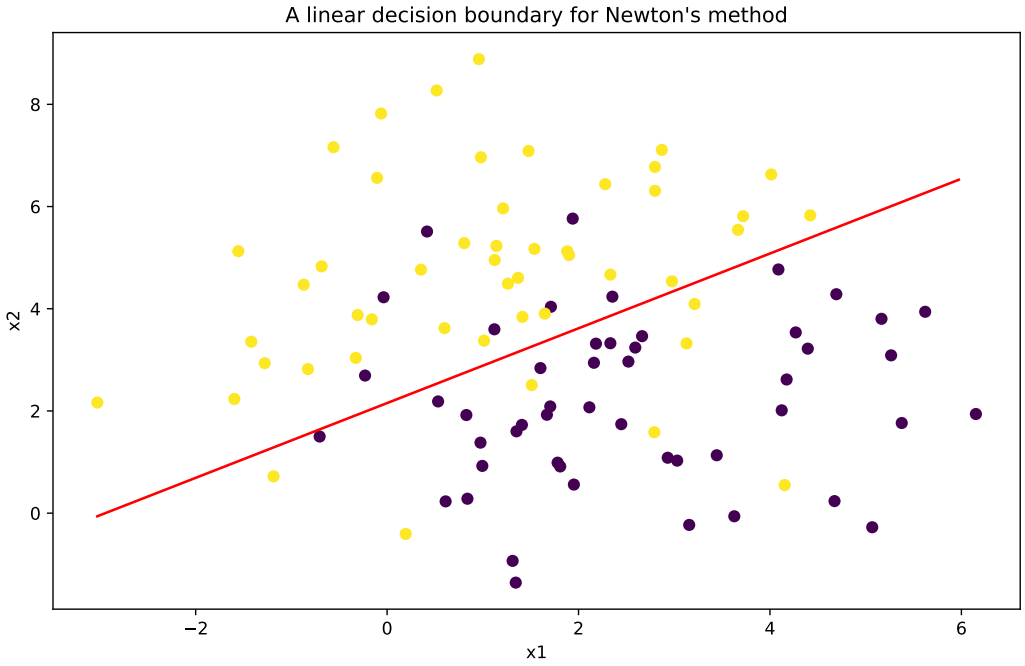
```
## ***************Newton's method********************
## This is Iteration 1 and the log likelihood is -44.39589201405799
## This is Iteration 2 and the log likelihood is -41.47986213111555
## This is Iteration 3 and the log likelihood is -41.214441345079045
## This is Iteration 4 and the log likelihood is -41.21099356552897
## This is Iteration 5 and the log likelihood is -41.21099284511821
## This is Iteration 6 and the log likelihood is -41.21099284511818
## The final weight is [[-1.84922892]
```

```
##  [-0.62814188]
##  [ 0.85846843]]
```

According to the running result, we can see that the it converges when thenumber of iteration is 6, and gives the final result for $\mathbf{w}$ to be [-1.84922892, -0.62814188, 0.85846843].

(c) Having the fitted coefficients derived from question (b), we can try to find a straight line as the decision boundary to separate regions for $h(\mathbf{x}) > 0.5$ and $h(\mathbf{x}) \leq 0.5$. We can see that when $h(\mathbf{x}) = 0.5$, $\mathbf{X}w = 0$. So I did the following job to generate the decision boudary:

```
import matplotlib.pyplot as plt
x1 = X[:,1]
x2 = X[:,2]

plt.figure(figsize=(10,6))
plt.scatter(x1, x2, c = y[:,0])
x_range = np.arange(min(x1), max(x1))
plt.plot(x_range, -(w[0,] + w[1, ]*x_range)/w[2,], c = "red")
plt.title("A linear decision boundary for Newton's method")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```



Question 2. Linear regression on a polynomial

```
import time

# Load the data:
X_train = np.load('data/q2xTrain.npy')
y_train = np.load('data/q2yTrain.npy')
X_test = np.load('data/q2xTest.npy')
y_test = np.load('data/q2yTest.npy')
```

```
def construct_polynomial(X_vec, degree):
    X = np.ones((X_vec.shape[0], 1))
    for i in range(1, degree + 1):
        X = np.append(X, np.reshape(X_vec**i, (X_vec.shape[0], 1)), axis=1)
    return X
```

(a) **i**. For this question, I use Python's numpy and apply the optimization methods one by one:

```
n = X_train.shape[0]
X = construct_polynomial(X_train, 1)
theta = np.zeros((2, 1))
y = np.reshape(y_train, (n, 1))

def cost_fuction(X, theta, y, lambda_ = 0):
    regularization = 0.5 * lambda_ * (theta.T.dot(theta))
    return 0.5 * np.sum((np.dot(X, theta) - y) ** 2) + np.asscalar(regularization)
```

```
def batchGD(X, theta, y, learning_rate = 0.001, num_iter = 10000):
    print("***************Batch GD********************")
    e = 0
    e_new = 1
    tolerance = 1e-10
    i = 0
    error_list = []
    while abs(e - e_new) > tolerance and i <= num_iter:
        i += 1
        e = e_new
        grad = np.dot(X.T, np.dot(X, theta) - y)
        theta = theta - learning_rate * grad
        e_new = cost_fuction(X, theta, y)
        error_list.append(e_new)
    print("It takes number of iteration # {} with final cost {}".format(i, e_new))
    print("The final result for theta is ", theta)

    return error_list

n = X_train.shape[0]
X = construct_polynomial(X_train, 1)
theta = np.zeros((2, 1))
y = np.reshape(y_train, (n, 1))

batchGD(X, theta, y)
```

```
## ***************Batch GD********************
## It takes number of iteration # 8495 with final cost 1.9875452873617634
## The final result for theta is  [[ 1.94676992]
##  [-2.82392259]]
## [11.177807585664526, 11.02542315874513, 10.87980092105017, 10.740613382423392, 10.607548979218027, 10
```

So for Batch gradient descent, it gives the coefficients for the weight $\mathbf{w} = [1.94676992, -2.82392259]$ and the training takes 8495 number of iterations to come to a convergence.

