

第三周学习笔记（算法实现）

2022-04-28

第一章 机器学习练习之线性回归

本次需要用到的包为：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

1.1 单变量线性回归

在本次的练习题中，将使用一个单变量线性回归的算法，以预测食品卡车的利润。假设自己是一家连锁餐厅的 *CEO*，并且正在考虑在不同的城市开设一家新店。该连锁店已经在各个城市拥有卡车，并且我们拥有来自城市的利润和人口数据。

将城市人口作为特征参数，用矩阵 X 表示，将获得的利润值作为预测值，用 y 表示。

文件 *ex1data1.txt* 包含我们的线性回归问题的数据集。第一列是一个城市的人口，第二列是该城市一辆食品卡车的利润。其中如果 *profit* 为负值，则表示营业状态为损失。

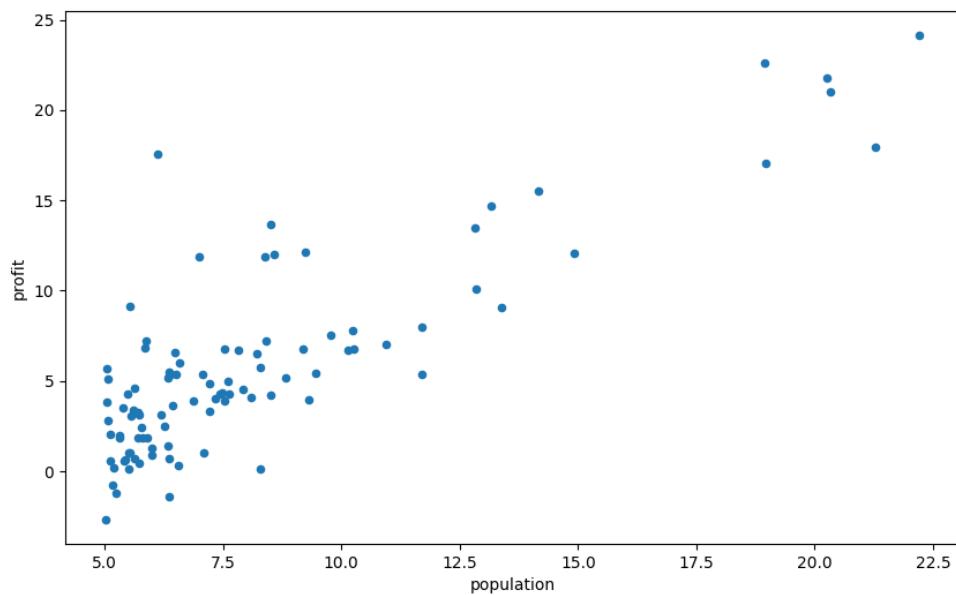
1.1.1 数据的可视化

在解决问题之前，通过可视化数据来理解数据通常很有用。对于此数据集，我们可以使用散点图来可视化数据，因为它只有两个要绘制的属性 (*profit* 和 *population*)。数据是以 *txt* 文件形式给出的，因此可以用 *pandas* 的 *read_csv()* 读取，但由于这个文件的数据没有表头，所以我们在读取时要加上表头。

```
path = ('ex1data1.txt')
data = pd.read_csv(path, header=None, names=['population', 'profit'])
```

```
data.plot(kind='scatter', x='population', y='profit', figsize=(10, 6))
plt.ylabel('profit')
plt.xlabel('population')
```

得到的散点图如下图所示：



```
cols = data.shape[1]
X = data.iloc[:, 0:cols - 1] # X是所有行，去掉最后一列
y = data.iloc[:, cols - 1:cols] # y是所有行，最后一列
```

因为我们需要使得到的代价函数是 *numpy* 矩阵，所以我们接下来转换 X 和 Y ，然后才能使用它们。我们还需要初始化 θ 。

```
X = np.matrix(X.values)
y = np.matrix(y.values)
theta = np.matrix(np.array([0, 0]))
```

接下来，可以检查一下 θ 是否被初始化，以及 X , y 和 θ 的维度，对后面的计算会有帮助。

```
print('----- 检查维数 -----')
print(X.shape, theta.shape, y.shape)
print('----- 初始化 theta -----')
print(theta)
```

输出的结果为：

```
----- 检查维数 -----
(97, 2) (1, 2) (97, 1)
----- 初始化 theta -----
[[0 0]]
```

上面的准备工作结束后，我们可以在 *Python* 中这样定义代价函数 *Cost*:

```
def Cost(X, y, theta):
    inner = np.power(((X * theta.T) - y), 2)
    return np.sum(inner) / (2 * len(X))
```

这样我们便可以计算代价函数最开始的数值：

```
print(Cost(X, y, theta))
```

完成计算后，我们便可以看到成本的最初数值，即：

```
----- 计算代价函数的值为 -----
32.072733877455676
```

我们拟合的最终目的是求得 $J(\theta)$ 最小值时的函数，此处可以使用两种方法：梯度下降法和正规方程法。

1.1.3 梯度下降

1. 算法

梯度下降算法：用来求解代价函数的最小值。梯度下降背后的思想是：开始时我们随机选择一个参数的组合 $(\theta_0, \theta_1, \dots, \theta_n)$ ，计算代价函数，然后我们寻找下一个能让代价函数值下降最多的参数组合。我们持续这么做直到找到一个局部最小值。

为什么说是局部最小值呢？因为我们并没有尝试完所有的参数组合，所以不能确定我们得到的局部最小值是否便是全局最小值。选择不同的初始参数组合，可能会找到不同的局部最小值。

在本章节，我们用梯度下降算法来求解代价函数 J 的最小值，其中 $J(\theta)$ 有两个参数 θ_0 和 θ_1 。

使用批量梯度下降算法。在批量梯度下降中，每次迭代都执行更新，规则如下：

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

同时更新所有 j 下的 θ_j 。

成本 $J(\theta)$ 由向量 θ 参数化，而不是 X 和 y 。也就是说，我们通过改变向量 θ 的值来最小化 $J(\theta)$ 的值，而不是通过改变 X 或 y 。

此处 α 为学习速率，需选择合适的值。若 α 过大，则可能会跳过最小值点，若 α 过小，则迭代需要较长的时间。

随着梯度下降的每一步，你的参数 θ_j 更接近将实现最低成本 $J(\theta)$ 的最优值。

2. 定义函数

我们首先初始化一些附加变量，即学习速率 α 和要执行的迭代次数 $iters$ 。

在这里，我们先选择学习速率为 0.01，执行的次数为 10000 次。

```
alpha = 0.01
iters = 10000
```

批量梯度下降算法用 *Python* 代码实现如下：

```
def gradientDescent(X, y, theta, alpha, iters):
    temp = np.matrix(np.zeros(theta.shape))
    parameters = int(theta.ravel().shape[1])
    cost = np.zeros(iters)
    for i in range(iters):
        error = (X * theta.T) - y
```

```

for j in range(parameters):
    term = np.multiply(error, X[:,j])
    temp[0,j] = theta[0,j] - ((alpha / len(X)) * np.sum(term))
theta = temp
cost[i] = Cost(X, y, theta)
return theta, cost

```

接下来运行梯度下降算法，便可以求出适合于训练集的参数 θ 。

```

final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
print(final_theta)

```

得到的结果如下：

```
[[ -3.89578082  1.19303364]]
```

其中，得到的 $cost$ 为二维数组的形式，里面是一系列的代价函数值。

最后，我们可以使用我们得到的拟合参数来计算训练模型的代价函数（误差）。

```

print('-----拟合的参数计算训练模型的代价函数的最小值-----')
print(Cost(X, y, final_theta))

```

计算的结果如下：

```

-----拟合的参数计算训练模型的代价函数的值-----
4.476971375975179

```

此时的代价函数值为 4.48，小于最初的 32，可见梯度下降是工作的，在下一节中我们会使用图像来检验模型是否工作。

接着上文的计算结果，我们可以输出得到的模型函数：

```

f = final_theta[0, 0] + (final_theta[0, 1] * x)
a = float(final_theta[0, 0])
b = float(final_theta[0, 1])
print('-----输出回归方程-----')
print('回归方程为:f=% .8f+%.8f*x' % (a, b))

```

```

-----输出回归方程-----
回归方程为:f=-3.89578082+1.19303364*x

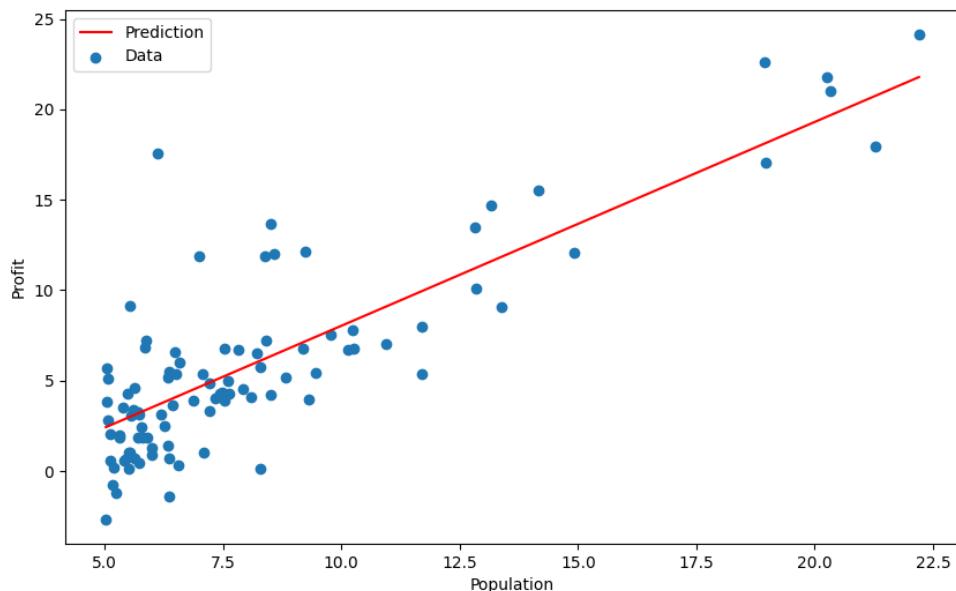
```

