Q1

February 11, 2018

ML Assignment 1
Madhur Singhal
2015CS10235

0.1 Q1 Linear Regression

In this problem, we will implement least squares linear regression to predict density of wine based on its acidity. We will also use 3D plotting and contour plotting to observe how our gradient descent converges. The loss function and update equations are shown below.

$$J(\theta) = \frac{1}{2}[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2]$$

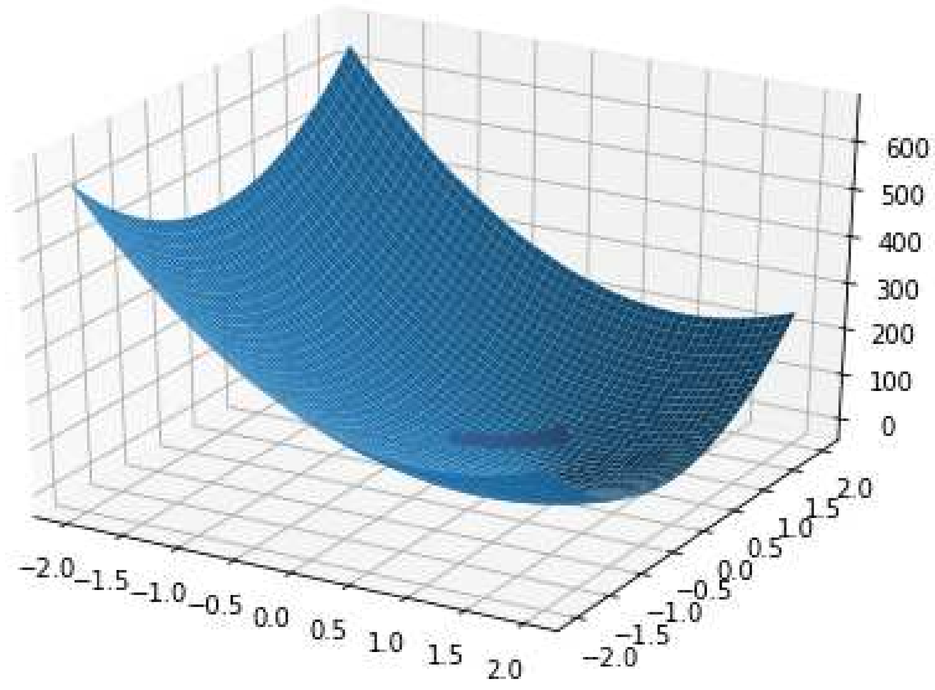
$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}]$$

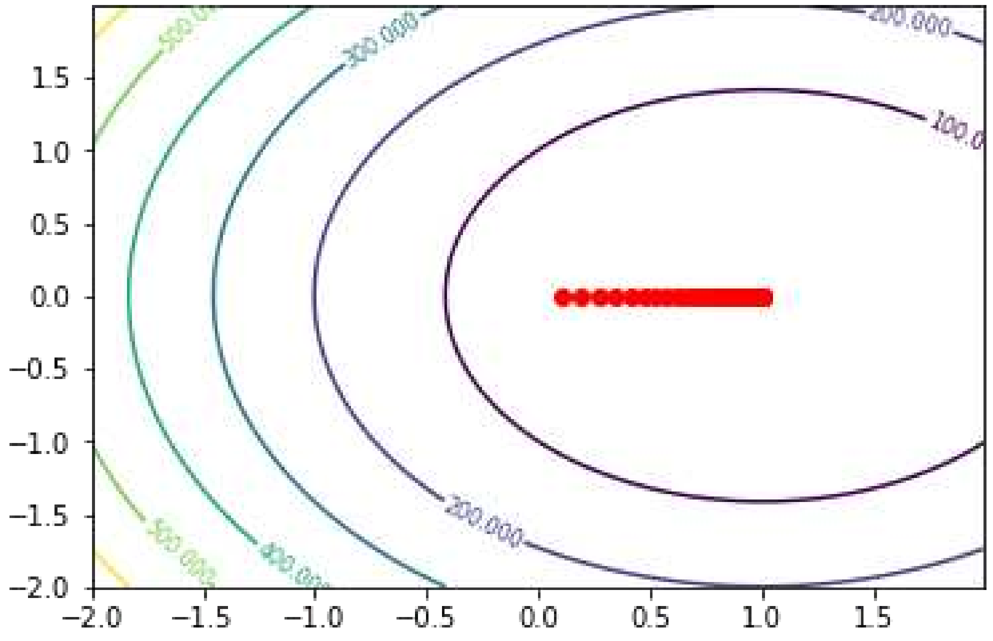
We use gradient descent to optimize the cost function and find out the values of parameters.

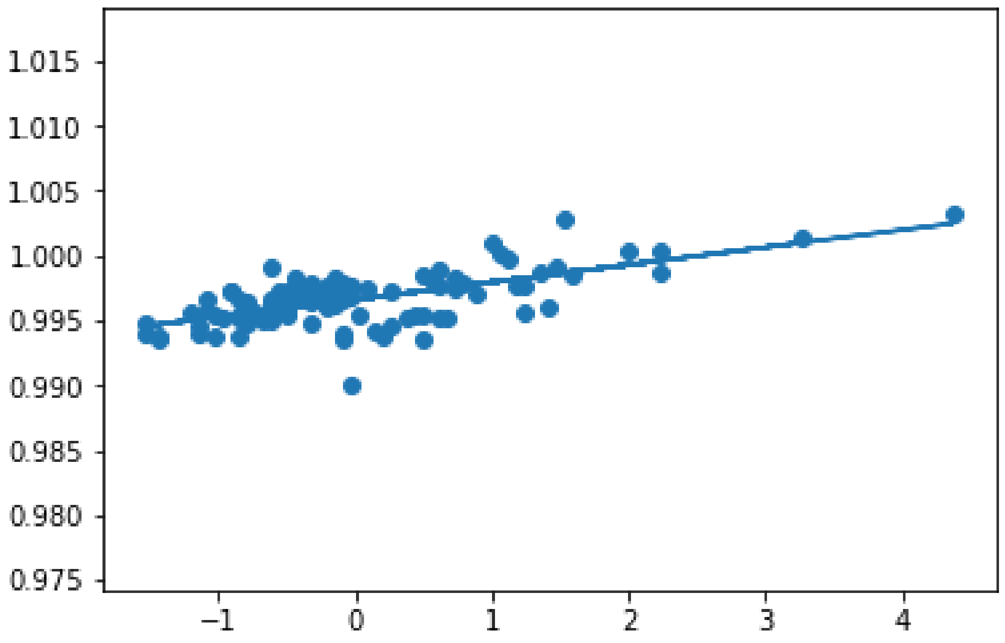
Since this is just a report so proper animation is not possible here, thus to view the plots interactively I have provided a separate python file in the folder (plots.py). In this report I have shown the final images produced only.

Below are the three plots produced for learning rate of 0.0001 and the parameters learnt.

```
Out[6]: [<matplotlib.lines.Line2D at 0x7effa0253c88>]
```