```
def SGD(X, theta, y, learning_rate = 0.001, num_iter = 10000):
    print("***************Stochastic GD********************")
    e = 0
```

```
    e_new = 1
    tolerance = 1e-10
    j = 0
    error_list = []
    while abs(e - e_new) > tolerance and j <= num_iter:
        j += 1
        e = e_new
        e_new = 0
        for i in range(n):
            X_i = X[i,:].reshape(1, X.shape[1])
            y_i = y[i].reshape(1,1)
            prediction_i = np.dot(X_i, theta)
            grad = np.dot(X_i.T, prediction_i - y_i)
            theta = theta - learning_rate * grad
            e_new += cost_fuction(X_i, theta, y_i)
        error_list.append(e_new)
    print("It takes number of iteration # {} with final cost {}".format(j, e_new))
    print("The final result for theta is ", theta)
    return error_list

SGD(X, theta, y)
```

```
## ***************Stochastic GD********************
## It takes number of iteration # 8157 with final cost 1.9849734848982807
## The final result for theta is  [[ 1.94634098]
##  [-2.82437173]]
## [11.2453898804368, 11.091070872677053, 10.943533718152368, 10.802453624159241, 10.667521415506483, 10
```

So for stochastic batch gradient descent, it gives the coefficients for the weight $\mathbf{w} = [1.94634098, -2.82437173]$ and the training takes 8157 number of iterations to come to a convergence.

```
def newton(X, theta, y, num_iter = 15):
    print("***************Newton's method*********************")
    H = X.T.dot(X)
    e = 0
    e_new = 1
    tolerance = 1e-10
    i = 0
    error_list = []
    while abs(e - e_new) > tolerance and i <= num_iter:
        i += 1
        e = e_new
        grad = np.dot(X.T, np.dot(X, theta) - y)
        theta = theta -  np.linalg.inv(H).dot(grad)
        e_new = cost_fuction(X, theta, y)
        error_list.append(e_new)
    print("It takes number of iteration # {} with final cost {}".format(i, e_new))
    print("The final result for theta is ", theta)
    return error_list, theta

newton(X, theta, y)
```

```
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 1.9875452421741249
## The final result for theta is  [[ 1.9468968 ]
```

```
##  [-2.82417908]]
## ([1.9875452421741249, 1.9875452421741249], array([[ 1.9468968 ],
##        [-2.82417908]]))
```

So for Newton's method, it gives the coefficients for the weight $\mathbf{w} = [1.9468968, -2.82417908]$ and the training takes 2 number of iterations to come to a convergence.

ii. Based on the knowledge of Newton's method, it converges to the optimal values for θ very fast, with only 2 iterations in my case (Ideally for linear regression it could converge in must one iteration, and the two here majorly is due to the precision of calculation), so it has a very high rate of convergence. As for the two gradient descent methods, we can compare the number of iterations they need to come to the convergence, where the batch GD takes 8495 iterations and SGD takes 8157 number of iterations, so we can say that the SGD is slightly faster than batch GD in the comparison of iteration numbers. But since SGD need to go over all the samples in one iteration, thus in this case, if we compare in time spent, batch GD has a higher rate of convergence than SGD.

(b)

i. Seen in the following generated chart and the coding results for coefficients:

```
X_test = np.load('data/q2xTest.npy')
y_test = np.load('data/q2yTest.npy')
ytest = np.reshape(y_test, (X_test.shape[0], 1))

M = range(0, 10)
ERMS_list = []
ERMS_list_test = []

for i in M:
    X = construct_polynomial(X_train, i)
    testX = construct_polynomial(X_test, i)
    theta = np.zeros((i + 1, 1))
    e, theta = newton(X, theta, y)
    print("The theta for the {}-degree is: {}".format(i, theta))
    e_test = cost_fuction(testX, theta, ytest)

    ERMS = np.sqrt(2 * e[-1] / X.shape[0])
    ERMS_list.append(ERMS)

    ERMS_test = np.sqrt(2 * e_test / testX.shape[0])
    ERMS_list_test.append(ERMS_test)
```

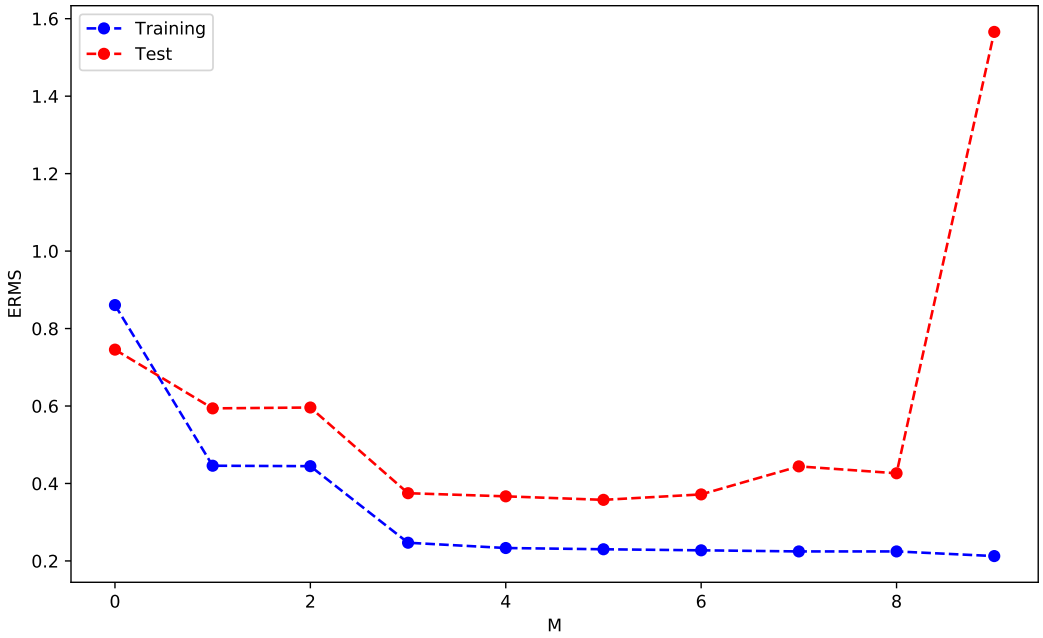
```
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 7.406775756796049
## The final result for theta is  [[0.62693881]]
## The theta for the 0-degree is: [[0.62693881]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 1.9875452421741249
## The final result for theta is  [[ 1.9468968 ]
##  [-2.82417908]]
## The theta for the 1-degree is: [[ 1.9468968 ]
##  [-2.82417908]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 1.9774912035147805
## The final result for theta is  [[ 2.02881908]
##  [-3.31190607]
```

