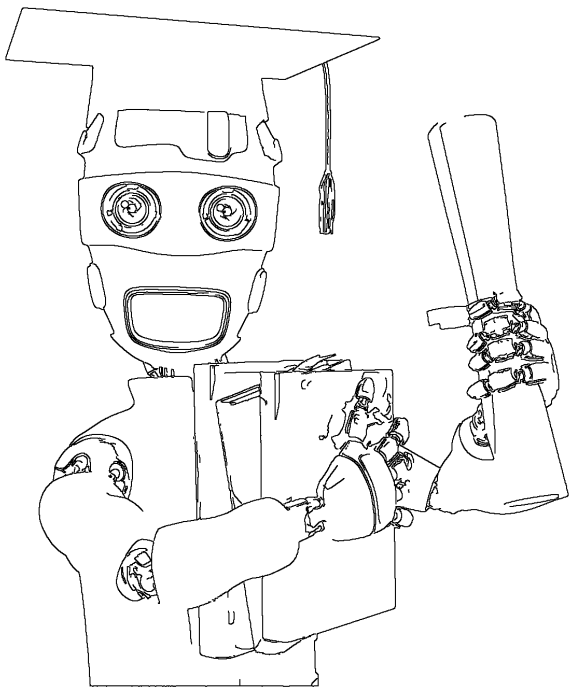
LECTURE NOTE

MACHINE LEARNING BY ANDREW NG



Yifeng Xu
Harbin Institute of Technology, Shenzhen

2020.12 – 2021.02

Course

Machine Learning by Andrew Ng (Coursera)

Lecturer

Prof. Andrew Ng

Slide

GitHub: `https://github.com/TheisTrue/MLofAndrew-Ng`

Online Video

BiliBili: `https://www.bilibili.com/video/BV164411b7dx`

目录

1 梯度下降解一元线性回归

1.1 一元线性回归

给定如下数据集：

$$\left\{\left(x^{(i)}, y^{(i)}\right),\ i = 1, 2, \cdots, m\right\}$$

其中，m 表示样本数量，$x^{(i)}$ 与 $y^{(i)}$ 分别表示第 i 个样本的输入和对应的观测值。一元线性回归的目标是使用线性函数 $y = \theta_1 x + \theta_0$ 对这些数据进行拟合。虽然该问题可以直接通过最小二乘法求得解析解，但本节以它为例介绍如何使用梯度下降法迭代求解。

1.2 梯度下降法

梯度下降法是一种常用的迭代优化算法。对于代价函数 $J(\theta_0, \theta_1)$，其基本思路是从一个初始参数点出发，沿当前点负梯度所指示的下降方向移动一小步，并反复执行这一过程，从而逐步减小代价函数的取值。若代价函数满足适当的凸性与光滑性条件，该过程可以收敛到其全局最小值；在更一般的情况下，梯度下降法通常用于寻找局部极小值或驻点。

函数 J 关于参数 θ_0 和 θ_1 的梯度定义为：

$$\text{grad } J = \left\{\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}\right\}$$

因此，可以按照下式不断更新参数：

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial J}{\partial \theta_j}, \quad j = 0, 1$$

其中，α 表示每次更新的步长，通常称为**学习率**。需要注意的是，在一次迭代中，θ_0 和 θ_1 应当依据更新前的参数值同步计算并更新。

学习率的选取会直接影响算法的收敛速度和稳定性。学习率过小时，每次参数更新的幅度有限，因而收敛过程可能较慢；学习率过大时，参数则可能越过极小值附近的区域并反复振荡，甚至导致算法无法收敛。

1.3 梯度下降解一元线性回归

为了衡量模型预测值与真实观测值之间的差异，定义代价函数为：

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_1 x^{(i)} + \theta_0 - y^{(i)}\right)^2$$

该函数是所有样本预测值与真实值的均方误差。

对代价函数分别关于 θ_0 和 θ_1 求偏导，可得：

$$\text{grad } J = \left\{\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}\right\} = \left\{\frac{1}{m} \sum_{i=1}^{m} \left(\theta_1 x^{(i)} + \theta_0 - y^{(i)}\right), \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \left(\theta_1 x^{(i)} + \theta_0 - y^{(i)}\right)\right\}$$

将上述两个偏导数代入参数更新公式，即可在每次迭代中沿负梯度方向调整 θ_0 和 θ_1.

1.4 代码实现

C++ 实现 对于该问题，可以预先计算样本中若干与参数无关的求和项，使每次梯度更新的时间复杂度降为 $O(1)$.

```cpp
#include<bits/stdc++.h>

using namespace std;

const double eps = 1e-8;
const double alpha = 0.01;

int m;
double sumx, sumy, sumxy, sumx2;
double th0, th1;

int main(){
    freopen("ex1data1.txt", "r", stdin);
    double x, y;
    while(scanf("%lf,%lf", &x, &y) != EOF)
        m++, sumx += x, sumy += y, sumx2 += x * x, sumxy += x * y;
    th0 = th1 = 0;
    while(1){
        double nth0, nth1;
        nth0 = th0 - alpha * 1.0 / m * (th1 * sumx - sumy + m * th0);
        nth1 = th1 - alpha * 1.0 / m * (th1 * sumx2 - sumxy + sumx * th0);
        if(fabs(th0 - nth0) < eps && fabs(th1 - nth1) < eps)    break;
        th0 = nth0, th1 = nth1;
    }
    printf("%f %f\n", th0, th1);
    return 0;
}
```

程序得到的结果为：$\theta_0 = -3.895775, \theta_1 = 1.193033$.

若将 α 设为 0.1，上述程序将无法获得收敛结果。这是因为学习率过大，参数更新可能在极小值附近反复跨越，导致代价函数无法稳定下降。除直接减小 α 之外，还可以让程序根据迭代过程自适应地调整学习率。具体地，当学习率取值合理时，代价函数通常应当随迭代逐步减小。因此，如果一次参数更新后代价函数反而增大，就可以将其视为当前学习率过大的信号，并相应减小 α。基于这一思路，可以得到如下实现：

```cpp
#include<bits/stdc++.h>

using namespace std;

const int N = 105;
const double eps = 1e-8;
double alpha = 100;

int m;
double x[N], y[N];
double th0, th1;

inline double gradJ(int k){
    double res = 0;
    for(int i = 1; i <= m; i++)
```

```
        res += (th1 * x[i] + th0 - y[i]) * (k == 1 ? x[i] : 1);
    return res / m;
}

inline double calc(){
    double res = 0;
    for(int i = 1; i <= m; i++)
        res += (th1 * x[i] + th0 - y[i]) * (th1 * x[i] + th0 - y[i]);
    return res / m / 2;
}

int main(){
    freopen("ex1data1.txt", "r", stdin);
    while(scanf("%lf,%lf", &x[0], &y[0]) != EOF)
        m++, x[m] = x[0], y[m] = y[0];
    th0 = th1 = 0;
    double preJ = 1e9, J = 0;
    while(1){
        double nth0, nth1;
        nth0 = th0 - alpha * gradJ(0);
        nth1 = th1 - alpha * gradJ(1);
        if(fabs(th0 - nth0) < eps && fabs(th1 - nth1) < eps)    break;
        th0 = nth0, th1 = nth1;
        J = calc();
        if(J > preJ)    alpha /= 5;
        preJ = J;
    }
    printf("%f\n%f %f\n", alpha, th0, th1);
    return 0;
}
```

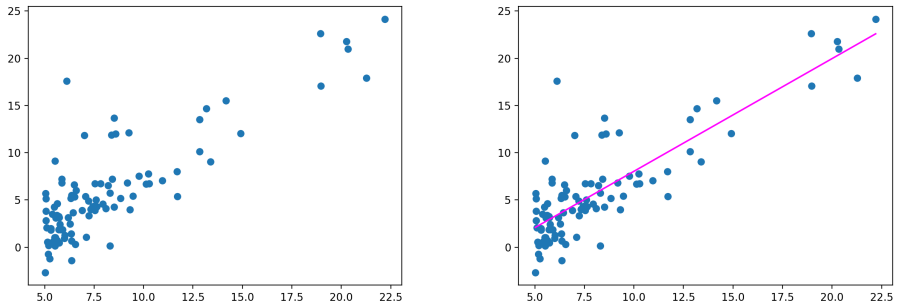
最终得到的结果为：$\alpha = 0.0064, \theta_0 = -3.895772, \theta_1 = 1.193033.$

Python 实现 Python 拥有较为完善的可视化工具，常用于机器学习算法的实现和实验分析。如下是学习率设置为 0.01 的 Python 代码：

```
import numpy as np
import matplotlib.pyplot as plt

X = []
Y = []
m = 0
alpha = 0.01
eps = 1e-8
th0, th1 = 0, 0

def gradJ(k):
    res = 0
    for x, y in zip(X, Y):
        res += (th1 * x + th0 - y) * (x if k == 1 else 1)
    return res / m

def calc():
    res = 0
    for x, y in zip(X, Y):
        res += (th1 * x + th0 - y) * (th1 * x + th0 - y)
    return res / 2 / m
```

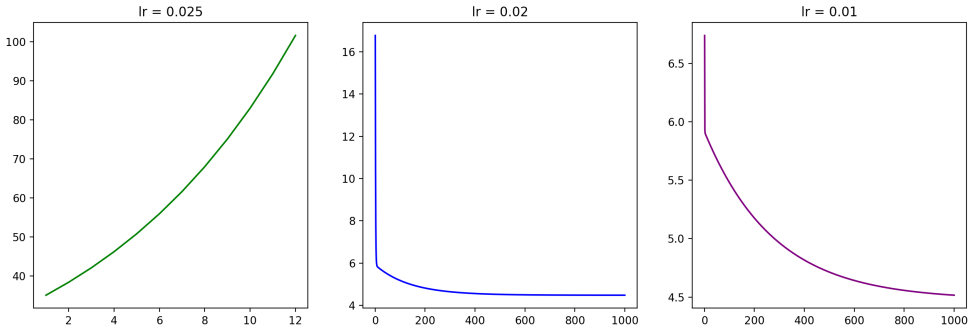
```
with open("ex1data1.txt", "r") as infile:
    data = infile.readlines()
    for line in data:
        x, y = line.split(',')
        X.append(float(x))
        Y.append(float(y))
        m += 1


while 1:
    nth0 = th0 - alpha * gradJ(0)
    nth1 = th1 - alpha * gradJ(1)
    if abs(th0 - nth0) < eps and abs(th1 - nth1) < eps:
        break
    th0 = nth0
    th1 = nth1
print(th0, th1)
plt.scatter(X, Y)
plt.plot([min(X), max(X)], [th0+min(X)*th1, th0+max(X)*th1], c="magenta")
plt.show()
```

程序得到的结果为：$\theta_0 = -3.895775, \theta_1 = 1.193033$，原始数据及拟合效果如下图所示：



图中的洋红色直线即为梯度下降法得到的线性回归模型。

前文已经指出，学习率过大会导致迭代无法收敛，即 $J(\theta_0, \theta_1)$ 可能随迭代次数增加而增大。为了比较不同学习率对收敛过程的影响，可以绘制代价函数随迭代次数变化的曲线：



由图可见，当学习率为 0.025 时，代价函数呈发散趋势；当学习率分别为 0.02 和 0.01 时，代价函数能够逐步收敛。这一结果说明，学习率需要结合具体问题进行选择，适当的取值才能兼顾收敛速度与迭代稳定性。

2 梯度下降解多元线性回归

2.1 多元线性回归

多元线性回归是一元线性回归的推广，其输入包含多个特征 $x_1, x_2, \cdots, x_n$.给定数据集：

$$\left\{\left(x_1^{(i)}, x_2^{(i)}, \cdots, x_n^{(i)}, y^{(i)}\right),\ i = 1, 2, \cdots, m\right\}$$

相应的回归模型也包含多个参数 $\theta_0, \theta_1, \cdots, \theta_n$：

$$h_\theta(x) = \theta^T x = \theta_0 x_0 + \cdots + \theta_n x_n$$

为表示方便，这里假定 x_0 恒等于 1，且记 $\theta = (\theta_0, \cdots, \theta_n)^T$, $x = (x_0, \cdots, x_n)^T$.

2.2 梯度下降解多元线性回归

代码函数及其梯度 与一元线性回归类似，定义代价函数为均方误差：

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

其关于参数向量的梯度为：

$$\frac{\partial J}{\partial \theta} = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right) x^{(i)}$$

因此，梯度下降的参数更新公式为：

$$\theta := \theta - \alpha \cdot \frac{\partial J}{\partial \theta}$$

反复执行上述更新，直至参数或代价函数收敛。

特征缩放与标准化 当不同特征的取值范围相差较大时，代价函数的等高线会较为狭长，梯度下降可能需要更多次迭代才能收敛。为改善收敛速度，可以对各个特征进行标准化：

$$x_i^{(j)} := \frac{x_i^{(j)} - \mu_i}{\sigma_i}$$

其中，$\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_i^{(j)}$ 是样本均值，$\sigma_i^2 = \frac{1}{m-1} \sum_{j=1}^{m} \left(x_i^{(j)} - \mu_i\right)^2$ 是样本方差。标准化后，样本均值为 0，方差为 1，不同特征具有相近的尺度，有利于梯度下降稳定而快速地收敛。

2.3 代码实现

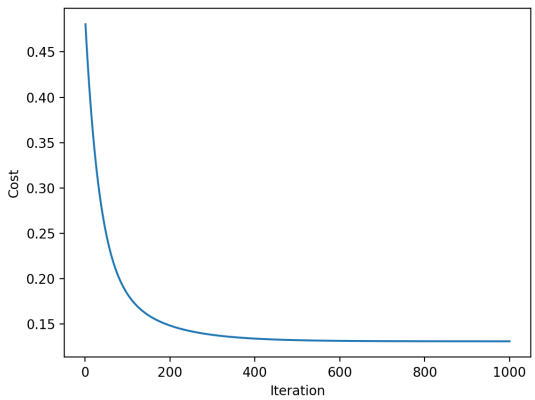
以下代码中，`Normalization` 函数用于标准化数据，`J` 函数用于计算 $J(\theta)$，`partJ` 函数用于计算 $\frac{\partial J}{\partial \theta}$，`GradientDescent` 函数则负责执行梯度下降。

```
import numpy as np
import matplotlib.pyplot as plt

alpha = 0.01
iteration = 10000
Z = []
```

```
def Normalization(data):
    return (data - data.mean(axis = 0)) / data.std(axis = 0, ddof = 1)

def J(T, X, Y):
    res = 0
    for i in range(m):
        res += (np.matmul(T.T, X[i:i+1, :].T) - Y[i:i+1, :]) ** 2
    res /= 2 * m;
    return res

def partJ(T, X, Y):
    res = np.zeros((n, 1))
    for i in range(m):
        res += (np.matmul(T.T, X[i:i+1, :].T) - Y[i:i+1, :]) * X[i:i+1, :].T
    res /= m
    return res

def GradientDescent(X, Y):
    T = np.zeros((n, 1))
    for t in range(iteration):
        T = T - alpha * partJ(T, X, Y)
        Z.append(J(T, X, Y)[0][0])
    return T

data = np.genfromtxt("ex1data2.txt", delimiter = ',')
(m, n) = data.shape
data = Normalization(data)
X = np.column_stack((np.ones((m, 1)), data[:, :-1]))
Y = data[:, -1:]
T = GradientDescent(X, Y)
print(T)

p1 = plt.subplot(111)
p1.plot(range(1, iteration+1), Z)
p1.set_xlabel('Iteration')
p1.set_ylabel('Cost')
plt.show()
```

最终得到的参数为：$\theta = (-1.11051830 \times 10^{-16}, 8.84765988 \times 10^{-1}, -5.31788197 \times 10^{-2})^T$.

当学习率取为 0.01 时，代价函数随迭代次数的变化如下图所示：



3 正规方程解多元线性回归

3.1 正规方程

正规方程利用多元微分直接求解线性回归的最优参数，无须像梯度下降那样反复迭代。

对于代价函数：

$$J(\theta) = J(\theta_0, \theta_1, \cdots, \theta_n)$$

若代价函数可微且为凸函数，则可令各偏导数为零，从而求得使其取最小值的参数 θ：

$$\frac{\partial J}{\partial \theta_j} = 0, \quad j = 0, 1, \cdots, n$$

写成向量形式即为：

$$\frac{\partial J}{\partial \theta} = \vec{0}$$

下面将这一方法应用于多元线性回归。

多元线性回归的代价函数为：

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

其梯度为：

$$\frac{\partial J}{\partial \theta} = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right) x^{(i)}$$

令其为零，解得：

$$\theta^T x^{(i)} = y^{(i)}, \quad i = 1, 2, \cdots, m$$

这些等式可以统一写成矩阵形式：

$$X\theta = y$$

其中，

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} = \begin{bmatrix} {x^{(1)}}^T \\ {x^{(2)}}^T \\ \vdots \\ {x^{(m)}}^T \end{bmatrix}, \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

两边同时乘以 X^T，并假设 $X^T X$ 可逆，解得：

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

这就是多元线性回归的解析解。

由于 $X^T X$ 是一个 $(n+1)\times(n+1)$ 的矩阵，直接求逆或求解线性方程组的时间复杂度通常为 $O(n^3)$. 当 n 不大时，正规方程能够一次得到解析解，也不需要选择学习率；但当 n 达到 $10^4, 10^5$ 或更高数量级时，矩阵运算的时间和空间开销都会迅速增加，此时通常应考虑梯度下降等迭代方法。

3.2 代码实现

仍使用第 2 节中的多元线性回归数据，代码如下：

```
import numpy as np

def J(T, X, Y):
    res = 0
    for i in range(m):
        res += (np.matmul(T.T, X[i:i+1, :].T) - Y[i:i+1, :]) ** 2
    res /= 2 * m;
    return res

data = np.genfromtxt("ex1data2.txt", delimiter = ',')
(m, n) = data.shape
X = np.column_stack((np.ones((m, 1)), data[:, :-1]))
Y = data[:, -1:]
T = np.matmul(np.matmul(np.linalg.inv(np.matmul(X.T, X)), X.T), Y)
print(T)
print(J(T, X, Y))
```

计算很快即可完成，结果为：$\theta = (89597.9095428, 139.21067402, -8738.01911233)^T$.

