HW1实验报告

1 线性回归 Linear Regression

1.1使用正规方程：

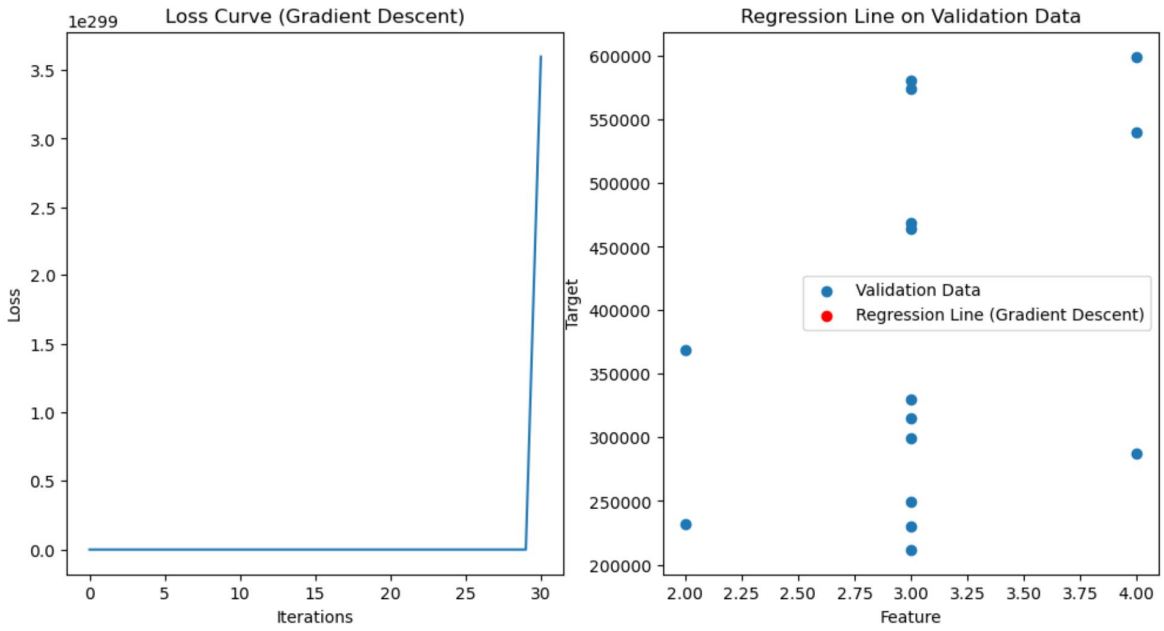
$$w = (X^T X)^{-1} X^T y$$

在没有进行归一化的时候：

使用正规方程计算斜率为[131.12253589 -2825.22573808]，截距为69599.05822335609

正规方程解的均方误差: 7611089615.0102

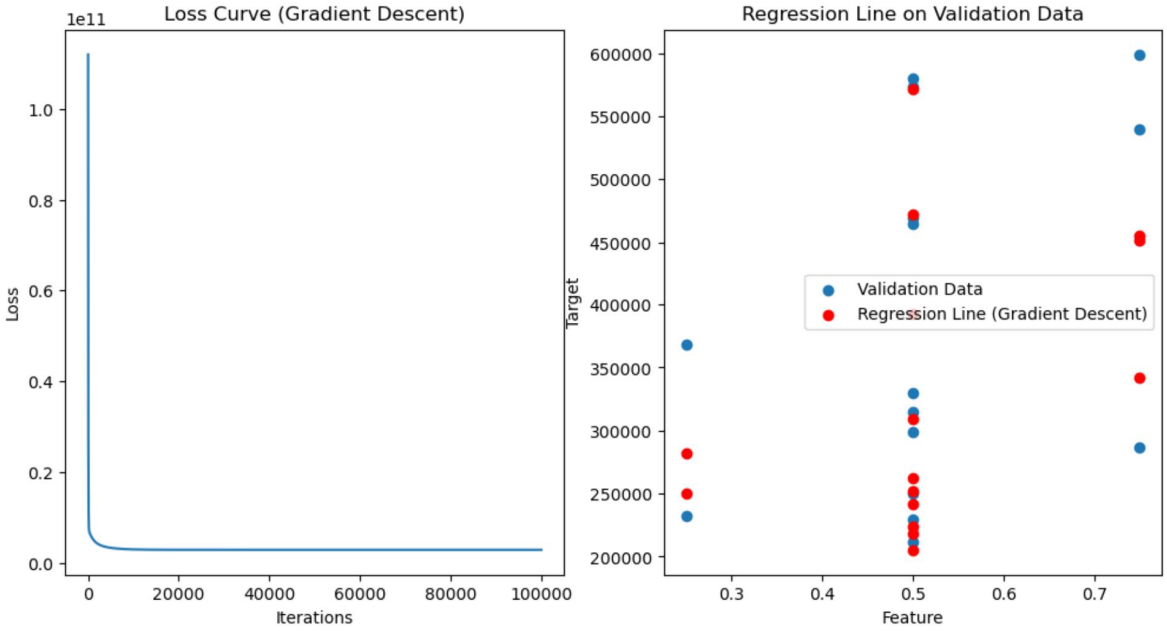
而使用梯度下降法无法很好的计算出结果，得到的Loss Curve和 验证集上的回归直线如下：



正规方程解的均方误差：7611089615.0102
梯度下降法的均方误差：nan

而进行了归一化后：

```
#归一化处理
def normalize_data(data_train, data_test):
    data_norm = data_train.copy()
    maximum = np.max(data_norm, axis=0, keepdims=True)
    minimun = np.min(data_norm, axis=0, keepdims=True)
    data_train = (data_train - minimun)/(maximum - minimun)
    data_test = (data_test-minimun)/(maximum-minimun)
    return data_train, data_test
```