```
##  [ 0.5099073 ]]
## The theta for the 2-degree is: [[ 2.02881908]
##  [-3.31190607]
##  [ 0.5099073 ]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.6100579287748409
## The final result for theta is  [[  0.72143522]
##  [ 10.6925568 ]
##  [-34.25867044]
##  [ 23.73399256]]
## The theta for the 3-degree is: [[  0.72143522]
##  [ 10.6925568 ]
##  [-34.25867044]
##  [ 23.73399256]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.5445978132804455
## The final result for theta is  [[  0.31700758]
##  [ 17.19240855]
##  [-61.74347052]
##  [ 65.73372287]
##  [-21.02990396]]
## The theta for the 4-degree is: [[  0.31700758]
##  [ 17.19240855]
##  [-61.74347052]
##  [ 65.73372287]
##  [-21.02990396]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.5296832351229446
## The final result for theta is  [[  0.68631137]
##  [  9.14396164]
##  [-11.21978608]
##  [-63.12577688]
##  [121.42886106]
##  [-56.75278845]]
## The theta for the 5-degree is: [[  0.68631137]
##  [  9.14396164]
##  [-11.21978608]
##  [-63.12577688]
##  [121.42886106]
##  [-56.75278845]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.5171812552282713
## The final result for theta is  [[   1.17337103]
##  [  -3.85879113]
##  [  99.39182601]
##  [-482.12388673]
##  [ 902.44548997]
##  [-756.24366057]
##  [ 240.00499352]]
## The theta for the 6-degree is: [[   1.17337103]
##  [  -3.85879113]
##  [  99.39182601]
##  [-482.12388673]
##  [ 902.44548997]
```

```
##  [-756.24366057]
##  [ 240.00499352]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.5042496816260292
## The final result for theta is  [[ 4.82561175e-01]
##  [ 1.88010755e+01]
##  [-1.55273420e+02]
##  [ 8.50251051e+02]
##  [-2.73875214e+03]
##  [ 4.59786346e+03]
##  [-3.76771954e+03]
##  [ 1.19660229e+03]]
## The theta for the 7-degree is: [[ 4.82561175e-01]
##  [ 1.88010755e+01]
##  [-1.55273420e+02]
##  [ 8.50251051e+02]
##  [-2.73875214e+03]
##  [ 4.59786346e+03]
##  [-3.76771954e+03]
##  [ 1.19660229e+03]]
## ***************Newton's method********************
## It takes number of iteration # 2 with final cost 0.5041822657588223
## The final result for theta is  [[ 3.84923550e-01]
##  [ 2.26064520e+01]
##  [-2.08301610e+02]
##  [ 1.20568441e+03]
##  [-4.03819931e+03]
##  [ 7.32669538e+03]
##  [-7.04882368e+03]
##  [ 3.29335455e+03]
##  [-5.51455257e+02]]
## The theta for the 8-degree is: [[ 3.84923550e-01]
##  [ 2.26064520e+01]
##  [-2.08301610e+02]
##  [ 1.20568441e+03]
##  [-4.03819931e+03]
##  [ 7.32669538e+03]
##  [-7.04882368e+03]
##  [ 3.29335455e+03]
##  [-5.51455257e+02]]
## ***************Newton's method********************
## It takes number of iteration # 3 with final cost 0.45181854649109093
## The final result for theta is  [[ 6.35099301e+00]
##  [-2.30843513e+02]
##  [ 3.77339470e+03]
##  [-3.00383303e+04]
##  [ 1.35360464e+05]
##  [-3.68462356e+05]
##  [ 6.16055234e+05]
##  [-6.17628407e+05]
##  [ 3.40252385e+05]
##  [-7.91054979e+04]]
## The theta for the 9-degree is: [[ 6.35099301e+00]
##  [-2.30843513e+02]
```

```
##   [ 3.77339470e+03]
##   [-3.00383303e+04]
##   [ 1.35360464e+05]
##   [-3.68462356e+05]
##   [ 6.16055234e+05]
##   [-6.17628407e+05]
##   [ 3.40252385e+05]
##   [-7.91054979e+04]]
```

```
plt.figure(figsize=(10,6))
train, = plt.plot(M, ERMS_list, '--o', color='blue', label='Training')
test, = plt.plot(M, ERMS_list_test, '--o', color='red', label='Test')
plt.legend(handles=[train, test])
plt.xlabel("M")
plt.ylabel("ERMS")
plt.show()
```



ii. From the generated chart in (i), I would say the 5th degree polynomial best fits the data. There is no evident clue that the model experienced underfitting for any particular degree, but when $M = 9$, we can see that overfitting exists since the RMS error for the test data explodes at this degree.

(c) **i**. For this question, I redefine another Newton's method function with considering the effect of regularization:

```
def newton_regularization(X, theta, y, lambda_, num_iter = 15):
    print("***************Newton's method with regularization*******************")
    H = X.T.dot(X)
    e = 0
    e_new = 1
    tolerance = 1e-10
    i = 0
```

```
    error_list = []
    while abs(e - e_new) > tolerance and i <= num_iter:
        i += 1
        e = e_new
        grad = np.dot(X.T, np.dot(X, theta) - y) + lambda_ * theta
        theta = theta - np.linalg.inv(np.add(H, lambda_ * np.identity(X.shape[1]))).dot(grad)
        e_new = cost_fuction(X, theta, y, lambda_)
        error_list.append(e_new)
    return error_list, theta
```

Different sets of lambda values are considered and plugged in to have a try, the result is shown in the following chart with the calculation of RMS errors:

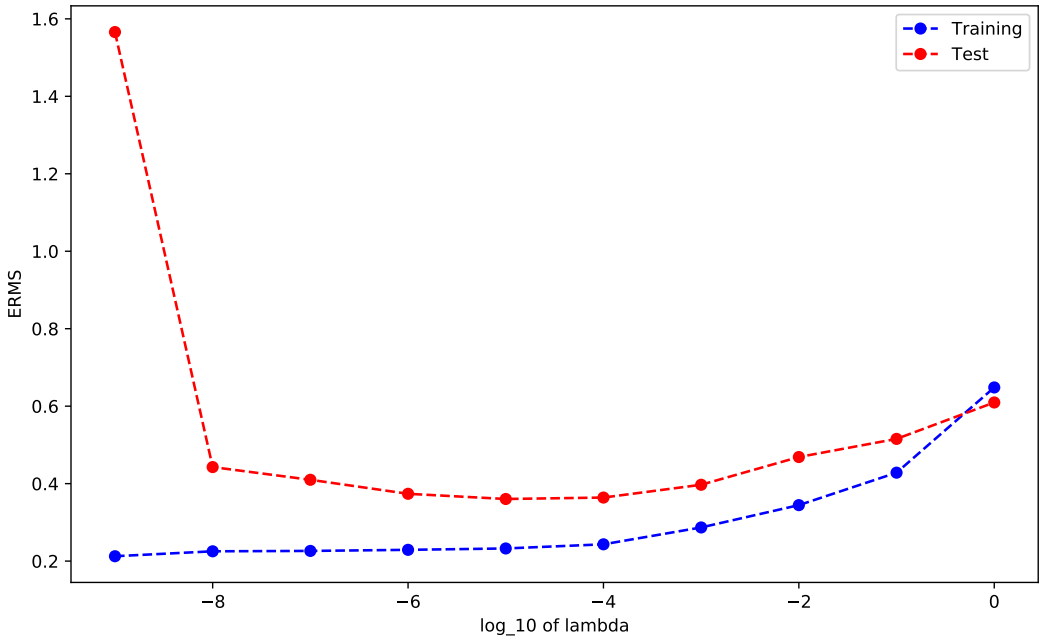
```
lambda_list = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 0.1, 1]
M = 9