3.3 不可逆情形

上述求解过程建立在 X^TX 可逆的假设上。当特征之间存在冗余或线性相关关系时，该矩阵可能不可逆。此时可以将代码中的 `inv()` 替换为 `pinv()`，通过**伪逆矩阵**求得一个最小二乘意义下的解。

```
import numpy as np

def J(T, X, Y):
    res = 0
    for i in range(m):
        res += (np.matmul(T.T, X[i:i+1, :].T) - Y[i:i+1, :]) ** 2
    res /= 2 * m;
    return res

data = np.genfromtxt("ex1data2.txt", delimiter = ',')
(m, n) = data.shape
X = np.column_stack((np.ones((m, 1)), data[:, :-1]))
Y = data[:, -1:]
T = np.matmul(np.matmul(np.linalg.pinv(np.matmul(X.T, X)), X.T), Y)
print(T)
print(J(T, X, Y))
```

4 逻辑回归之二分类

4.1 二分类问题与逻辑回归的引入

给定数据集：

$$\left\{\left(x^{(i)}, y^{(i)}\right),\ i = 1, 2, \cdots, m\right\}$$

其中，$x^{(i)}$ 是 n 维特征向量 $(x_0^{(i)}, \cdots, x_n^{(i)})^T$，$y^{(i)} \in \{0, 1\}$ 表示输入 $x^{(i)}$ 所属的类别。二分类的目标是学习一个能够预测样本类别的模型。

由于 y 的取值是离散的，而线性回归的输出可以取任意实数，直接使用线性回归很难得到具有明确概率意义的分类结果，因此引入逻辑回归。

在线性函数 $h_\theta(x) = \theta^T x$ 的基础上应用 sigmoid 函数，可得逻辑回归的假设函数：

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

> 注解. sigmoid 函数：
>
> $$g(z) = \frac{1}{1 + e^{-z}}$$
>
> 是一个值域为 $(0, 1)$ 的单调递增 S 形函数。

对于参数为 θ 的逻辑回归模型，输入 x 后得到 $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$. 由于该值始终处于 $(0, 1)$，可以将它解释为 x 属于类别 1 的概率。当该概率 $\geqslant 0.5$ 时，将样本分类为 1；否则分类为 0.

根据 sigmoid 函数的性质，$h_\theta(x) \geqslant 0.5 \iff \theta^T x \geqslant 0$. 因而只需判断 $\theta^T x$ 与 0 的大小：当 $\theta^T x \geqslant 0$ 时分类为 1，否则分类为 0. 等式 $\theta^T x = 0$ 所表示的直线或超平面称为**决策边界**，它将特征空间划分为两个分类区域。

4.2 代价函数

接下来需要从数据中求解逻辑回归参数 θ，仍采用最小化代价函数的方法。

一般地，代价函数可以写为：

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right)$$

其中，$\text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right)$ 衡量单个样本的预测值 $h_\theta(x^{(i)})$ 与真实值 $y^{(i)}$ 之间的差异。例如，在线性回归中可以使用二者差值的平方作为单个样本的代价，再对全部样本取平均。

对于逻辑回归，单个样本的代价定义为：

$$\begin{aligned}
\text{Cost}\left(h_\theta(x), y\right) &= \begin{cases} -\ln(h_\theta(x)), & y = 1 \\ -\ln(1 - h_\theta(x)), & y = 0 \end{cases} \\
&= -y \ln(h_\theta(x)) - (1 - y) \ln(1 - h_\theta(x)) \\
&= y \ln\left(1 + e^{-\theta^T x}\right) + (1 - y) \ln\left(1 + e^{\theta^T x}\right)
\end{aligned}$$

当真实类别为 1 时，预测概率越接近 1，代价越小；当预测概率接近 0 时，代价会迅速增大。真实类别为 0 时则正好相反。这种定义不仅符合分类任务的直觉，也能得到便于优化的凸代价函数。

注解 (上式的来源). 由 $h_\theta(x) = \mathbb{P}(y=1)$ 可知 $1 - h_\theta(x) = \mathbb{P}(y=0)$，因此：

$$\mathbb{P}(y=k) = [h_\theta(x)]^k [1-h_\theta(x)]^{1-k}, \quad k \in \{0,1\}$$

采用**极大似然法**，并假设各样本相互独立。对于数据集 $\left\{\left(x^{(i)}, y^{(i)}\right), i = 1, 2, \cdots, m\right\}$，所有观测同时出现的似然函数为：

$$L(\theta) = \prod_{i=1}^{m} \mathbb{P}\left(y = y^{(i)}\right) = \prod_{i=1}^{m} \left[h_\theta(x^{(i)})\right]^{y^{(i)}} \left[1 - h_\theta(x^{(i)})\right]^{1-y^{(i)}}$$

取对数可得：

$$\ln L(\theta) = \sum_{i=1}^{m} y^{(i)} \ln(h_\theta(x^{(i)})) + (1-y^{(i)}) \ln(1 - h_\theta(x^{(i)}))$$

极大似然估计要求选择参数，使 $L(\theta)$ 或 $\ln L(\theta)$ 取得最大值。为了将该目标转化为通常的最小化问题，可以对对数似然取负，并定义：

$$J(\theta) = -\frac{1}{m} \ln L(\theta)$$

系数 $\frac{1}{m}$ 仅用于对所有样本的代价取平均，不会改变极大值或极小值对应的参数位置。
因此，**逻辑回归本质上是在拟合类别的概率，并通过极大似然法求解参数**。

4.3 梯度下降解逻辑回归

上述逻辑回归代价函数是凸函数，适合使用梯度下降法求解。

首先计算单个样本代价关于参数的偏导：

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathrm{Cost}(h_\theta(x), y) &= \frac{\partial}{\partial\theta}\left[y \ln\left(1 + e^{-\theta^T x}\right) + (1-y)\ln\left(1 + e^{\theta^T x}\right)\right] \\
&= \frac{-yxe^{-\theta^T x}}{1 + e^{-\theta^T x}} + \frac{(1-y)xe^{\theta^T x}}{1 + e^{\theta^T x}} \\
&= \frac{-yx + (1-y)xe^{\theta^T x}}{1 + e^{\theta^T x}} \\
&= \left(-y + \frac{1}{1 + e^{-\theta^T x}}\right) x \\
&= (h_\theta(x) - y)x
\end{aligned}$$

因此，

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

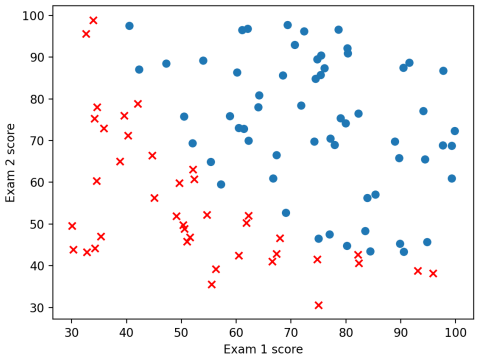
该梯度与线性回归中的表达式形式完全相同，二者的区别仅在于 $h_\theta(x)$ 的定义：逻辑回归的假设函数在线性函数外增加了非线性的 sigmoid 函数，将输出映射到 $(0,1)$. 也正是由于这一非线性变换，逻辑回归不能像线性回归那样直接使用正规方程给出解析解，通常需要采用梯度下降等数值方法。

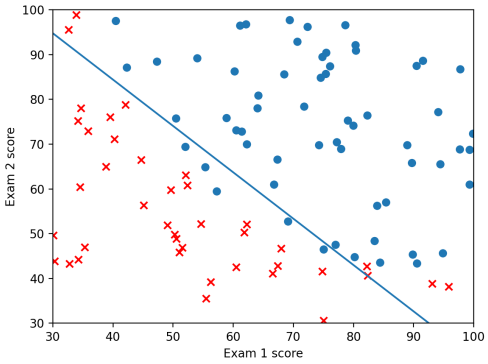
4.4 代码实现

逻辑回归的 Python 实现如下：

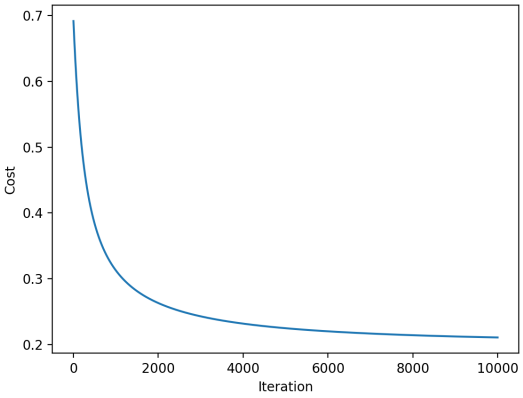
```
import numpy as np
import matplotlib.pyplot as plt
import math

alpha = 0.01
iteration = 10000
Z = []

def Normalization(data):
    return (data - data.mean(axis = 0)) / data.std(axis = 0, ddof = 1)

def h(T, x):
    return 1 / (1 + np.e ** (-np.matmul(T.T, x)[0][0]))

def J(T, X, Y):
    res = 0
    for i in range(m):
        res -= Y[i] * math.log(h(T, X[i:i+1, :].T)) + \
        (1 - Y[i]) * math.log(1 - h(T, X[i:i+1, :].T))
    res /= m
    return res

def partJ(T, X, Y):
    res = np.zeros((n, 1))
    for i in range(m):
        res += (h(T, X[i:i+1, :].T) - Y[i]) * X[i:i+1, :].T
    res /= m
    return res

def GradientDescent(X, Y):
    T = np.zeros((n, 1))
    for t in range(iteration):
        T = T - alpha * partJ(T, X, Y)
        Z.append(J(T, X, Y))
    return T

data = np.genfromtxt("ex2data1.txt", delimiter = ',')
(m, n) = data.shape
data[:, :-1] = Normalization(data[:, :-1])
X = np.column_stack((np.ones((m, 1)), data[:, :-1]))
Y = data[:, -1]
T = GradientDescent(X, Y)
print(T)
print(J(T, X, Y))

p1 = plt.subplot(111)
p1.plot(range(1, iteration+1), Z)
p1.set_xlabel("Iteration")
p1.set_ylabel("Cost")
plt.show()
```

取学习率 0.01 并迭代 10000 次，得到 $\theta = (1.2677702, 3.05550587, 2.81891901)^T$，此时 $J(\theta) = 0.21065763610049573$. 将得到的参数代入 $\theta^T x = 0$，即可绘制对应的决策边界：





$J(\theta)$ 随迭代次数的变化如下：



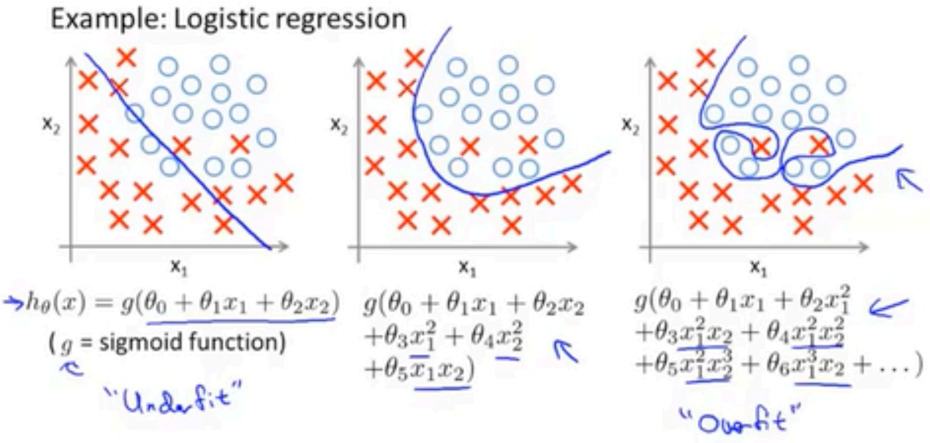
5 正则化

5.1 过拟合

多项式回归通过在线性回归中人为引入高阶项，并将它们视为新的特征，从而拟合非线性数据。例如，可以将样本 $\left(x^{(i)}, y^{(i)}\right)$ 扩展为 $\left(x^{(i)}, {x^{(i)}}^2, y^{(i)}\right)$，使假设函数变为 $y = \theta_0 + \theta_1 x + \theta_2 x^2$，从而用二次函数而不是直线拟合数据。

尽管增加多项式次数能够提高模型的表达能力，但次数并非越高越好。当特征维数 n 接近样本数量 m 时，模型可能近乎完美地拟合训练数据，却无法对训练集之外的新数据作出可靠预测，这种现象称为过拟合。例如，当 $n = m$ 时，根据插值原理，可以找到一个 $n-1$ 次多项式经过所有样本点，使训练集中每个点的误差都为 0；但这条曲线可能在样本点之间剧烈波动，泛化能力很差，如下图所示：

逻辑回归同样可能出现过拟合：



缓解过拟合的方法有很多，本节主要介绍正则化（Regularization）。

5.2 正则化

仍以多项式回归为例。若假设函数为 $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$，并且假定高阶项 θ_3, θ_4 导致了过拟合，那么一个自然的想法是在优化时限制这两个参数。在原代价函数的基础上，可以增加两项：$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + 1000\theta_3^2 + 1000\theta_4^2\right]$，此时，较大的惩罚系数会使非零

的 θ_3 和 θ_4 显著增大代价。因而在最小化 $J(\theta)$ 时，这两个参数会被压缩到接近 0，高阶项对模型的影响也随之减弱。

实际问题中通常无法预先判断究竟哪些参数会导致过拟合，因此不只惩罚某几个指定参数，而是对除偏置项外的所有参数统一加入正则化项：

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\text{Cost}\left(h_\theta(x^{(i)}) - y^{(i)}\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

其中，λ 是正则化参数，$\lambda\sum_{j=1}^{n}\theta_j^2$ 是正则化项，即对**除 θ_0 以外**的参数进行惩罚。

这个代价函数同时考虑了拟合能力和模型复杂度。第一项 $\frac{1}{m}\sum_{i=1}^{m}\text{Cost}\left(h_\theta(x^{(i)}) - y^{(i)}\right)$ 衡量模型对训练数据的拟合误差；第二项 $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ 限制各 θ_j 的规模，以避免模型过度依赖某些特征。正则化系数 λ 用于权衡二者：λ 过大时，参数会被压缩得过小，模型可能无法充分拟合数据，从而产生欠拟合；λ 过小时，正则化作用不足，模型仍可能过拟合。因此，选择合适的 λ 十分重要。

5.3 线性回归的正则化

线性回归加入正则化项后的代价函数为：

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

对其求导：

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T x^{(i)} - y^{(i)}\right)x_j^{(i)} + [j \neq 0]\frac{\lambda}{m}\theta_j, \quad j = 0, 1, \cdots, n$$

将该梯度代入参数更新公式，梯度下降的迭代过程为：

$$\begin{aligned}
\theta_j &:= \theta_j - \alpha\frac{\partial J}{\partial \theta_j} \\
&= \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(\theta^T x^{(i)} - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \\
&= \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(\theta^T x^{(i)} - y^{(i)}\right)x_j^{(i)}, \quad j = 1, \cdots, n
\end{aligned}$$

由最后一行可以看出，正则化会在每次迭代时以 $(1 - \alpha\frac{\lambda}{m})$ 的倍率额外缩小参数。

除梯度下降外，正则化线性回归也可以使用正规方程求解。为便于推导，先暂时假设 θ_0 也参与惩罚，并将 $J(\theta)$ 展开后写成矩阵形式：

$$\begin{aligned}
J(\theta) &= \frac{1}{2m}\left[\sum_{i=1}^{m}\left(\theta^T x^{(i)}\right)^2 - 2\sum_{i=1}^{m}\theta^T x^{(i)} y^{(i)} + \sum_{i=1}^{m}\left(y^{(i)}\right)^2 + \lambda\theta^T\theta\right] \\
&= \frac{1}{2m}\left[\left(\theta^T X^T\right)\left(\theta^T X^T\right)^T - 2\theta^T X^T y + y^T y + \lambda\theta^T\theta\right] \\
&= \frac{1}{2m}\left[\theta^T X^T X\theta - 2\theta^T X^T y + y^T y + \lambda\theta^T\theta\right]
\end{aligned}$$

令

$$\frac{\partial J}{\partial \theta} = \frac{1}{m}\left[X^T X\theta - X^T y + \lambda\theta\right] = 0$$

则：

$$(X^T X + \lambda)\theta = X^T y$$

解得：

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

再考虑到 $j = 0$ 不参与正则化，故最终结果为：

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \right)^{-1} X^T y$$

5.4 逻辑回归的正则化

逻辑回归加入正则化项后的代价函数为：

$$\begin{aligned} J(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln(h_\theta(x^{(i)})) + (1-y^{(i)})\ln(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln\left(1+e^{-\theta^T x^{(i)}}\right) + \left(1-y^{(i)}\right)\ln\left(1+e^{\theta^T x^{(i)}}\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \end{aligned}$$

对其求导：

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + [j \neq 0]\frac{\lambda}{m}\theta_j, \quad j = 0, 1, \cdots, n$$

因此，梯度下降的迭代过程为：

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha\frac{\partial J}{\partial \theta_j} \\ &= \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha\frac{\partial J}{\partial \theta_j} \\ &= \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \\ &= \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}, \quad j = 1, \cdots, n \end{aligned}$$

为了便于运算，下面将上述表达式改写为矩阵形式。需要注意的是，以下写法与 `numpy` 的广播机制相对应，其中指数、对数以及乘法均按元素作用，因此数学记号并不完全严谨。

代价函数的矩阵形式　逻辑回归的代价函数为：

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

其中，

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

设：

$$X=\begin{bmatrix}(x^{(1)})^T\\(x^{(2)})^T\\\vdots\\(x^{(m)})^T\end{bmatrix}\quad\theta=\begin{bmatrix}\theta_0\\\theta_1\\\vdots\\\theta_n\end{bmatrix}\quad y=\begin{bmatrix}y^{(0)}\\y^{(1)}\\\vdots\\y^{(m)}\end{bmatrix}$$

于是：

$$X\theta=\begin{bmatrix}(x^{(1)})^T\theta\\(x^{(2)})^T\theta\\\vdots\\(x^{(m)})^T\theta\end{bmatrix}=\begin{bmatrix}\theta^T(x^{(1)})\\\theta^T(x^{(2)})\\\vdots\\\theta^T(x^{(m)})\end{bmatrix}$$

记：

$$\bar{h}_\theta(X)=1/(1+e^{-X\theta})\in\mathbb{R}^m$$

则：

$$J(\theta)=-\frac{1}{m}\left[y^T\ln(\bar{h}_\theta(X))+(1-y^T)\ln(1-\bar{h}_\theta(X))\right]$$

代价函数偏导的矩阵形式　逻辑回归代价函数的梯度向量为：

$$\frac{\partial J}{\partial\theta}=\frac{1}{m}\sum_{i=1}^{m}\left[\left(h_\theta(x^{(i)})-y^{(i)}\right)x^{(i)}\right]=\frac{1}{m}X^T(\bar{h}_\theta(X)-y)$$

正则化后的矩阵形式　正则化后的逻辑回归代价函数及其梯度为：

$$\begin{aligned}J(\theta)&=\frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\ln(h_\theta(x^{(i)}))-(1-y^{(i)})\ln(1-h_\theta(x^{(i)}))\right]+\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2\\\frac{\partial J}{\partial\theta}&=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)})-y^{(i)}\right)x^{(i)}+\frac{\lambda}{m}\hat{\theta}\end{aligned}$$

其中，$\hat{\theta}=(0,\theta_1,\cdots,\theta_n)^T$，即将 θ 中的 θ_0 替换为 0.