正规方程解的均方误差：7611089615.0102
梯度下降法的均方误差：7611090182.5280

梯度下降法实现效果比较好，能够很好的得到回归方程。

原因：通过分析data1.txt里面的数据我们可以得知第一个特征向量是第二个特征向量的1000倍，如果不进行正则化处理那么第一个变量的影响对目标值非常的大，而第二个变量的影响几乎很小。因此我们需要进行正则化处理。

```
array([[2.104e+03, 3.000e+00],
       [1.600e+03, 3.000e+00],
       [2.400e+03, 3.000e+00],
       [1.416e+03, 2.000e+00],
       [3.000e+03, 4.000e+00],
       [1.985e+03, 4.000e+00],
       [1.534e+03, 3.000e+00],
       [1.427e+03, 3.000e+00],
       [1.380e+03, 3.000e+00],
       [1.494e+03, 3.000e+00],
       [1.940e+03, 4.000e+00],
       [2.000e+03, 3.000e+00],
       [1.890e+03, 3.000e+00],
       [4.478e+03, 5.000e+00],
       [1.268e+03, 3.000e+00],
       [2.300e+03, 4.000e+00],
       [1.320e+03, 2.000e+00],
       [1.236e+03, 3.000e+00],
       [2.609e+03, 4.000e+00],
       [3.031e+03, 4.000e+00],
       [1.767e+03, 3.000e+00],
       [1.888e+03, 2.000e+00],
       [1.604e+03, 3.000e+00],
       [1.962e+03, 4.000e+00],
       [3.890e+03, 3.000e+00],
```

1.2 比较正规方程和梯度下降法：

1. 从方法的实现过程比较：梯度下降法需要选择学习速率α，运行多次尝试不同的α直到找到合适的α。而使用正规方程只需要根据公式计算一次即可。
2. 从计算量上来看：梯度下降法需要多次迭代，取决于细节，计算可能较慢，而正规方程不需要进行迭代，使用最小二乘法计算J(θ)
3. 从时间复杂度上分析：正规方程中需要求逆矩阵，而逆矩阵的计算复杂度是O(n^3)，当矩阵维度n很大的时候，计算量很大。而梯度下降法根据其原理在维度n比较大的时候，通常都是比较有效的。

查阅网上资料也可得出相似的结论：

梯度下降	正规方程
$J(\Theta)=\frac{1}{2m}\sum_{i=1}^{m}\left(h_\Theta\left(x^{(i)}\right)-y^{(i)}\right)^2$	$\Theta=\left(X^TX\right)^{-1}X^Ty$
需要选择学习速率α，运行多次尝试不同的α直到找到合适的α	不需要选取需要选取学习速率α，只需要运行一次计算即可
需要多次迭代，取决于细节，计算可能较慢	不需要进行迭代，计算$J(\Theta)$来检查收敛性
当n很大（上百万）时表现很好，通常很有效	求解Θ时需要计算$\left(X^TX\right)^{-1}$这一项，$\left(X^TX\right)$该项为$n*n$的矩阵，对于大多数计算机而言，实现逆矩阵计算的代价是以矩阵维度的三次方$O\left(n^3\right)$增长，计算较慢
当n>10000时，选择梯度下降效率更高	当n<10000时（几百几千维向量），选择正规方程能更好的求解参数

2. 逻辑回归 Logitstic Regression

2.1梯度计算求导得到的公式：

- 梯度计算

$$g = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

- 梯度计算（L2正则化）

$$g_j = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + 2 * \lambda * \theta_j$$

求导过程：

$$l(\theta) = \log L(\theta) = \sum_{i=1}^{m}\left(y_i \log h_\theta(x_i) + (1-y_i)\log(1-h_\theta(x_i))\right)$$

$$\frac{\delta}{\delta\theta_j}J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \frac{1}{h_\theta(x_i)}\frac{\delta}{\delta\theta_j}h_\theta(x_i) - (1-y_i)\frac{1}{1-h_\theta(x_i)}\frac{\delta}{\delta\theta_j}h_\theta(x_i)\right)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \frac{1}{g(\theta^T x_i)} - (1-y_i)\frac{1}{1-g(\theta^T x_i)}\right)\frac{\delta}{\delta\theta_j}g(\theta^T x_i)$$