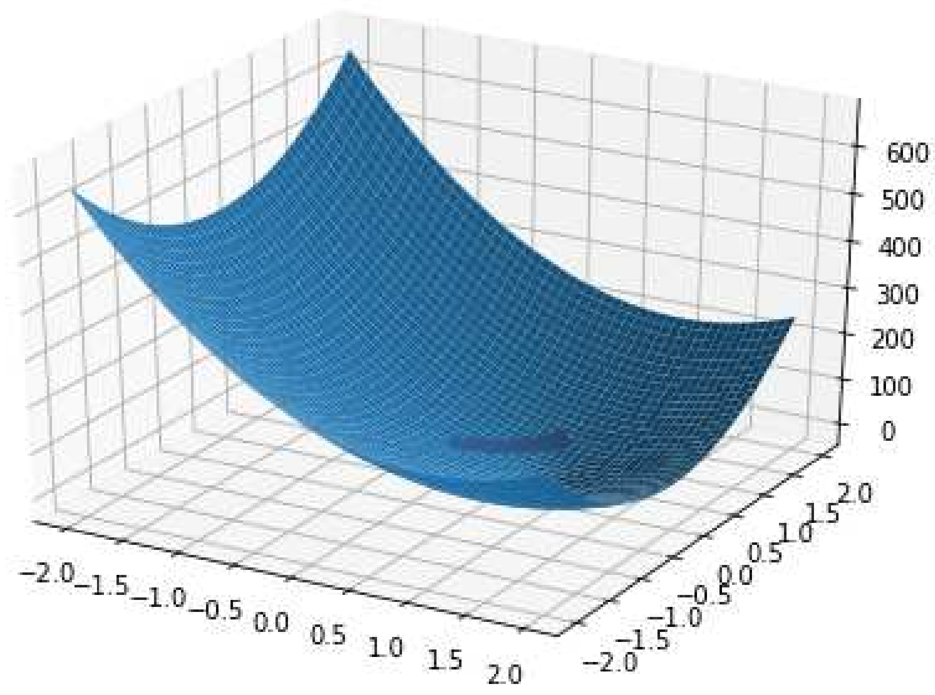

600
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

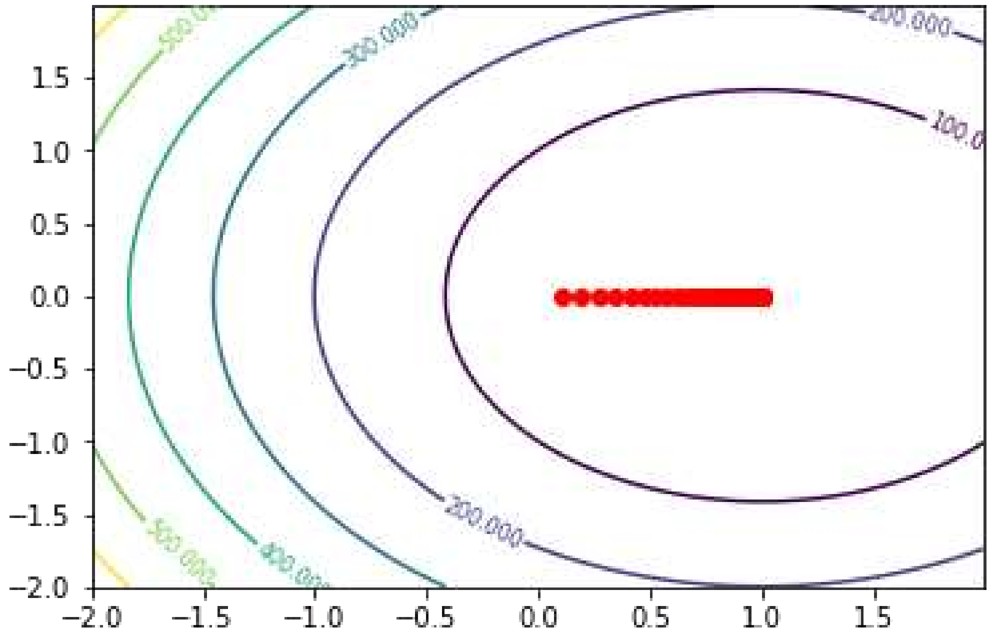

500.000
300.000
200.000
100.000
400.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

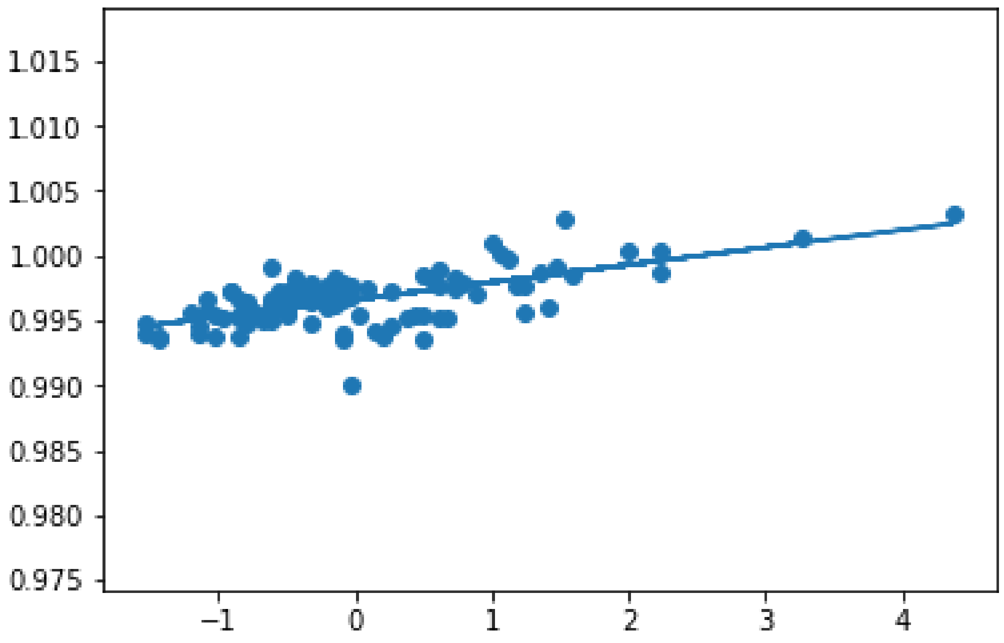


Finally we do the last part and see the contours for different learning rates.
For LR = 0.001, following are the stats and the plots obtained.

```
---------------
LR -  0.001
Iterations needed -  122
Loss -  0.00011947940058425674
Change in Loss -  9.839792527869054e-11
Theta --  [0.00134019 0.9966172 ]
----------------
```

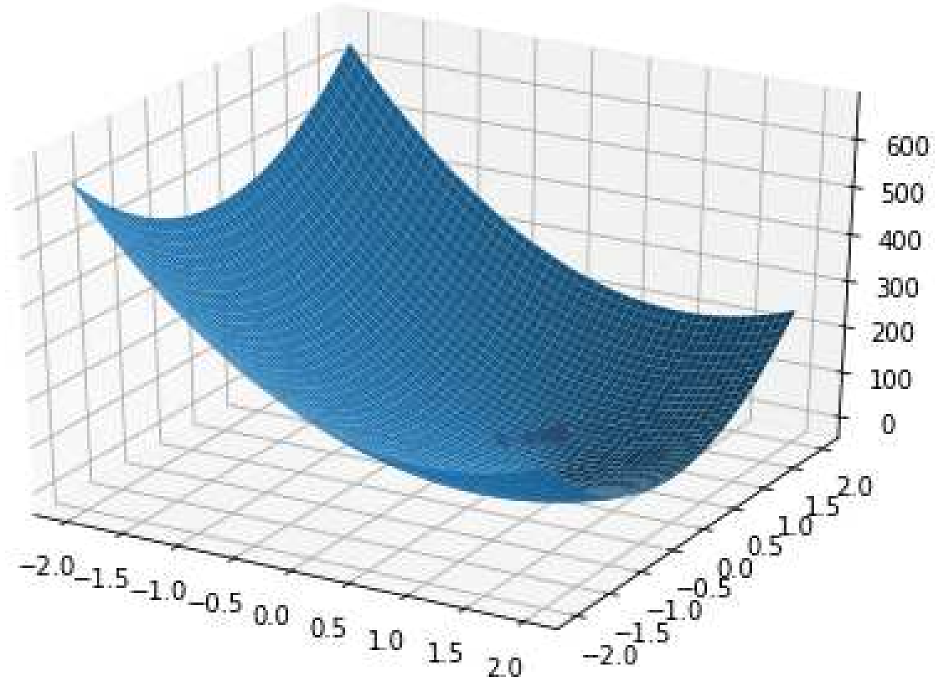

600
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

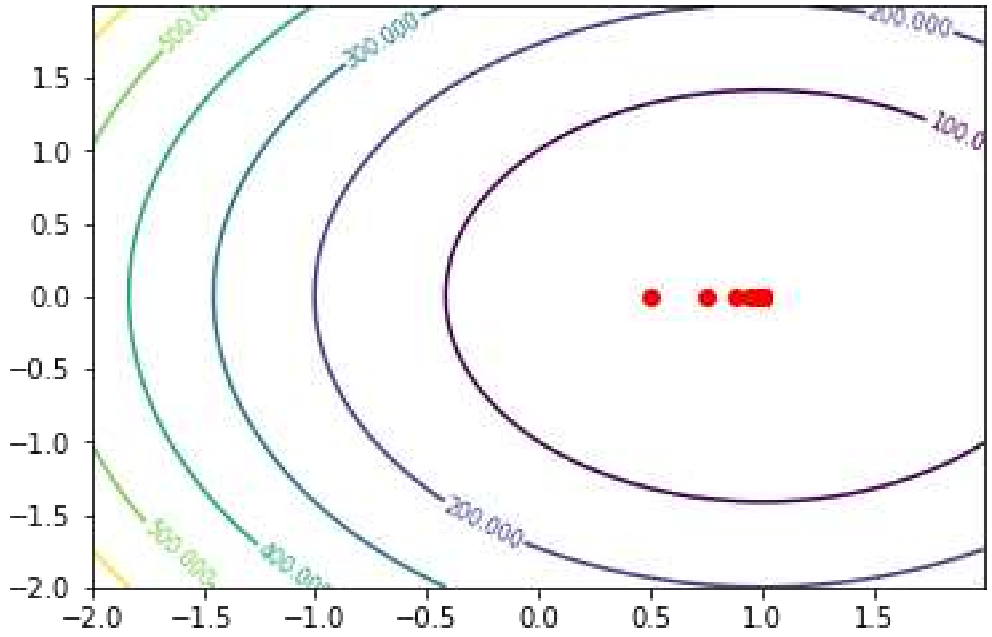

500.000
300.000
200.000
100.0
400.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