ERMS_list = []
ERMS_list_test = []

X = construct_polynomial(X_train, M)
testX = construct_polynomial(X_test, M)

for l in lambda_list:
    theta = np.zeros((M+1, 1))
    e, theta = newton_regularization(X, theta, y, l)
    print("The theta for the regularization with lambda equal to {} is: {}".format(l, theta))
    e_test = cost_fuction(testX, theta, ytest, lambda_ = l)

    ERMS = np.sqrt(2 * e[-1] / X.shape[0])
    ERMS_list.append(ERMS)

    ERMS_test = np.sqrt(2 * e_test / testX.shape[0])
    ERMS_list_test.append(ERMS_test)
```

```
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 0 is: [[ 6.35099301e+00]
##  [-2.30843513e+02]
##  [ 3.77339470e+03]
##  [-3.00383303e+04]
##  [ 1.35360464e+05]
##  [-3.68462356e+05]
##  [ 6.16055234e+05]
##  [-6.17628407e+05]
##  [ 3.40252385e+05]
##  [-7.91054979e+04]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 1e-08 is: [[   0.87281516]
##  [   5.68109738]
##  [  -1.15857179]
##  [ -13.23532598]
##  [-136.08338633]
##  [ 222.45617486]
##  [ 140.03781262]
##  [-200.50437254]
```

```
##  [-238.96452666]
##  [ 223.1751212 ]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 1e-07 is: [[   0.9080066 ]
##  [   4.21682887]
##  [  17.19639558]
##  [-104.09953582]
##  [  50.76245505]
##  [ 121.85142874]
##  [   9.43281124]
##  [-119.42264541]
##  [ -99.86869151]
##  [ 120.7364416 ]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 1e-06 is: [[  0.67778364]
##  [  9.26697861]
##  [-13.10679556]
##  [-43.55076818]
##  [ 44.4067339 ]
##  [ 50.20791682]
##  [ -7.66810429]
##  [-52.1792701 ]
##  [-36.03023406]
##  [ 48.87151966]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 1e-05 is: [[  0.59106034]
##  [ 11.58939844]
##  [-30.01201196]
##  [ -0.61146409]
##  [ 20.93512034]
##  [ 13.97639282]
##  [ -3.13002479]
##  [-13.13277559]
##  [ -8.93475101]
##  [  8.99366139]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 0.0001 is: [[  0.76397647]
##  [  9.18088699]
##  [-22.62646106]
##  [ -2.26667382]
##  [ 10.25041855]
##  [ 10.18741054]
##  [  4.7031529 ]
##  [ -0.87709574]
##  [ -4.28496401]
##  [ -5.14799425]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 0.001 is: [[  1.14781374]
##  [  4.96043921]
##  [-13.02789327]
##  [ -3.51704705]
##  [  4.38087189]
##  [  6.45474174]
##  [  4.81393522]
```

```
##  [  1.64522555]
##  [ -1.74954177]
##  [ -4.75946016]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 0.01 is: [[ 1.61509788]
##  [ 0.24850035]
##  [-4.18021188]
##  [-1.97672047]
##  [ 0.25860156]
##  [ 1.31974107]
##  [ 1.50045914]
##  [ 1.21698491]
##  [ 0.75022723]
##  [ 0.25443342]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 0.1 is: [[ 1.71910715]
##  [-1.26573909]
##  [-1.73323498]
##  [-1.00186049]
##  [-0.27404594]
##  [ 0.22745282]
##  [ 0.53336578]
##  [ 0.70658846]
##  [ 0.79631279]
##  [ 0.83493196]]
## ***************Newton's method with regularization*******************
## The theta for the regularization with lambda equal to 1 is: [[ 1.22947449]
##  [-0.72301958]
##  [-0.74894872]
##  [-0.49070049]
##  [-0.25533698]
##  [-0.0835526 ]
##  [ 0.03434322]
##  [ 0.11330398]
##  [ 0.16522674]
##  [ 0.1984788 ]]
```