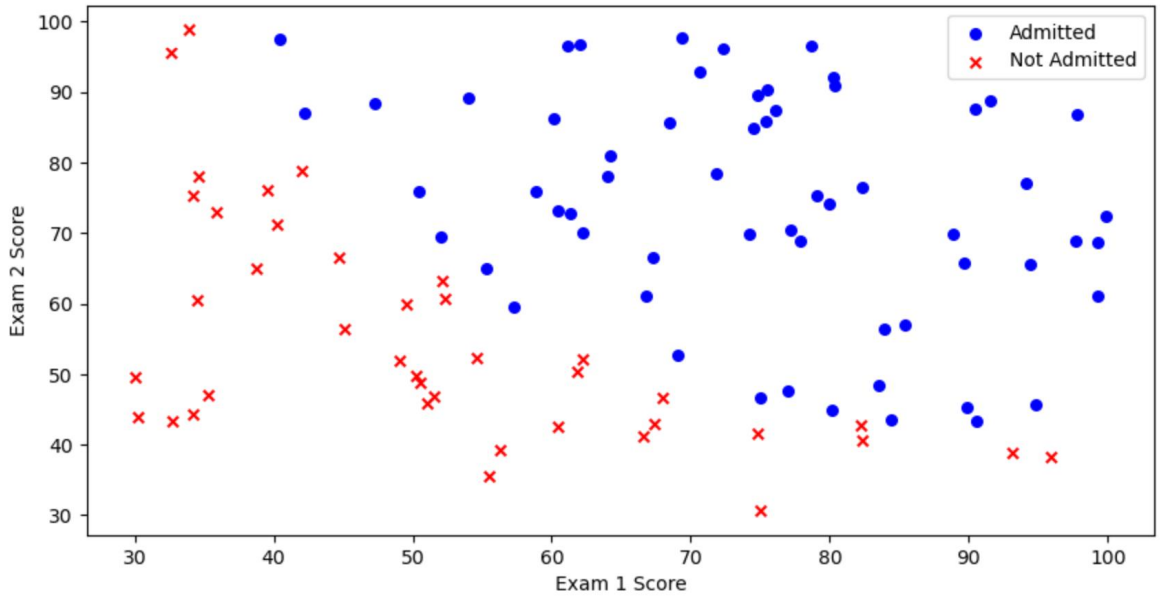
$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \frac{1}{g(\theta^T x_i)} - (1-y_i)\frac{1}{1-g(\theta^T x_i)}\right)g(\theta^T x_i)(1-g(\theta^T x_i))\frac{\delta}{\delta\theta_j}\theta^T x_i$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i(1-g(\theta^T x_i)) - (1-y_i)g(\theta^T x_i)\right)x_i^j$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i - g(\theta^T x_i)\right)x_i^j$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)x_i^j$$

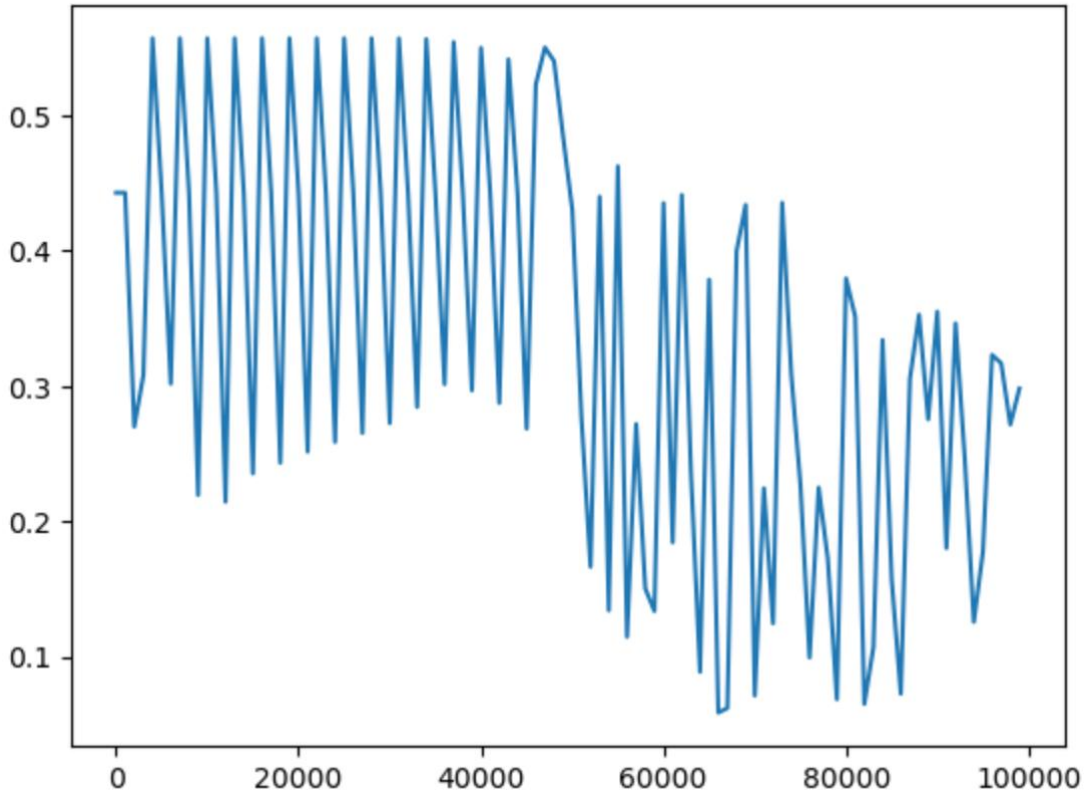
首先将数据读进来得到散点图:



2.2不进行正则化的情况

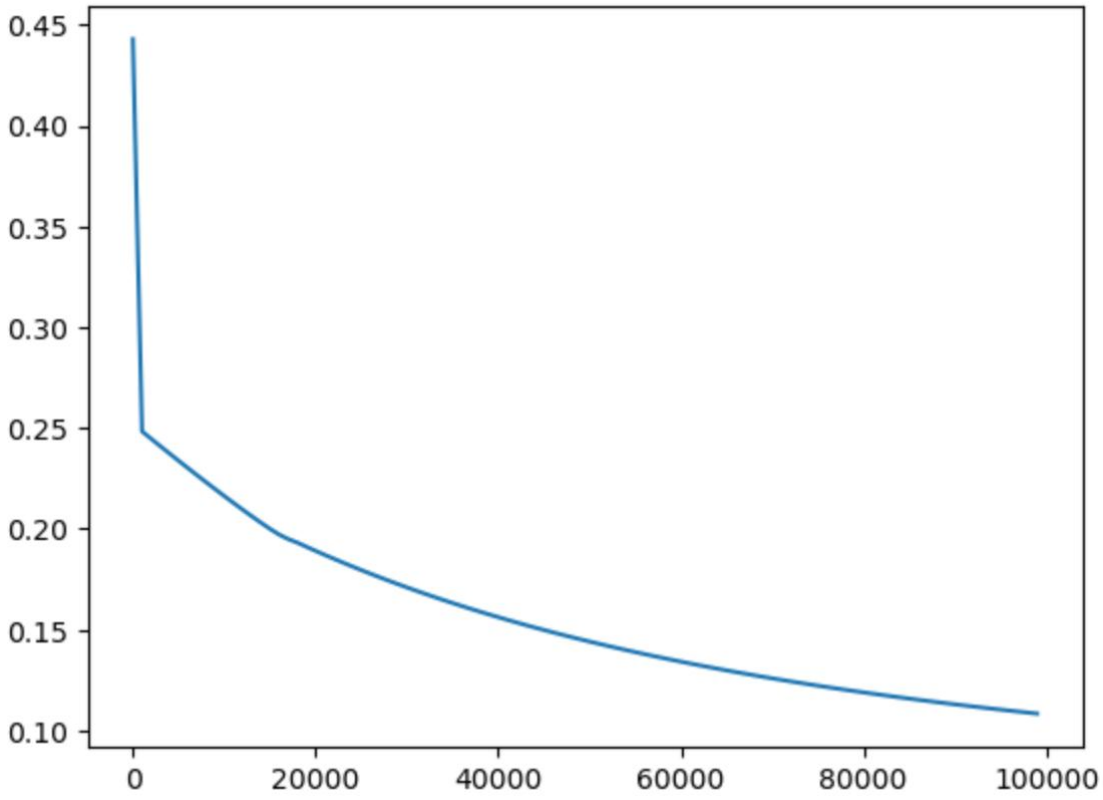
2.2.1不进行正则化我们举一般梯度下降法的损失的例子：

不进行归一化的时候这是我们把学习率设置为0.01，让它不那么小，同时设置迭代次数为10000次，这时候求迭代次数与损失函数之间的关系，关系如下：



能够看出损失函数非常的不稳定，这个时候尝试把学习率设置的小一点，看看情况会不会好一点，

设置学习率为0.001，然后迭代10000次，得到如下图像：



能够看到初始时很快，但稳定性不太好，需要设置很小的学习率。

2.3 进行正则化的情况

2.3.1 一般梯度下降法

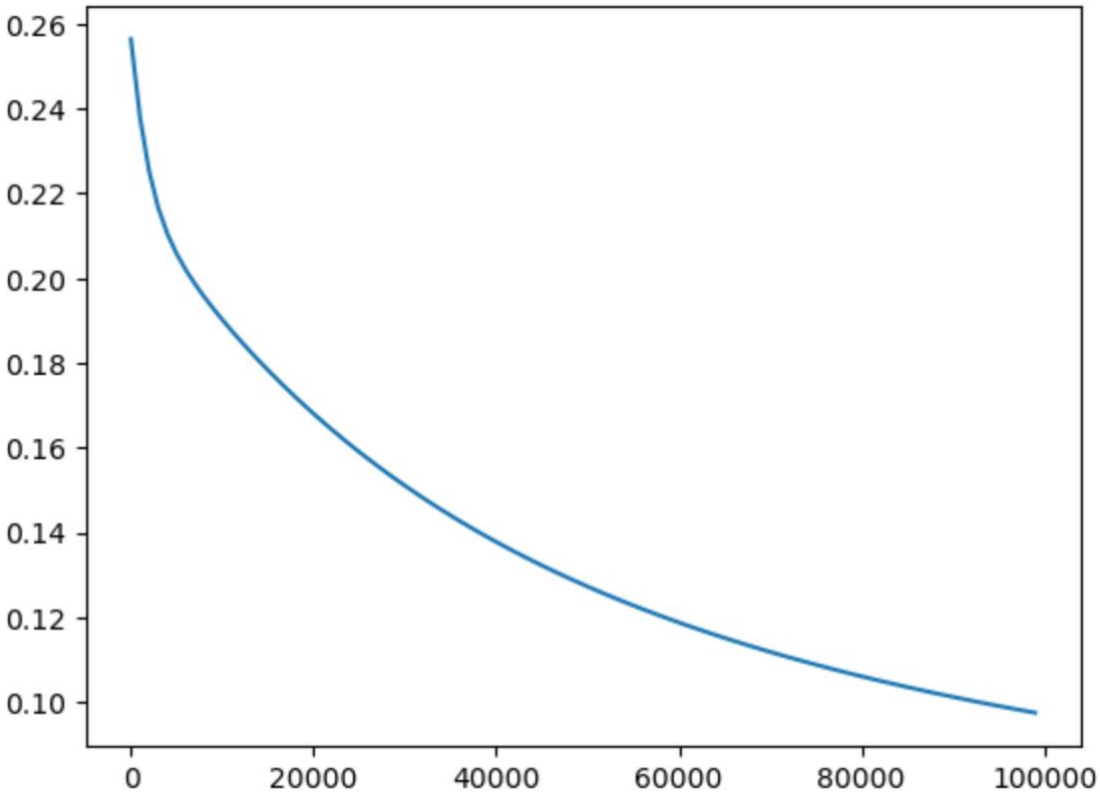
```
log_normol = []
test_log_normol = []
for i in range(iterations):
    h = np.matmul(X_train, w)
    y_pred = np.array(sigmoid(h))
    term = lr*np.mean((y_pred-y_train.reshape(-1,1))*X_train,
axis=0).reshape(-1,1)
    w -= term
    y_test_pred = np.matmul(X_val, w)
    loss = L2_loss(y_pred.reshape(-1,1),y_train.reshape(-1,1))
    test_loss = L2_loss(y_test_pred.reshape(-1,1),y_val.reshape(-1,1))
    if i % 1000 == 0:
        print('iter:{},loss:{}'.format(i,loss))
        log_normol.append([i,loss])
        test_log_normol.append([i,test_loss])
```

```
iter:0,loss:0.25640580782655226
iter:1000,loss:0.2379037318426969
iter:2000,loss:0.22521782725814882
iter:3000,loss:0.21658136864424438
iter:4000,loss:0.21045875188804047
iter:5000,loss:0.2058085437605662
iter:6000,loss:0.20200652154424475
iter:7000,loss:0.19870160359303943
iter:8000,loss:0.19570193898470442
iter:9000,loss:0.19290325659292748
iter:10000,loss:0.1902481612721992
iter:11000,loss:0.18770415402936713
iter:12000,loss:0.1852520440247857
iter:13000,loss:0.1828798829684409
iter:14000,loss:0.18057977367317807
iter:15000,loss:0.17834616477584317
iter:16000,loss:0.17617491724484416
iter:17000,loss:0.1740627771898593
iter:18000,loss:0.17200706735405913
iter:19000,loss:0.17000549983472796
iter:20000,loss:0.1680560583986317
iter:21000,loss:0.16615692227482262
iter:22000,loss:0.16430641558773335
iter:23000,loss:0.1625029731672017
iter:24000,loss:0.16074511710458794
iter:25000,loss:0.15903144050701362
iter:26000,loss:0.15736059614402673
iter:27000,loss:0.15573128845095544
iter:28000,loss:0.1541422678471368
iter:29000,loss:0.15259232665323116
iter:30000,loss:0.15108029611185633
iter:54000,loss:0.12368131412005352
iter:55000,loss:0.12282635469310633
```