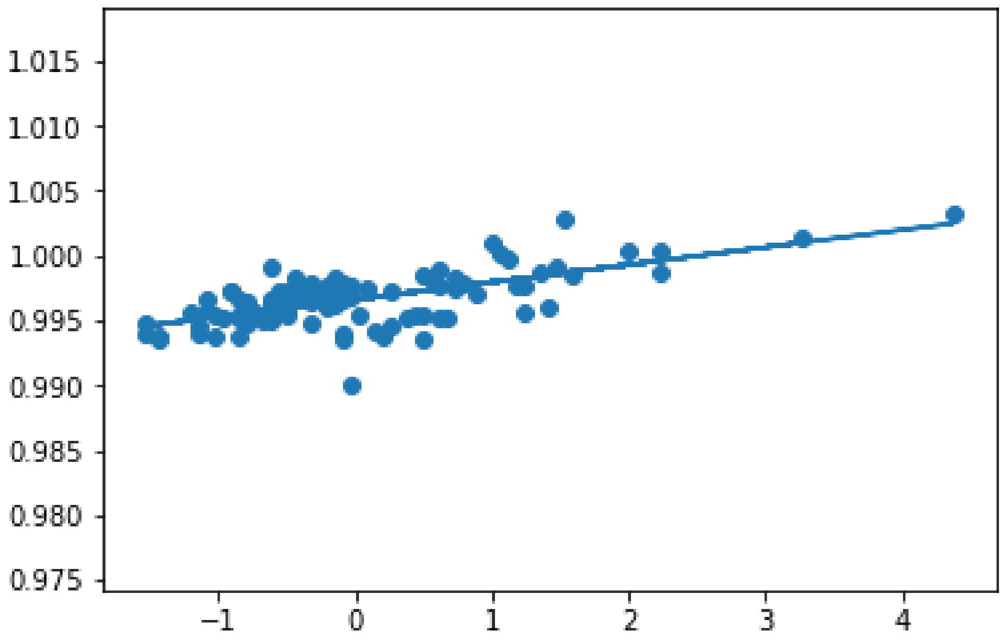


For LR = 0.005, following are the stats and the plots obtained.

```
---------------
LR -  0.005
Iterations needed -  22
Loss -  0.00011947899239035003
Change in Loss -  3.3875952444670926e-11
Theta --  [0.0013402  0.99661962]
----------------
```

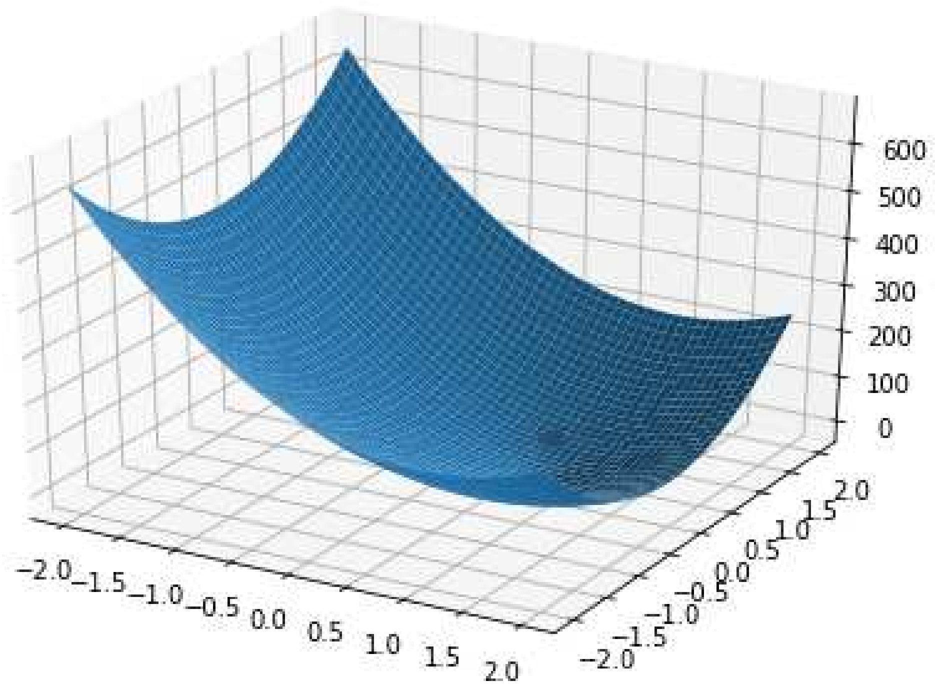

600
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

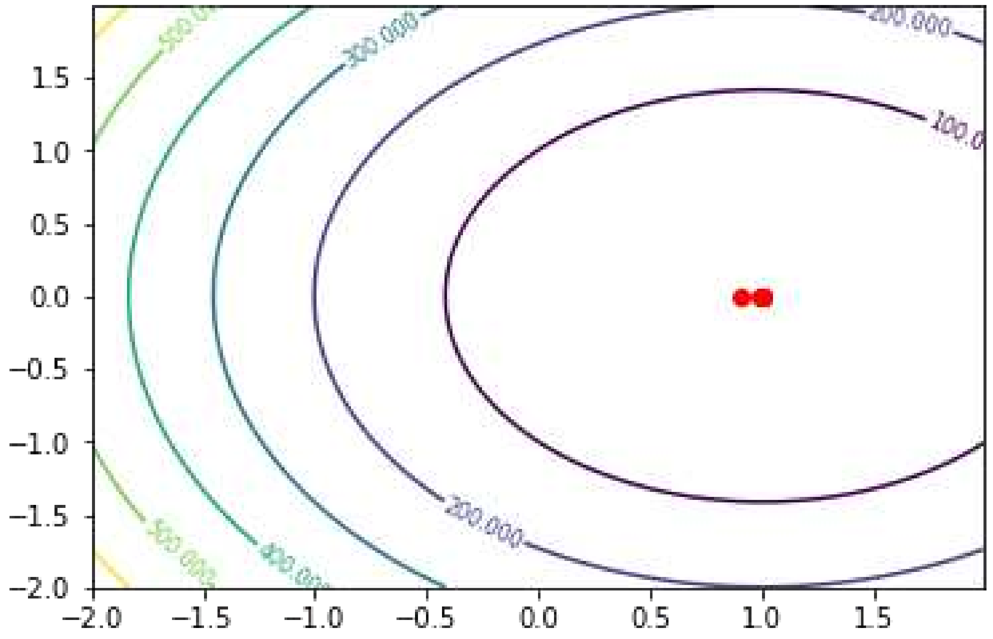

500.000
300.000
200.000
100.0
200.000
400.000
500.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