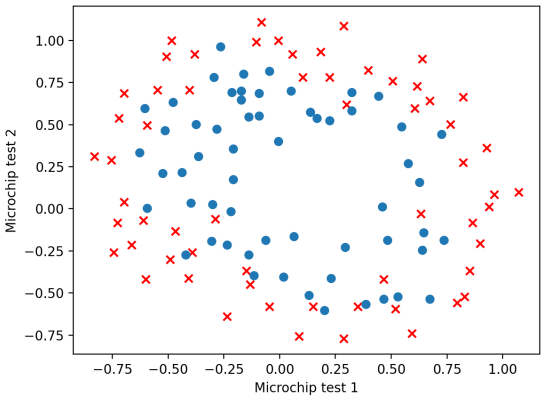
将前面得到的非正则化矩阵形式与正则化项结合，即可得到：

$$\begin{aligned}J(\theta)&=-\frac{1}{m}\left[y^T\ln(\bar{h}_\theta(X))+(1-y^T)\ln(1-\bar{h}_\theta(X))\right]+\frac{\lambda}{2m}\hat{\theta}^T\hat{\theta}\\\frac{\partial J}{\partial\theta}&=\frac{1}{m}X^T(\bar{h}_\theta(X)-y)+\frac{\lambda}{m}\hat{\theta}\end{aligned}$$

这也是使用 `numpy` 实现正则化逻辑回归时采用的表达式。

5.5 代码实现

首先观察数据集：

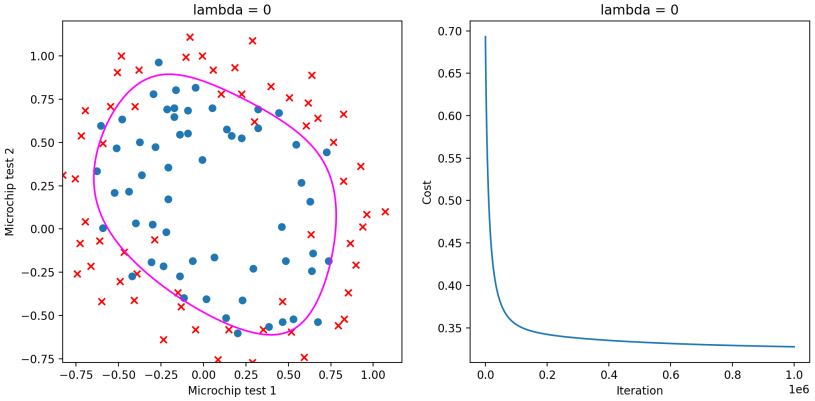


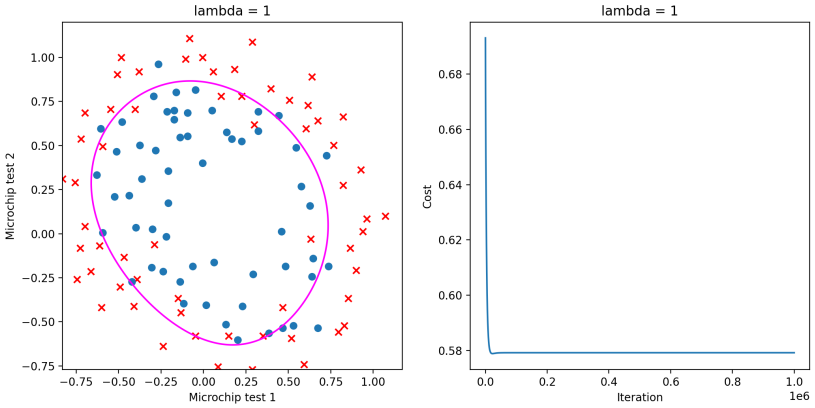
从散点分布可以看出，线性决策边界无法很好地区分两类样本，因此需要引入多项式特征。这里将两个原始特征分别扩展至 3 次，组合得到 16 维特征，再进行正则化逻辑回归。

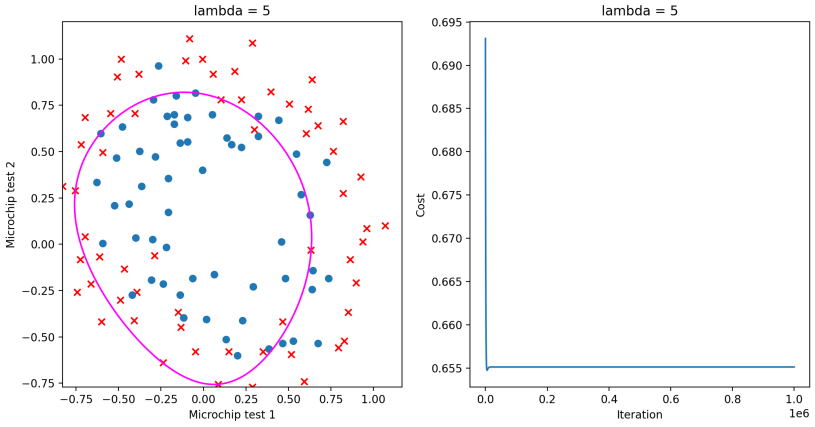
```
import numpy as np
import matplotlib.pyplot as plt

alpha = 0.01
lamb = 1
iteration = 1000000
Z = []

def h(Theta, x):
    return 1 / (1 + np.exp(-np.matmul(Theta.T, x)[0][0]))

def h_(Theta, X):
    return 1 / (1 + np.exp(-np.matmul(X, Theta)))

def J_reg(Theta, X, y):
    tmp = h_(Theta, X)
    return (-(np.matmul(y.T, np.log(tmp)) + np.matmul(1 - y.T, np.log(1 - tmp))) / m\
         + lamb / 2 / m * np.matmul(Theta.T, Theta))[0][0]

def partJ_reg(Theta, X, y):
    return (np.matmul(X.T, h_(Theta, X) - y) + lamb * np.vstack((np.zeros((1, 1)), Theta[1:, :]))) /
          m

def GradientDescent(X, Y):
    T = np.zeros((n, 1))
    for t in range(iteration):
        T = T - alpha * partJ_reg(T, X, Y)
        Z.append(J_reg(T, X, Y))
    return T

def extend(x1, x2):
    X = []
    for j in range(4):
        for k in range(4):
            X.append(np.power(x1, j) * np.power(x2, k))
```

```
    return X

data = np.genfromtxt("ex2data2.txt", delimiter = ',')
(m, n) = data.shape
Y = data[:, -1:]
n = 16
X = np.zeros((m, n))
for i in range(m):
    X[i] = extend(data[i][0], data[i][1])

T = GradientDescent(X, Y)
print(T.T)
print(J_reg(T, X, Y))

#============================== draw the picture ==============================#

def calc(x1, x2):
    res = np.zeros((x1.shape[0], x1.shape[1]))
    for ix in range(x1.shape[0]):
        for iy in range(x1.shape[1]):
            tmp = np.array(extend(x1[ix][iy], x2[ix][iy]), ndmin = 2)
            res[ix][iy] = h(T, tmp.T)
    return res

minx1, maxx1 = data[:, 0].min(), data[:, 0].max()+0.1
minx2, maxx2 = data[:, 1].min(), data[:, 1].max()+0.1
delta = 0.01
x1, x2 = np.meshgrid(np.arange(minx1, maxx1, delta), np.arange(minx2, maxx2, delta))

p1 = plt.subplot(121)
plt.contour(x1, x2, calc(x1, x2), [0.5], colors = 'magenta')

X0 = data[data[:, -1] == 0, :]
X1 = data[data[:, -1] == 1, :]
p1.set_title("lambda = " + str(lamb))
p1.set_xlabel("Microchip test 1")
p1.set_ylabel("Microchip test 2")
p1.scatter(X0[:, 0:1], X0[:, 1:2], marker = 'x', c = 'r')
p1.scatter(X1[:, 0:1], X1[:, 1:2], marker = 'o')

p2 = plt.subplot(122)
p2.plot(range(1, iteration+1), Z)
p2.set_title("lambda = " + str(lamb))
p2.set_xlabel("Iteration")
p2.set_ylabel("Cost")

plt.show()
```

下面分别令 $\lambda = 0, 1, 5$，观察正则化强度变化对决策边界和收敛过程的影响：



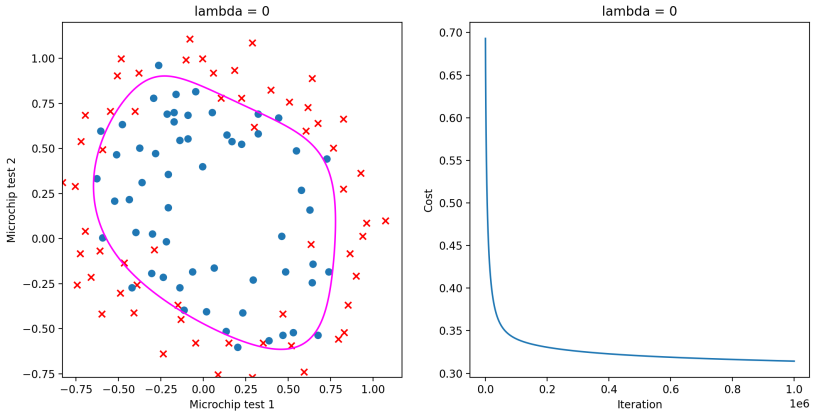


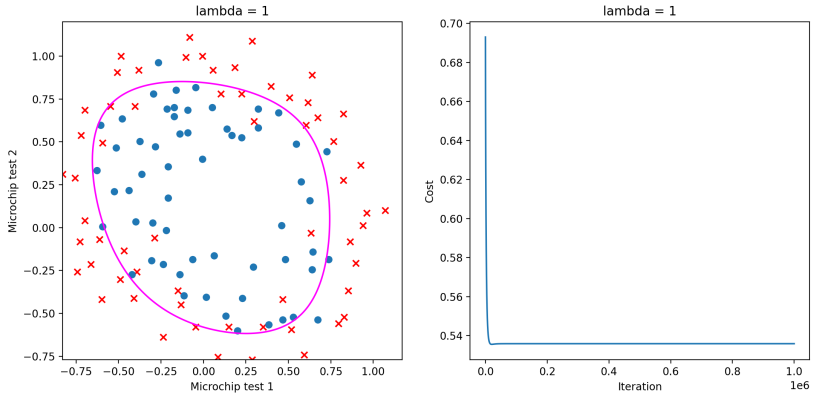


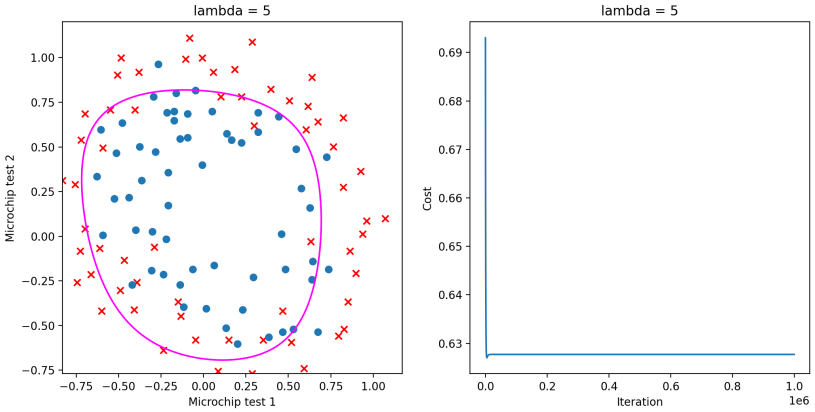
另一种方式生成特征的方式：

```
def extend(x1, x2):
    X = []
    for j in range(6 + 1):
        for k in range(j + 1):
            X.append(np.power(x1, k) * np.power(x2, j-k))
    return X
```

再次比较不同正则化参数，相应结果如下：







6 逻辑回归之多分类（数字识别）

6.1 数据读入与显示

数据采用.mat 格式，可以使用 scipy 中的 loadmat 方法读取：

```
import numpy as np
from scipy.io import loadmat

data = loadmat('ex3data1.mat')
print(data)
print(data['X'].shape)
print(data['y'].shape)
```

```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Oct 16 13:09:09 2011', '
    __version__': '1.0', '__globals__': [], 'X': array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]]), 'y': array([[10],
       [10],
       [10],
       ...,
       [ 9],
       [ 9],
       [ 9]], dtype=uint8)}
(5000, 400)
(5000, 1)
```

文件中共有 5000 个样本。每个样本的输入是由 20×20 灰度图像展开得到的 400 维向量，输出为图像对应的数字标签。

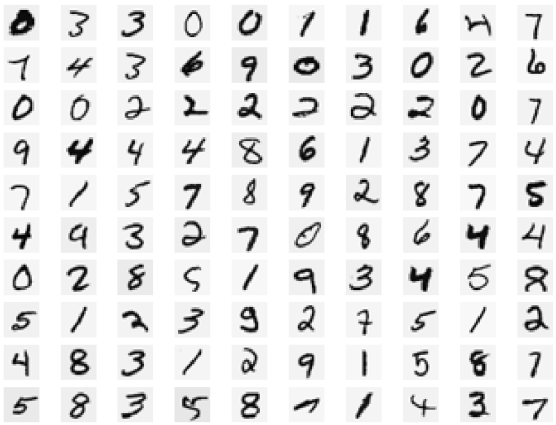
> 注意. 由于 MATLAB 或 Octave 的数据约定，数字 0 被标记为 10，使用 Python 时可以将其恢复为 0. 此外，图像数据按列展开，还原为 20×20 矩阵后需要转置。

```
data = loadmat('ex3data1.mat')
X = data['X']
X = np.transpose(X.reshape((5000, 20, 20)), [0, 2, 1]).reshape(5000, 400)
y = data['y']
y[y==10] = 0
```

随机选取 100 幅图像，显示结果如下：

```
def show_a_number(num, ax):
    """
    num.shape: (400, )
    """
    ax.matshow(num.reshape((20, 20)), cmap=matplotlib.cm.binary)
    ax.axis('off')

fig, ax = plt.subplots(10, 10)
```

```
for i in range(10):
    for j in range(10):
        id = np.random.randint(0, 5000)
        show_a_number(X[id, :], ax[i][j])
plt.show()
```



6.2 多分类

这里采用“一对多”策略完成多分类。对于每一个数字，都单独训练一个逻辑回归分类器：属于当前数字的样本标记为正类，其余数字全部标记为负类。这样便可将一个十分类问题转化为 10 个二分类问题。预测时，分别计算所有分类器给出的概率，并选择概率最大的类别作为最终结果。

沿用上一节的矩阵化正则逻辑回归代码，分别训练 10 个分类器。注意需要在 X 的每一行前添加一个 1 作为偏置项。

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import matplotlib

lamb = 1
alpha = 0.1
iteration = 10000
m = 5000
n = 401

data = loadmat('ex3data1.mat')
X = data['X']
X = np.transpose(X.reshape((m, 20, 20)), [0, 2, 1]).reshape(m, 400)
X = np.hstack((np.ones((m, 1)), X))
y = data['y']
y[y==10] = 0

def h(Theta, x):
    return 1 / (1 + np.exp(-np.matmul(Theta.T, x)[0][0]))

def h_(Theta, X):
    return 1 / (1 + np.exp(-np.matmul(X, Theta)))
```

```
def J_reg(Theta, X, y):
    tmp = h_(Theta, X)
    return (-(np.matmul(y.T, np.log(tmp)) + np.matmul(1 - y.T, np.log(1 - tmp))) / m\
         + lamb / 2 / m * np.matmul(Theta.T, Theta))[0][0]

def partJ_reg(Theta, X, y):
    return (np.matmul(X.T, h_(Theta, X) - y) + lamb * np.vstack((np.zeros((1, 1)), Theta[1:, :]))) /
          m

def GradientDescent(X, y):
    Theta = np.zeros((n, 1))
    for t in range(iteration):
        Theta = Theta - alpha * partJ_reg(Theta, X, y)
    return Theta

def classify(k):
    y_ = y.copy()
    y_[y==k] = 1
    y_[y!=k] = 0
    Theta = GradientDescent(X, y_)
    return Theta