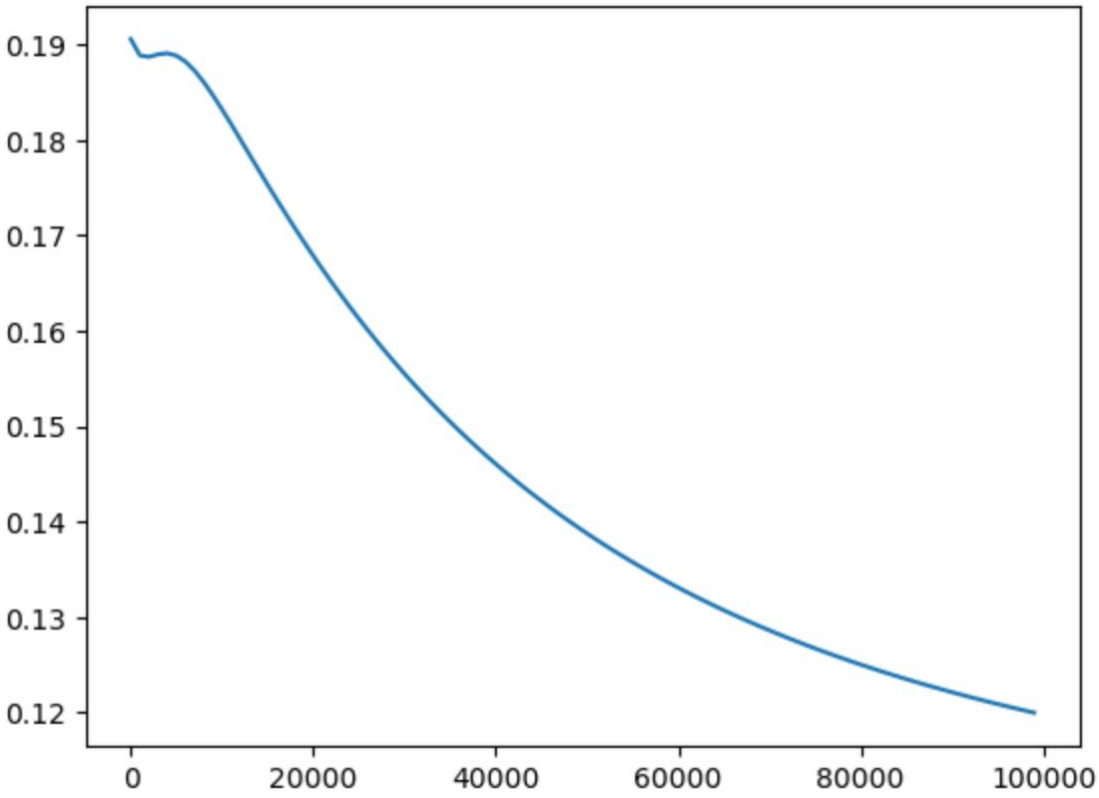
```
iter:56000,loss:0.12198897504015595
iter:57000,loss:0.12116868774898992
iter:58000,loss:0.12036502130273896
iter:59000,loss:0.11957751953862968
iter:60000,loss:0.11880574112222185
iter:61000,loss:0.11804925903714386
iter:62000,loss:0.1173076600902735
iter:63000,loss:0.11658054443225499
iter:64000,loss:0.11586752509319641
iter:65000,loss:0.11516822753334517
iter:66000,loss:0.114482289208506
iter:67000,loss:0.11380935914993671
iter:68000,loss:0.11314909755843011
iter:69000,loss:0.11250117541227303
iter:70000,loss:0.11186527408875134
iter:71000,loss:0.11124108499886591
iter:72000,loss:0.11062830923490771
iter:73000,loss:0.11002665723053807
iter:74000,loss:0.1094358484330139
iter:75000,loss:0.10885561098719598
iter:85000,loss:0.10358049379345495
iter:86000,loss:0.10310080251130271
iter:87000,loss:0.10262899824383892
iter:88000,loss:0.10216489816929411
iter:89000,loss:0.10170832478201157
iter:90000,loss:0.10125910571249627
iter:96000,loss:0.09870918408624833
iter:97000,loss:0.09830701551189557
iter:98000,loss:0.09791098769616663
iter:99000,loss:0.09752096741870579
```