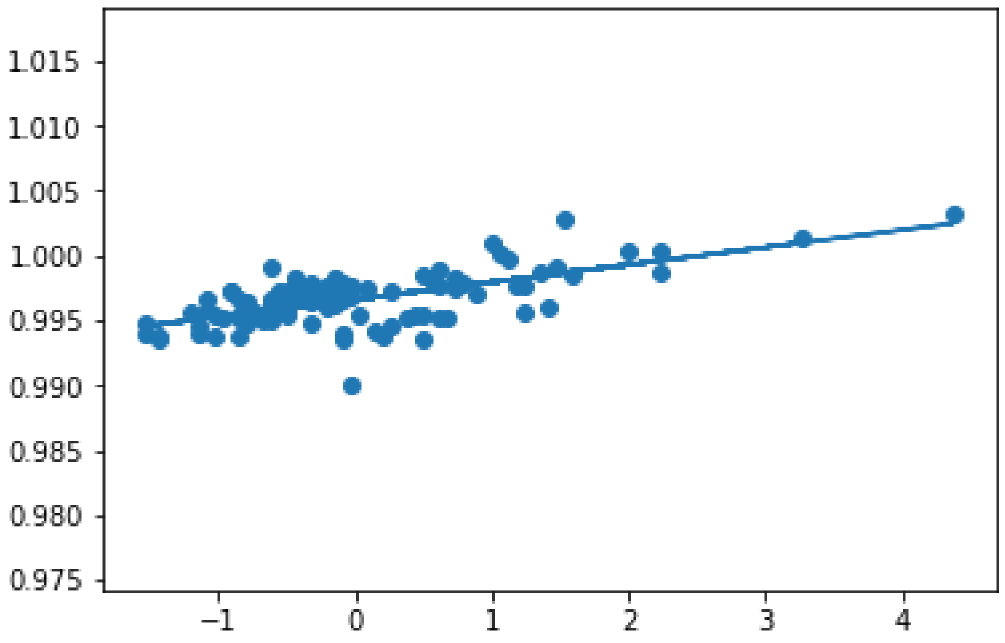


For LR = 0.009, following are the stats and the plots obtained.

```
---------------
LR -  0.009
Iterations needed -  8
Loss -  0.0001194789815949993
Change in Loss -  4.916604441824492e-11
Theta --  [0.0013402 0.99662  ]
----------------
```

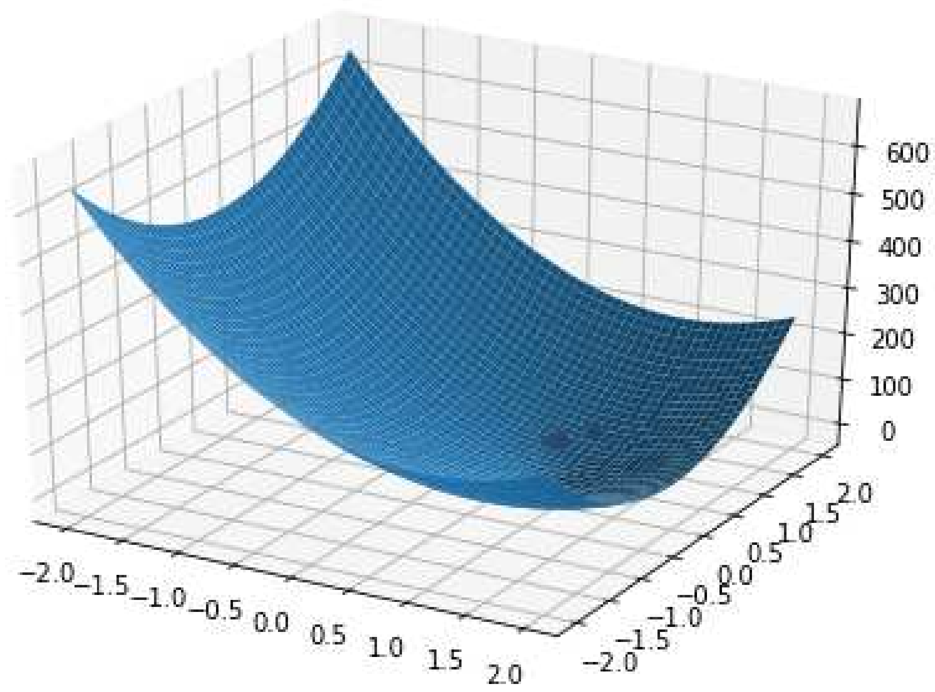

600
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

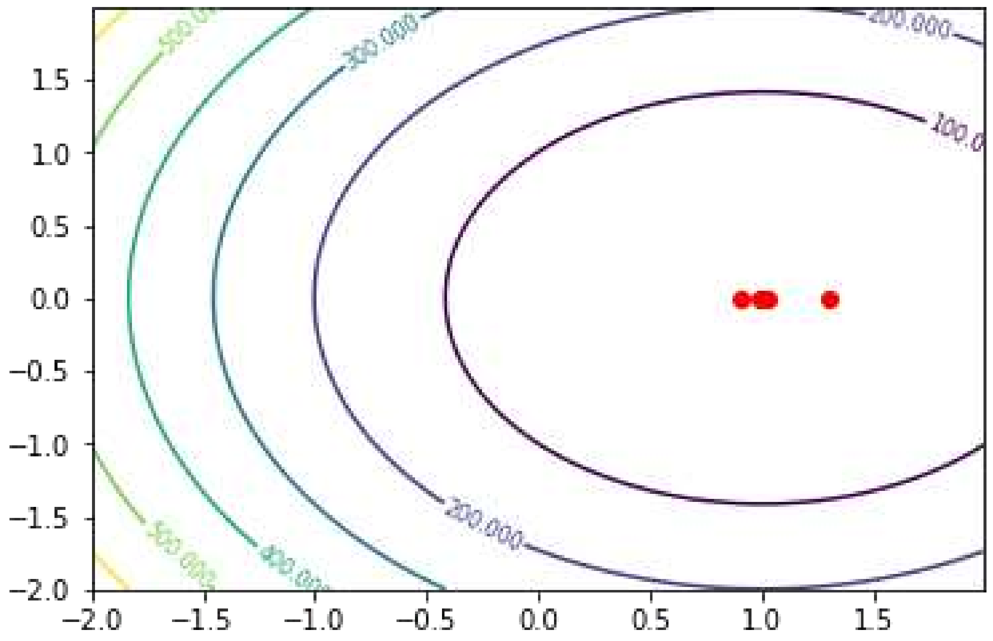

500.000
300.000
200.000
100.0
200.000
500.000
400.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

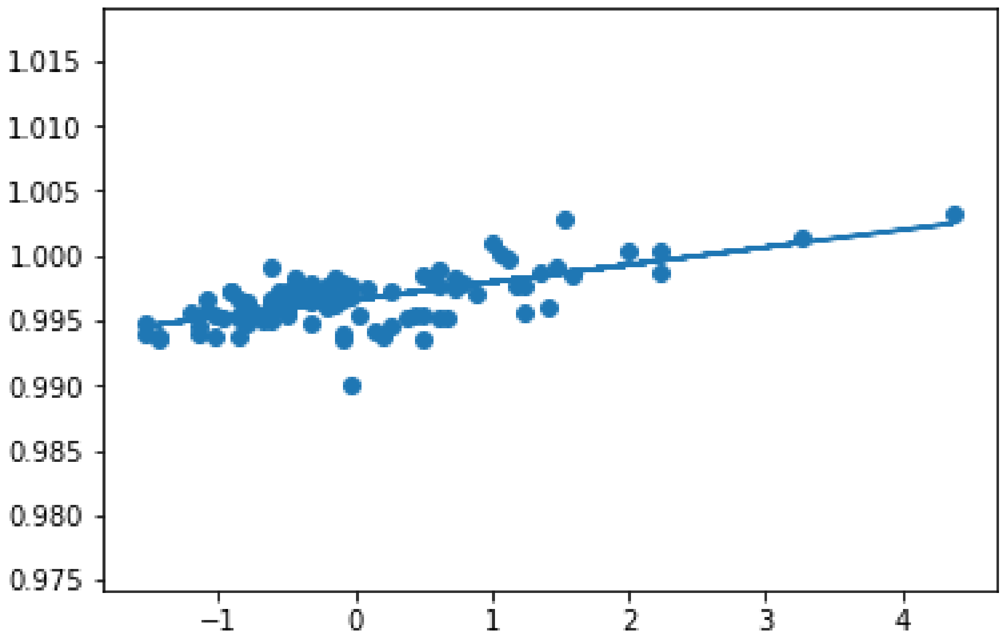


For LR = 0.013, following are the stats and the plots obtained.

```
---------------
LR -  0.013
Iterations needed -  14
Loss -  0.00011947898236072468
Change in Loss -  1.2763846511838761e-11
Theta --  [0.0013402  0.99662026]
----------------
```