3. 拟合效果

现在来绘制线性模型以及数据，直观地看出它的拟合。我们可以使用梯度下降得到的最终参数来绘制线性图，代码实现为：

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x, f, 'r', label='Prediction')
ax.scatter(data.population, data.profit, label='Data')
ax.legend(loc=2)
ax.set_xlabel('population')
ax.set_ylabel('profit')
```

得到的图片为：



从图像上来看，拟合效果较好。

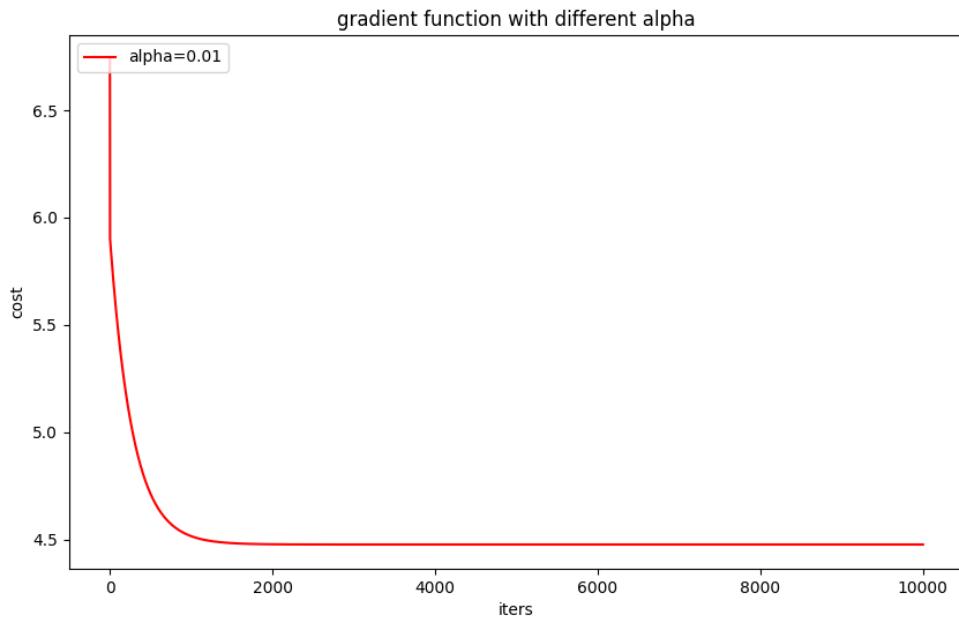
4. 检验算法

验证梯度下降是否正常工作的一个常用方法是查看 $J(\theta)$ 的值并检查它是否随着每一步而减小。假设你已经正确地实现了梯度下降和 $Cost$ ，你的 $J(\theta)$ 值永远不会增加，并且应该在算法结束时收敛到一个稳定的值。

由于梯度方程式函数也在每个训练迭代中输出一个代价的向量，所以我们也就可以绘制。请注意，代价总是降低，同时这也是凸优化问题的一个例子。

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.arange(iters), cost, 'r', label='alpha=0.01')
ax.set_xlabel('iters')
ax.set_ylabel('cost')
plt.show()
```

我们可以看到代价函数的图像如下，可以看出我们选择学习率为 0.01 时，在迭代 10000 次之前，代价函数已经收敛。



5. 预测结果

计算的最终值将用于预测，这也是我们建立模型的初衷。比如预测人口为 35,000 和 70,000 的地区的利润。

```
x_predict1 = np.matrix([[1, 3.5]])
x_predict2 = np.matrix([[1, 7]])
predict1 = np.dot(x_predict1, final_theta.T)
predict2 = np.dot(x_predict2, final_theta.T)
print('-----输出预测值1-----')
print(predict1)
print('-----输出预测值2-----')
print(predict2)
```

得到的预测结果为：

```
-----输出预测值1-----
[[0.27983691]]
-----输出预测值2-----
[[4.45545465]]
```

到这一步，我们的初步学习任务已经完成。

6. 学习率

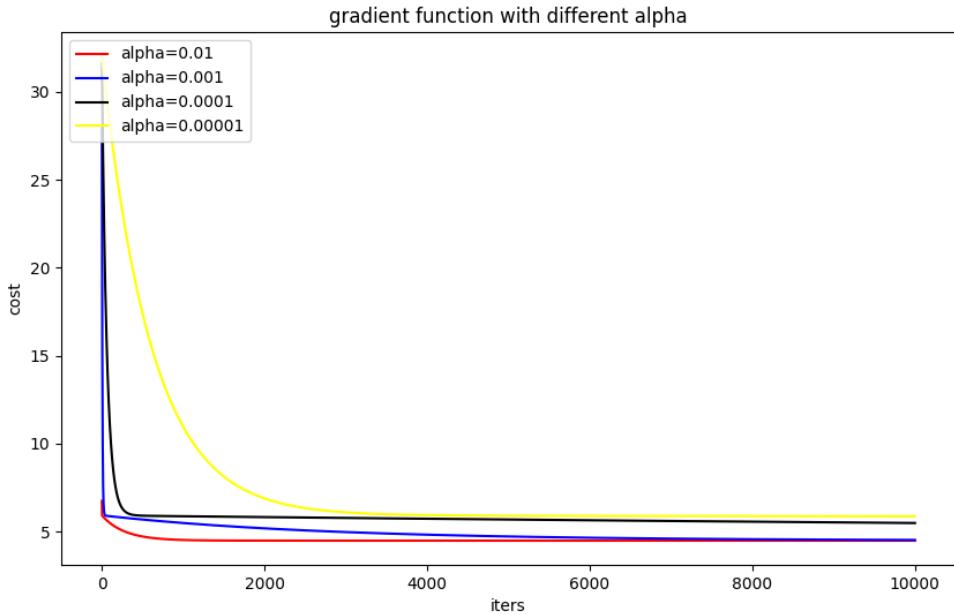
通过前面的学习我们知道， α 为学习速率，需选择合适的值，若 α 过大，则可能会跳过最小值点，若 α 过小，则迭代需要较长的时间。

但如何确定自己选择的学习率是否合适呢？可以使用不同的学习率绘图来查看收敛情况。

因此我接下来选择依次改变学习率，使得 α 分别取值为 0.01, 0.001, 0.0001, 0.00001 来观察学习率对代价函数的影响。

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.arange(iters), cost, 'r', label='alpha=0.01')
alpha = 0.001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'b', label='alpha=0.001')
alpha = 0.0001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'black', label='alpha=0.0001')
alpha = 0.00001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'yellow', label='alpha=0.00001')
ax.legend(loc=2)
ax.set_xlabel('iters')
ax.set_ylabel('cost')
ax.set_title('gradient function with different alpha')
plt.show()
```

结果如下图所示：



可以看见学习速率 α 对于收敛速率的影响。并且之前的模型选择 $\alpha=0.01$ 比较合适。

因为我们选择的迭代次数很大，所以不同学习率在迭代 10000 次后均达到了收敛效果。如果降低迭代次数，比如将迭代次数由 10000 次降为 1000 次，学习率较低的情况下，应该是完成不了收敛的，这里就不再演示。

1.1.4 正规方程

正规方程求解 θ 是按照矩阵方程进行计算。特别是对于某些线性回归问题，它会使我们更好地求 θ 的最优值。

正规方程，又叫最小二乘估计。

正规方程的计算公式为：

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

需要注意的是：

- (1) 使用正规方程求解 θ 时，不用特征缩放；
- (2) 正规方程原理：令 $\frac{\partial}{\partial \theta_j} J(\theta) = 0$ ，解出 $(\theta_0, \theta_1, \dots, \theta_n)$ ，即 θ 的最优解。

正规方程的 Python 代码实现为：

```
theta = np.matrix(np.array([0, 0]))
theta_n = (X.T * X).I * X.T * y
print('----利用正规方程求系数----')
```