对于一般梯度下降设置学习率为0.001，迭代次数为10000次，得到如下图像：

对于测试集一般梯度下降图像：

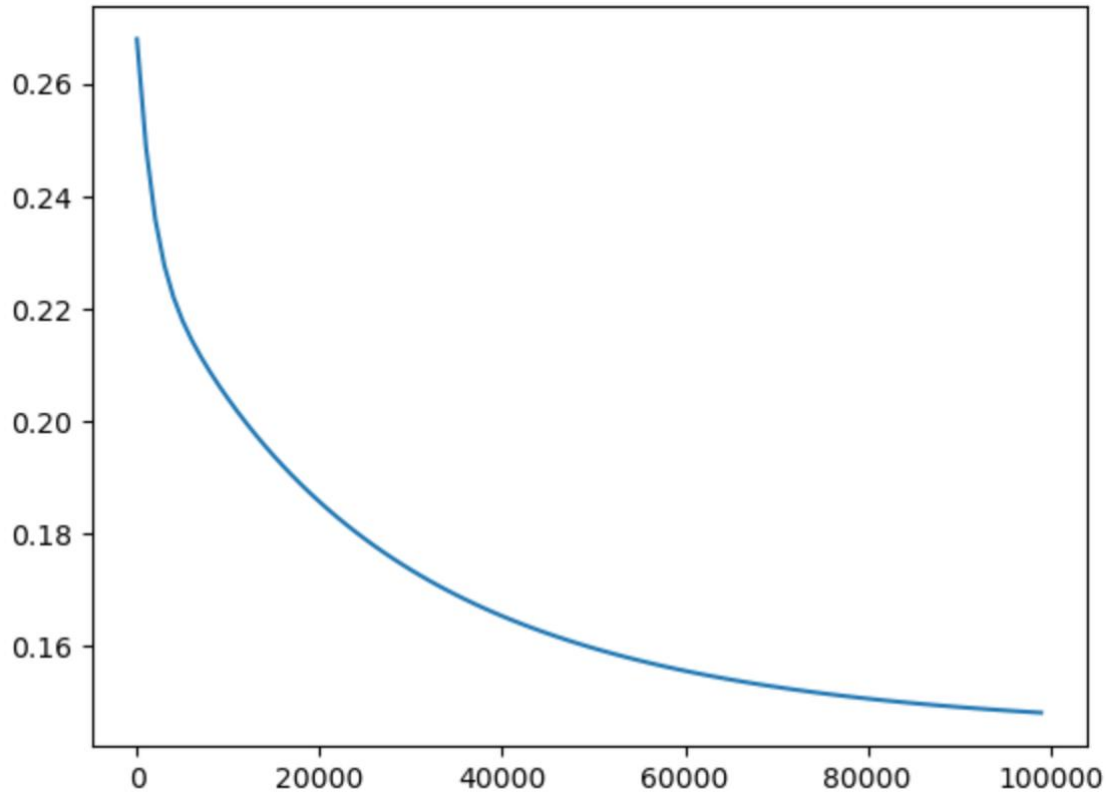


对于验证集L2正则化梯度下降图像：

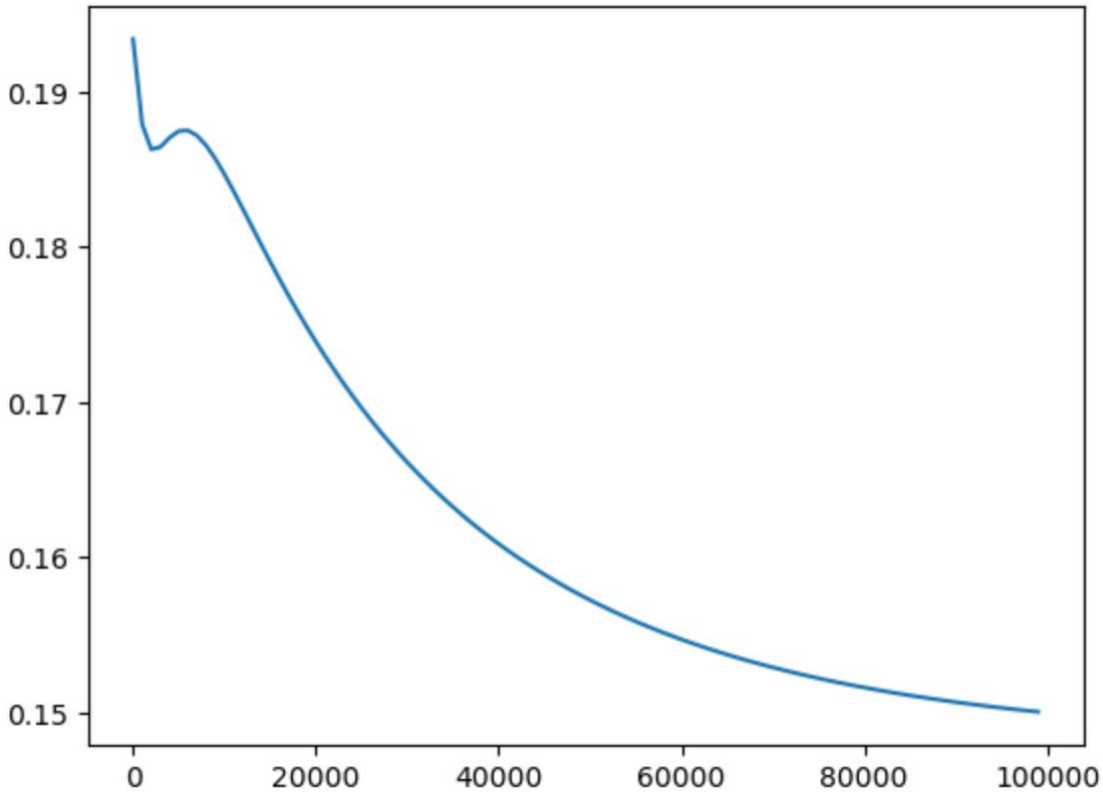


2.3.2 使用L2正则化梯度下降图像，设置lamda为0.01：

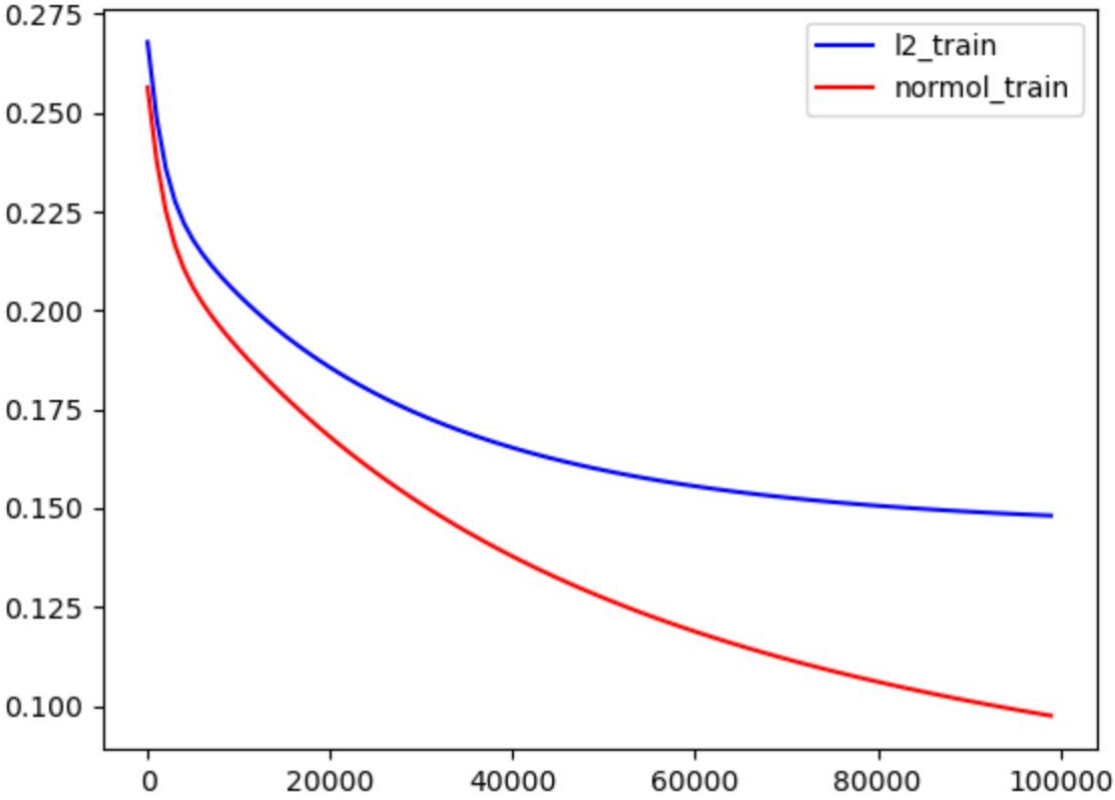
对于测试集L2正则化梯度下降图像：



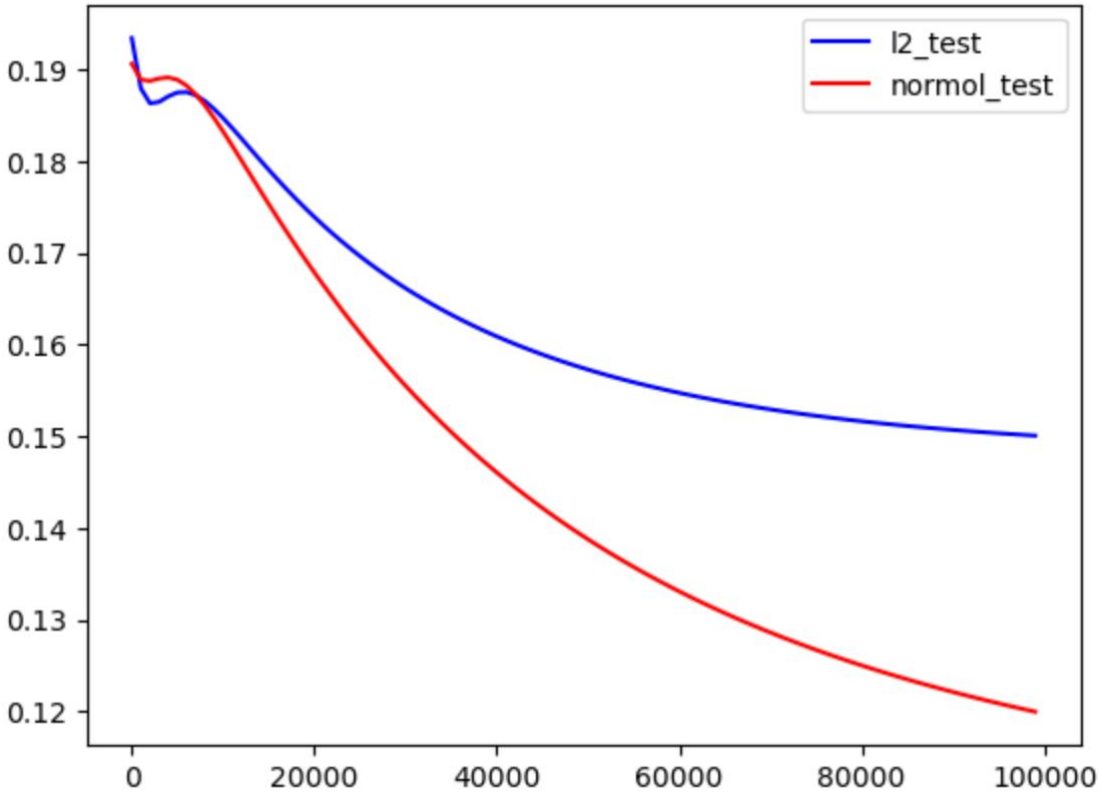
验证集损失函数对比如下：