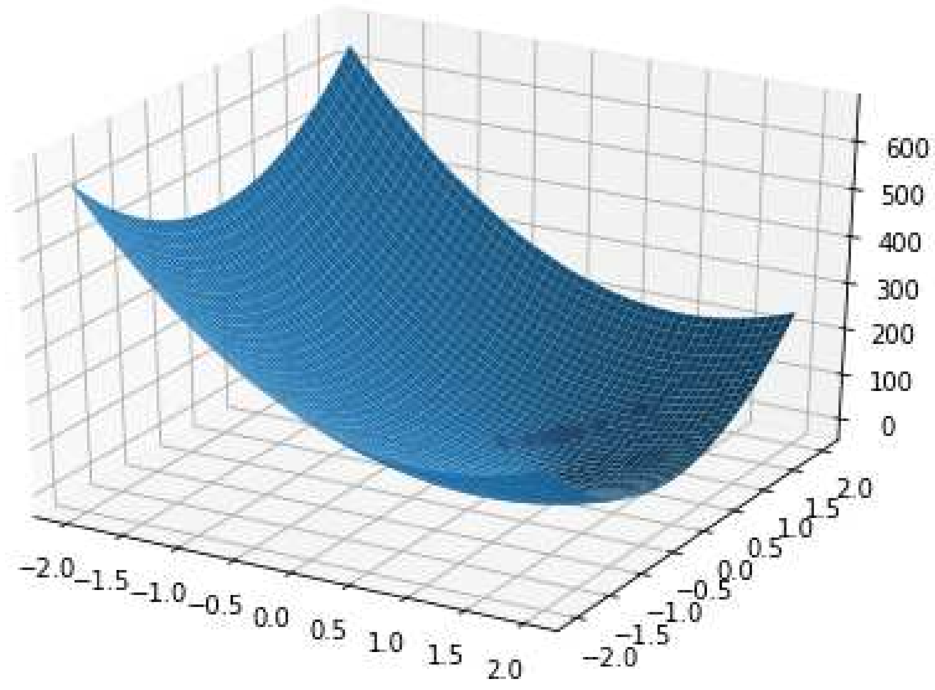

600
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


500.000
300.000
200.000
100.000
200.000
500.000
400.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

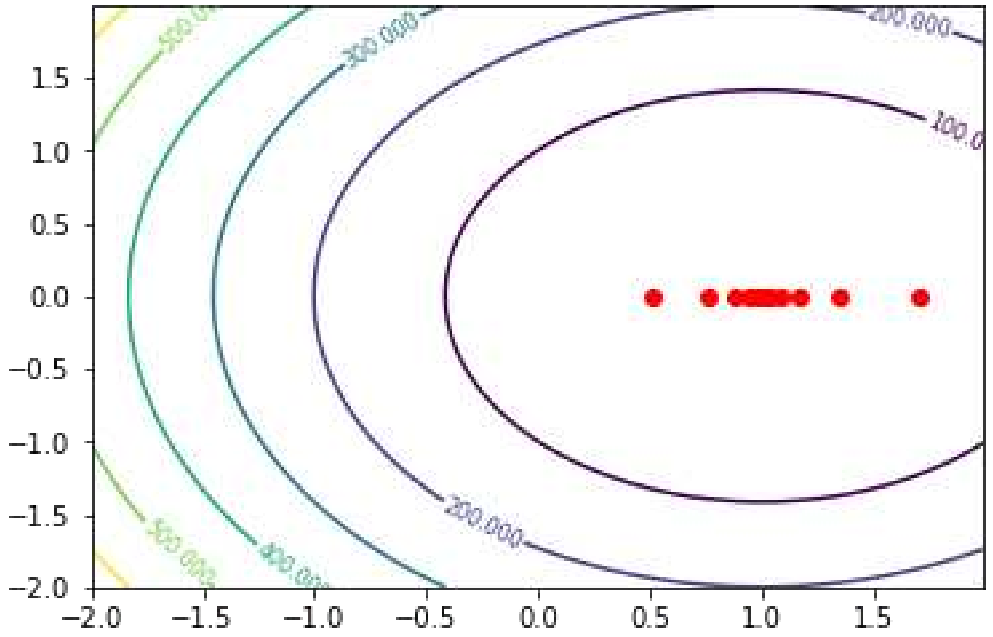


For LR = 0.017, following are the stats and the plots obtained.

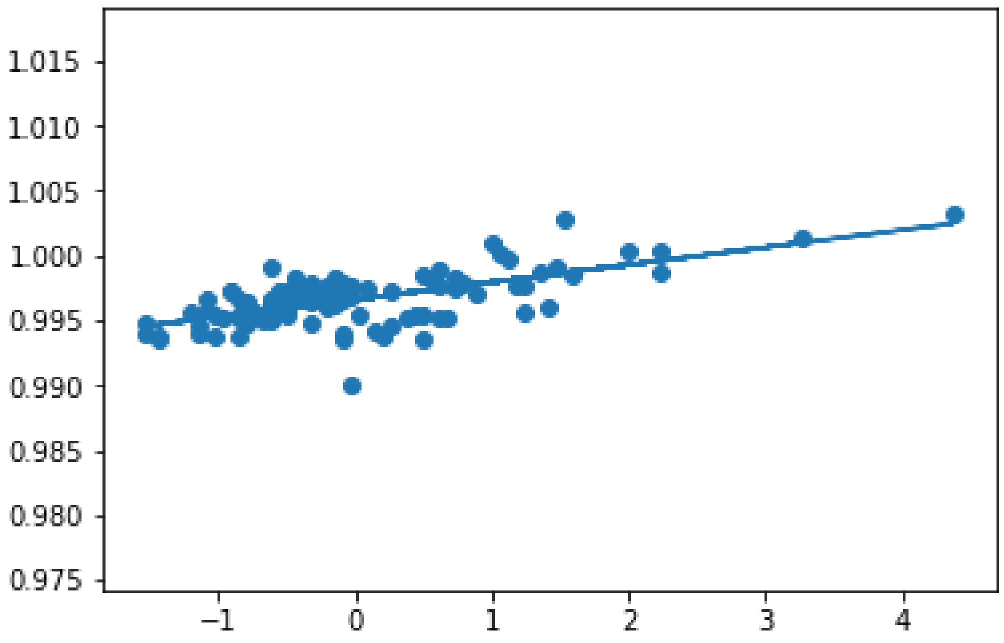
```
---------------
LR -  0.017
Iterations needed -  39
Loss -  0.00011947906494412838
Change in Loss -  8.72680353545055e-11
Theta --  [0.00134019 0.99661881]
----------------
```



600
500
400
300
200
100
0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0



500.000
400.000
300.000
200.000
100.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5



For the other two values of LR given, gradient descent diverges and python gives an overflow error after some iterations.

The main observation is that a moderate value of learning rate (0.009 here) is the best as there are problems on both sides of the spectrum. For large learning rates gradient descent can oscillate about the minimum for quite some time as seen here with LR = 0.017. Also gradient descent can diverge too if the learning rate is too high (more than 0.2 here). For small learning rates, gradient descent takes a large time to converge as can be seen by comparing the different number of iterations taken using different learning rates. With LR = 0.009 our gradient descent converges in just 8 iterations with a final loss of 10^{-4}.

Q2

February 11, 2018

```
<IPython.core.display.HTML object>
```

0.1 Locally Weighted Linear Regression

In this problem, we will generalize the ideas of linear regression to implement locally weighted linear regression where we want to weigh different training examples differently.

This is the initial loss with parameters equal to zero.

```
Out[21]: 121.390984336008
```

We obtain the parameters learned by unweighted linear regression implemented using normal equation, the solution being

$$\theta = (X^T X)^{-1} X^T Y$$

The first output is the **weight** and second is the **bias**.