Theta = np.zeros((10, n))
for k in range(10):
    Theta[k, :] = classify(k).flatten()
    print("Trained for:", k)

np.savetxt('theta.txt', Theta, delimiter=',')
```

训练得到的参数保存至 `theta.txt`. 预测时读入该矩阵，对同一个输入同时计算 10 个分类器的输出，再取最大值对应的数字：

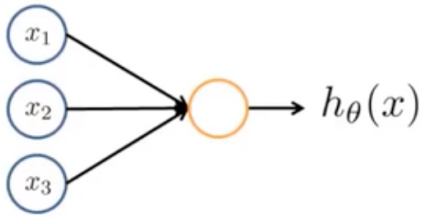
```
def predict(Theta, x):
    return np.matmul(Theta, x).argmax(axis=0)
```

训练集上各数字的准确率如下。注意：训练集准确率并不能作为模型评估结果，详见第 9 章。

数字	0	1	2	3	4	5	6	7	8	9	Total
准确率	99.0%	97.8%	88.2%	90.0%	93.4%	89.4%	97.0%	93.8%	91.4%	91.2%	93.1%

7 初识神经网络

7.1 神经元模型



神经元可以视为一个函数：它接收上一层节点的输出，将各输入按权重组合，再经过激活函数得到新的输出，即：

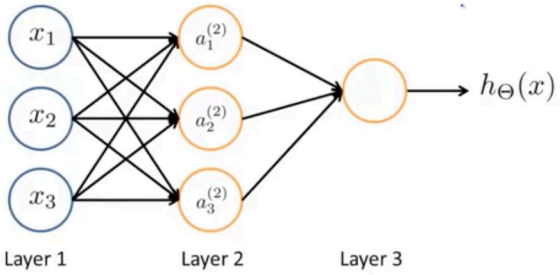
$$h_\theta(x) = f(\theta, x)$$

这里 $\theta = (\theta_0, \theta_1, \cdots, \theta_n)^T$, $x = (x_0, x_1, \cdots, x_n)^T$.

> 注意. 上图省略了 $x_0 \equiv 1$ 这一偏置项。

神经元输出 $h_\theta(x)$ 由激活函数计算。激活函数为神经网络引入非线性，使多层网络能够表示比单纯线性变换更复杂的关系。此处采用 sigmoid 函数，即 $h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$.

7.2 神经网络



神经网络由多层相互连接的神经元组成。第一层称为输入层，负责接收原始特征；最后一层称为输出层，给出模型的预测；位于二者之间的各层称为隐藏层。信息从输入层开始，逐层计算并传递到输出层，这一过程称为前向传播。

> 注意. 上图同样省略了 $x_0(a_0^{(1)}) \equiv a_0^{(2)} \equiv 1$ 偏置项。

下面单独分析第 i 层到第 $i+1$ 层的传播过程。设 s_i 为第 i 层的神经元数量（**不包含偏置项**），则第 $i+1$ 层的每个神经元都与一组独立参数对应。对于第 k 个神经元，其参数记为 $\theta_k^{(i)}$，并满足 $a_k^{(i+1)} = h_{\theta_k^{(i)}}(\hat{a}^{(i)}) = g\left({\theta_k^{(i)}}^T \hat{a}^{(i)}\right)$. 其中，$\hat{a}^{(i)} \in \mathbb{R}^{s_i+1}$ 表示在 $a^{(i)}$ 前加入偏置项后的向量。将下一层所有神经元的计算合并，即可写成矩阵形式：

$$a^{(i+1)} = g\left(\Theta^{(i)} \hat{a}^{(i)}\right)$$

其中，$\Theta^{(i)} \in \mathbb{R}^{s_{i+1} \times (s_i+1)}$.

注意. 也可以将偏置项对应的参数向量 b 单独列出，将上式写为：

$$a^{(i+1)} = g\left(\Theta^{(i)} a^{(i)} + b^{(i)}\right)$$

其中，$\Theta^{(i)} \in \mathbb{R}^{s_{i+1} \times s_i}$, $b \in \mathbb{R}^{s_{i+1}}$. 本文仍沿用前一种写法。

7.3 神经网络实现门电路

下面通过门电路示例直观说明神经网络参数如何控制输出。

考虑一个输入层有 2 个神经元（不包括偏置项）且取值 $\in \{0,1\}$、输出层有 1 个神经元且取值 $\in \{0,1\}$ 的神经网络。如果我们将其参数设置为：$\Theta^{(1)} = (-30, 20, 20)^T$，那么我们有：

输入 1	输入 2	输出
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

从表中可以看出，只有两个输入同时为 1 时，线性组合才为正且输出接近 1；其他情况下输出均接近 0. 因此，这个神经元实现了与门。

通过选择不同参数，同样可以实现或门和非门。进一步将这些基本门组合起来，多层神经网络还可以表示单个线性分类器无法直接实现的异或门、同或门等逻辑电路。这也说明隐藏层能够逐层组合较简单的特征，形成更复杂的决策规则。

7.4 代码实现

本节仍以手写数字识别为例，但暂不训练参数。参数 Θ 已在 `ex3weights.mat` 中给出，只需按照神经网络结构逐层执行前向传播。

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import matplotlib

m = 5000
n = 401

data = loadmat('ex3data1.mat')
X = data['X'] # (5000, 400)
X = np.hstack((np.ones((m, 1)), X)) # (5000, 401)
y = data['y'] # (5000, 1)

data = loadmat('ex3weights.mat')
Theta1 = data['Theta1'] # (25, 401)
Theta2 = data['Theta2'] # (10, 26)

def sigmoid(z):
```

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

def predict(x):
    """
    x: (401, )
    """
    a1 = x.reshape((401, 1))
    a2 = np.matmul(Theta1, a1)
    a2 = sigmoid(a2) # (25, 1)
    a2 = np.vstack((np.ones((1, 1)), a2)) # (26, 1)
    a3 = np.matmul(Theta2, a2)
    a3 = sigmoid(a3)
    return a3.argmax(axis=0)[0] + 1

Sum = np.zeros(11)
Hit = np.zeros(11)
Accuracy = np.zeros(11)
hit = 0
for id in range(5000):
    Sum[y[id][0]] += 1
    if predict(X[id, :].T) == y[id][0]:
        Hit[y[id][0]] += 1
        hit += 1
for i in range(1, 11):
    Accuracy[i] = Hit[i] / Sum[i]
    print(str(i)+":", str(Accuracy[i] * 100)+"%")
print("tot:", str(hit / 5000 * 100)+"%")
```

各数字及整体的准确率如下：

数字	1	2	3	4	5	6	7	8	9	0	Total
准确率	98.2%	97.0%	96.0%	96.8%	98.4%	98.6%	97.0%	98.2%	95.8%	99.2%	97.5%

注意：训练集准确率并不能作为模型评估结果。

8 神经网络之反向传播

8.1 代价函数

回顾正则化逻辑回归的代价函数：

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

第 4 节已通过极大似然法说明其来源。该函数称为**交叉熵代价函数**；它与线性回归中衡量连续预测误差的二次代价函数不同。

神经网络同样采用交叉熵代价函数。对于 K 分类问题，输出层包含 K 个分量，每个分量表示输入属于相应类别的预测结果，因此整个网络可以看作从输入到 K 个二分类输出的映射：$x \mapsto h_\Theta(x)$. 代价函数定义为各输出分量对应的**交叉熵代价之和**，并对全部训练样本取平均：

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \ln\left(h_\Theta(x^{(i)})_k\right) + (1-y_k^{(i)}) \ln\left(1-h_\Theta(x^{(i)})_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$$

其中，$h_\Theta(x^{(i)}) \in \mathbb{R}^K$ 表示第 i 个输入经过神经网络得到的输出，$h_\Theta(x^{(i)})_k$ 表示其第 k 个分量。

> 注意．这里存在符号复用：$x^{(i)}$ 和 $y^{(i)}$ 的上标 $^{(i)}$ 表示第 i 个样本，其维数分别为 $\in \mathbb{R}^{n+1}$ 和 $\in \mathbb{R}^K$；$\Theta^{(l)}$ 的上标表示第 l 层的参数矩阵，其维数为 $\in \mathbb{R}^{s_{l+1}\times(s_l+1)}$.

训练神经网络的目标是找到使代价函数 $J(\Theta)$ 最小的参数 Θ. 若使用梯度下降法，就需要计算代价函数关于每一层每一个参数的偏导数 $\frac{\partial J}{\partial \Theta_{ji}^{(l)}}$. 直接逐项求导十分繁琐，而反向传播算法能够利用链式法则，从输出层向输入层高效地完成这一计算。

8.2 反向传播算法

推导　为简化推导，先只考虑单个样本 (x, y)，从而省略样本上标和对样本的求和。对第 $l+1$ 层的第 j 个神经元定义误差：

$$\delta_j^{(l+1)} = \frac{\partial J}{\partial z_j^{(l+1)}}$$

其中，$z_j^{(l+1)} = \sum_{k=0}^{s_l} \Theta_{jk}^{(l)} a_k^{(l)}$ 表示神经元 $a_j^{(l+1)}$ 在 sigmoid 前得到的加权和，即 $a_j^{(l+1)} = g\left(z_j^{(l+1)}\right)$.

根据链式法则，有：

$$\frac{\partial J}{\partial \Theta_{ji}^{(l)}} = \frac{\partial J}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial \Theta_{ji}^{(l)}} = a_i^{(l)} \delta_j^{(l+1)}$$

写成矩阵形式：

$$\boxed{\Delta^{(l)} = \delta^{(l+1)} \cdot (a^{(l)})^T}$$

因此，一旦得到各层的 $\delta^{(l+1)}$，就能直接计算相应参数矩阵的梯度，问题转化为求解这些误差向量。

误差需要从输出层开始递推。首先计算输出层的误差 $\delta_j^{(L)}$：

$$\begin{aligned}\delta_j^{(L)} &= \frac{\partial J}{\partial z_j^{(L)}} = \frac{\partial J}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \\ &= \frac{\partial J}{\partial a_j^{(L)}} \cdot g'\left(z_j^{(L)}\right) \\ &= -\left(\frac{y_j}{a_j^{(L)}} - \frac{1-y_j}{1-a_j^{(L)}}\right) \cdot \left(a_j^{(L)}(1-a_j^{(L)})\right) \\ &= a_j^{(L)} - y_j\end{aligned}$$

所以：

$$\boxed{\delta^{(L)} = a^{(L)} - y}$$

注意. 上述推导的第二行使用了 $a^{(L)} = g(z^{(L)})$，第三行使用了 sigmoid 函数的重要性质 $g'(z) = g(z)(1-g(z))$. 此外，由 $a^{(L)} = h_\Theta(x)$，故 $J(\Theta)$ 在当前假设条件下可写为：

$$J(\Theta) = -\left[\sum_{k=1}^{K} y_k \ln(a_k^{(L)}) + (1-y_k)\ln(1-a_k^{(L)})\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$$

得到输出层误差后，再通过链式法则向前一层传播。下面递推计算第 l 层（$2 \leqslant l < L$）的 $\delta^{(l)}$：

$$\begin{aligned}\delta_j^{(l)} &= \frac{\partial J}{\partial z_j^{(l)}} \\ &= \frac{\partial J}{\partial z^{(l+1)}} \cdot \frac{\partial z^{(l+1)}}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\ &= {\delta^{(l+1)}}^T \Theta_{\bullet,j}^{(l)} \cdot g'(z_j^{(l)}) \\ &= {\delta^{(l+1)}}^T \Theta_{\bullet,j}^{(l)} \cdot \left(a_j^{(l)} * (1-a_j^{(l)})\right)\end{aligned}$$

所以：

$$\boxed{\delta^{(l)} = \left({\Theta^{(l)}}^T \delta^{(l+1)}\right) * \left(a^{(l)} * (1-a^{(l)})\right)}$$

其中，$*$ 表示两个向量对应位置逐元素相乘。

步骤总结 综合前面的推导，对于单个样本可以得到以下三个重要结果：

$$\begin{aligned}&\Delta^{(l)} = \delta^{(l+1)} \cdot (a^{(l)})^T && 1 \leqslant l < L \\ &\delta^{(L)} = a^{(L)} - y \\ &\delta^{(l)} = \left({\Theta^{(l)}}^T \delta^{(l+1)}\right) * \left(a^{(l)} * (1-a^{(l)})\right) && 2 \leqslant l < L\end{aligned}$$

对于 m 个样本，计算过程本身不变，只需累加即可。具体而言，设数据集为 $\left\{(x^{(i)}, y^{(i)}) \mid 1 \leqslant i \leqslant m\right\}$，反向传播算法的步骤如下：

1. 所有 $\Delta^{(l)}$ 置零；
2. 遍历数据集，设当前数据为 $(x^{(i)}, y^{(i)})$：
   (a) 以 $x^{(i)}$ 为输入做前向传播，得到输出 $a^{(L)}$；
   (b) 置 $\delta^{(L)} = a^{(L)} - y^{(i)}$，进行反向传播：$\delta^{(l)} = {\Theta^{(l)}}^T \delta^{(l+1)} * \left[a^{(l)}(1 - a^{(l)})\right]$，$2 \leqslant l < L$；
   (c) 更新 Δ 矩阵：$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \cdot (a^{(l)})^T$，$1 \leqslant l < L$；
3. 计算 D 矩阵：

$$D_{ij}^{(l)} := \begin{cases} \frac{1}{m}\left(\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}\right) & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$$

   由此得到梯度矩阵：$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)}$.

随后即可使用 $D_{ij}^{(l)}$ 更新每一层参数。参数 $\Theta^{(l)}$ 应初始化为 $[-\epsilon, \epsilon]$ 内的随机值，而不能全部初始化为同一个数；否则同一层的神经元会得到完全相同的梯度，并始终学习相同特征，无法打破对称性。

8.3 梯度检验

反向传播涉及多层矩阵运算、偏置项和正则化，代码很容易出现错误，其中一些错误甚至不会让最终结果立即表现出明显异常。梯度检验通过独立的数值方法近似梯度，可以用来验证反向传播计算是否正确。

具体而言，可以使用差分数值近似 $J(\theta)$ 的各偏导数：

$$\frac{\partial}{\partial \theta_k} J(\theta) \approx \frac{J(\theta_1, \cdots, \theta_k + \epsilon, \cdots, \theta_n) - J(\theta_1, \cdots, \theta_k - \epsilon, \cdots, \theta_n)}{2\epsilon}$$

将这些数值估计与反向传播得到的 $D_{ij}^{(l)}$ 对应位置逐一比较，二者应当非常接近。

> 注意. 梯度检验需要为每个参数多次计算代价函数，计算开销很大，因此仅应在调试阶段使用。确认反向传播实现正确后，应将相关代码删除或关闭，正式训练时只保留反向传播。

8.4 代码实现

本节仍使用手写数字数据集，共 5000 个样本。每个输入是由 20×20 灰度图像展开得到的 400 维向量，输出为 0 到 9 之间的整数。

第一部分 · 代价函数与前向传播 前向传播与上一节基本相同，不过这里将实现一般化，使其能够处理任意 L 层神经网络，并通过矩阵运算同时计算 m 个样本：

```
def forwardPropagation():
    a = [None] * (L+1)
    a[1] = np.hstack((np.ones((m, 1)), X))
    for l in range(2, L):
        a[l] = sigmoid(np.matmul(a[l-1], Theta[l-1].T))
        a[l] = np.hstack((np.ones((m, 1)), a[l]))
    a[L] = sigmoid(np.matmul(a[L-1], Theta[L-1].T))
    return a
```

若只需要处理单个样本，也可以使用相同的逐层计算方式：

```
def forwardPropagation(x):
    """
    x: (n, 1)
    """
    a = [None] * (L+1)
    a[1] = np.vstack((np.ones((1, 1)), x))
    for l in range(2, L):
        a[l] = sigmoid(np.matmul(Theta[l-1], a[l-1]))
        a[l] = np.vstack((np.ones((1, 1)), a[l]))
    a[L] = sigmoid(np.matmul(Theta[L-1], a[L-1]))
    return a
```

代价函数及其正则化实现如下：

```
def J(lamb):
    res = 0
    FP = forwardPropagation()
    res -= np.sum(Y * np.log(FP[L]) + (1-Y) * np.log(1-FP[L]))
    for l in range(1, L):
        res += lamb / 2 * np.sum(np.power(Theta[l][:, 1:], 2))
    res /= m
    return res
```

先使用 `ex4weights.mat` 中给定的参数 Θ 检验前向传播和代价函数实现。无正则化时，准确率为 97.52%，代价为 0.28762916516131865；有正则化时（$\lambda = 1$），准确率为 97.52%，代价为 0.3844877962428937.

第二部分 · 反向传播 反向传播算法：

```
def backPropagation(lamb):
    Delta = [None] * L # (s_{l+1}, s_l+1)
    for l in range(1, L):
        Delta[l] = np.zeros((s[l+1], s[l]+1))
    delta = [None] * (L+1) # (s_l, )
    a = forwardPropagation()
    for i in range(m):
        delta[L] = a[L][i:i+1, :].T - Y[i:i+1, :].T
        for l in range(L-1, 1, -1):
            delta[l] = (np.matmul(Theta[l].T, delta[l+1]) * (a[l][i:i+1, :].T * (1 - a[l][i:i+1, :].
                T)))[1:, :]
        for l in range(1, L):
            Delta[l] += np.matmul(delta[l+1], a[l][i:i+1, :])
    D = [None] * L # (s_{l+1}, s_l+1)
    for l in range(1, L):
        D[l] = (Delta[l] + lamb * np.hstack((np.zeros((s[l+1], 1)), Theta[l][:, 1:]))) / m
    return D
```

梯度下降：

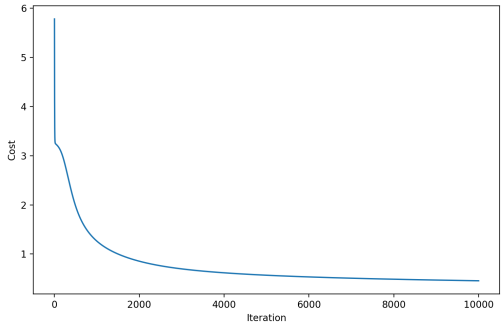
```
def GradientDescent(alpha, iteration):
    for l in range(1, L):
        Theta[l] = np.random.random((s[l+1], s[l]+1))
        Theta[l] = (Theta[l] - 0.5) / 4
    for t in range(iteration):
        D = backPropagation(lamb=1)
```

```
        for l in range(1, L):
            Theta[l] -= alpha * D[l]
        Z.append(J(lamb=1))
    return Theta
```

梯度检验（把这段代码插入到梯度下降计算出 D 矩阵之后的地方即可）：

```
for l in range(1, L):
    for i in range(s[l+1]):
        for j in range(s[l]+1):
            Theta[l][i, j] -= 0.001
            J1 = J(lamb=1)
            Theta[l][i, j] += 0.002
            J2 = J(lamb=1)
            Theta[l][i, j] -= 0.001
            print(D[l][i, j], (J2 - J1) / 0.002)
```

现在即可开始训练。代价随迭代次数增加的变化如下：

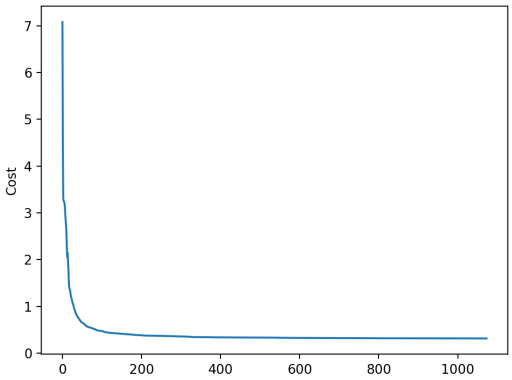


准确率如下：

数字	0	1	2	3	4	5	6	7	8	9	Total
准确率	98.6%	97.8%	93.4%	93.4%	97.0%	93.8%	97.8%	95.6%	95.8%	94.0%	95.7%

附 · 使用 `scipy` 优化 逐样本计算的梯度下降实现运行速度较慢。由于前面已经实现了代价函数及其梯度，也可以直接交给通用优化器求解。这里使用 `scipy.optimize.minimize` 最小化 $J(\Theta)$，并将 `method` 设为 CG. 结果如下：