```
lambda_list[0] = 1e-9
plt.figure(figsize=(10,6))
train2, =  plt.plot(np.log10(lambda_list), ERMS_list, '--o', color='blue', label='Training')
test2, = plt.plot(np.log10(lambda_list), ERMS_list_test, '--o', color='red', label='Test')
plt.legend(handles=[train2, test2])
plt.xlabel("log_10 of lambda")
plt.ylabel("ERMS")
plt.show()
```



Notice that I use the log 10 of lambda to represent the values shown on the x axis, and because we know the log of 0 is negative infinity, so in order to show the RMS error when there's no regularization, I shift the point of negative infinity to the point of $log_{10}(e^{-9}) = -9$ on x axis, so we can see the trend better in the plot.

ii. From the generated chart above, it seems that the $\lambda = 1e-5$ makes the best model in terms of error control.

Question 3. Locally weighted linear regression

(a) For $E_D(w)$, the original expression can be transformed to:

$$\begin{aligned} E_D(w) &= \frac{1}{2}\sum_{i=1}^{N}(\mathbf{w}^T\mathbf{X}^{(i)} - y^{(i)})^2 \\ &= \frac{1}{2}\mathbf{w}^T\mathbf{X}^T R^{'}\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T R^{'}\mathbf{y} + \frac{1}{2}\mathbf{y}^T R^{'}\mathbf{y} \\ &= \frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{y})^T R^{'}(\mathbf{X}\mathbf{w} - \mathbf{y}) \\ &= (\mathbf{X}\mathbf{w} - \mathbf{y})^T R(\mathbf{X}\mathbf{w} - \mathbf{y}). \end{aligned}$$

Here $R^{'}$ is a diagonal matrix with dimension (N,N) and $r^{(i)}$ falls on the diagonal line of this matrix. Therefore, R is a diagonal matrix where for $R_{ii} = \frac{1}{2}r^{(i)}$ and zeros at non-diagonal positions.

(b) Based on the previous conclusion,

$$\begin{aligned} \nabla_{\mathbf{w}} E_D(\mathbf{w}) &= \nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{X}^T R\mathbf{X}\mathbf{w} - 2\mathbf{w}^T\mathbf{X}^T R\mathbf{y} + \mathbf{y}^T R\mathbf{y}) \\ &= 2\mathbf{X}^T R\mathbf{X}\mathbf{w} - 2\mathbf{X}^T R\mathbf{y} \\ &= 0. \end{aligned}$$

To solve this equation, we can easily get the normal equation for the new value of $\mathbf{w}^*$ is then

$$\mathbf{w}^* = (\mathbf{X}^T R\mathbf{X})^{-1}\mathbf{X}^T R\mathbf{y}.$$

(c) By applying the maximum likelihood estimate of $\mathbf{w}$, we need to derive the log likelihood:

$$\begin{aligned}\log p(y^{(1)}, y^{(2)}, ..., y^N|\mathbf{X}, \mathbf{w}) &= \log \prod_{i=1}^{N} (\frac{1}{\sqrt{(2\pi)}\sigma^{(i)}} \exp(-\frac{(y^{(i)} - \mathbf{w}^T\mathbf{X}^{(i)})^2}{2(\sigma^{(i)})^2})) \\ &= \sum_{i=1}^{N} (-\frac{1}{2}\log\ 2\pi - \log\sigma^{(i)} - \frac{(y^{(i)} - \mathbf{w}^T\mathbf{X}^{(i)})^2}{2(\sigma^{(i)})^2}) \\ &= -\frac{N}{2}\log 2\pi - \sum_{i=1}^{N}\log\sigma^{(i)} - \sum_{i=1}^{N}\frac{(y^{(i)} - \mathbf{w}^T\mathbf{X}^{(i)})^2}{2(\sigma^{(i)})^2}),\end{aligned}$$

and the goal is to let $\nabla_{\mathbf{w}} \log p(y^{(1)}, y^{(2)}, ..., y^N|\mathbf{X}, \mathbf{w}) = 0$, so we can derive:

$$\begin{aligned}\nabla_{\mathbf{w}} \log p(y^{(1)}, y^{(2)}, ..., y^N|\mathbf{X}, \mathbf{w}) &= -\sum_{i=1}^{N}\frac{1}{(\sigma^{(i)})^2}(y^{(i)} - \mathbf{w}^T\mathbf{X}^{(i)})\mathbf{X}^{(i)} \\ &= 0.\end{aligned}$$

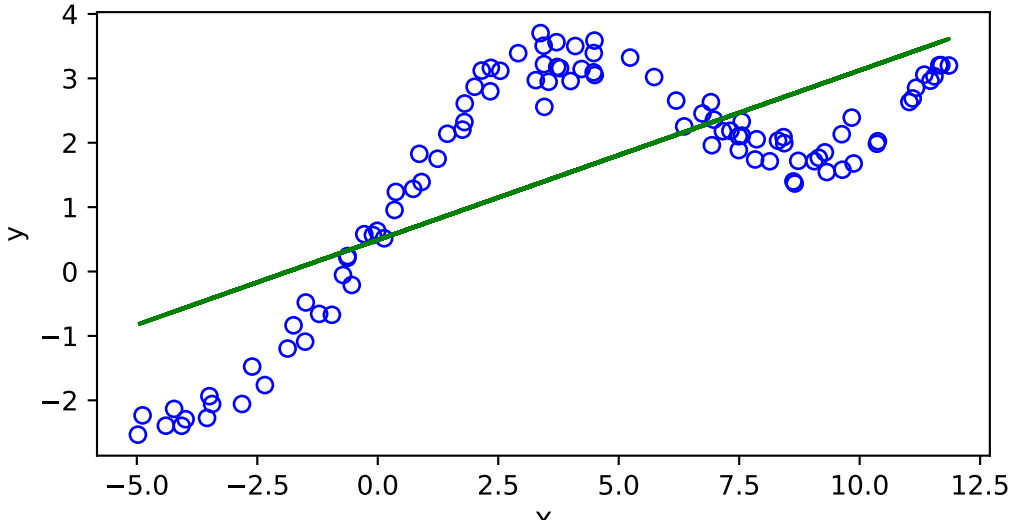
We can find that if we replace $\frac{1}{(\sigma^{(i)})^2}$ with $r^{(i)}$ in the previous example, we can transform this equation to the same one that we solved for problem (b). Thus we can see that finding the maximum likelihood estimate of $\mathbf{w}$ reduces to solving a weighted linear regression problem. And the $r^{(i)}$ can be represented as $\frac{1}{(\sigma^{(i)})^2}$.

(d) **i**. Using the normal equation $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$, apply the following python code and get the coefficients for $\mathbf{w}$ as $[0.49073707, 0.26333931]$:

```
# Load the data:
X = np.load('data/q3x.npy')
y = np.load('data/q3y.npy')
n = X.shape[0]
X = np.append(np.ones((n, 1)), np.reshape(X, (n, 1)), axis = 1)
# Use normal equation to get the result for the weight:
theta = np.linalg.inv(np.dot(X.T, X)).dot(X.T).dot(y)
print(theta)
```

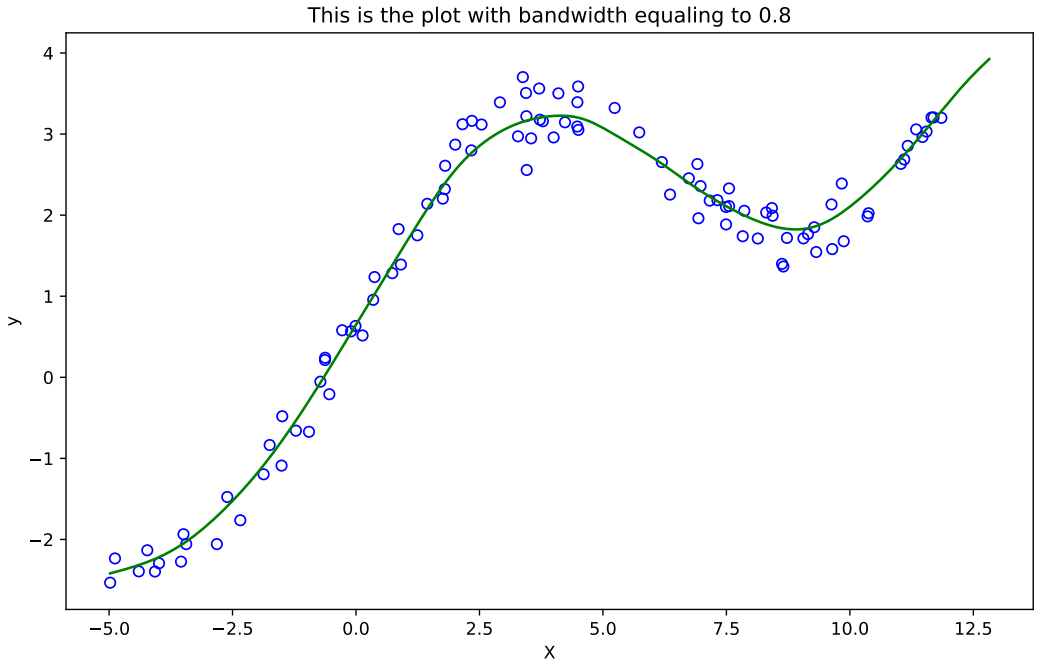
```
## [0.49073707 0.26333931]
```