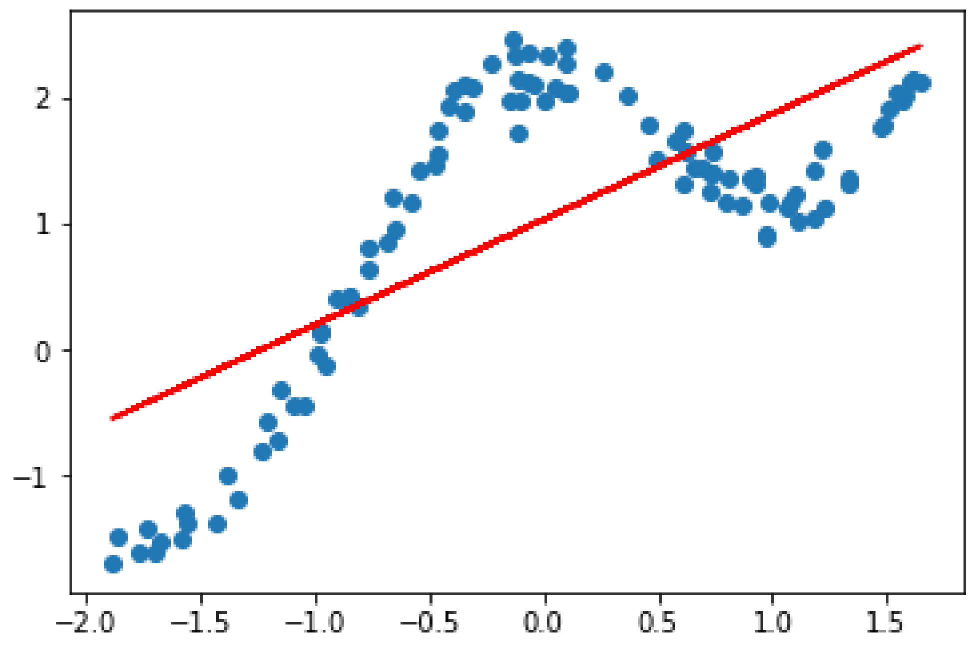
```
Out[40]: array([ 0.83519315,  1.03128116])
```

This is the new loss, it is apparent that the loss has decreased but not by too much.

```
Out[42]: 33.336562701909685
```

This is the plot of unweighted linear regression output and the data, it is clear that our regression has missed important underlying patterns in the data.

```
Out[43]: [<matplotlib.lines.Line2D at 0x7f82fb8d0ef0>]
```

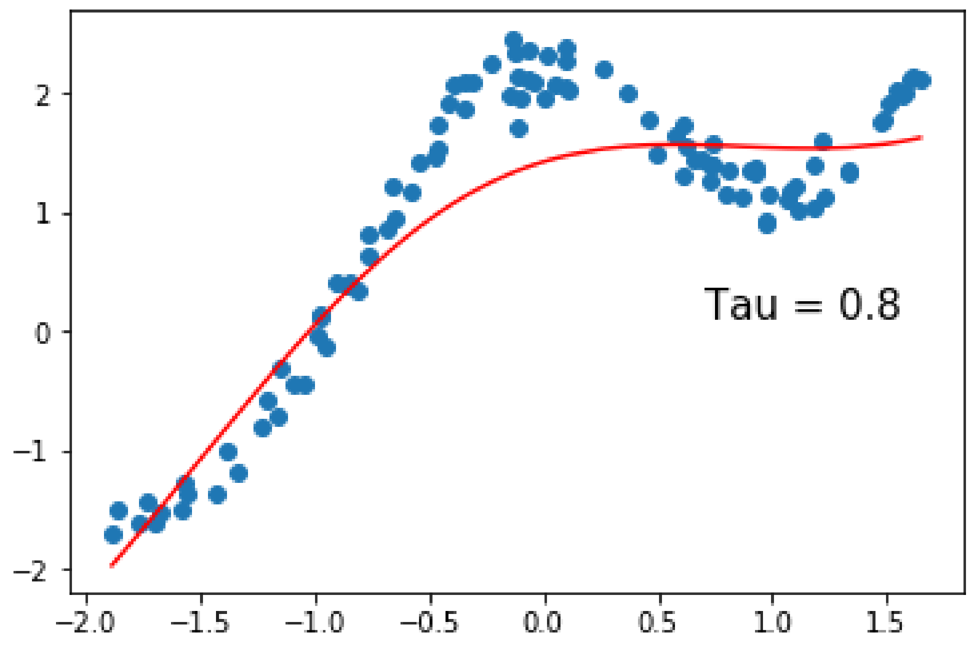


After this we implement locally weighted linear regression. This is not learning per se, as the parameters get learnt at inference time. I did the derivation and found that the normal equation solution for locally weighted LR is

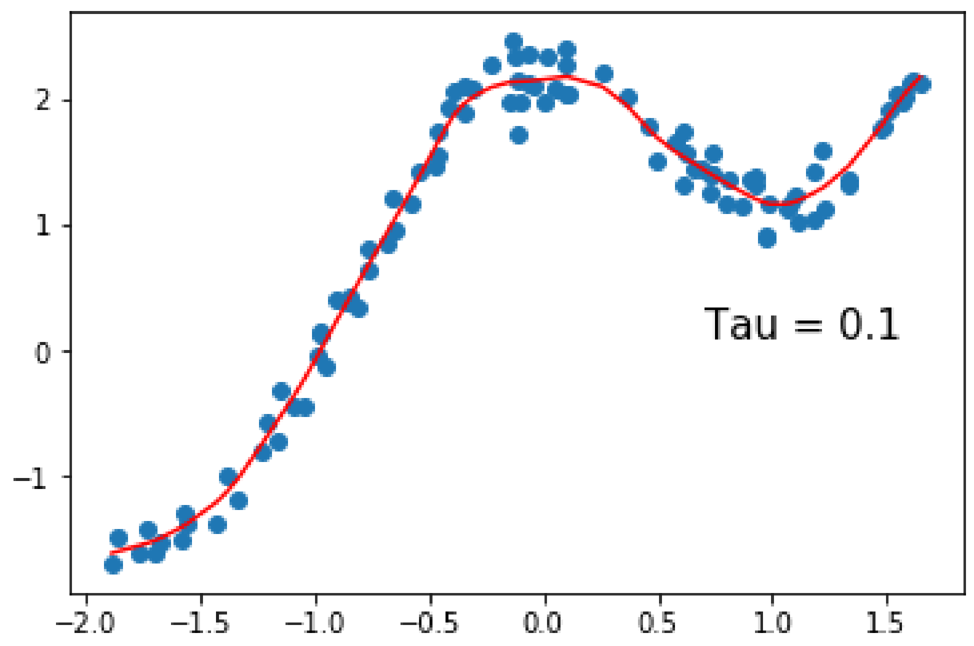
$$\theta = (X^T W X)^{-1} X^T W Y$$

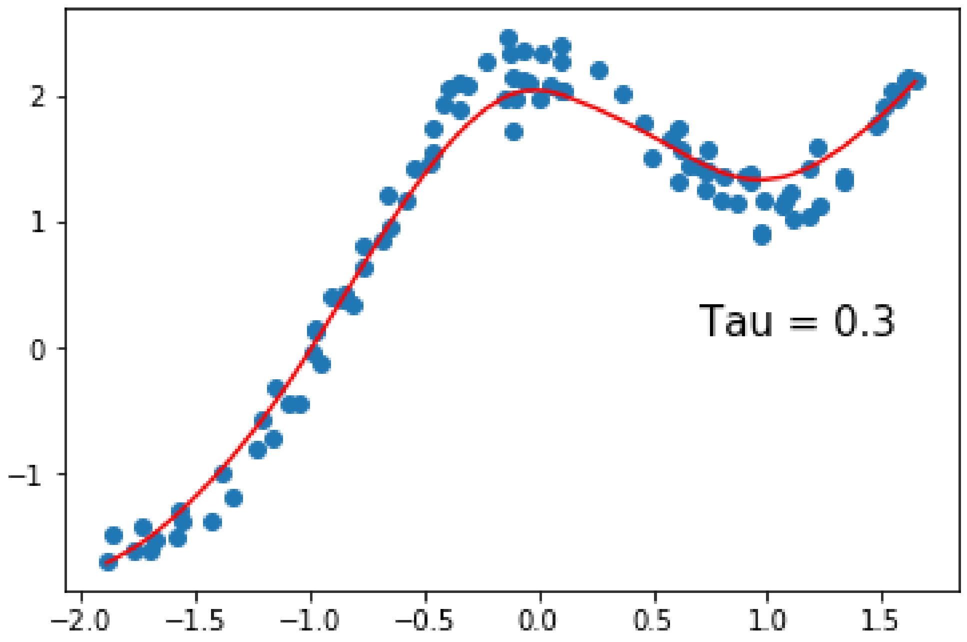
Below is the plot of the output we obtain for locally weighted LR with bandwidth parameter $\tau = 0.8$.