```
     fun: 0.3139915007113895
     jac: array([-2.06059133e-05, -3.07615047e-11,  7.43614742e-10, ...,
       -2.86333807e-05,  4.03018555e-06,  5.10265929e-05])
 message: 'Optimization terminated successfully.'
    nfev: 1074
     nit: 461
    njev: 1074
  status: 0
 success: True
       x: array([-9.31407860e-01, -1.53807524e-07,  3.71577070e-06, ...,
       -1.27597236e-01, -1.69568829e-01,  3.47560466e-01])
```



数字	0	1	2	3	4	5	6	7	8	9	Total
准确率	99.8%	100.0%	99.4%	99.4%	99.2%	99.8%	99.6%	99.6%	100.0%	99.0%	99.6%

注意：训练集准确率并不能作为模型评估结果。

9 偏差与方差

9.1 训练集、验证集与测试集

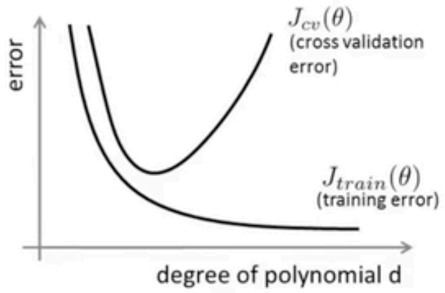
此前的实验使用全部数据训练模型，并在同一批数据上评估准确率。这种做法并不可靠：模型可能只是记住了训练样本，即使训练准确率很高，也可能已经发生过拟合，无法对新数据作出准确预测。正确的做法是保留一个不参与训练的独立测试集，只在模型建立完成后使用它进行最终评估，这样得到的结果才能反映模型的泛化能力。

如果模型包含需要人工选择的超参数，例如正则化参数 λ，仅划分训练集和测试集仍然不够。若反复根据测试结果选择表现最好的超参数，参数选择过程实际上也利用了测试集信息，测试集便不再真正独立。为此还要引入验证集：使用训练集拟合模型，使用验证集比较不同超参数，确定最终方案后再在测试集上评估。测试集自始至终都不参与模型训练和超参数选择。

如果训练过程包含正则项，计算训练集、验证集或测试集误差时应去除该项。正则化的作用是在训练阶段约束参数 θ，帮助得到复杂度更合适的模型；而评价这个参数的实际预测效果时，真正需要衡量的是预测值与真实值之间的差异，因此应使用不含正则项的原始代价函数。

9.2 高偏差与高方差

模型发生欠拟合时，通常称其具有高偏差；发生过拟合时，则称其具有高方差。以多项式回归为例，随着多项式次数增加，模型表达能力逐渐增强，并由欠拟合过渡到合适拟合，最终可能出现过拟合。在这一过程中，训练误差 $J_{\text{train}}(\theta)$ 通常持续减小，因为更复杂的模型总能更充分地拟合训练数据；验证误差 $J_{\text{valid}}(\theta)$ 则会先减小后增大，在中间某个复杂度处达到最小值，如下图所示：

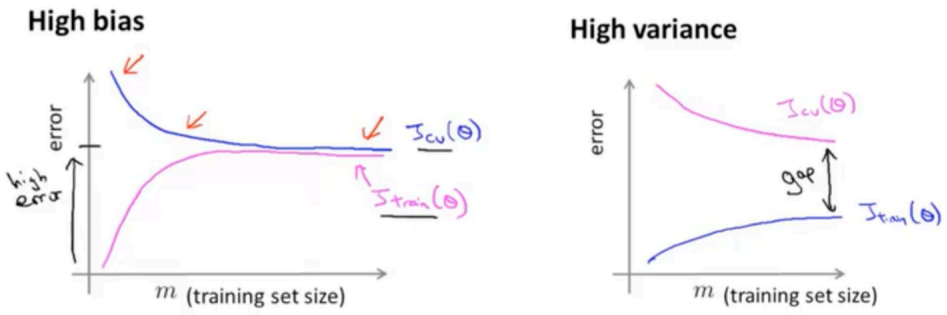


9.3 学习曲线

误差随训练集规模变化的曲线称为学习曲线。分别绘制训练误差与验证误差，可以观察增加训练数据时模型表现如何变化，从而帮助判断模型处于高偏差还是高方差状态。

对于高偏差模型，当训练集很小时，模型容易拟合少量样本，因此 $J_{\text{train}}(\theta)$ 较小，而 $J_{\text{valid}}(\theta)$ 较大。随着训练集增大，训练任务变得更困难，$J_{\text{train}}(\theta)$ 会迅速增大；$J_{\text{valid}}(\theta)$ 虽然有所减小，但幅度通常有限。最终二者逐渐接近，并共同稳定在较高水平，说明模型本身的表达能力不足。

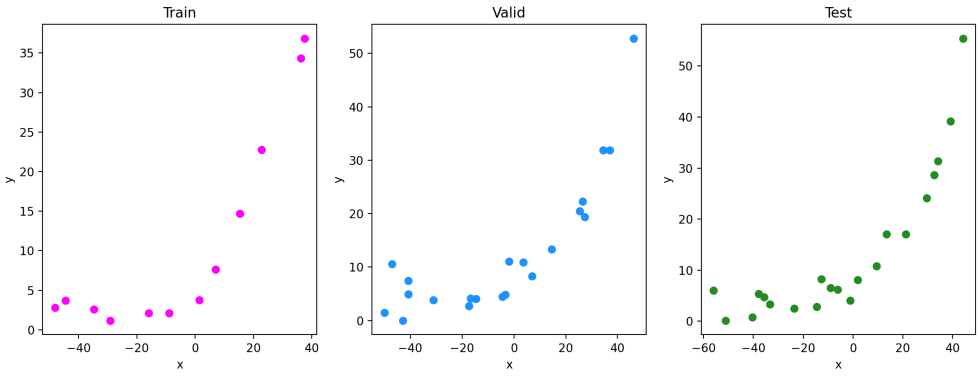
对于高方差模型，训练集很小时，模型可以近乎记住所有样本，因此 $J_{\text{train}}(\theta)$ 很小，而 $J_{\text{valid}}(\theta)$ 较大。随着训练集增大，$J_{\text{train}}(\theta)$ 会略有上升，$J_{\text{valid}}(\theta)$ 会逐渐下降；但即使训练集已经较大，训练误差仍可能明显小于验证误差，二者之间保持较大间隔。



由上述分析可知，如果模型处于高偏差状态，继续增加训练数据通常没有明显帮助，因为模型本身过于简单；更有效的方法是增加特征或提高模型复杂度。对于高方差模型，增加训练数据则可能缩小训练误差与验证误差之间的差距，从而缓解过拟合。

9.4 代码实现

正则化线性回归 首先观察数据集：



正则化线性回归的矩阵形式为：

$$J(\theta) = \frac{1}{2m}\left[\theta^T X^T X\theta - 2\theta^T X^T y + y^T y + \lambda\hat{\theta}^T\hat{\theta}\right]$$
$$\frac{\partial J}{\partial\theta} = \frac{1}{m}\left[X^T X\theta - X^T y + \lambda\hat{\theta}\right]$$

其中，$\hat{\theta}$ 表示将 θ_0 置为 0 后的 θ，因为偏置项不参与正则化。

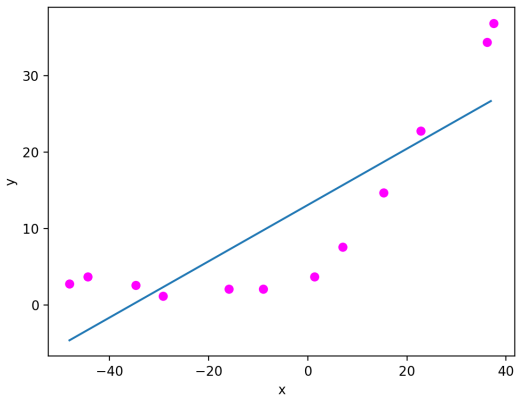
```
import numpy as np
from scipy.io import loadmat
from scipy.optimize import minimize
import matplotlib.pyplot as plt

data = loadmat('ex5data1.mat')
X, y, Xval, yval, Xtest, ytest = \
data['X'], data['y'], data['Xval'], data['yval'], data['Xtest'], data['ytest']
X = np.hstack((np.ones((X.shape[0], 1)), X))
Xval = np.hstack((np.ones((Xval.shape[0], 1)), Xval))
Xtest = np.hstack((np.ones((Xtest.shape[0], 1)), Xtest))
```

```
def unseq(theta):
    return theta.reshape(theta.shape[0], 1)

def seq(theta):
    return theta.flatten()

def J(theta, X, y, lamb):
    m = X.shape[0]
    thetahat = np.vstack((np.zeros((1, 1)), theta[1:, :]))
    return ((theta.T@X.T@X@theta-2*theta.T@X.T@y+y.T@y+lamb*thetahat.T@thetahat)/m/2)[0][0]

def partJ(theta, X, y, lamb):
    m = X.shape[0]
    thetahat = np.vstack((np.zeros((1, 1)), theta[1:, :]))
    return (X.T@X@theta-X.T@y+lamb*thetahat)/m

def Train(X, y):
    return minimize(fun = lambda theta, X, y, lamb: J(unseq(theta), X, y, lamb),
                    x0 = np.array([1, 1]),
                    jac = lambda theta, X, y, lamb: seq(partJ(unseq(theta), X, y, lamb)),
                    args = (X, y, 1),
                    method = 'CG')

res = Train(X, y)
print(res)
```

回归结果为：

```
     fun: 22.3795418229475
     jac: array([ 3.74520898e-06, -1.25949765e-07])
 message: 'Optimization terminated successfully.'
    nfev: 28
     nit: 18
    njev: 28
  status: 0
 success: True
       x: array([13.08771802,  0.36774202])
```

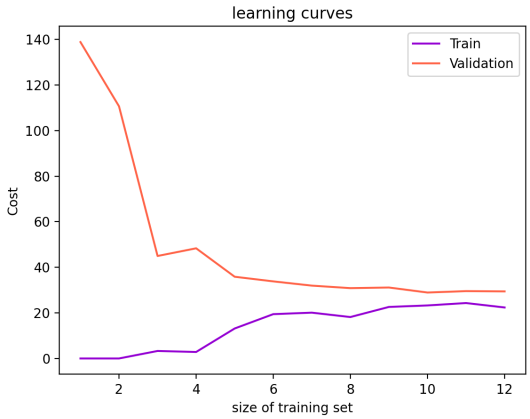
回归曲线如下：



学习曲线的绘制 绘制学习曲线时，依次增大训练集规模，每次重新训练参数，再分别计算当前训练子集的误差和验证集误差。需要注意，训练参数时可以使用正则化，但评估误差时应取 $\lambda = 0$：

```
Z_train = []
Z_valid = []
for i in range(1, 1+X.shape[0]):
    res = Train(X[:i, :], y[:i, :])
    Z_train.append(J(unseq(res.x), X[:i, :], y[:i, :], 0))
    Z_valid.append(J(unseq(res.x), Xval, yval, 0))

ax = plt.subplot(1, 1, 1)
ax.set_xlabel('size of training set')
ax.set_ylabel('Cost')
ax.set_title('learning curves')
ax.plot(range(1, 1+len(Z_train)), Z_train, color='darkviolet', label='Train')
ax.plot(range(1, 1+len(Z_valid)), Z_valid, color='tomato', label='Validation')
ax.legend()
plt.show()
```

作图如下：



图中训练误差和验证误差都较大，并且随着样本数量增加逐渐接近。这是典型的高偏差学习曲线，说明当前线性模型发生了欠拟合。

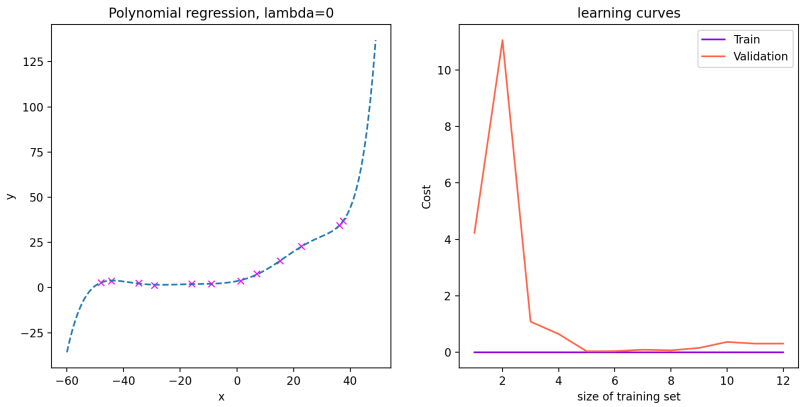
多项式回归 欠拟合的主要原因是当前使用了线性回归，而数据明显呈现非线性关系。为了提高模型的表达能力，下面引入高次特征，改用多项式回归。

加入高次特征后，特征取值范围会随着次数迅速增大，不同维度的尺度可能相差悬殊。为了保证优化过程稳定，需要对扩展后的特征进行标准化：

```
data = loadmat('ex5data1.mat')
X, y, Xval, yval, Xtest, ytest = \
data['X'], data['y'], data['Xval'], data['yval'], data['Xtest'], data['ytest']