```
print(theta_n)
```

求得的系数为：

```
-----利用正规方程求系数-----  
[[-3.89578088]  
 [ 1.19303364]]
```

通过与梯度下降求得的系数 ($[-3.89578082 \ 1.19303364]$) 对比，发现两者之间还是有些许差距的，但在单变量线性回归中几乎可以忽略不计。

在下面的多变量线性回归中，正规方程和梯度下降求出的系数结果是有差距的。

1.2 多变量线性回归

在这一部分中，将学习使用多个变量实现线性回归来预测房价。

假设房子的拥有者要卖掉自己的房子，拥有者想知道一个好的市场价格是多少。需要的方法是首先收集最近售出房屋的信息并制作房价售价模型。

1.2.1 数据预处理—归一化

文件 *ex1data2.txt* 是整理房间价格的训练集。第一列是房子的大小（平方英尺），第二列是卧室数量，第三列是房子的价格。

最开始拿到数据集，通过查看这些数值，不难发现房屋面积大约是卧室数量的 1000 倍，而且不同特征之间的单位也不相同，房子的大小的单位是平方英尺，卧室的数量是个，房子的价格是美元。

当特征相差几个数量级时，我们首先需要的操作是特征缩放，它可以使梯度下降收敛得更快。

特征缩放的步骤如下：

1. 从数据集中减去每个特征的平均值。
2. 减去平均值后，再将新的特征值除以它们各自的标准差得到特征缩放后的数据。

对特征进行归一化处理时，重要的是存储用于归一化的值，即用于计算的平均值和标准差。在从模型中学习到参数之后，我们经常想要预测我们以前没见过的房子的价格。给定一个新的 X 值（房屋面积和卧室数量），我们必须首先使用我们之前从训练集计算的均值和标准差对 X 进行归一化处理。

这里的操作用 *Pandas* 很容易就可以实现。

用 *Python* 代码实现为：

```
path = 'ex1data2.txt'
data = pd.read_csv(path, header=None, names=['Size', 'Bedrooms', 'Price'])
print('-----展示前十行数据-----')
print(data.head(10))
new_data = (data - data.mean()) / data.std()
print('-----特征归一化后的数据-----')
print(new_data.head(10))
```

特征归一后的数据结果展示为：

```
-----展示前十行数据-----
   Size  Bedrooms  Price
0    2104         3  399900
1    1600         3  329900
2    2400         3  369000
3    1416         2  232000
4    3000         4  539900
5    1985         4  299900
6    1534         3  314900
7    1427         3  198999
8    1380         3  212000
9    1494         3  242500
-----特征归一化后的数据-----
      Size  Bedrooms     Price
0  0.130010 -0.223675  0.475747
1 -0.504190 -0.223675 -0.084074
2  0.502476 -0.223675  0.228626
3 -0.735723 -1.537767 -0.867025
4  1.257476  1.090417  1.595389
5 -0.019732  1.090417 -0.323998
6 -0.587240 -0.223675 -0.204036
7 -0.721881 -0.223675 -1.130948
8 -0.781023 -0.223675 -1.026973
9 -0.637573 -0.223675 -0.783051
```

通过结果展示，我们不难发现，所有的特征数据都集中在区间 $[-1, 1]$ 之间。对于我们接下来的学习会很有帮助。

1.2.2 梯度下降

之前，我们在单变量回归问题上实现了梯度下降。现在唯一的区别是矩阵 X 中多了一个特征。假设函数和批量梯度下降更新规则保持不变。

在多变量情况下，代价函数也可以写成以下向量化形式：

$$J(\theta) = \frac{1}{2m} (X\theta - y)^T (X\theta - y)$$

$$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{pmatrix} \quad (1.1)$$

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \quad (1.2)$$

数据的初始化为：

```
new_data.insert(0, 'Ones', 1)
cols = new_data.shape[1]
X = new_data.iloc[:, 0:cols - 1]
y = new_data.iloc[:, cols - 1:cols]
X = np.matrix(X.values)
y = np.matrix(y.values)
theta = np.matrix(np.array([0, 0, 0]))
```

现在我们重复单变量线性回归的步骤，并对代价函数进行定义：

```
def Cost(X, y, theta):
    inner = np.power(((X * theta.T) - y), 2)
    return np.sum(inner) / (2 * len(X))
```

梯度下降的算法实现为：

```
def gradientDescent(X, y, theta, alpha, iters):
    temp = np.matrix(np.zeros(theta.shape))
    parameters = int(theta.ravel().shape[1])
    cost = np.zeros(iters)
    for i in range(iters):
        error = (X * theta.T) - y
```

```

for j in range(parameters):
    term = np.multiply(error, X[:, j])
    temp[0, j] = theta[0, j] - ((alpha / len(X)) * np.sum(term))
theta = temp
cost[i] = computeCost(X, y, theta)
return theta, cost

```

我们输出回归方程的结果:

```

final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
print('-----代价函数值-----')
print(Cost(X, y, final_theta))
print('-----输出系数值-----')
print(final_theta)

```

输出的结果为:

```

-----代价函数值-----
0.13070365481036997
-----输出系数值-----
[[ -1.11000452e-16   8.78450989e-01  -4.68640133e-02]]

```

从系数矩阵中，我们不难看出，截距项的数值非常的小，在实际情况中，我们可以忽略截距项。

我们利用和单变量回归相似的方法，做出拟合图像，观察模型的拟合效果。

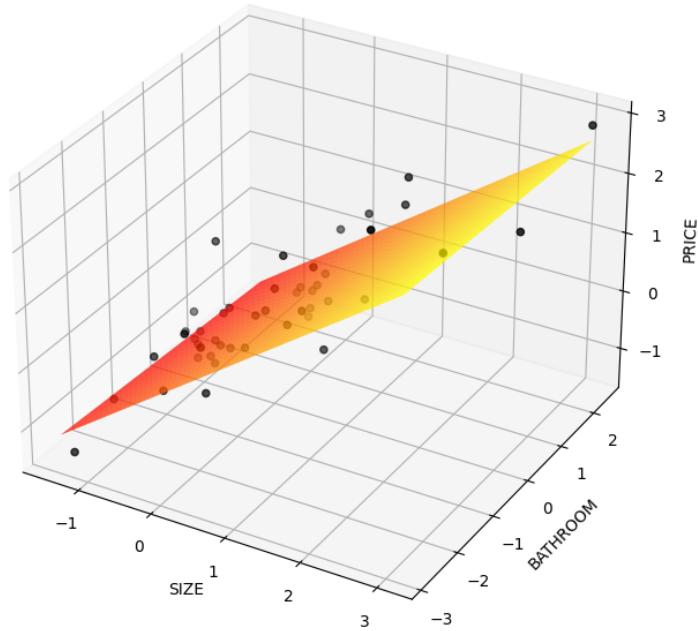
Python 代码的实现方式为:

```

alpha=0.001
iters=10000
x1 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
x2 = np.linspace(X[:, 2].min(), X[:, 2].max(), 100)
x1, x2 = np.meshgrid(x1, x2)
fig = plt.figure(figsize=(10,6))
ax = Axes3D(fig)
ax.plot_surface(x1, x2, f, rstride=1, cstride=1, cmap=cm.autumn, label='prediction')
ax.scatter(X[:100, 1], X[:100, 2], y[:100, 0], c='black')
ax.set_zlabel('PRICE')
ax.set_ylabel('BATHROOM')
ax.set_xlabel('SIZE')

```

结果如下图所示:

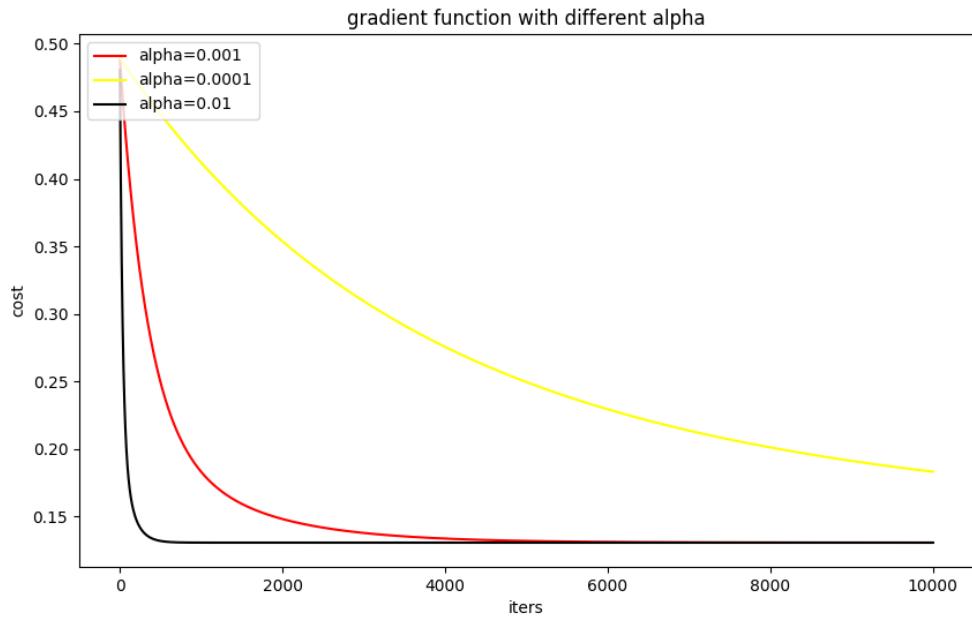


同上面的单变量线性回归的步骤，接下来讨论不同的学习率对于代价函数的影响。

Python 代码实现结果如下：

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.arange(iters), cost, 'r', label='alpha=0.001')
ax.set_xlabel('iters')
ax.set_ylabel('cost')
alpha = 0.0001
theta = np.matrix(np.array([0, 0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'yellow', label='alpha=0.0001')
alpha = 0.01
theta = np.matrix(np.array([0, 0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'black', label='alpha=0.01')
ax.legend(loc=2)
ax.set_title('gradient function with different alpha')
plt.show()
```