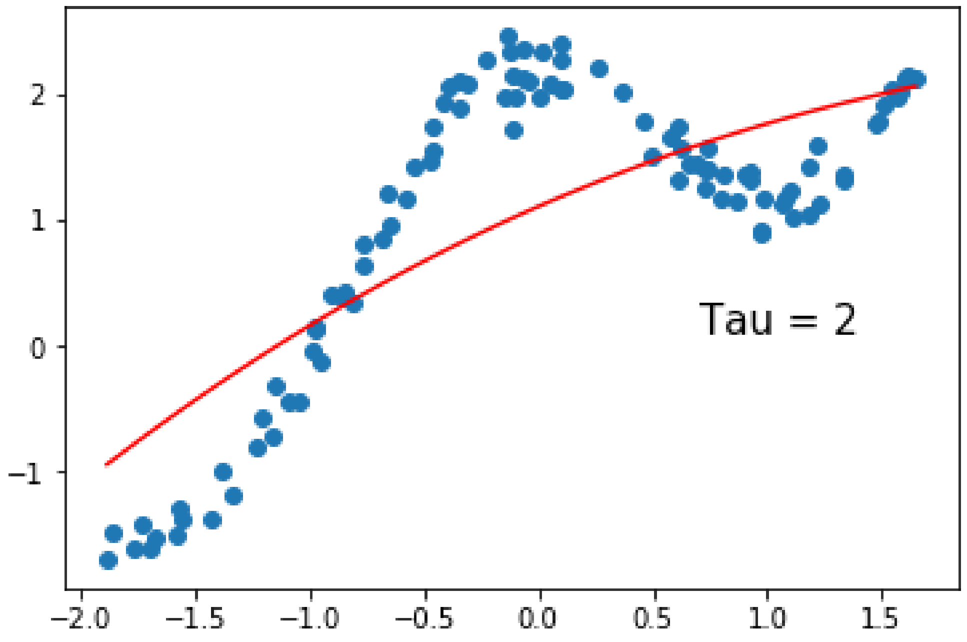
```
Out[49]: Text(0.7,0.1,'Tau = 0.8')
```

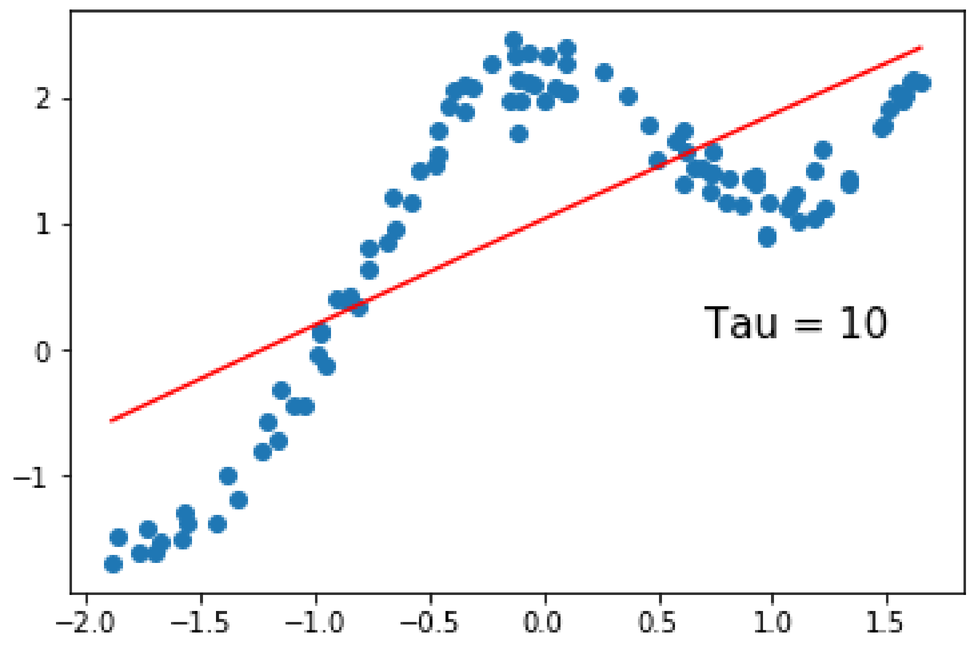


Finally we vary the bandwidth τ and obtain the following plots. It seems that a value of 0.3 for the bandwidth results in the best regression output.




Tau = 0.3
2
1
0
−1
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Tau = 2
2
1
0
−1
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5



When tau is too small the model starts fitting the small errors in the data and not the underlying pattern, on the other hand when tau is too large the "local" part matters less and we approach unweighted regression, that is the regression starts giving near equal weight to all points no matter how close they are to the target.

Q3

February 11, 2018

0.1 Logistic Regression using Newton's Method

In this part we implement classification using logistic regression using Newton's method for optimization.

We calculate the Grad and Hessian matrices and use the following update rule to update our parameters (initialized at zero).

$$\theta_{t+1} = \theta_t - H(f(\theta_t))^{-1} \nabla f(\theta_t)$$

Following is the output of our optimization. The first parameter corresponds to the first feature, second to the second feature and last one is the bias. It is clear that our algorithm converges sufficiently in only one iteration and after this all the parameters are just scaled by some number. After five iterations overflow is encountered and the algorithm terminates.

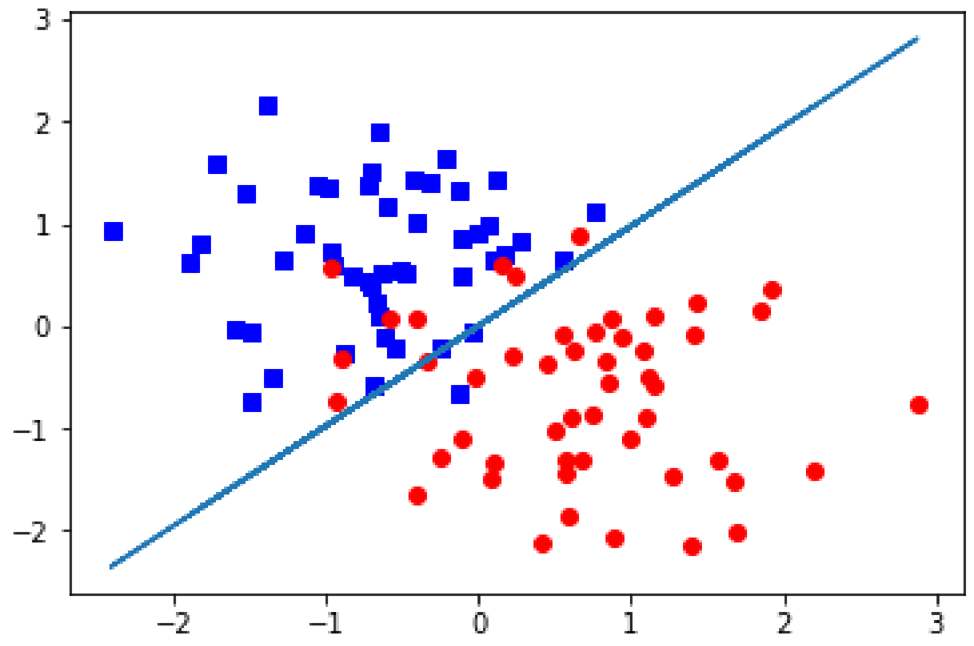
```
Iteration =  1
Theta array =  [ -8.97341408e-01   9.20112283e-01   3.85905135e-16]
Iteration =  2
Theta array =  [-4.88262336  5.02247361 -0.28086918]
Iteration =  3
Theta array =  [-326.74629425  321.51992179  -63.86126337]
Iteration =  4
Theta array =  [ -5.65776800e+10   5.55708777e+10  -1.15565907e+10]
Overflow in iteration 5


/home/madhur/.local/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning: overfl
```

Thus a reasonable equation of the boundary is $ -8.97341408e-01 * x_0 + 9.20112283e-01* x_1 + 3.85905135e-16

Below is the plot of our decision boundary and the data points, it can be seen that our boundary separates the two classes reasonably well.

```
Out[9]: [<matplotlib.lines.Line2D at 0x7f356c952b00>]
```



3
2
1
0
−1
−2
−2
−1
0
1
2
3

Q4

February 11, 2018

0.1 Gaussian Discriminant Analysis

In this problem, we implement GDA for separating out salmons from Alaska and Canada. Alaska is assigned identifier 0 and Canada is assigned 1.