dim = 9
meanX, stdX = [], []
meany, stdy = 0, 0

def featurePrepare(X, y):
```

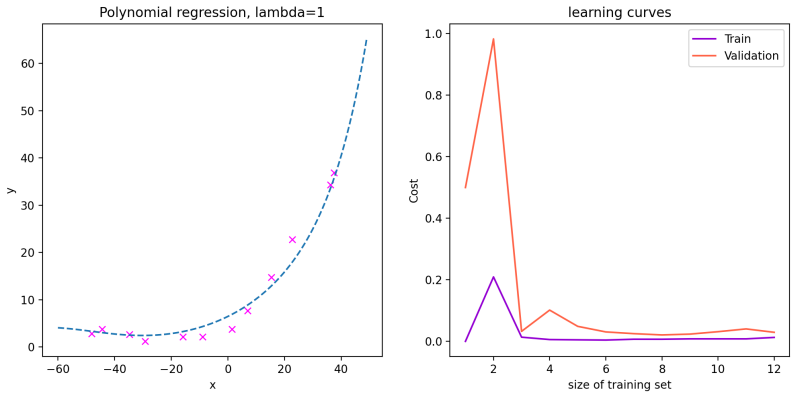
```
    """
    feature extension & normalization
    """
    global meanX, stdX, meany, stdy
    res = np.empty((X.shape[0], 0))
    for i in range(dim):
        tmpX = X ** i
        meanX.append(np.mean(tmpX, axis=0))
        stdX.append(np.std(tmpX, axis=0))
        if i:
            tmpX = (tmpX - meanX[i]) / stdX[i] if stdX[i] else tmpX - meamX[i]
        res = np.hstack((res, tmpX))
    meany = np.mean(y, axis=0)
    stdy = np.std(y, axis=0)
    y = (y - meany) / stdy if stdy else y - meany
    return res, y
```

首先使用最高次数为 8 的多项式，并取正则化参数 $\lambda = 0$. 此时模型具有较强表达能力，但完全没有正则化约束。拟合结果和学习曲线如下：

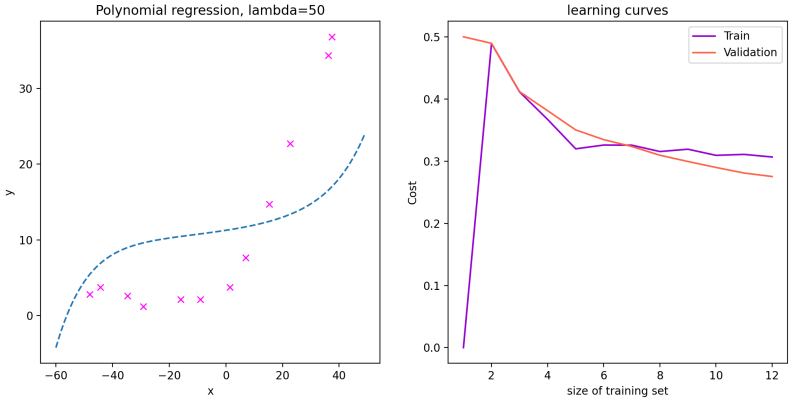


此时模型出现过拟合。

取 $\lambda = 1$，得到拟合效果和学习曲线如下：

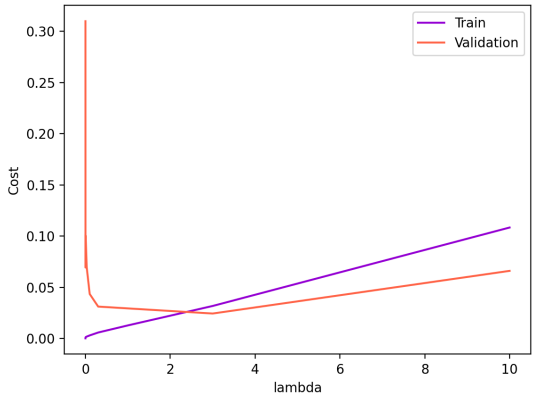


取 $\lambda = 50$，得到拟合效果和学习曲线如下：



此时模型出现欠拟合。

为了系统选择正则化参数，接下来依次使用若干 λ 训练模型，结果如下：



由图可见，较小的 λ 容易产生高方差，较大的 λ 又会带来高偏差；当 $\lambda = 3$ 时验证误差最小，因此最终选取 $\lambda = 3$. 此时测试误差为 $J_{\text{test}}(\theta) = 0.01393303991254464$.

10 支持向量机

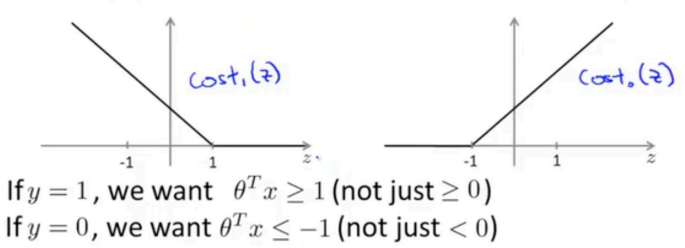
10.1 优化目标

逻辑回归的优化目标为：

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

将 $-\ln h_\theta(x^{(i)})$ 记为 $\text{cost}_1(\theta^T x^{(i)})$，它表示样本类别为 1 时的代价；将 $-\ln(1-h_\theta(x^{(i)}))$ 记为 $\text{cost}_0(\theta^T x^{(i)})$，它表示类别为 0 时的代价。支持向量机不再通过 sigmoid 函数得到平滑的对数代价，而使用如下分段线性函数近似：

$$\text{cost}_1(z)=\begin{cases}0 & z\geqslant 1\\ k_1(z-1) & z<1\end{cases}\qquad \text{cost}_0(z)=\begin{cases}0 & z\leqslant -1\\ k_0(z+1) & z>-1\end{cases}$$



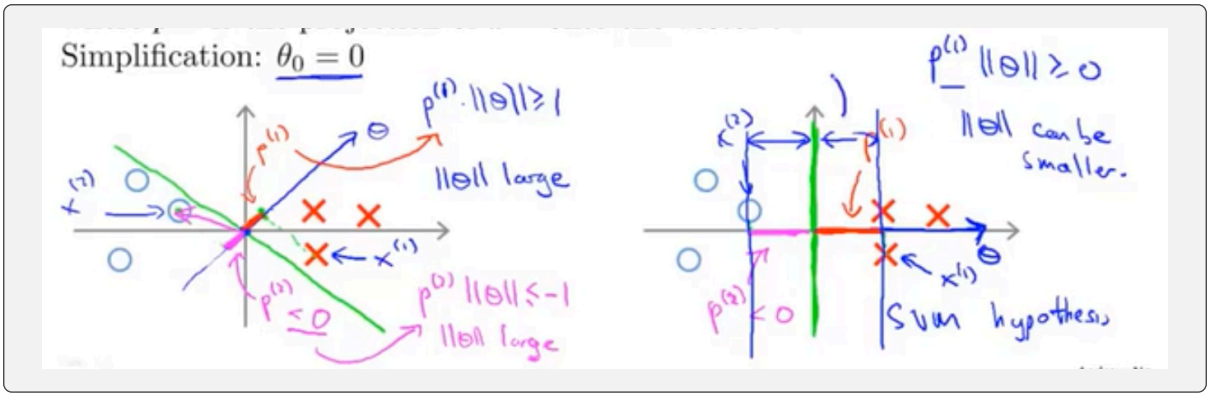
在支持向量机的常用写法中，通常省略除以样本数 m 的系数，并将正则化参数置于第一项而不是第二项。故支持向量机的优化目标为：

$$\boxed{\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}\text{cost}_1(\theta^T x^{(i)})+(1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right]+\frac{1}{2}\sum_{j=1}^{n}\theta_j^2}$$

其中，C 是正则化参数，用于权衡分类误差与参数规模，其作用与逻辑回归中的 $\frac{1}{\lambda}$ 类似：C 越大，对训练样本分类错误的惩罚越强。

注解. 先考察 $y\text{cost}_1(\theta^T x)+(1-y)\text{cost}_0(\theta^T x)$ 这一分类代价。理想情况下，希望 $\text{cost}_y(\theta^T x)=0$，这对应 $\begin{cases}\theta^T x\geqslant 1 & \text{if } y=1\\ \theta^T x\leqslant -1 & \text{if } y=0\end{cases}$. 以类别 1 为例，逻辑回归只需要比较 $\theta^T x$ 与 0 的大小便可作出分类，而支持向量机还要求 $\theta^T x$ 至少达到 1. 这意味着样本不仅要位于决策边界正确的一侧，还要与边界保持一定间隔。

从几何角度看，$\theta^T x$ 是向量 θ 与 x 的点积，可以理解为 x 在 θ 方向上的投影与 θ 长度的乘积。$||\theta||$ 表示参数向量的长度，而正则化项 $\sum_{j=1}^{n}\theta_j^2$ 等于 $||\theta||$ 的平方。减小参数长度相当于增大决策边界两侧的几何间隔，因此第二项体现了最大间隔的目标。C 较大时，模型更重视让训练样本满足分类条件，即使这需要较大的参数；C 较小时，模型允许少量样本被错误分类或落入间隔内部，以换取更大的分类间隔和更强的正则化。



10.2 核函数

对于在原特征空间中无法用直线或超平面分开的数据，可以引入核函数 $\phi : x \mapsto \phi(x)$，将原始 n 维向量映射到更高维的特征空间。原空间中的非线性边界可能对应新空间中的线性边界，因此仍可使用线性支持向量机进行分类。常用核函数包括：

线性核：保持原特征不变

$$\phi(x_i) = x_i$$

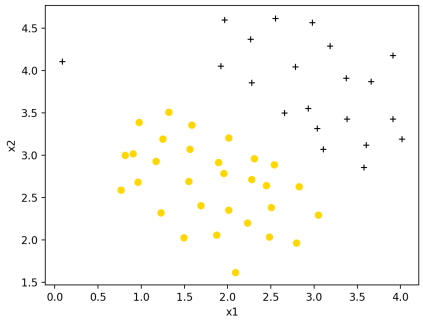
高斯核

$$\phi(x) = \begin{bmatrix} \exp\left(-||x - l^{(1)}||^2/2\sigma^2\right) \\ \exp\left(-||x - l^{(2)}||^2/2\sigma^2\right) \\ \vdots \\ \exp\left(-||x - l^{(m)}||^2/2\sigma^2\right) \end{bmatrix}$$

其中 σ 控制相似度随距离衰减的速度，$l^{(1)}, \cdots, l^{(m)}$ 称为 landmark. 每个新特征衡量输入 x 与某个 landmark 的接近程度；实践中通常直接采用训练样本 $x^{(1)}, \cdots, x^{(m)}$ 作为 landmark.

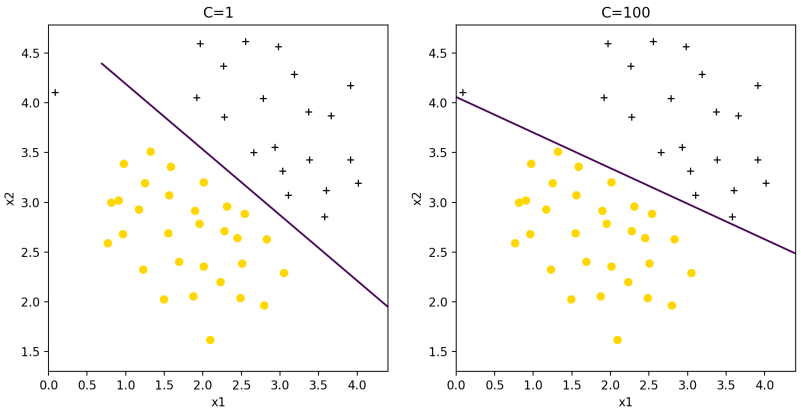
10.3 代码实现

线性可分情形 首先观察数据集：

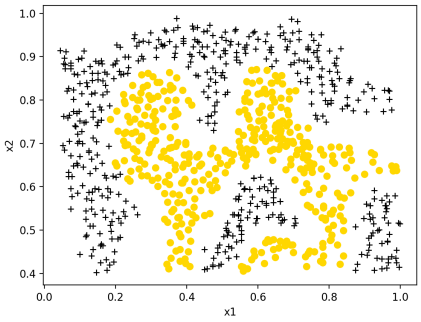



```
from sklearn import svm

clf = svm.SVC(C=1, kernel='linear')
clf.fit(X, y)
```

训练结果如下：

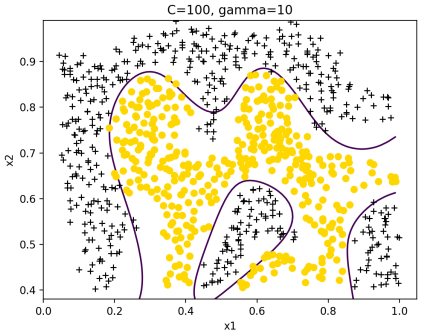


带高斯核的 SVM　首先观察数据集：

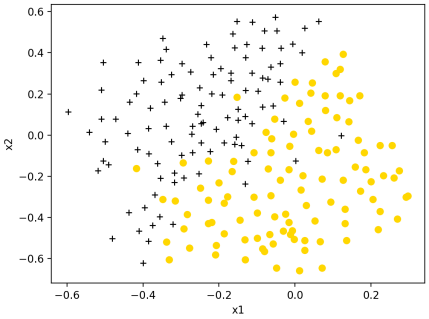



```
from sklearn import svm

clf = svm.SVC(C=100, kernel='rbf', gamma=10, probability=True)
clf.fit(X, y)
```

训练结果如下：

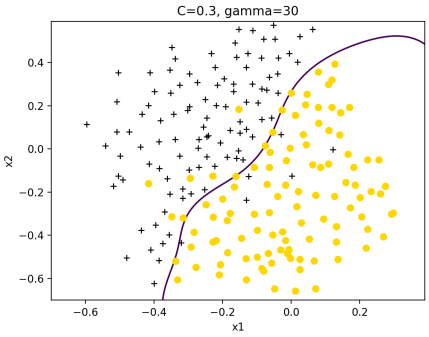


调整参数 C, σ　数据集：




```
maxscore, maxC, maxgamma = 0, 0, 0
for C in [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]:
    for gamma in [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]:
        clf = svm.SVC(C=C, kernel='rbf', gamma=gamma, probability=True)
        clf.fit(X, y)
        score = clf.score(X, y)
        print(C, gamma, score)
        if score > maxscore:
            maxscore, maxC, maxgamma = score, C, gamma

clf = svm.SVC(C=maxC, kernel='rbf', gamma=maxgamma, probability=True)
clf.fit(X, y)
```

训练结果如下：



11 K-means 聚类

11.1 聚类问题

聚类属于无监督学习。与有监督学习不同，训练数据不包含预先标注的类别标签，算法需要依据样本本身的结构或相似性，将数据自动划分为若干组。每一组称为一个簇，同一簇内的样本应当相对接近，不同簇之间则应当具有较明显的差异。

11.2 K-means 算法

K-means 是解决聚类问题的常用算法，其思想较为直观。若要将数据分为 K 类，首先初始化 K 个聚类中心，然后反复执行以下两个步骤：

1. 将每个数据点分配给距离最近的聚类中心；
2. 将每个聚类中心更新为该类所有数据点的均值。

第一步固定聚类中心并更新每个样本所属的类别，第二步固定类别划分并重新计算聚类中心。重复上述过程，直至聚类中心不再变化，或其变化幅度已经小于预设阈值。

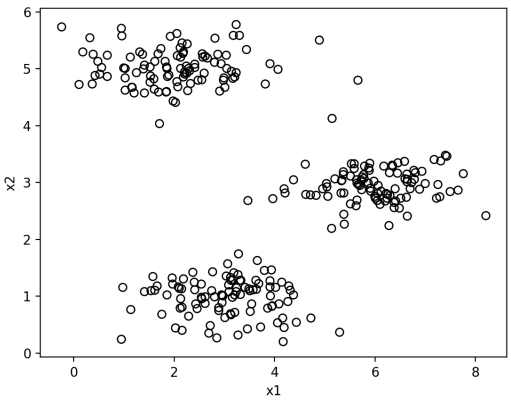
K-means 的代价函数可以定义为各数据点到所属聚类中心的距离平方和。分配类别时，每个样本都选择距离最近的中心，因此不会增大代价；更新中心时，均值又是使簇内距离平方和最小的位置，因此同样不会增大代价。由此可知，正确实现的 K-means 算法中，代价应随迭代逐步减小，最终收敛到一个稳定值。

实践中还需注意：

1. 可以从数据集中随机选取 K 个不同样本作为初始聚类中心；
2. 如果更新聚类中心时没有任何数据属于某一中心，就无法计算该簇的均值。此时可以删除该中心，使分类数减少；也可以重新随机选择一个位置，以保持分类数不变；
3. K-means 的结果依赖初始聚类中心，不同初始化可能得到不同划分，并收敛到不同的局部最优解。因此可以使用多组随机初始值重复运行算法，再选择最终代价最小的结果。

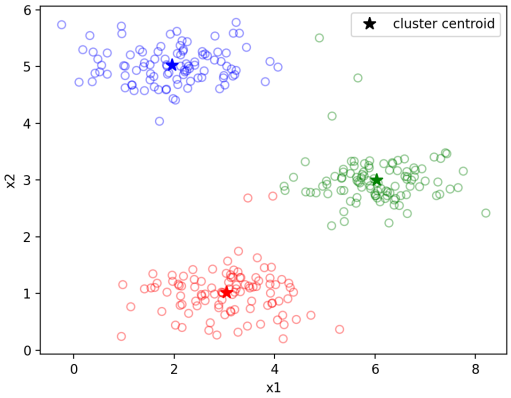
11.3 代码实现

平面点集分类 首先观察数据集：



```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt

X = loadmat('ex7data2.mat')['X']

def J(X, cluster, centroid):
    res = 0
    for i in range(X.shape[0]):
        res += np.dot(X[i]-centroid[cluster[i]], X[i]-centroid[cluster[i]])
    return res

def K_means(K, X, iteration=100):
    (m, n) = X.shape
    bestCluster, bestCentroid, bestJ = np.empty(m), np.empty((K, n)), np.inf
    for iter in range(iteration):
        centroid = X[np.random.randint(0, m, K)]
        cluster = np.empty(m, dtype='int')
        while True:
            ncentroid = np.zeros((K, n))
            cnt = np.zeros(K)
            for i in range(m):
                cluster[i] = np.argmin(np.sum((X[i]-centroid)**2, axis=1))
                ncentroid[cluster[i]] += X[i]
                cnt[cluster[i]] += 1
            ncentroid[cnt!=0] /= cnt[cnt!=0][:, np.newaxis]
            ncentroid[cnt==0] = X[np.random.randint(0, m, len(cnt[cnt==0]))]
            if (centroid == ncentroid).all():
                break
            centroid = ncentroid.copy()
        cost = J(X, cluster, centroid)
        if cost < bestJ:
            bestCluster, bestCentroid, bestJ = cluster.copy(), centroid.copy(), cost.copy()
    return bestCluster, bestCentroid

cl, ce = K_means(3, X, iteration=100)

ax = plt.subplot(1, 1, 1)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.plot(X[cl==0][:, 0], X[cl==0][:, 1], 'o', color='blue', markerfacecolor='none', alpha=0.4)
ax.plot(X[cl==1][:, 0], X[cl==1][:, 1], 'o', color='green', markerfacecolor='none', alpha=0.4)
ax.plot(X[cl==2][:, 0], X[cl==2][:, 1], 'o', color='red', markerfacecolor='none', alpha=0.4)
ax.plot(ce[0, 0], ce[0, 1], '*', color='blue', ms=10)
ax.plot(ce[1, 0], ce[1, 1], '*', color='green', ms=10)
ax.plot(ce[2, 0], ce[2, 1], '*', color='red', ms=10)
ax.plot([], [], '*', color='black', ms=10, label='cluster centroid')
ax.legend()

plt.show()
```

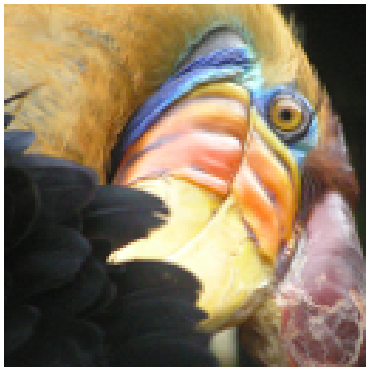
结果如下：



图像压缩 彩色图像由大量像素组成，每个像素通常使用 3 个字节分别记录 RGB 三个颜色分量。一幅图像可能包含成百上千种不同颜色；若将它们归并为 16 种代表颜色，每个像素只需用 4 位二进制数记录颜色编号。忽略颜色表本身的少量开销，像素颜色数据理论上可以压缩到原来的 $\frac{1}{6}$.

可以把每个像素的颜色视为三维空间中的一个数据点，再使用 K-means 算法将所有颜色聚为 16 类。每个聚类中心就是一种代表颜色，而原像素则替换为其所属聚类中心的颜色。

实验图像包含 128×128 个像素，可整理为形状为 $(128 \times 128, 3)$ 的二维数组。每一行表示一个像素，并包含 3 个 RGB 分量，这个数组就是 K-means 的输入数据。原图如下：



```
import numpy as np
from PIL import Image

def readin():
    data = np.array(Image.open('bird_small.png'))
    data = data.reshape((128*128, 3))
    return data

def J(X, cluster, centroid):
    res = 0
    for i in range(X.shape[0]):
        res += np.dot(X[i]-centroid[cluster[i]], X[i]-centroid[cluster[i]])
    return res

def K_means(K, X, iteration=100):
    (m, n) = X.shape
    bestCluster, bestCentroid, bestJ = np.empty(m), np.empty((K, n)), np.inf
```

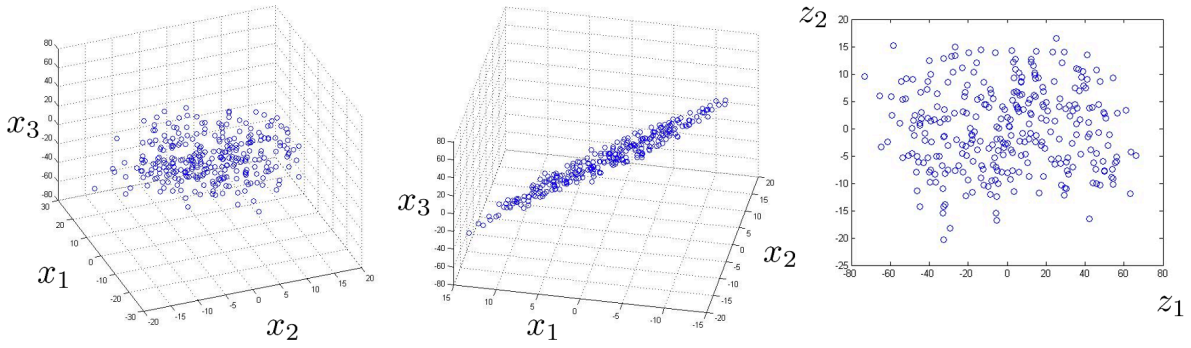
```
    for iter in range(iteration):
        centroid = X[np.random.randint(0, m, K)]
        cluster = np.empty(m, dtype='int')
        while True:
            ncentroid = np.zeros((K, n))
            cnt = np.zeros(K)
            for i in range(m):
                cluster[i] = np.argmin(np.sum((X[i]-centroid)**2, axis=1))
                ncentroid[cluster[i]] += X[i]
                cnt[cluster[i]] += 1
            ncentroid[cnt!=0] /= cnt[cnt!=0][:, np.newaxis]
            ncentroid[cnt==0] = X[np.random.randint(0, m, len(cnt[cnt==0]))]
            if (centroid == ncentroid).all():
                break
            centroid = ncentroid.copy()
        cost = J(X, cluster, centroid)
        if cost < bestJ:
            bestCluster, bestCentroid, bestJ = cluster.copy(), centroid.copy(), cost.copy()
    return bestCluster, bestCentroid