图像结果显示如下：



从图像中可以很明显的看出，当学习率过低时，此时迭代 10000 次，代价函数还未收敛。

1.2.3 正规方程

```
theta_n = (X.T * X).I * X.T * y
print('-----利用正规方程求系数-----')
print(theta_n)
```

结果为：

```
-----利用正规方程求系数-----
[[-8.67361738e-17]
 [ 8.84765988e-01]
 [-5.31788197e-02]]
```

与上面梯度下降算法求得的系数 ($[-1.11000452e-16 \ 8.78450989e-01 \ -4.68640133e-02]$)，这次的结果相比单变量线性回归就有明显的不同。

第二章 机器学习练习之逻辑回归

本次需要用到的包:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.optimize as opt
from sklearn.metrics import classification_report
```

在这部分的练习中，我们将尝试建立一个逻辑回归模型来预测一个学生是否能进入大学。这是一个分类问题。

假设这所大学的行政管理人员想根据两门考试的结果来决定每个申请的学生是否被录取。他有以前申请人的历史数据，可以将其用作逻辑回归训练集。对于每一个训练样本，管理人员有申请人两次测评的分数以及录取的结果。

为了完成这个预测任务，我们准备构建一个可以基于两次测试评分来评估录取可能性的分类模型。

2.1 逻辑回归

2.1.1 数据的可视化

首先，我们先简单看一下数据集中的数据的构成:

```
path = 'ex2data1.txt'
data = pd.read_csv(path, header=None, names=['Exam 1', 'Exam 2', 'Admitted'])
print(data.head(10))
```

展示的数据为:

	Exam 1	Exam 2	Admitted
0	34.623660	78.024693	0

1	30.286711	43.894998	0
2	35.847409	72.902198	0
3	60.182599	86.308552	1
4	79.032736	75.344376	1
5	45.083277	56.316372	0
6	61.106665	96.511426	1
7	75.024746	46.554014	1
8	76.098787	87.420570	1
9	84.432820	43.533393	1

让我们创建两个分数的散点图，并使用不同颜色来进行区分，如果样本是正的，则表示为被接纳，即为 *AdmittedYes*; 样本是负的，则表示未被接纳，即为 *AdmittedNo*。

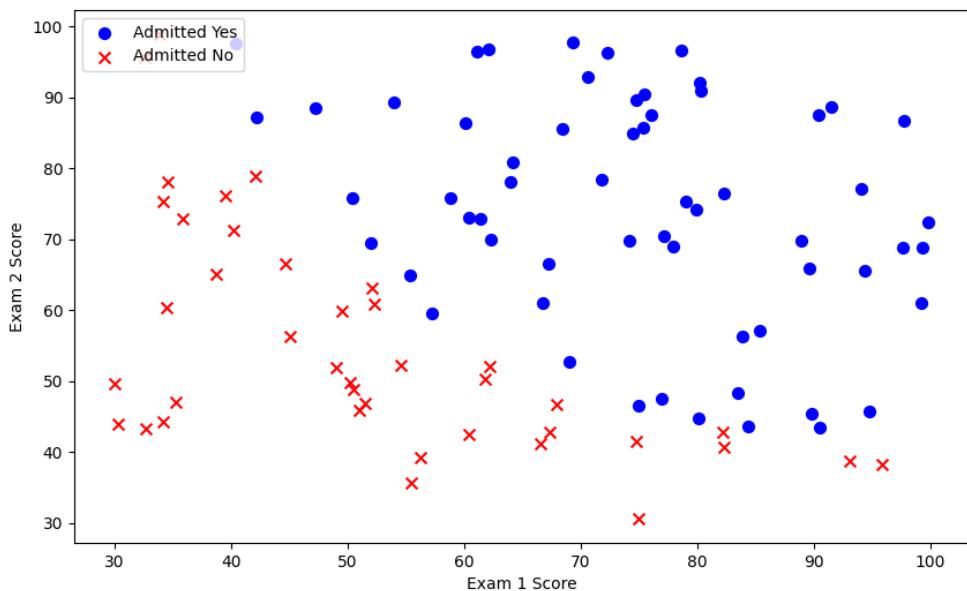
```
positive = data[data['Admitted'].isin([1])]
negative = data[data['Admitted'].isin([0])]

fig, ax = plt.subplots(figsize=(10, 6))

ax.scatter(positive['Exam 1'], positive['Exam 2'], s=50, c='blue', marker='o', label='Admitted Yes')
ax.scatter(negative['Exam 1'], negative['Exam 2'], s=50, c='red', marker='x', label='Admitted No')

ax.legend(loc=2)
ax.set_xlabel('Exam 1 Score')
ax.set_ylabel('Exam 2 Score')
plt.show()
```

图片的展示结果为：



从图片结果可以看出，在两类数据之间，有一个清晰的分界线，在机器学习中，我们把这个分界线叫做决策边界，即 *DecisionBoundary*。现在我们需要实现逻辑回归，那样就可以训练一个模型来预测结果。

2.1.2 定义

在开始实际成本函数之前，回忆一下逻辑回归假设定义为：

$$h_{\theta}(x) = g(\theta^T x)$$

其中函数 g 是 *sigmoid* 函数。

sigmoid 函数定义为：

$$g(z) = \frac{1}{1 + e^{(-z)}}$$

合起来，我们得到逻辑回归模型的假设函数：

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T X}}$$

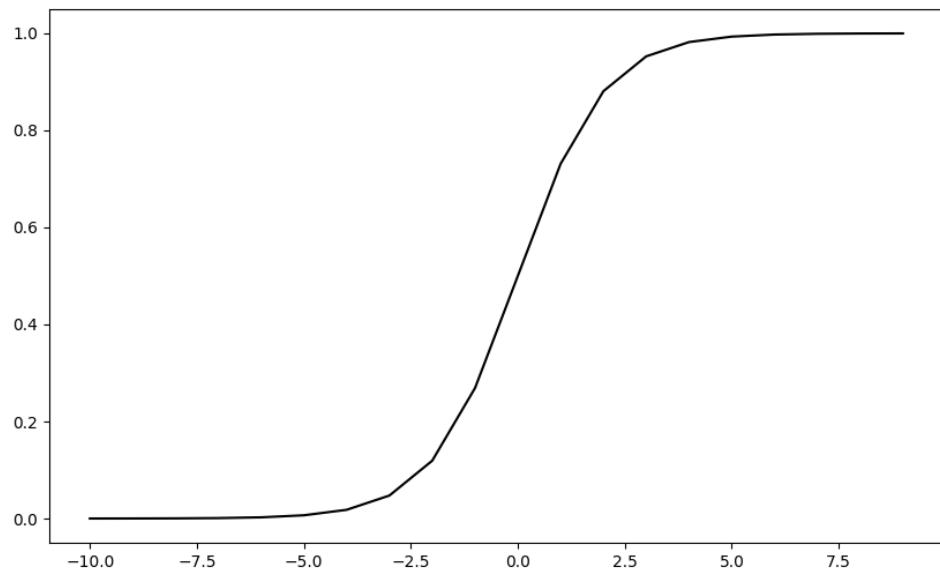
这个定义用 *Python* 实现如下：

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

让我们做一个快速的检查，来确保它可以工作。

```
random_numbers = np.arange(-10, 10, step=1)
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(random_numbers, sigmoid(random_numbers), 'black')
plt.show()
```

最后的结果为：



通过上面的效果图，我们看出很明显的符合 S 曲线，说明我们的函数定义正确。

2.1.3 代价函数

回想一下，逻辑回归中的成本函数是：

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

并且代价的梯度是一个与第 j 个元素（对于 $j = 0; 1; \dots; n$ ）长度相同的向量，定义如下：

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

请注意，虽然这个梯度看起来与线性回归梯度相同，但公式实际上是不同的，因为线性回归和逻辑回归对 $h_\theta(x)$ 的定义不同。

```
def cost(theta, X, y):
    theta = np.matrix(theta)
```

```
X = np.matrix(X)
y = np.matrix(y)

first = np.multiply(-y, np.log(sigmoid(X * theta.T)))
second = np.multiply((1 - y), np.log(1 - sigmoid(X * theta.T)))
return np.sum(first - second) / (len(X))
```

现在需要做一些设置，和之前在线性回归的练习中的设置很相似。

```
data.insert(0, 'Ones', 1)
cols = data.shape[1]
X = data.iloc[:, 0:cols - 1]
y = data.iloc[:, cols - 1:cols]
X = np.matrix(X.values)
y = np.matrix(y.values)
theta = np.matrix(np.array([0, 0, 0]))
```

我们可以检查 θ 和 X, y 的维度，结果如下所示：

```
[[0 0 0]]
(100, 3) (100, 1) (1, 3)
```

最后，我们计算得到的代价函数的值：

```
print(cost(theta, X, y))
```

结果如下所示：

```
-----输出的代价函数的值为-----
0.6931471805599453
```

这也是得到的最初的代价函数的值。

2.1.4 梯度下降

批量梯度下降转化为向量化计算的算法如下：

$$\frac{1}{m} X^T (Sigmoid(X\theta) - y)$$

Python 代码实现为：

```
def gradient(theta, X, y):
    theta = np.matrix(theta)
    X = np.matrix(X)
    y = np.matrix(y)
```