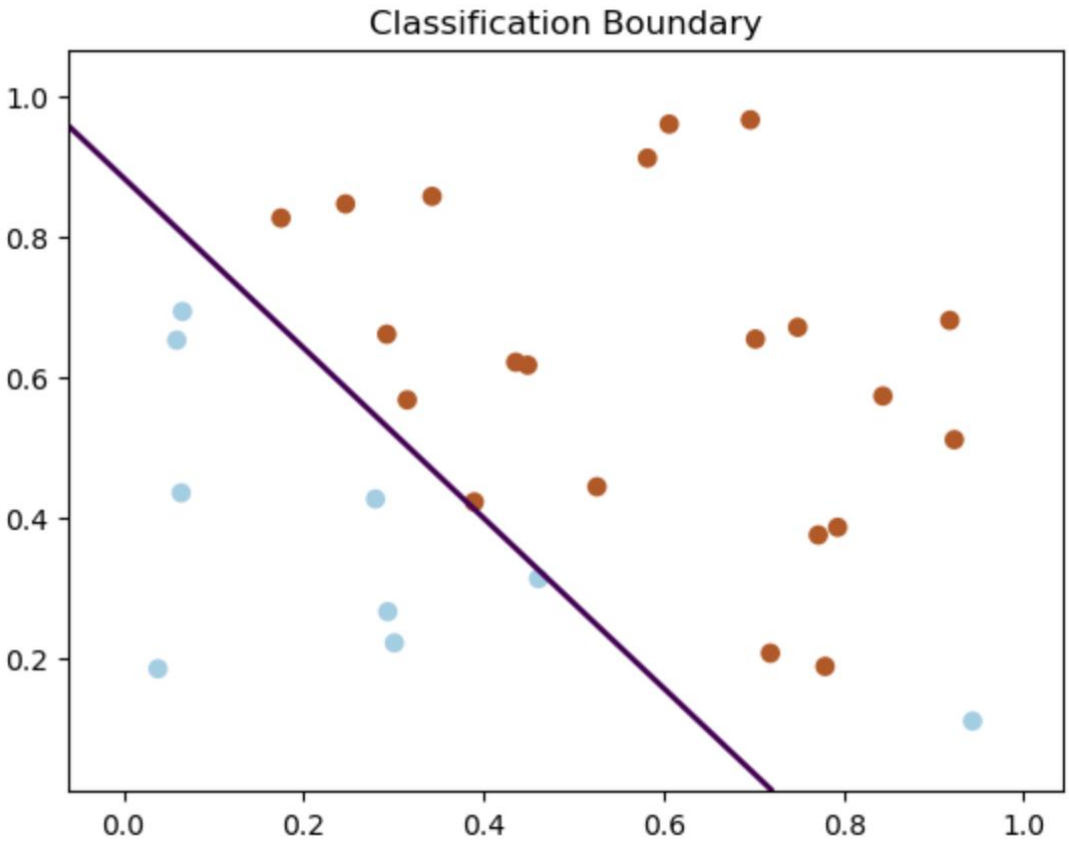
测试集L2正则化梯度下降损失函数和梯度下降损失函数对比如下：



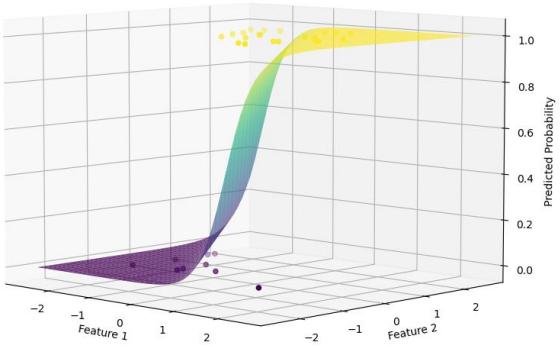
验证集损失函数对比如下：



最后绘制出分类直线的2D,3D图像：



Classification Boundary and Data Points



综上所述，L2正则化权重更新如下（2α 也是常数项，可以合并到一起用整体 α 替代）：

$$\theta_j^{n+1} = \theta_j^n(1 - \eta * \alpha) - \eta * \sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

其中 α 就是正则化参数，η 表示学习率。从上式可以看到，与未添加L2正则化的迭代公式相比，每一次迭代，θ_j 都要先乘以一个小于1的因子（即 $(1 - \eta * \alpha)$），从而使得 θ_j 加速减小，因此总的来看，θ 相比不加L2正则项的线性回归可以获得更小的值。从而，实现了防止过拟合的效果，增加模型的鲁棒性~

有的书本上，公式写法可能**不同**：其中 λ 表示正则化参数。

$$\theta_j^{n+1} = \theta_j^n(1 - \eta * \lambda) - \eta * \sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$