```
plt.figure(figsize=(6,3))
plt.scatter(X[:,1], y, facecolors='none', edgecolors='blue')
plt.plot(X[:,1], X.dot(theta), color='green')
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```



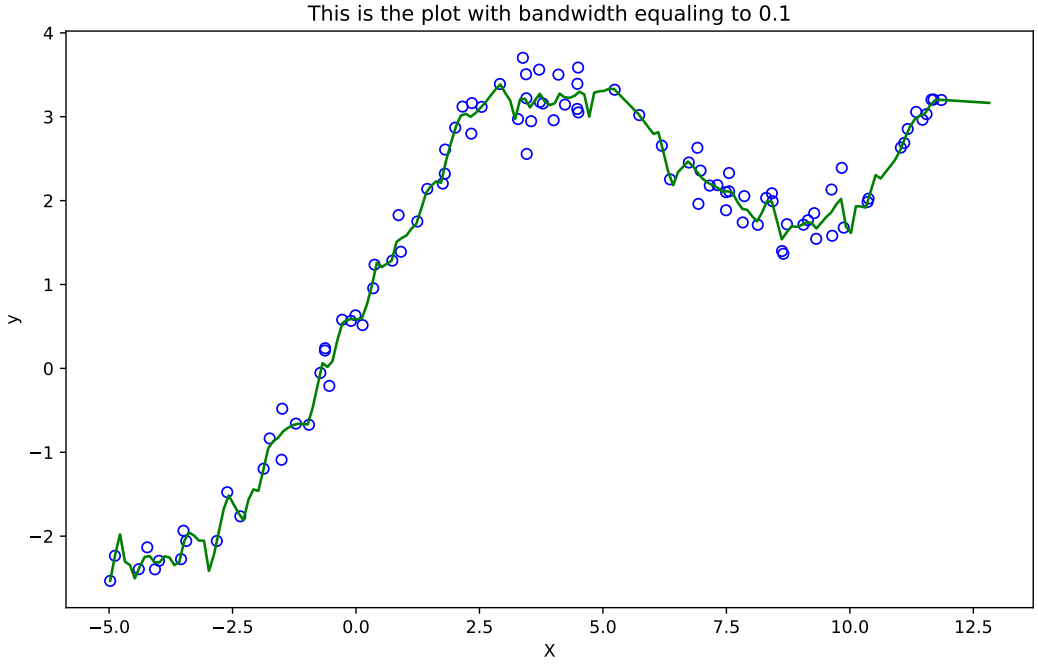
ii. Generalize the Python code, we have:

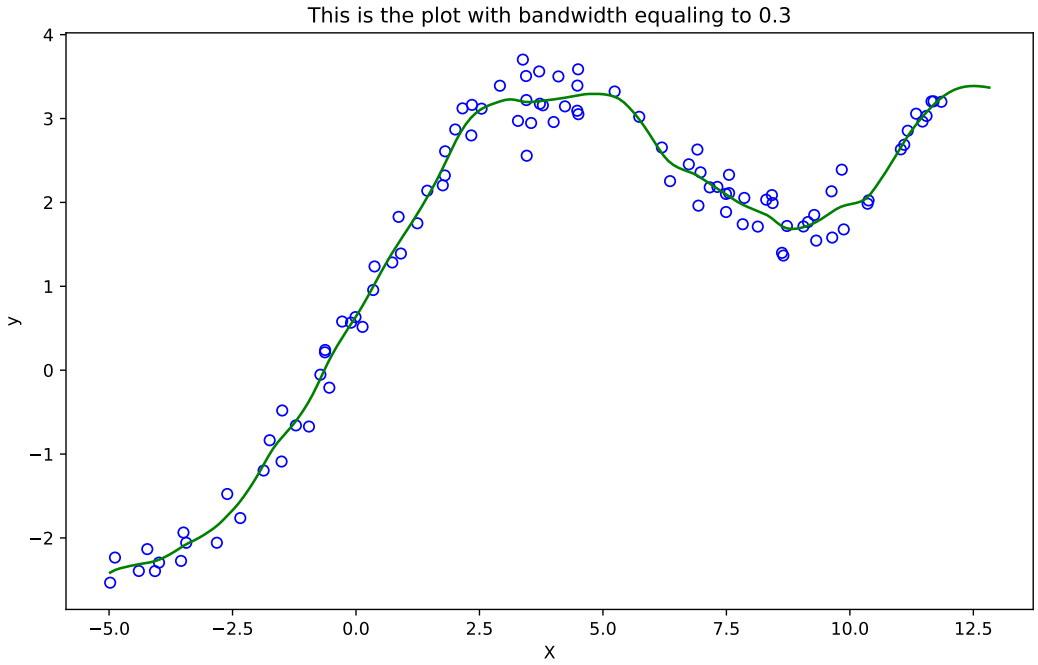
```
import math
bandwidth = 0.8
def LWLR(X, y, bandwidth):
    # Choose each point between the minimum and the maximum as with step 0.1 as the center points
    x_list = np.arange(min(X[:,1]), max(X[:,1]) + 1, 0.1)
    prediction_list = []
    for k in range(len(x_list)):
        R = np.zeros((n, n))
        for i in range(n):
            center = x_list[k]
            R[i,i] = math.exp(-(center - X[i,1])**2/(2*bandwidth**2))
        # Calculate the theta matrix and then get the predictions:
        theta = np.linalg.inv(X.T.dot(R).dot(X)).dot(X.T).dot(R).dot(y)
        prediction_list.append(theta[0] + theta[1]*x_list[k])
    plt.figure(figsize=(10,6))
    plt.scatter(X[:,1], y, facecolors='none', edgecolors='blue')
    plt.plot(x_list, prediction_list, color='green')
    plt.title("This is the plot with bandwidth equaling to {}".format(bandwidth))
    plt.xlabel("X")
    plt.ylabel("y")
    plt.show()
LWLR(X, y, 0.8)
```