```

parameters = int(theta.ravel().shape[1])
grad = np.zeros(parameters)
error = sigmoid(X * theta.T) - y
for i in range(parameters):
    term = np.multiply(error, X[:,i])
    grad[i] = np.sum(term) / len(X)
return grad

```

我们看看用我们的数据和初始参数为 0 的梯度下降法的结果：

```
print(gradient(theta, X, y))
```

```
[ -0.1      -12.00921659 -11.26284221]
```

值得注意的是，在这一步中，实际上没有在这个定义的梯度函数中执行梯度下降，我们仅仅在计算一个梯度步长。由于我们使用 *Python*，通过查阅资料，我发现可以使用 *SciPy* 的 *optimize* 命名空间来做同样的事情。

现在可以用 *Newton-CG* 实现寻找最优参数。只需传入 *cost* 函数，已经所求的变量 *theta*，和梯度，*fun = cost* 表示传入我们的 *cost* 函数，*x0 = theta*，表示传入初始点即 *theta* 的初值。

```

res = opt.minimize(fun=cost, x0=theta, jac=gradient, args=(X, y), method='Newton-CG')
print(res)

```

结果展示如下：

```

fun: 0.2035657548120845
jac: array([ 0.00015425, -0.00354585, -0.00287975])
message: 'Optimization terminated successfully.'
nfev: 63
nhev: 0
nit: 25
njev: 160
status: 0
success: True
x: array([-24.49260996, 0.20088557, 0.19605783])

```

此时的 $x = \text{array}([-24.49260996, 0.20088557, 0.19605783])$ ，即为 *theta* 的最优解。

2.1.5 逻辑回归的预测

接下来，需要编写一个函数，用之前所学的参数 θ 来为数据集 X 输出预测。

首先，先取出上面求的 x ，即取出 θ 的最优解。

```
final_theta = res.x
print(final_theta)
```

取出的 θ 最优解为：

```
[-24.49260996  0.20088557  0.19605783]
```

接下来，我们来建立我们需要的预测函数如下：

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$$

当 $h_{\theta}(x)$ 大于等于 0.5 时，预测 $y = 1$ ；当 $h_{\theta}(x)$ 小于 0.5 时，预测 $y = 0$ 。

其中， $h_{\theta}(x)$ 的作用是，对于给定的输入变量，根据选择的参数计算输出变量等于 1 的可能性，即 $h_{\theta}(x) = P(y = 1|x; \theta)$ 。

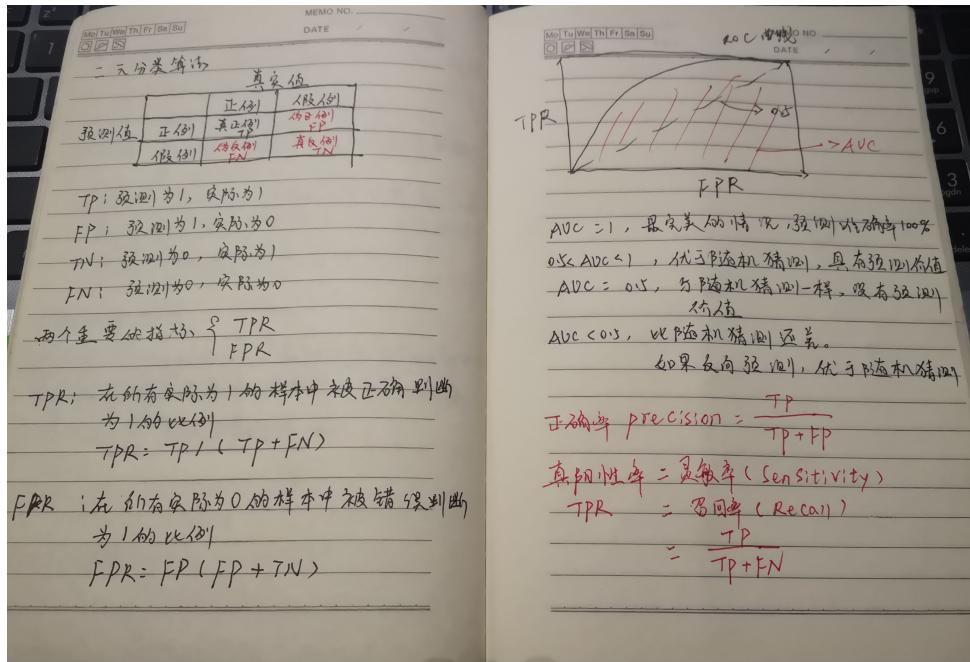
用 Python 代码实现如下：

```
def predict(theta, X):
    probability = sigmoid(X * theta.T)
    return [1 if x >= 0.5 else 0 for x in probability]
```

模型评估这一块，我们引入 AUC 指标。

AUC 模型评价指标只能用于二分类模型的评价，对于二分类模型，还有损失函数 *logloss*，正确率 *accuracy*，准确率 *precision*，但相比之下 AUC 和 *logloss* 要比 *accuracy* 和 *precision* 用的多，原因是因为很多的机器学习模型计算结果都是概率的形式，那么对于概率而言，我们就需要去设定一个阈值来判定分类，那么这个阈值的设定就会对我们的正确率和准确率造成一定成都的影响。

二元分类算法，通过 $AUC(Area under the Curve of ROC(receiver operating characteristic))$ 进行评估。



• 正确率(Precision):

$$Precision = \frac{TP}{TP + FP}$$

• 真阳性率(True Positive Rate, TPR), 敏感度(Sensitivity), 召回率(Recall):

$$Sensitivity = Recall = TPR = \frac{TP}{TP + FN}$$

力
SK16

		预测结果		MEMO NO.
		正例	假例	DATE
真实 结果	正例	真正例 TP	假反例 FN	
	假例	假正例 FP	真反例 TN	

精确率：预测结果为正的样本中样本中真实为正的比例

召回率：真实为正例的样本中预测结果为正例的比例
(查得全，对正样本的区分能力)

还有其他的评估标准， F_1 -score 反映了模型的稳健性

$$F_1 = \frac{2TP}{2TP + FN + FP} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

可以使用 sklearn 中的方法来检验模型的准确度：

```
final_theta=np.matrix(final_theta)
y_predict = predict(final_theta, X)
from sklearn.metrics import classification_report
print(classification_report(y, y_predict))
```

预测的结果为：

	precision	recall	f1-score	support
0	0.87	0.85	0.86	40
1	0.90	0.92	0.91	60
accuracy			0.89	100
macro avg	0.89	0.88	0.88	100
weighted avg	0.89	0.89	0.89	100

我们的逻辑回归分类器预测正确，如果一个学生被录取或没有录取，达到 89
其中 $f1 - score$ 即 F 。

2.1.6 决策边界

为了更直观地展示出所拟合的参数是否被录取的效果，所以需要根据参数画出决策边界（线性函数），公式定义如下：

$$X \times \theta = 0$$

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$$

因此我们得出斜率矩阵如下：

```
coef = -(res.x / res.x[2])
print(coef)
```

得出决策边界对应的线性函数的系数为：

```
[124.92543689 -1.02462405 -1.]
```

因此接下来是函数实现，并且根据原始数据画散点图，根据函数画出决策边界。

```
x = np.arange(130, step=0.1)
y = coef[0] + coef[1] * x

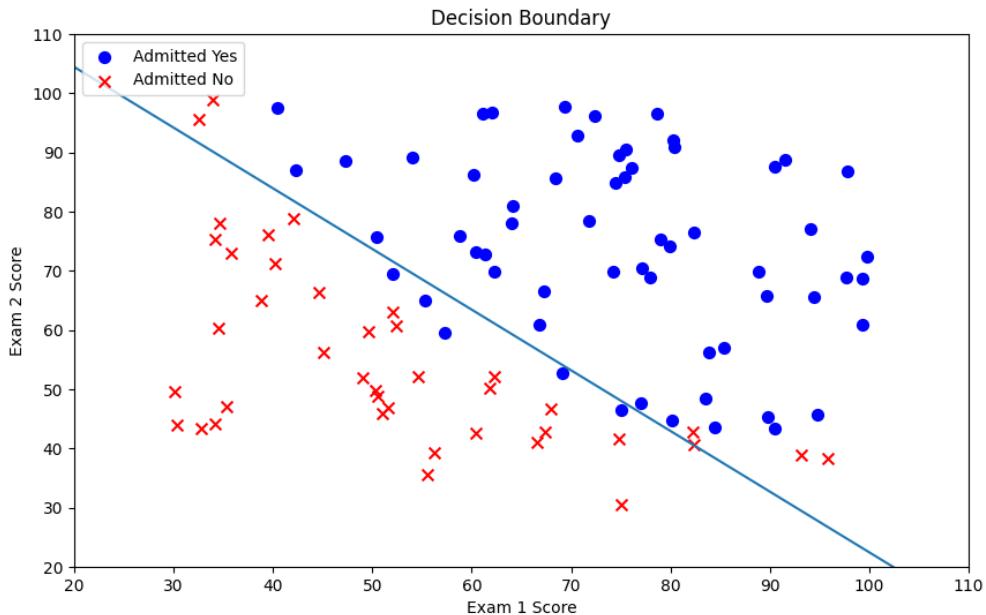
positive = data[data['Admitted'].isin([1])]
negative = data[data['Admitted'].isin([0])]

fig, ax = plt.subplots(figsize=(10, 6))

ax.scatter(positive['Exam 1'], positive['Exam 2'], s=50, c='blue', marker='o', label='Admitted Yes')
ax.scatter(negative['Exam 1'], negative['Exam 2'], s=50, c='red', marker='x', label='Admitted No')

ax.plot(x,y)
ax.set_xlim(20, 110)
ax.set_ylim(20, 110)
ax.legend(loc=2)
ax.set_xlabel('Exam 1 Score')
ax.set_ylabel('Exam 2 Score')
ax.set_title('Decision Boundary')
plt.show()
```

结果如下图所示：



2.2 正则化逻辑回归

在训练的第二部分，我们将要通过加入正则项提升逻辑回归算法。简而言之，正则化是成本函数中的一个术语，它使算法更倾向于“更简单”的模型（在这种情况下，模型将更小的系数）。这个理论助于减少过拟合，提高模型的泛化能力。

设想有一些芯片在两次测试中的测试结果。对于这两次测试，工厂的生产主管想决定是否芯片要被接受或抛弃。为了帮助生产主管做出艰难的决定，根据拥有的过去芯片的测试数据集，从其中可以构建一个逻辑回归模型。

2.2.1 数据的可视化

与上面的思路一样，从可视化做起。