Firstly we assume that both normal distributions have same covariance matrices and calculate the poisson parameter, means and the covariance matrix. Following are the parameters learned, first value corresponds to the first feature.

```
Phi =  0.5


mu_0 =  [-0.75529433  0.68509431]


mu_1 =  [ 0.75529433 -0.68509431]


Covariance matrix =  [[ 0.42953048 -0.02247228]
 [-0.02247228  0.53064579]]
```

Expanding decision boundary expression to an explicit form gives a complicated expression in terms of covariances, means and determinants of covariances. For same covariance matrix assumption, expression is linear, let it be $k_1 * x_1 + k_2 * x_2 + k_3 = 0$

$$det = np.linalg.det(cov)$$

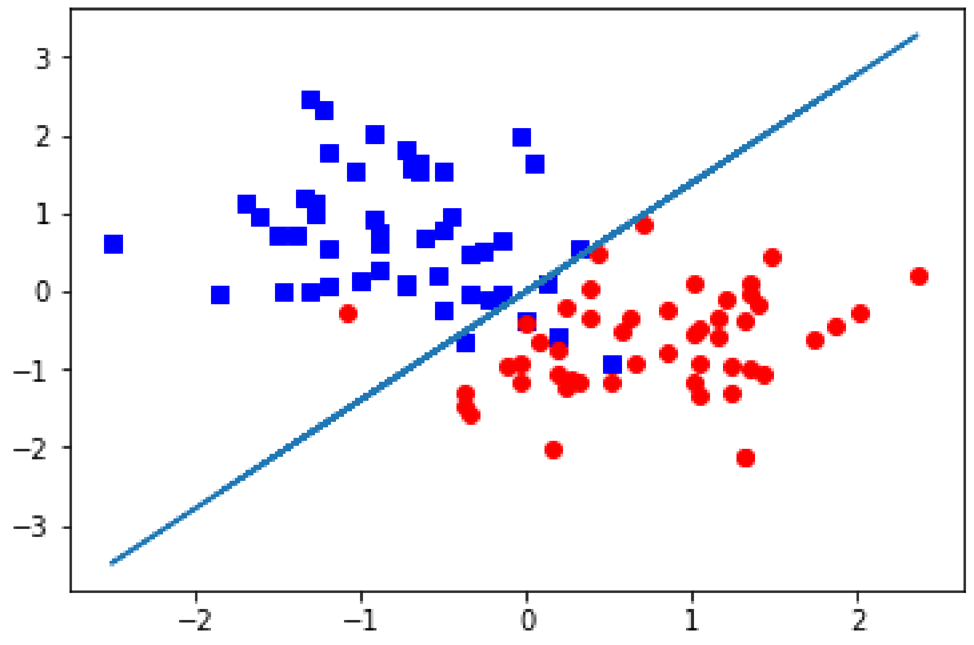
$k_1 = (1/np.abs(det)) * (-2 * mu_1[0] * cov[1,1] + 2 * mu_1[1] * cov[0,1] + 2 * mu_0[0] * cov[1,1] - 2 * mu_0[1] * cov[0,1])$

$k_2 = (1/np.abs(det)) * (-2 * mu_1[1] * cov[0,0] + 2 * mu_1[0] * cov[0,1] + 2 * mu_0[1] * cov[0,0] - 2 * mu_0[0] * cov[0,1])$

$k_3 = (1/np.abs(det)) * (mu_1[0] * mu_1[0] * cov[1,1] - 2 * mu_1[0] * mu_1[1] * cov[0,1] + mu_1[1] * mu_1[1] * cov[0,0] - mu$

Following is a plot of the data and the linear decision boundary. Blue points correspond to Alaska and redpoints correspond to Canada.

```
Red = Canada , Blue = Alaska
```



After this we relax the assumption of same covariance matrices and allow them to be different. The new covariance matrices are as follows (other parameters are unchanged).

```
Cov_0 is  [[ 0.38158978 -0.15486516]
 [-0.15486516  0.64773717]]


Cov_1 is  [[ 0.47747117  0.1099206 ]
 [ 0.1099206   0.41355441]]
```

The expression for decision boundary is now quadratic, let it be $l_1 * x_1^2 + l_2 * x_2^2 + l_3 * x_1 * x_2 + l_4 * x_1 + l_5 * x_2 + l_6 = 0$

The exact expressions are given below.

$$det_0 = np.linalg.det(cov_0)$$

$$det_1 = np.linalg.det(cov_1)$$

$$l_1 = (1/np.abs(det_1)) * cov_1[1,1] - (1/np.abs(det_0)) * cov_0[1,1]$$

$$l_2 = (1/np.abs(det_1)) * cov_1[0,0] - (1/np.abs(det_0)) * cov_0[0,0]$$

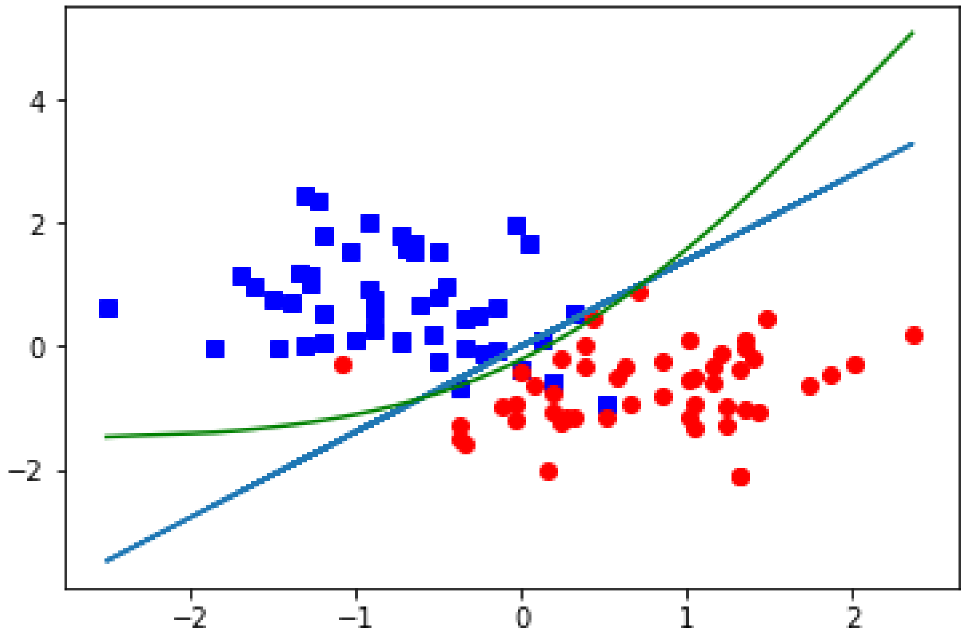
$$l_3 = -2 * (1/np.abs(det_1)) * cov_1[0,1] + 2 * (1/np.abs(det_0)) * cov_0[0,1]$$

$l_4 = (-2 * (1/np.abs(det_1)) * mu_1[0] * cov_1[1,1] + 2 * (1/np.abs(det_1)) * mu_1[1] * cov_1[0,1] + 2 * (1/np.abs(det_0)) * mu_0[0] * cov_0[1,1] - 2 * (1/np.abs(det_0)) * mu_0[1] * cov_0[0,1])$
$l_5 = (-2 * (1/np.abs(det_1)) * mu_1[1] * cov_1[0,0] + 2 * (1/np.abs(det_1)) * mu_1[0] * cov_1[0,1] + 2 * (1/np.abs(det_0)) * mu_0[1] * cov_0[0,0] - 2 * (1/np.abs(det_0)) * mu_0[0] * cov_0[0,1])$

$l_6 = ((1/np.abs(det_1)) * mu_1[0] * mu_1[0] * cov_1[1,1] - (1/np.abs(det_1)) * 2 * mu_1[0] * mu_1[1] * cov_1[0,1] + (1/np.abs(det_1)) * mu_1[1] * mu_1[1] * cov_1[0,0] - (1/np.abs(det_0)) * mu_0[0] * mu_0[0] * cov_0[1,1] + (1/np.abs(det_0)) * 2 * mu_0[0] * mu_0[1] * cov_0[0,1] - (1/np.abs(det_0)) * mu_1[1] * mu_1[1] * cov_0[0,0] + np.log(np.abs(det_1/det_0)))$

Finally we plot the data, linear decision boundary and quadratic decision boundary on the same plot.

```
Red = Canada , Blue = Alaska
```



It is clear from the graph that the quadratic boundary seems to be a better fit as it takes care of a few points which protrude out of the linear boundary. We can also observe that the different spreading out of the two classes has an effect on the quadratic boundary. The blue points seem more vertically spread out than the red points and hence the boundary bends to accomodate those points.