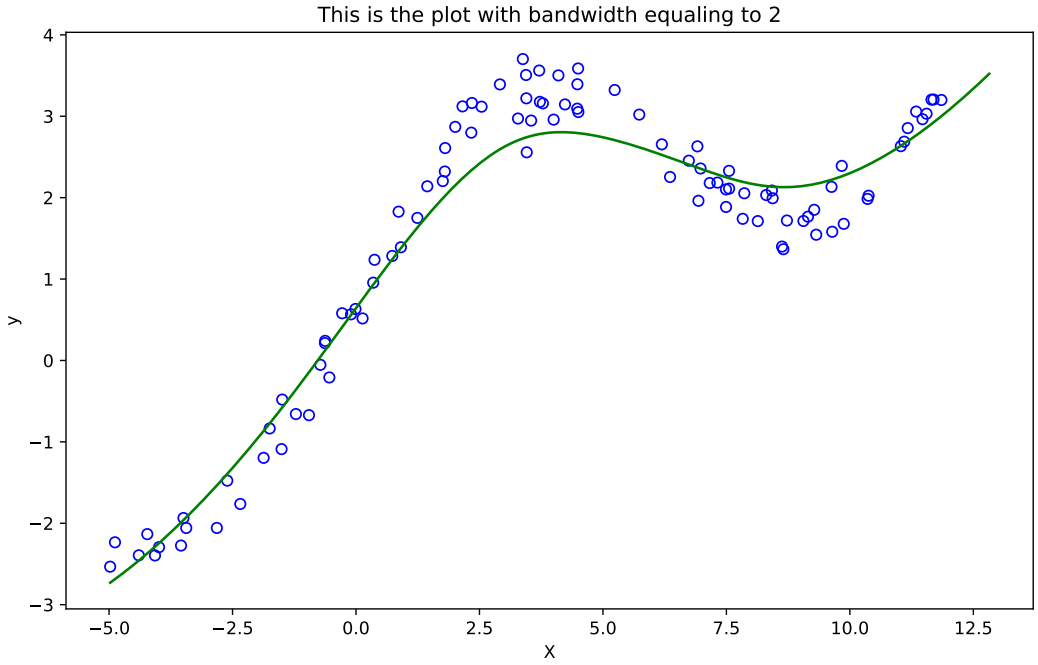


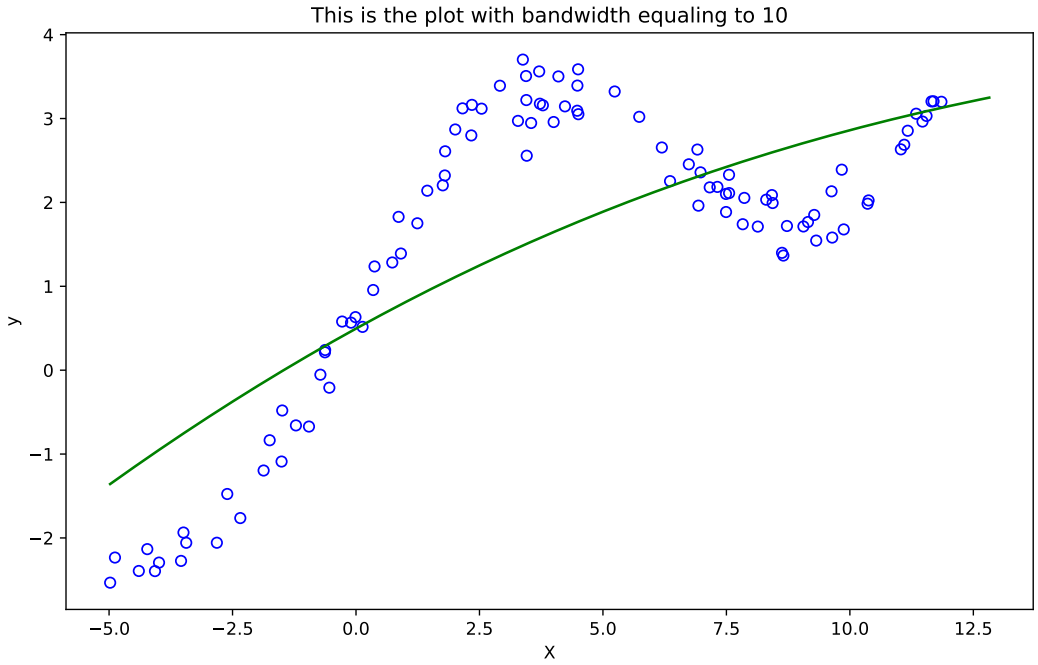
iii. Initialize a bandwidth lists, and apply the above Python function iteratively:

```
bandwidth_list = [.1,.3,2,10]
for i in bandwidth_list:
    LWLR(X, y, i)
```




This is the plot with bandwidth equaling to 0.3
y
X
4
3
2
1
0
−1
−2
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5


This is the plot with bandwidth equaling to 2
y
X
4
3
2
1
0
−1
−2
−3
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5



So, we can observe that when τ is too small, like 0.2 or 0.3, the model is obviously overfitting. And reversely, when when τ is too large, like 10, the model is underfitting.

Question 4. Derivation and Proof

(a)

Plug in $h(x) = w_1 x + w_0$ into the formula of the mean squared error, we can get:

$$\begin{aligned} L &= \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - h(x^{(i)}))^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}[y^{(i)} - (w_1 x^{(i)} + w_0)]^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}[y^{(i)2} - 2y^{(i)}(w_1 x^{(i)} + w_0) + (w_1 x^{(i)} + w_0)^2]. \end{aligned}$$

In order to get the minimum value for L, we can set $\frac{\partial L}{\partial w_0} = 0$ and $\frac{\partial L}{\partial w_1} = 0$, so we can have the following two relations:

$$\begin{aligned} \frac{\partial L}{\partial w_0} &= \frac{1}{2}\sum_{i=1}^{N}[2(w_1 x^{(i)} + w_0) - 2y^{(i)}] \\ &= \sum_{i=1}^{N}[w_1 x^{(i)} + w_0 - y^{(i)}] \\ &= Nw_0 + w_1\sum_{i=1}^{N}x^{(i)} - \sum_{i=1}^{N}y^{(i)} \\ &= N(w_0 + w_1\overline{X} - \overline{Y}) \\ &= 0, \end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= \frac{1}{2}\sum_{i=1}^{N}[2x^{(i)}(w_1x^{(i)}+w_0)-2x^{(i)}y^{(i)}]\\ &= \sum_{i=1}^{N}[w_1x^{(i)2}+w_0x^{(i)}-x^{(i)}y^{(i)}]\\ &= w_1\sum_{i=1}^{N}x^{(i)2}+w_0\sum_{i=1}^{N}x^{(i)}-\sum_{i=1}^{N}x^{(i)}y^{(i)}\\ &= w_1\sum_{i=1}^{N}x^{(i)2}+Nw_0\overline{X}-\sum_{i=1}^{N}x^{(i)}y^{(i)}\\ &= 0.\end{aligned}$$