```
path = 'ex2data2.txt'

data = pd.read_csv(path, header=None, names=['Test 1', 'Test 2', 'Accept'])

positive = data[data['Accept'].isin([1])]
negative = data[data['Accept'].isin([0])]

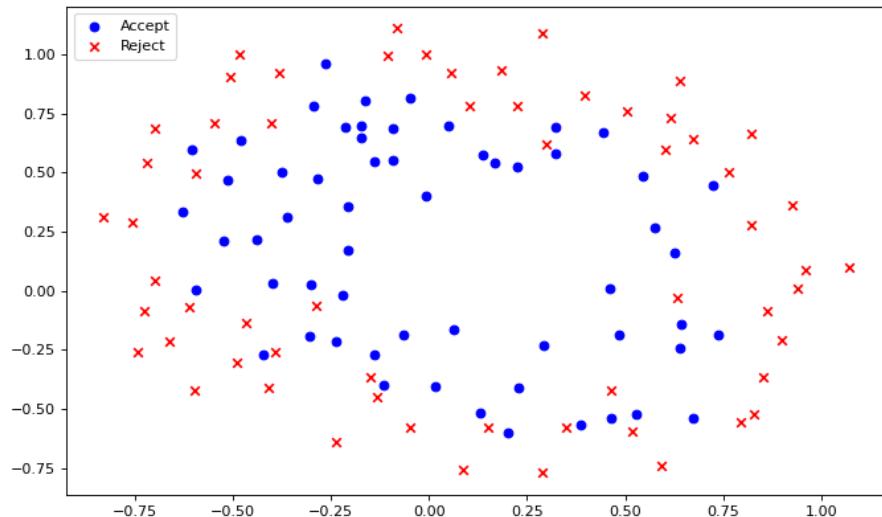
fig, ax = plt.subplots(figsize=(10, 6))

ax.scatter(positive['Test 1'], positive['Test 2'], s=50, c='b', marker='o',
           label='Accept')

ax.scatter(negative['Test 1'], negative['Test 2'], s=50, c='r', marker='*',
           label='Reject')

ax.legend(loc=2)
```

```
ax.set_xlabel('Test 1 Score')
ax.set_ylabel('Test 2 Score')
plt.show()
```



这个数据看起来复杂得多。特别地，其中没有线性决策界限，来良好的分开两类数据。

因此直接用 *logistic* 回归在这个数据集上并不能表现良好，因为它只能用来寻找一个线性的决策边界。

提供一个新的方法：用像逻辑回归这样的线性技术来构造从原始特征的多项式中得到的特征。通过原特征来增加特征量总量（线性组合多次方）

2.2.2 特征映射

查阅资料，获得的理论来源是：已有的这些特征映射到所有的 x_1 和 x_2 的多项式项上，（升高幂数使线性直线变成曲线）直到第六次幂，即 x_1 和 x_2 的次方和从 1 升为 6， $h(x)$ 从 3 项升为 28 项，同时 θ 变为 28 个。

$$mapFeature(x) = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1x_2 \\ x_2^2 \\ x_1^3 \\ \dots \\ x_1x_2^5 \\ x_2^6 \end{pmatrix} \quad (2.1)$$

代码实现如下：

```
def feature_mapping(x, y, power, as_ndarray=False):
    data = {"f'{}{}'.format(i - p, p): np.power(x, i - p) * np.power(y, p)
            )
            for i in np.arange(power + 1)
            for p in np.arange(i + 1)
            }

    if as_ndarray:
        return np.array(pd.DataFrame(data))
    else:
        return pd.DataFrame(data)

x1 = np.array(data.Test1)
x2 = np.array(data.Test2)

new_data = feature_mapping(x1, x2, power=6)
```

	f'00	f'10	f'01	f'20	...	f'33	f'24	f'15	f'06
0	1.0	0.051267	0.69956	0.002628	...	0.000046	0.000629	0.008589	0.
					117206				
1	1.0	-0.092742	0.68494	0.008601	...	-0.000256	0.001893	-0.013981	0.
					103256				
2	1.0	-0.213710	0.69225	0.045672	...	-0.003238	0.010488	-0.033973	0.
					110047				
3	1.0	-0.375000	0.50219	0.140625	...	-0.006679	0.008944	-0.011978	0.
					016040				

```

4    1.0 -0.513250  0.46564  0.263426 ... -0.013650  0.012384 -0.011235  0.
      010193

[5 rows x 28 columns]

```

经过映射，我们将有两个特征的向量转化成了一个 28 维的向量。

在这个高维特征向量上训练的 *logistic* 回归分类器将会有个更复杂的决策边界，当我们在二维图中绘制时，会出现非线性。

虽然特征映射允许我们构建一个更有表现力的分类器，但它也更容易过拟合。在接下来的练习中，我们将实现正则化的 *logistic* 回归来拟合数据，并且可以看到正则化如何帮助解决过拟合的问题。

2.2.3 正则化代价函数

正则化代价函数的公式为：

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

λ 的大小控制新产生的特征变量对代价函数的影响，注意不惩罚第一项即 θ_0 。

因为 θ_0 控制的是截距，截距不是越小越好，而应该符合实际要求。

模型用到的变量为：

```

theta = np.zeros(new_data.shape[1])
X = feature_mapping(x1, x2, power=6, as_ndarray=True)
y = np.array(data.iloc[:, -1])

```

$X.shape$ 为 (118, 28), $y.shape$ 为 (118,)。

定义正则化下的代价函数为：

```

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

```

```

def regularized_cost(theta, X, y, l=1):
    thetaReg = theta[1:]
    first = (-y * np.log(sigmoid(X @ theta))) - (1 - y) * np.log(1 -
                                                                sigmoid(X @ theta))
    reg = (thetaReg @ thetaReg) * 1 / (2 * len(X))
    return np.mean(first) + reg

```

我们利用函数计算代价函数的值：

```
print(regularized_cost(theta, X, y, l=1))
```

最后得到的结果为：

```
0.6931471805599454
```

2.2.4 正则化梯度下降

我们要使用梯度下降法令上面的代价函数最小化，与正常的梯度下降相比多了后面 λ 那项，所以需要重新定义梯度下降函数。算法的实现过程如下：

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m [h_\theta(x^{(i)}) - y^{(i)}] x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m [h_\theta(x^{(i)}) - y^{(i)}] x_j^{(i)} + \frac{\lambda}{m} \theta_j$$

注意：不需要对 θ_0 进行正则化，因为 θ_0 是截距项通常不考虑， j 从 1 到 n ，不惩罚第一项。

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

综上所述，利用 Python 代码实现如下：

```
def regularized_gradient(theta, X, y, l=1):
    thetaReg = theta[1:]
    first = (X.T @ (sigmoid(X @ theta) - y)) / len(X)
    reg = np.concatenate([np.array([0]), (l / len(X)) * thetaReg])
    return first + reg

print(regularized_gradient(theta, X, y))
```

结果如下所示：

```
[8.47457627e-03 1.87880932e-02 7.77711864e-05 5.03446395e-02
 1.15013308e-02 3.76648474e-02 1.83559872e-02 7.32393391e-03
 8.19244468e-03 2.34764889e-02 3.93486234e-02 2.23923907e-03
 1.28600503e-02 3.09593720e-03 3.93028171e-02 1.99707467e-02
 4.32983232e-03 3.38643902e-03 5.83822078e-03 4.47629067e-03
 3.10079849e-02 3.10312442e-02 1.09740238e-03 6.31570797e-03
 4.08503006e-04 7.26504316e-03 1.37646175e-03 3.87936363e-02]
```

2.2.5 正则化模型评估

操作方法与逻辑回归模型的评估方法类似。

```
res = opt.minimize(fun=regularized_cost, x0=theta, args=(X, y), method='Newton-CG', jac=regularized_gradient)
print(res)
```

结果展示如下：