X = readin()
cl, ce = K_means(16, X, iteration=20)

comImg = np.empty((128*128, 3), dtype='uint8')
for i in range(128*128):
    comImg[i] = np.floor(ce[cl[i]])
comImg = comImg.reshape((128, 128, 3))
im = Image.fromarray(comImg)
im.save('bird_compression.png')
```

得到的压缩结果如下：

12 主成分分析

12.1 数据降维

数据往往包含较多特征，其中部分维度可能存在冗余。最极端的情形是，某个特征能够由其他若干特征的线性组合精确表示，那么它不会提供新的信息，可以直接舍去。实际数据通常不会满足如此精确的关系，但如果某个特征只是在其他特征的线性组合上叠加了很小的扰动，也可以在允许少量信息损失的前提下将其去除。这一过程称为数据降维。



主成分分析是常用的数据降维方法之一。

12.2 主成分分析

主成分分析（Principal Component Analysis，PCA）的基本思想是：假设原始数据具有 n 维特征，希望将其压缩到 k 维，就在原来的 n 维空间中寻找一个 k 维子空间，使所有样本到该子空间的距离平方和最小。这个子空间尽可能保留了数据变化最显著的方向，而每个样本在子空间上的投影就是新的 k 维表示。

数学推导 为便于推导，首先对数据进行中心化，使所有样本的平均位置位于原点。求解所需 k 维子空间的一种方式是：先在原空间中寻找一组合适的 n 维标准正交基，再选择其中前 k 个基向量张成目标子空间。对于数据点 $x = (x_1, x_2, \cdots, x_n)^T \in \mathbb{R}^n$，设这组标准正交基为 $u_1, u_2, \ldots, u_n$，则 x 在新基下第 j 个方向的坐标为：

$$y_j = x \cdot u_j = x_1 u_{j1} + x_2 u_{j2} + \cdots + x_n u_{jn}$$

于是，$y = (y_1, y_2, \cdots, y_k)^T$ 到前 k 维形成的子空间（即以 $u_1, u2, \cdots, u_k$ 为基底的子空间）的距离之平方为：

$$y_{k+1}{}^2 + y_{k+2}{}^2 + \cdots + y_n{}^2$$

对于 m 个样本 $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$，将每个样本的这部分距离相加，得到优化目标：

$$\min \sum_{i=1}^{m} {y_{k+1}^{(i)}}^2 + {y_{k+2}^{(i)}}^2 + \cdots + {y_n^{(i)}}^2$$

由于正交变换不会改变向量长度，$||x||^2$ 在不同正交基下都是定值。一个样本在前 k 个方向上的投影长度平方与其到子空间的距离平方之和等于原向量长度平方，因此最小化上述距离等价于最大化

前 k 个方向上的投影平方和：

$$\max \sum_{i=1}^{m} {y_1^{(i)}}^2 + {y_2^{(i)}}^2 + \cdots + {y_k^{(i)}}^2$$

其充分条件为：

$$\max \sum_{i=1}^{m} {y_r^{(i)}}^2, \quad r = 1, 2, \ldots, k$$

由于

$$\begin{aligned}
\sum_{i=1}^{m} {y_r^{(i)}}^2 &= \sum_{i=1}^{m} (x^{(i)} \cdot u_r)^2 \\
&= \sum_{i=1}^{m} \left(u_r^T x^{(i)}\right) \left({x^{(i)}}^T u_r\right) \\
&= u_r^T \left(\sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T \right) u_r
\end{aligned}$$

这是一个正定二次型，其中矩阵 $\sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T$ 是一个正定矩阵。对该矩阵进行特征值分解：

$$\sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T = U \Lambda U^T$$

其中，U 是正交矩阵；Λ 是对角矩阵 $\begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix}$，$\lambda_1, \ldots, \lambda_n$ 是特征值，$\lambda_1 > \cdots > \lambda_n$.

令 $v_r = U^T u_r$，由于 U 正交，所以 v_r 也是单位向量，代回得到：

$$\begin{aligned}
\sum_{i=1}^{m} {y_r^{(i)}}^2 &= u_r^T U \Lambda U^T u_r \\
&= (U^T u_r)^T \Lambda (U^T u_r) \\
&= v_r^T \Lambda v_r \\
&= \lambda_1 v_{r1}^2 + \lambda_2 v_{r2}^2 + \cdots + \lambda_n v_{rn}^2
\end{aligned}$$

故优化目标写作：

$$\max \sum_{i=1}^{n} \lambda_i v_{ri}^2 \qquad \text{s.t.} \begin{cases} \sum_{i=1}^{n} v_{ri}^2 = 1 \\ \lambda_1 > \cdots > \lambda_n \end{cases}$$

为了使加权和最大，应优先把单位向量的分量放在最大特征值对应的位置，因此其解为 $v_{r1} = 1, v_{r2} = \cdots = v_{rn} = 0$，即 $v_r = (1, 0, \cdots, 0)^T$. 对多个主成分还要满足彼此正交的约束，因此依次选择后续特征值对应的方向。又由于 $u_r = U v_r$，所需标准正交基就是矩阵 $\sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T$ 的各个特征向量。选取前 k 个特征向量作为目标子空间的基，再对原数据进行基变换，即可得到降维后的数据。

步骤总结　虽然上述推导涉及子空间、二次型与特征值分解，但主成分分析的实际实现较为直接，可以概括为以下步骤：

1. 对数据中心化后，计算矩阵 $X^TX = \sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T$；
2. 进行特征值分解，得到特征向量；
3. 选取前 k 个特征向量作为降维后的空间的基向量，构成基变换矩阵 C；
4. 对于原数据 x，取 $z = C^Tx$ 为其降维后的数据。

主成分数量的选择　实际应用中，k 过小会损失过多信息，过大又无法达到理想的压缩效果，因此需要根据重构误差选择主成分数量。定义平均重构误差：

$$\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - x_{\text{approx}}^{(i)}||^2$$

其中，$x_{\text{approx}}^{(i)}$ 表示数据 $x^{(i)}$ 在所选 k 维子空间上的投影重新映射回原空间后得到的近似。再定义数据的总方差：

$$\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)}||^2$$

用平均重构误差除以总方差，可以衡量降维丢失的信息比例。通常选择满足下式的最小 k：

$$\frac{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - x_{\text{approx}}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)}||^2} \leqslant 0.01$$

该比例不超过 0.01，表示重构误差只占总方差的 1%，因此称为保留了 99% 的方差。

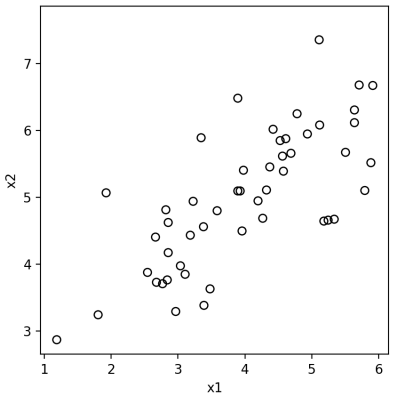
直接对每个候选 k 重构全部样本并计算误差较为繁琐。利用特征值可以更方便地计算该比例；对于给定的 k，有：

$$\frac{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - x_{\text{approx}}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)}||^2} = 1 - \frac{\sum_{i=1}^{k} \lambda_k}{\sum_{i=1}^{n} \lambda_i}$$

因此，只需考察前 k 个特征值占全部特征值之和的比例，就可以选择主成分数量。

12.3　代码实现

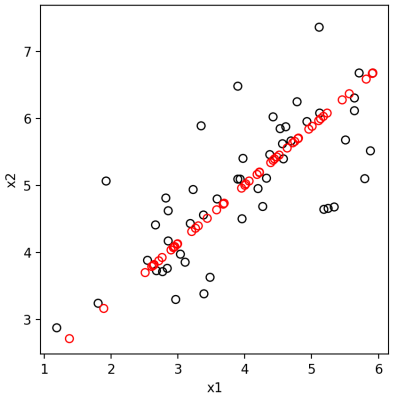
二维压缩为一维　二维平面上的原始数据：



主成分分析的代码如下：

```
def PCA(X, dim = -1):
    """
    X is the input data: (m, n)

    dim is the dimension after reduction
    if dim=-1, then the program select the smallest dim
    such that 99% of variance is retained

    return the data after reduction: (m, dim)
    and the data recovered from reduced data: (m, n)
    """
    Xmean = np.empty((1, X.shape[1]))
    Xstd = np.empty((1, X.shape[1]))
    def normalization(X, k):
        global Xmean, Xstd
        if k == 1:
            Xmean = np.mean(X, axis=0)
            Xstd = np.std(X, axis=0, ddof=1)
            return (X - Xmean) / Xstd
        else:
            return X * Xstd + Xmean

    Xnorm = normalization(X, 1)
    u, s, v = np.linalg.svd(Xnorm.T @ Xnorm)
    if dim == -1:
        dim = 1
        while s[:dim].sum() / s.sum() < 0.99:
            dim += 1
    return Xnorm @ u[:, :dim], normalization(Xnorm @ u[:, :dim] @ u[:, :dim].T, 0)
```

投影结果如下：



人脸特征压缩　数据集包含 5000 张人脸图像，每张图像由 32×32 个灰度像素组成。将像素矩阵展开后，每张人脸对应一个 1024 维特征向量。由于相邻像素和不同人脸之间存在较强相关性，这类高维数据适合使用主成分分析进行压缩。前 100 张图像如下：

分别将数据压缩至 $\dim = 36, 100$ 维，再从低维表示恢复到原来的 1024 维空间。通过重构图像可以直观比较不同主成分数量保留的信息，结果如下：

13 异常检测

13.1 多元正态分布

多元正态分布的概率密度函数为：

$$p(x;\mu,\Sigma)=\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

其中，$x\in\mathbb{R}^n$ 是 n 维随机变量，$\mu\in\mathbb{R}^n$ 是 x 的均值，$\Sigma\in\mathbb{R}^{n\times n}$ 是协方差矩阵。均值决定分布的中心位置，协方差矩阵则描述各维度的变化范围以及不同维度之间的相关关系。

特别地，当 x 的各个维度不相关、协方差矩阵为对角矩阵时，联合概率密度可以分解为各分量概率密度的乘积，即：

$$p(x;\mu,\Sigma)=\prod_{i=1}^{n}p(x_i;\mu_i,\sigma_i^2)$$

13.2 异常检测

异常检测的基本思想是先利用正常样本估计数据通常出现的区域，再将概率很低的新样本视为异常。设正常数据集为 $\{x^{(1)},x^{(2)},\dots,x^{(m)}\}$，可以使用样本均值和协方差矩阵构建多元正态分布：

$$\mu=\frac{1}{m}\sum_{i=1}^{m}x^{(i)}$$

$$\Sigma=\frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T$$

$$p(x;\mu,\Sigma)=\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

如果不希望估计完整协方差矩阵，也可以进一步假设 x 的各维度相互独立，分别拟合每一维的正态分布，再取各概率密度之积：

$$p(x;\mu,\Sigma)=\prod_{i=1}^{n}p(x_i;\mu_i,\sigma_i^2)$$

> 注意. 后者计算更快，且允许 $m<n$ 的情况；而前者在 $m<n$ 时 Σ 不可逆。

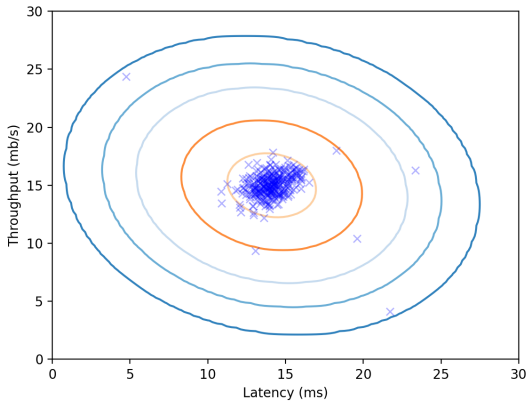
模型建立后，对于待检测样本 x，计算其在所拟合分布下的概率密度。若 $p(x;\mu,\Sigma)$ 小于阈值 ε，说明该样本在正常数据中出现的可能性很低，因此将其判定为异常；否则判定为正常。

若部分数据带有是否异常的标注，可以将数据划分为训练集、验证集和测试集。训练集主要包含正常样本，用于估计 μ 和 Σ 并建立概率模型；验证集同时包含正常与异常样本，用于比较不同阈值并选择 ε；测试集同样包含两类样本，但只用于最终评估。

> 注意. 异常检测通常属于类别极不平衡的问题，正常样本远多于异常样本。此时即使把所有样本都预测为正常，也可能得到很高的准确率，因此准确率并不是合适的主要指标。应使用 precision, recall, F1 等指标综合评估异常样本的识别效果。

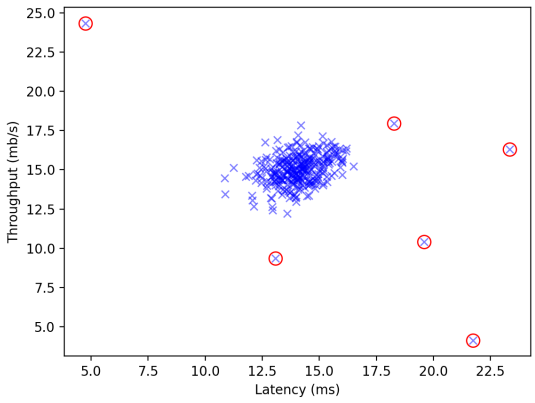
13.3 代码实现

根据训练集构建的二元正态分布如下：




```
Xmeans = X.mean(axis=0)
Xcov = ((X - Xmeans).T @ (X - Xmeans)) / X.shape[0]
normDist = multivariate_normal(mean=Xmeans, cov=Xcov)
```

依次尝试不同阈值，并根据验证集上的 F1 选择最佳值。最终得到的最佳 ε 约为 2.72×10^{-5}，此时测试集上的 F1 约为 0.75.



14 推荐系统

14.1 基于内容的推荐算法

以向用户推荐电影为例。假设已经为每部电影构建了特征向量，例如不同类型或题材所占的程度，并且已知每个用户对部分电影的评分。对于某个用户，可以将电影特征作为自变量 x、该用户的评分作为因变量 y，从而为每个用户建立一个独立的**线性回归**模型。

设共有 n_u 个用户和 n_m 部电影，第 i 部电影的特征向量为 $x^{(i)} \in \mathbb{R}^{n+1}$（包含偏置项）。$r(i,j)$ 表示用户 j 是否评价了电影 i；若已经评分，则对应分数记为 $y^{(i,j)}$. 对于用户 j，只使用他已经评价过的电影训练参数 $\theta^{(j)}$，优化目标为：

$$\min_{\theta^{(j)}} \quad \frac{1}{2} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

每个用户的线性回归在形式上相互独立，但可以把所有用户的目标函数相加，统一训练全部参数：

$$J(\theta^{(1)}, \ldots, \theta^{(n_u)}) := \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

为了使用梯度下降等方法优化，所需的偏导数为：

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) \theta_k^{(j)} + \lambda \theta_k^{(j)} [k > 0]$$

训练完成后，对于特征向量为 x 的任意电影，用户 j 的预测评分为 $(\theta^{(j)})^T x$. 可以据此对用户尚未评价的电影排序，并推荐预测评分较高的项目。

基于内容的推荐算法有一个明显限制：必须预先知道每部电影的特征向量。然而，哪些特征最能解释用户偏好往往难以确定，人工标注大量电影也需要较高成本。为避免依赖显式内容特征，可以使用协同过滤，从用户评分数据中同时学习电影和用户的表示。

14.2 协同过滤算法

初始版本 现在假设电影特征未知，但可以通过询问用户等方式获得用户参数 $\theta^{(j)}$，例如用户对不同类型电影的偏好。这样便可以反过来把用户参数作为已知量，根据评分学习电影特征 $x^{(i)}$. 对于第 i 部电影，只使用评价过该电影的用户，优化目标为：

$$\min_{\theta^{(j)}} \quad \frac{1}{2} \sum_{j:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

与前面按用户训练参数类似，各电影特征的优化也相互独立，可以将它们的目标相加后统一写为：

$$J(x^{(1)}, \ldots, x^{(n_m)}) := \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

为了优化电影特征，所需的偏导数为：

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) \theta_k^{(i)} + \lambda x_k^{(i)} [k > 0]$$

总结而言，已知 $\theta^{(j)}$ 时可以学习 $x^{(i)}$，已知 $x^{(i)}$ 时也可以学习 $\theta^{(j)}$. 因此，可以先随机初始化一组用户参数，用它们学习所有电影特征；再固定新得到的电影特征，重新学习用户参数；随后继续使用新的用户参数更新电影特征。如此在两组参数之间**交替优化**，直至电影特征和用户参数都趋于稳定，这构成了协同过滤算法的初始版本。

改进版本 实际上，没有必要把两组参数拆开并手工反复交替训练。观察“固定用户参数学习电影特征”和“固定电影特征学习用户参数”这两个优化目标，可以发现它们的非正则化项完全相同，均为 $\sum_{(i,j):r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2$. 因此，只需合并两边的正则化项，就可以把电影特征和用户参数放入同一个目标函数中同时优化：

$$J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) =$$
$$\frac{1}{2} \sum_{(i,j):r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i)}\right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

值得注意的是，在改进版本中，特征由模型自动学习。由于电影特征和用户参数可以共同吸收整体偏移，模型中**无须额外加入偏置项**，因此 $x^{(i)} \in \mathbb{R}^n, \theta^{(j)} \in \mathbb{R}^n$.

上式的偏导数为：

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) \theta_k^{(j)} + \lambda \theta_k^{(j)}$$
$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) x_k^{(i)} + \lambda x_k^{(i)}$$

得到目标函数和梯度后，可以使用梯度下降或其他算法（如 LBFGS 等）进行优化。

向量化版本 逐项遍历所有电影和用户会产生较大开销。为提高代码运行效率，需要将协同过滤算法写成矩阵形式并进行并行计算。

构造评分矩阵 $Y := \left[y^{(i,j)}\right] \in \mathbb{R}^{n_m \times n_u}$，其第 i 行第 j 列表示用户 j 对电影 i 的评分；构造电影特征矩阵 $X := \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(n_m)})^T \end{bmatrix} \in \mathbb{R}^{n_m \times n}$，其第 i 行是电影 i 的特征向量；再构造用户参数矩阵 $\Theta := \begin{bmatrix} (\theta^{(1)})^T \\ \vdots \\ (\theta^{(n_u)})^T \end{bmatrix} \in \mathbb{R}^{n_u \times n}$，其第 j 行是用户 j 的参数向量。由此，所有用户对所有电影的预测评分可以一次写成矩阵：

$$X\Theta^T \in \mathbb{R}^{n_m \times n_u}$$

利用 `numpy` 的语法可以简洁地实现代价函数及其梯度的向量化计算，详见代码。

14.3 代码实现