So to simplify, we can have $w_0 = \overline{Y} - w_1\overline{X}$ and then we can plug this result into the other formula, and we get:

$$\begin{aligned}w_1 &= \frac{\sum_{i=1}^{N}x^{(i)}y^{(i)}-Nw_0\overline{X}}{\sum_{i=1}^{N}x^{(i)2}}\\ &= \frac{\sum_{i=1}^{N}x^{(i)}y^{(i)}-N(\overline{Y}-w_1\overline{X})\overline{X}}{\sum_{i=1}^{N}x^{(i)2}},\end{aligned}$$

so w_1 can be simplified to $w_1 = \frac{\frac{1}{N}\sum_{i=1}^{N}x^{(i)}y^{(i)}-\overline{XY}}{\frac{1}{N}\sum_{i=1}^{N}x^{(i)2}-\overline{X}^2}$.

(b)

i.

First, prove $\Leftarrow$:

since $\mathbf{A}$ can be written in the spectral decomposition that $\mathbf{A} = \mathbf{U}\Lambda\mathbf{U}^T$ with $\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_d)$. Here in order to show that $\mathbf{A}$ is PD given all the λs are positive, we can expand in this way:

$$\begin{aligned}\mathbf{z}^T\mathbf{A}\mathbf{z} &= \mathbf{z}^T\mathbf{U}\Lambda\mathbf{U}^T\mathbf{z}\\ &= (\mathbf{U}^T\mathbf{z})^T\Lambda(\mathbf{U}^T\mathbf{z}).\end{aligned}$$

Now, we can let $\mathbf{y} = \mathbf{U}^T\mathbf{z}$, and represent the original equation with its quadratic form, we can see:

$$\begin{aligned}\mathbf{z}^T\mathbf{A}\mathbf{z} &= \mathbf{y}^T\Lambda\mathbf{y}\\ &= \lambda_1y_1^2+\lambda_2y_2^2+...+\lambda_dy_d^2.\end{aligned}$$

Therefore, given that all the λs are positive, the sum of the product with a sqaured term must be postive, if we know $\mathbf{z} \neq \mathbf{0}$, we can see that $\mathbf{y} \neq \mathbf{0}$.

Therefore we can prove the direction of $\mathbf{A}$ is PD if $\lambda_i > 0$ for each i.

Next, prove $\Rightarrow$:

given that $\mathbf{A}$ is PD, we can expand that $\mathbf{z}^T\mathbf{U}\Lambda\mathbf{U}^T\mathbf{z} > 0$ for all $\mathbf{z} \neq \mathbf{0}$. Similarly, we can let $\mathbf{y} = \mathbf{U}^T\mathbf{z}$ again, so that we can know that $\mathbf{y}^T\Lambda\mathbf{y} > 0$ is true, and again, quadratic form of this can be written:

$$\mathbf{y}^T\Lambda\mathbf{y} = \lambda_1y_1^2+\lambda_2y_2^2+...+\lambda_dy_d^2,$$

the only way to make this equation positive given any $\mathbf{z}$ that can lead to different $\mathbf{y}$ is to make every λ positive, so we can finish this direction of proof.

ii.

For normal linear regression, as we can see the solution for coefficient *theta* is $(\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}$, and for ridge regression, the solution is the addition of a regularization term, thus it is $(\Phi^T\Phi+\beta\mathbf{I})^{-1}\Phi^T\mathbf{y}$. So the symmetric matrix for the two different regressions are $\Phi^T\Phi$ and $\Phi^T\Phi+\beta\mathbf{I}$ correspondingly.

Now use the spectral decomposition to decompose these two symmetric matrices, we can suppose that

$$\Phi^T\Phi = \mathbf{U}\Lambda\mathbf{U}^T,$$

where the eigenvalues, as mentioned in the statement of this problem, are diagonal elements on the Λ matrix. So we can plug this equation into $\Phi^T\Phi+\beta\mathbf{I}$, and utilizing the property that $\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}$, we can further simplify:

$$\begin{aligned}\Phi^T\Phi+\beta\mathbf{I} &= \mathbf{U}\Lambda\mathbf{U}^T+\beta\mathbf{I}\\ &= \mathbf{U}\Lambda\mathbf{U}^T+\mathbf{U}\beta\mathbf{I}\mathbf{U}^T\\ &= \mathbf{U}(\Lambda+\beta\mathbf{I})\mathbf{U}^T,\end{aligned}$$

therefore, we can easily see that for the ridge regression, the eigenvalues for the symmetric matrix $\Phi^T\Phi+\beta\mathbf{I}$, are the diagonal elements that are equal to $\lambda_i+\beta$ at the i-th position on the diagonal line. Here according to SVD method, we can see that the eigenvalues are equal to the singualr values, thus we can draw the conclusion that the ridge regression has an effect of shifting all singular values by a β.

Similar with the last problem, to solve that the matrix $\Phi^T\Phi+\beta\mathbf{I}$ is PD for any $\beta>0$, we again use the quadratic form, again let $\mathbf{y}=\mathbf{U}^T\mathbf{z}$:

$$\mathbf{z}^T(\Phi^T\Phi+\beta\mathbf{I})\mathbf{z} = (\lambda_1+\beta)y_1^2+(\lambda_2+\beta)y_2^2+...+(\lambda_d+\beta)y_d^2.$$

To prove that this formula is always positive, we need to first take a consideration to $\Phi^T\Phi$, show it is PSD:

$$\mathbf{z}^T(\Phi^T\Phi)\mathbf{z} = (\Phi\mathbf{z})^T(\Phi\mathbf{z}) = ||\Phi\mathbf{z}||_2^2 \geq 0.$$

Since for any $\mathbf{z}$, the above equation holds, so the eigenvalues for $\Phi^T\Phi$ should always be non-negative, therefore, go back to $\Phi^T\Phi+\beta\mathbf{I}$ with the eigenvalue to be $\lambda_i+\beta$, if $\beta>0$, $\lambda_i+\beta$ must be absolutely positive, thus $\Phi^T\Phi+\beta\mathbf{I}$ must absolutely be PD.

Collaboration For this homework, I formed all the solutions my own but discussed the problem sets together with people in my assignment group, they are Junwei Deng, Dongjian Chen and Yunzhe Jiang.