```
fun: 0.5290027297130131
jac: array([ 3.38416081e-07,  4.01937968e-08,  5.43735571e-08, -6.
           27273610e-09,
           1.26956590e-08,  6.40907408e-08,  2.69228765e-08, -4.51563057e-09,
           1.50775067e-08,  2.01944806e-08,  8.27631258e-09,  4.27458179e-09,
           2.83024550e-09, -5.76783073e-10,  3.12592429e-08,  1.28289199e-08,
          -5.78226849e-09,  4.59893812e-09,  4.29356683e-09,  4.28094325e-09,
           7.21687570e-09,  1.23194381e-08,  4.56164518e-10, -1.98406451e-09,
          9.54598916e-10, -6.60088763e-11,  6.83325961e-09,  3.14103603e-08])
message: 'Optimization terminated successfully.'
nfev: 7
nhev: 0
nit: 6
njev: 57
status: 0
success: True
x: array([ 1.27274175,  0.62527239,  1.18108994, -2.01996653, -0.
         91742318,
         -1.43166726,  0.12400716, -0.36553523, -0.35723936, -0.17513155,
         -1.45815693, -0.05098901, -0.61555554, -0.27470685, -1.19281737,
         -0.24218799, -0.2060068 , -0.04473071, -0.27778419, -0.2953783 ,
         -0.45635874, -1.04320168,  0.02777154, -0.29243171,  0.01556673,
        -0.32737987, -0.14388645, -0.92465133])
```

可以使用预测函数来查看我们的方案在训练数据上的准确度。

```
def predict(theta, X):
    probability = sigmoid(X @ theta)
    return [1 if x >= 0.5 else 0 for x in probability]
from sklearn.metrics import classification_report
final_theta = res.x
y_predict = predict(final_theta, X)
```

```
print(classification_report(y, y_predict))
```

结果展示如下：

	precision	recall	f1-score	support
0	0.90	0.75	0.82	60
1	0.78	0.91	0.84	58
accuracy			0.83	118
macro avg	0.84	0.83	0.83	118
weighted avg	0.84	0.83	0.83	118

模型的准确率为 83%，符合预测的要求，模型可以使用。

2.2.6 决策边界

此题中要找出 n 个点，使其代入后满足 $\theta^T X = 0$ ，那么可以设 $z = \theta^T X$ ，即画出 $z = 0$ 时的图像问题，此时 $(xx\ yy\ z)$ 为三维空间上的一点。而这个问题恰好可以用 `plt.contour()` 函数来实现。

```
positive = data[data['Accept'].isin([1])]
negative = data[data['Accept'].isin([0])]

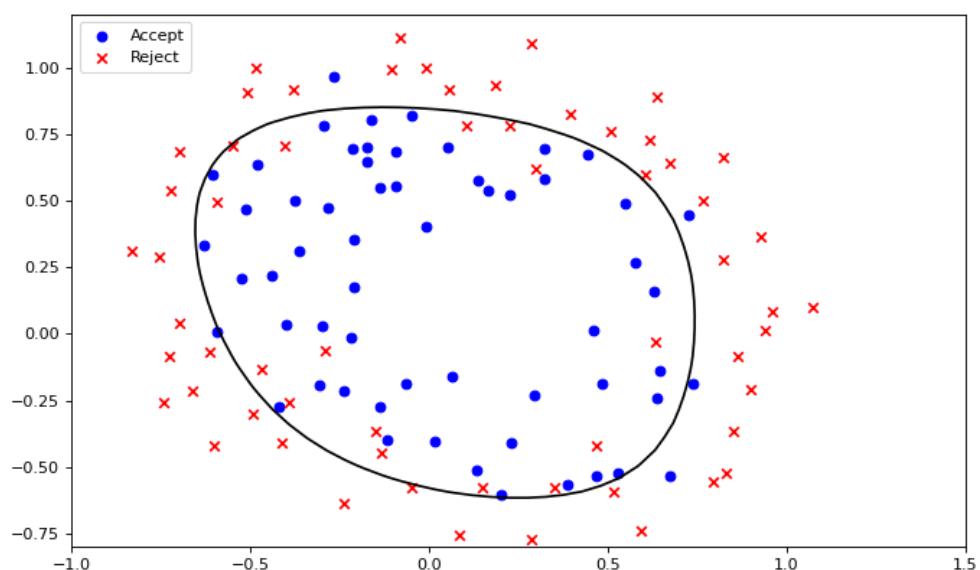
fig, ax = plt.subplots(figsize=(10, 6), dpi=80)
ax.scatter(positive['Test1'], positive['Test2'], marker='o', c='b', label='Accept')
ax.scatter(negative['Test1'], negative['Test2'], marker='x', c='r', label='Reject')

ax.legend(loc=2)
x_label = 'test1'
x_label = 'test2'

x = np.linspace(-1, 1.5, 50)
xx, yy = np.meshgrid(x, x) # 将x里的数组合成50*50=250个坐标
z = np.array(feature_mapping(xx.ravel(), yy.ravel(), 6))
z = z @ final_theta
z = z.reshape(xx.shape)

plt.contour(xx, yy, z, 0, colors='black')
# 等高线是三维图像在二维空间的投影，0表示z的高度为0
plt.ylim(-.8, 1.2)
plt.show()
```

最后得到的图像如下所示：



2.3 附录—Python 代码

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import scipy.optimize as opt
def regression():
    path = ('ex1data1.txt')
    data = pd.read_csv(path, header=None, names=['population', 'profit'])
    print('-----展示前十行代码-----')
    print(data.head(10))
    """
    看下数据长什么样子
    """
    data.plot(kind='scatter', x='population', y='profit', figsize=(10, 6))
    plt.ylabel('profit')
    plt.xlabel('population')
    """
    现在让我们使用梯度下降来实现线性回归，以最小化成本函数。
    """
    def Cost(X, y, theta):
        inner = np.power(((X * theta.T) - y), 2)
        return np.sum(inner) / (2 * len(X))
    """
    让我们在训练集中添加一列，以便我们可以使用向量化的解决方案来计算代价和
    梯度
    """
    data.insert(0, 'Ones', 1)
    alpha = 0.01
    iters = 10000
    """
    现在我们来做一些变量初始化
    """
    # set X (training data) and y (target variable)
    cols = data.shape[1]
    X = data.iloc[:, 0:cols - 1] # X是所有行，去掉最后一列
    y = data.iloc[:, cols - 1:cols] # y是所有行，最后一列

```

```
"""
代价函数是应该是numpy矩阵，所以我们需要转换X和Y，然后才能使用它们。 我们还需要初始化theta。
"""


```

```
X = np.matrix(X.values)
y = np.matrix(y.values)
theta = np.matrix(np.array([0, 0]))
theta_n = (X.T * X).I * X.T * y
print('-----利用正规方程求系数-----')
print(theta_n)
print('-----初始化theta-----')
print(theta)
print('-----检查维数-----')
print(X.shape, theta.shape, y.shape)
print('-----计算代价函数的值为-----')
print(Cost(X, y, theta))
print('-----批量梯度下降-----')

def gradientDescent(X, y, theta, alpha, iters):
    temp = np.matrix(np.zeros(theta.shape))
    parameters = int(theta.ravel().shape[1])
    cost = np.zeros(iters)
    for i in range(iters):
        error = (X * theta.T) - y
        for j in range(parameters):
            term = np.multiply(error, X[:, j])
            temp[0, j] = theta[0, j] - ((alpha / len(X)) * np.sum(term))
        theta = temp
        cost[i] = Cost(X, y, theta)
    return theta, cost
"""


```

现在让我们运行梯度下降算法来将我们的参数 适合于训练集。

```
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
print('-----随机梯度下降法求得的参数-----')
print(final_theta)
"""


```

最后，我们可以使用我们拟合的参数计算训练模型的代价函数（误差）。

```
"""


```

```
print('-----拟合的参数计算训练模型的代价函数-----')
print(Cost(X, y, final_theta))
"""

绘制线性模型以及数据，直观地看出它的拟合。
"""

x = np.linspace(data.population.min(), data.population.max(), 100)
f = final_theta[0, 0] + (final_theta[0, 1] * x)
a = float(final_theta[0, 0])
b = float(final_theta[0, 1])
print('-----输出回归方程-----')
print('回归方程为:f=% .8f+%.8f*x' % (a, b))
x_predict1 = np.matrix([[1, 3.5]])
x_predict2 = np.matrix([[1, 7]])
predict1 = np.dot(x_predict1, final_theta.T)
predict2 = np.dot(x_predict2, final_theta.T)
print('-----输出预测值1-----')
print(predict1)
print('-----输出预测值2-----')
print(predict2)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x, f, 'r', label='Prediction')
ax.scatter(data.population, data.profit, label='Data')
ax.legend(loc=2)
ax.set_xlabel('population')
ax.set_ylabel('profit')

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.arange(iters), cost, 'r', label='alpha=0.01')
alpha = 0.001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'b', label='alpha=0.001')
alpha = 0.0001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'black', label='alpha=0.0001')
alpha = 0.00001
theta = np.matrix(np.array([0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'yellow', label='alpha=0.00001')
```

```
ax.legend(loc=2)
ax.set_xlabel('iters')
ax.set_ylabel('cost')
ax.set_title('gradient function with different alpha')
plt.show()
return None

def regression2():
    path = 'ex1data2.txt'
    data = pd.read_csv(path, header=None, names=['Size', 'Bedrooms', 'Price'])
    print('-----展示前十行数据-----')
    print(data.head(10))
    """
对于此任务，我们添加了另一个预处理步骤 - 特征归一化。
"""

new_data = (data - data.mean()) / data.std()
print('-----特征归一化后的数据-----')
print(new_data.head(10))
"""

重复第1部分的预处理步骤，并对新数据集运行线性回归程序。
"""

new_data.insert(0, 'Ones', 1)
cols = new_data.shape[1]
X = new_data.iloc[:, 0:cols - 1]
y = new_data.iloc[:, cols - 1:cols]
X = np.matrix(X.values)
y = np.matrix(y.values)
print(X.shape, y.shape)
theta = np.matrix(np.array([0, 0, 0]))
theta_n = (X.T * X).I * X.T * y
print('-----利用正规方程求系数-----')
print(theta_n)
alpha=0.001
iters=10000

def Cost(X, y, theta):
    inner = np.power(((X * theta.T) - y), 2)
    return np.sum(inner) / (2 * len(X))

def gradientDescent(X, y, theta, alpha, iters):
    temp = np.matrix(np.zeros(theta.shape))
```

```

parameters = int(theta.ravel().shape[1])
cost = np.zeros(iters)

for i in range(iters):
    error = (X * theta.T) - y
    for j in range(parameters):
        term = np.multiply(error, X[:, j])
        temp[0, j] = theta[0, j] - ((alpha / len(X)) * np.sum(term))
    )
theta = temp
cost[i] = Cost(X, y, theta)

return theta, cost

final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
print('-----代价函数值-----')
print(Cost(X, y, final_theta))
print('-----输出系数值-----')
print(final_theta)

x1 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
x2 = np.linspace(X[:, 2].min(), X[:, 2].max(), 100)
x1, x2 = np.meshgrid(x1, x2)
f = final_theta[0, 0] + final_theta[0, 1] * x1 + final_theta[0, 2] * x2
a=float(final_theta[0,0])
b1=float(final_theta[0,1])
b2=float(final_theta[0,2])
print('输出的回归方程为:f=% .8f*x1% .8f*x2%(b1,b2)')
fig = plt.figure(figsize=(10, 6))
ax = Axes3D(fig)
ax.plot_surface(x1, x2, f, rstride=1, cstride=1, cmap=cm.autumn, label='prediction')

ax.scatter(X[:100, 1], X[:100, 2], y[:100, 0], c='black')
ax.set_zlabel('PRICE')
ax.set_ylabel('BATHROOM')
ax.set_xlabel('SIZE')
"""

快速查看这一个的训练进程。
"""

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.arange(iters), cost, 'r', label='alpha=0.001')
ax.set_xlabel('iters')
ax.set_ylabel('cost')

```

```
alpha = 0.0001
theta = np.matrix(np.array([0, 0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'yellow', label='alpha=0.0001')
alpha = 0.01
theta = np.matrix(np.array([0, 0, 0]))
final_theta, cost = gradientDescent(X, y, theta, alpha, iters)
ax.plot(np.arange(iters), cost, 'black', label='alpha=0.01')
ax.legend(loc=2)
ax.set_title('gradient function with different alpha')
plt.show()
return None

def logisticregression():
    path = 'ex2data1.txt'
    data = pd.read_csv(path, header=None, names=['Exam 1', 'Exam 2', 'Admitted'])
    print(data.head(10))
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))
    random_numbers = np.arange(-10, 10, step=1)
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.plot(random_numbers, sigmoid(random_numbers), 'black')
    def cost(theta, X, y):
        theta = np.matrix(theta)
        X = np.matrix(X)
        y = np.matrix(y)
        first = np.multiply(-y, np.log(sigmoid(X * theta.T)))
        second = np.multiply((1 - y), np.log(1 - sigmoid(X * theta.T)))
        return np.sum(first - second) / (len(X))
    data.insert(0, 'Ones', 1)
    cols = data.shape[1]
    X = data.iloc[:, 0:cols - 1]
    y = data.iloc[:, cols - 1:cols]
    X = np.matrix(X.values)
    y = np.matrix(y.values)
    theta = np.matrix(np.array([0, 0, 0]))
    print(theta)
    print(X.shape, y.shape, theta.shape)
    print('-----输出的代价函数的值为-----')
```

```
print(cost(theta, X, y))

def gradient(theta, X, y):
    theta = np.matrix(theta)
    X = np.matrix(X)
    y = np.matrix(y)
    parameters = int(theta.ravel().shape[1])
    grad = np.zeros(parameters)
    error = sigmoid(X * theta.T) - y
    for i in range(parameters):
        term = np.multiply(error, X[:, i])
        grad[i] = np.sum(term) / len(X)

    return grad

print(gradient(theta, X, y))
res = opt.minimize(fun=cost, x0=theta, jac=gradient, args=(X, y), method
                    = 'Newton-CG')

print(res)

def predict(theta, X):
    probability = sigmoid(X * theta.T)
    return [1 if x >= 0.5 else 0 for x in probability]

print(res.x)
final_theta = res.x
print(final_theta)
final_theta = np.matrix(final_theta)
y_predict = predict(final_theta, X)
from sklearn.metrics import classification_report
print(classification_report(y, y_predict))

coef = -(res.x / res.x[2])
print(coef)

x = np.arange(130, step=0.1)
y = coef[0] + coef[1] * x
positive = data[data['Admitted'].isin([1])]
negative = data[data['Admitted'].isin([0])]

fig, ax = plt.subplots(figsize=(10, 6))
ax.scatter(positive['Exam 1'], positive['Exam 2'], s=50, c='blue',
           marker='o', label='Admitted Yes')
ax.scatter(negative['Exam 1'], negative['Exam 2'], s=50, c='red',
           marker='x', label='Admitted No')

ax.plot(x, y)
ax.set_xlim(20, 110)
```

```
ax.set_xlim(20, 110)
ax.legend(loc=2)
ax.set_xlabel('Exam 1 Score')
ax.set_ylabel('Exam 2 Score')
ax.set_title('Decision Boundary')
plt.show()
return None

def logisticregression2():
    path = 'ex2data2.txt'
    data= pd.read_csv(path, header=None, names=['Test1', 'Test2', 'Accept'])
    )
    print(data.head())
    def feature_mapping(x, y, power, as_ndarray=False):
        data = {"f'{}{}'".format(i - p, p): np.power(x, i - p) * np.power(y,
            p)
            for i in np.arange(power + 1)
            for p in np.arange(i + 1)
            }
        if as_ndarray:
            return np.array(pd.DataFrame(data))
        else:
            return pd.DataFrame(data)
    x1 = np.array(data.Test1)
    x2 = np.array(data.Test2)
    new_data = feature_mapping(x1, x2, power=6)
    print(new_data.shape)
    print(new_data.head())
    theta = np.zeros(new_data.shape[1])
    X = feature_mapping(x1, x2, power=6, as_ndarray=True)
    print(X.shape) # (118, 28)
    y = np.array(data.iloc[:, -1])
    print(y.shape) # (118,)
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))
    def regularized_cost(theta, X, y, l=1):
        thetaReg = theta[1:]
        first = (-y * np.log(sigmoid(X @ theta))) - (1 - y) * np.log(1 -
            sigmoid(X @ theta))
        reg = (thetaReg @ thetaReg) * l / (2 * len(X))
```

```
    return np.mean(first) + reg
print(regularized_cost(theta, X, y, l=1))

def regularized_gradient(theta, X, y, l=1):
    thetaReg = theta[1:]
    first = (X.T @ (sigmoid(X @ theta) - y)) / len(X)
    reg = np.concatenate([np.array([0]), (l / len(X)) * thetaReg])
    return first + reg
print(regularized_gradient(theta,X,y))
print('init cost = {}'.format(regularized_cost(theta, X, y)))
res = opt.minimize(fun=regularized_cost, x0=theta, args=(X, y), method=
                    'Newton-CG', jac=
                    regularized_gradient)

print(res)

def predict(theta, X):
    probability = sigmoid(X @ theta)
    return [1 if x >= 0.5 else 0 for x in probability] # return a list
from sklearn.metrics import classification_report
final_theta = res.x
y_predict = predict(final_theta, X)
predict(final_theta, X)
print(classification_report(y, y_predict))
positive = data[data['Accept'].isin([1])]
negative = data[data['Accept'].isin([0])]
fig, ax = plt.subplots(figsize=(10, 6), dpi=80)
ax.scatter(positive['Test1'], positive['Test2'], marker='o', c='b',
           label='Accept')
ax.scatter(negative['Test1'], negative['Test2'], marker='x', c='r',
           label='Reject')
ax.legend(loc=2)
x_label = 'test1'
x_label = 'test2'
x = np.linspace(-1, 1.5, 50)
# 从 -1到1.5等间距取出50个数
xx, yy = np.meshgrid(x, x)
# 将x里的数组合成为50*50=2500个坐标
z = np.array(feature_mapping(xx.ravel(), yy.ravel(), 6))
z = z @ final_theta
z = z.reshape(xx.shape)
```

```
plt.contour(xx, yy, z, 0, colors='black')
# 等高线是三维图像在二维空间的投影，0表示z的高度为0
plt.ylim(-.8, 1.2)
plt.show()

if __name__ == '__main__':
    logisticregression2()
```