```
import numpy as np
from scipy.io import loadmat
from scipy.optimize import minimize

def J(Y, R, X, Theta, lamb):
    return 0.5 * (np.sum(((X @ Theta.T - Y) * R) ** 2) + \
        lamb * np.sum(X ** 2) + lamb * np.sum(Theta ** 2))

def partJ(Y, R, X, Theta, lamb):
    return np.concatenate((
        ( ((X @ Theta.T - Y) * R) @ Theta + lamb * X ).flatten(),
        ( ((X @ Theta.T - Y) * R).T @ X + lamb * Theta ).flatten()
    ))

def train(Y, R, lamb, n):
    (n_m, n_u) = Y.shape
    xt = np.empty(n*n_m+n*n_u)
    return minimize(fun = lambda xt, Y, R, lamb : J(Y, R, xt[:n*n_m].reshape((n_m, n)), \
                                                    xt[n*n_m:].reshape((n_u, n)), lamb),
                    x0 = np.random.randn(n*n_m+n*n_u),
                    args = (Y, R, lamb),
                    method = 'TNC',
                    jac = lambda xt, Y, R, lamb: partJ(Y, R, xt[:n*n_m].reshape((n_m, n)), \
                                                    xt[n*n_m:].reshape((n_u, n)), lamb)
                    )

def predict(Y, R, xt, n):
    (n_m, n_u) = Y.shape
    X, Theta = xt[:n*n_m].reshape((n_m, n)), xt[n*n_m:].reshape((n_u, n))
    return X @ Theta.T

data = loadmat('ex8_movies.mat')
Y = data['Y']
R = data['R']
Ymean = Y.mean(axis=1, keepdims=True)
res = train(Y-Ymean, R, lamb=1, n=50)
print(res)
np.save('xt.npy', res.x)
xt = res.x
```

优化结果为：

```
     fun: 11078.825074101991
     jac: array([ 4.75273039e-07, -8.33595791e-07, -4.85091646e-07, ...,
        9.67607422e-07,  3.80539749e-06,  1.86386969e-06])
 message: 'Converged (|f_n-f_(n-1)| ~= 0)'
    nfev: 14671
     nit: 463
  status: 1
 success: True
       x: array([ 0.30269431, -1.3577984 , -0.19821756, ...,  0.12718836,
       -0.40793964, -0.60753772])
```

优化得到电影特征和用户参数后，先计算完整的预测评分矩阵，再从中选取第一个用户预测评分最高的 10 部电影：

```
xt = np.load('xt.npy')
pred = predict(Y, R, xt, n=50) + Ymean
movie_list = []
with open('movie_ids.txt', encoding='latin-1') as file:
    for line in file:
        movie_list.append(' '.join(line.strip().split(' ')[1: ]))
movie_list = np.array(movie_list)
idx = np.argsort(pred[:, 0])[::-1]
print('Top 10 movies for user 1:')
for movie in movie_list[idx][:10]:
    print(movie)
```

结果为:

```
Top 10 movies for user 1:
Titanic (1997)
In the Name of the Father (1993)
Philadelphia (1993)
Duck Soup (1933)
Ice Storm, The (1997)
Saint, The (1997)
William Shakespeare's Romeo and Juliet (1996)
Boot, Das (1981)
People vs. Larry Flynt, The (1996)
Manhattan (1979)
```

15 总结与使用 scikit-learn

15.1 总结

至此，本课程的主要内容已经学习完毕。课程从线性回归出发，逐步介绍分类、神经网络、无监督学习与推荐系统等主题，使我对机器学习的基本概念、常用算法和实践方法形成了较为系统的认识。这段学习经历不仅完成了机器学习领域的入门，也为后续理解更复杂的模型奠定了基础。

课程主要内容如下：

- 监督学习（Supervised Learning）
  - 线性回归（Linear Regression，第 1,2,3 章）
  - 逻辑回归（Logistic Regression，第 4,5,6 章）
  - BP 神经网络（Neural Networks，第 7,8 章）
  - 支持向量机（Support Vector Machines，第 10 章）
- 无监督学习（Unsupervised Learning）
  - K-Means（第 11 章）
  - 主成分分析（Principal Component Analysis，第 12 章）
  - 异常检测（Anomaly Detection，第 13 章）
- 应用与专题
  - 推荐系统（Recommender Systems）与协同过滤（Collaborative Filtering，第 14 章）
  - 大规模机器学习（Large-scale Machine Learning）
- 机器学习模型的构建与评估
  - 偏差与方差（Bias/Variance，第 9 章）
  - 正则化（Regularization，第 5 章）
  - 评价指标：Precision、Recall 与 F1 Score
  - 学习曲线（Learning Curves，第 9 章）
  - 误差分析（Error Analysis）
  - 上界分析（Ceiling Analysis）

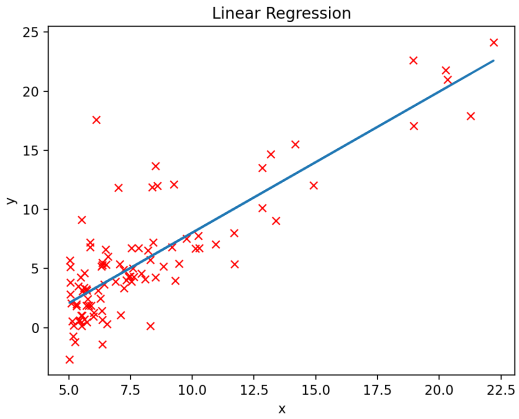
15.2 使用 scikit-learn 进行机器学习

前文代码大多从公式出发自行实现，主要目的在于理解算法原理和训练过程，因此没有特别优化运行效率、数值稳定性和易用性。掌握基本原理后，实际应用中可以使用成熟的机器学习库避免重复实现底层细节。下面使用 scikit-learn 重新完成若干典型任务。

线性回归 使用 LinearRegression 拟合一元线性回归数据，并绘制样本点与回归直线。

```
"""
class sklearn.linear_model.LinearRegression(*, fit_intercept=True,
normalize=False, copy_X=True, n_jobs=None, positive=False)
"""

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

data = np.loadtxt('data/ex1data1.txt', delimiter=',')
```

```
y = data[:, -1]
X = data[:, :-1]

reg = LinearRegression(normalize=True)
reg.fit(X, y)
print(reg.score(X, y))

ax = plt.subplot(1, 1, 1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Linear Regression')
ax.plot(X.flatten(), y, 'x', color='red')
ax.plot(X, reg.predict(X))
plt.show()
```

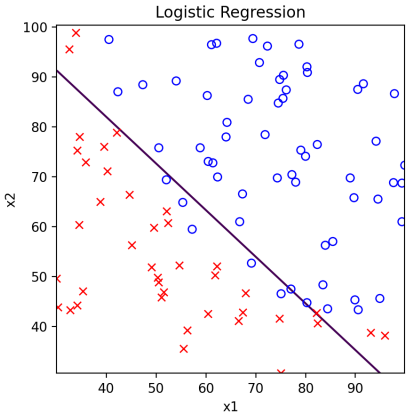



```
"""
class sklearn.linear_model.LinearRegression(*, fit_intercept=True,
normalize=False, copy_X=True, n_jobs=None, positive=False)
"""

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

data = np.loadtxt('data/ex1data2.txt', delimiter=',')
y = data[:, -1]
X = data[:, :-1]

reg = LinearRegression(normalize=True)
reg.fit(X, y)
print(reg.score(X, y))
print(reg.predict(X))
```

逻辑回归 使用 LogisticRegression 完成二分类，并分别观察线性特征与扩展特征下的决策边界。

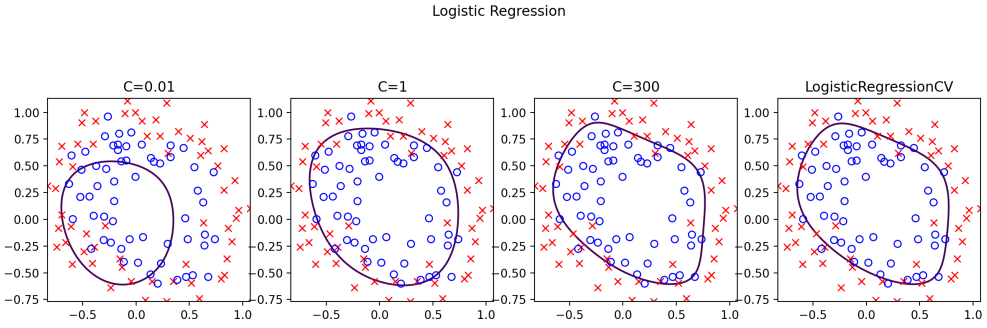
```
"""
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False,
tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1,
class_weight=None, random_state=None, solver='lbfgs', max_iter=100,
multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
```

```
"""

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

data = np.loadtxt('data/ex2data1.txt', delimiter=',')
y = data[:, -1]
X = data[:, :-1]

clf = LogisticRegression()
clf.fit(X, y)
print(clf.score(X, y))

ax = plt.subplot(1, 1, 1)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_title('Logistic Regression')
ax.plot(X[y==0][:, 0], X[y==0][:, 1], 'x', color='red')
ax.plot(X[y==1][:, 0], X[y==1][:, 1], 'o', color='blue', markerfacecolor='none')
x1, x2 = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
    np.linspace(X[:, 1].min(), X[:, 1].max(), 100))
ax.contour(x1, x2,
    clf.predict_proba(np.array([x1, x2]).\
    transpose(2, 1, 0).reshape((10000, 2)))[:, 0].reshape(100, 100), [0.5])
ax.axis('square')

plt.show()
```




```
"""
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False,
tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1,
class_weight=None, random_state=None, solver='lbfgs', max_iter=100,
multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)

class sklearn.preprocessing.PolynomialFeatures(degree=2, *,
interaction_only=False, include_bias=True, order='C')
"""

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
```

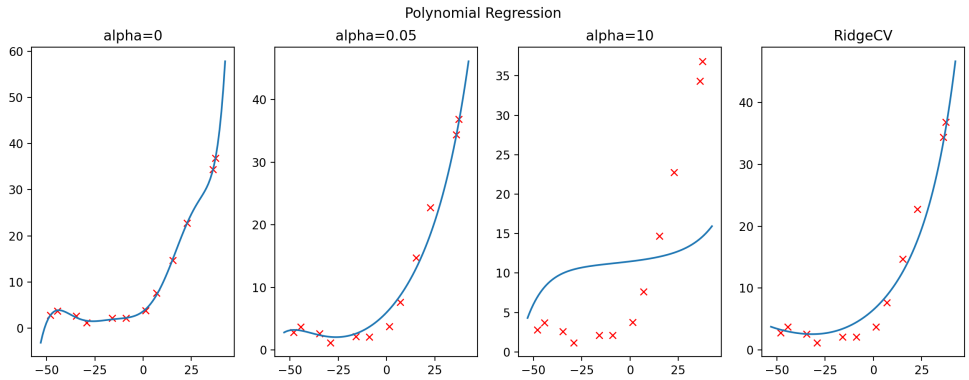
```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.preprocessing import PolynomialFeatures

data = np.loadtxt('data/ex2data2.txt', delimiter=',')
y = data[:, -1]
X = data[:, :-1]
poly = PolynomialFeatures(degree=5)
newX = poly.fit_transform(X)

fig, ax = plt.subplots(1, 4)
fig.suptitle('Logistic Regression')
Cs = [0.01, 1, 300, -1]
for i in range(4):
    C = Cs[i]
    if i < 3:
        clf = LogisticRegression(C=C, max_iter=1000)
    else:
        clf = LogisticRegressionCV(max_iter=1000)
    clf.fit(newX, y)
    print(clf.score(newX, y))

    if i < 3:
        ax[i].set_title('C={0}'.format(C))
    else:
        ax[i].set_title('LogisticRegressionCV')
    ax[i].plot(X[y==0][:, 0], X[y==0][:, 1], 'x', color='red')
    ax[i].plot(X[y==1][:, 0], X[y==1][:, 1], 'o', color='blue', markerfacecolor='none')
    x1, x2 = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
        np.linspace(X[:, 1].min(), X[:, 1].max(), 100))
    ax[i].contour(x1, x2, clf.predict_proba(
        poly.transform(np.array([x1, x2]).transpose(1, 2, 0).reshape((10000, 2)))
        )[:, 0].reshape(100, 100), [0.5])
    ax[i].axis('square')

plt.show()
```



多项式回归 先生成多项式特征，再使用线性回归拟合非线性数据，从而复现多项式回归流程。

```
"""
class sklearn.linear_model.Ridge(alpha=1.0, *, fit_intercept=True, normalize=False,
copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None)

class sklearn.preprocessing.PolynomialFeatures(degree=2, *,
interaction_only=False, include_bias=True, order='C')
```

```
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.preprocessing import PolynomialFeatures

data = loadmat('data/ex5data1.mat')
X = data['X']
y = data['y']
poly = PolynomialFeatures(degree=8)
newX = poly.fit_transform(X)

alphas = [0, 0.05, 10, -1]
fig, ax = plt.subplots(1, 4)
fig.suptitle('Polynomial Regression')
for i in range(4):
    alpha = alphas[i]
    if alpha == -1:
        reg = RidgeCV(normalize=True)
    else:
        reg = Ridge(alpha=alpha, normalize=True)
    reg.fit(newX, y)
    print(reg.score(newX, y))

    if alpha >= 0:
        ax[i].set_title('alpha={0}'.format(alpha))
    else:
        ax[i].set_title('RidgeCV')
    ax[i].plot(X.flatten(), y, 'x', color='red')
    ax[i].plot(np.linspace(X.min()-5, X.max()+5, 100), \
        reg.predict(poly.transform( np.linspace(X.min()-5, X.max()+5, 100).reshape((100, 1))) ))

plt.show()
```



BP 神经网络 使用多层感知机分类器完成前向传播与反向传播训练，无须手动实现各层梯度。

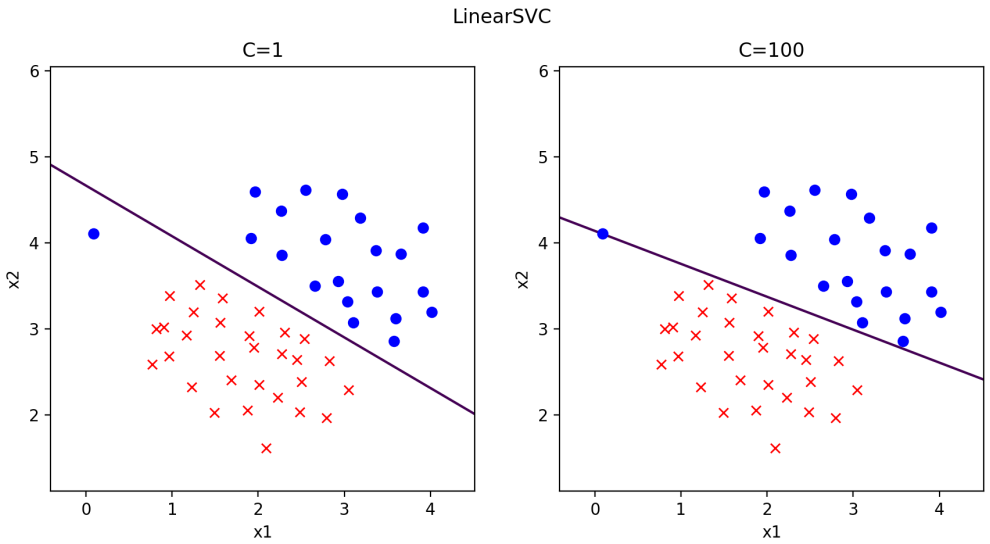
```
"""
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=100, activation='relu', *,
solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant',
learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None,
tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True,
```

```
early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
"""

import numpy as np
from scipy.io import loadmat
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

data = loadmat('data/ex4data1.mat')
X = data['X']
m = X.shape[0]
X = np.transpose(X.reshape(m, 20, 20), [0, 2, 1]).reshape(m, 400)
y = data['y']
y[y==10] = 0
y = y.flatten()
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.8)

clf = MLPClassifier(hidden_layer_sizes=(25,),
                    random_state=True,
                    max_iter=1000)
clf.fit(X_train, y_train)
print(clf.score(X_test, y_test))
```

支持向量机 分别使用线性支持向量分类器和带核函数的支持向量机，比较它们得到的决策边界。

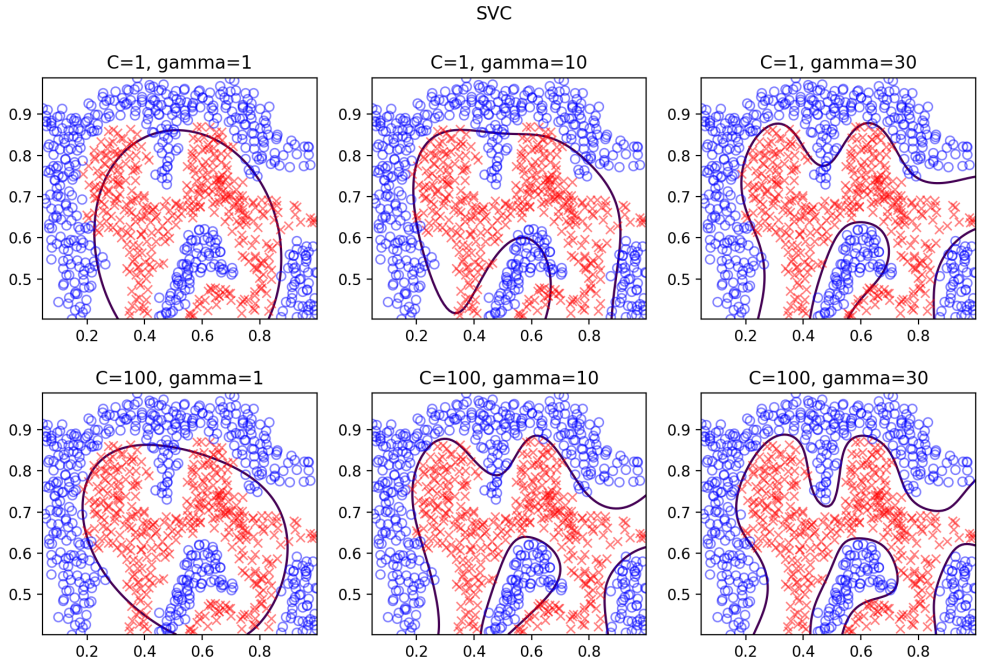
```
"""
class sklearn.svm.LinearSVC(penalty='l2', loss='squared_hinge', *, dual=True,
tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1,
class_weight=None, verbose=0, random_state=None, max_iter=1000)
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.svm import LinearSVC

data = loadmat('data/ex6data1.mat')
X = data['X']
y = data['y'].flatten()

Cs = [1, 100]
fig, ax = plt.subplots(1, 2)
fig.suptitle('LinearSVC')
x1, x2 = np.meshgrid(np.linspace(X[:, 0].min()-0.5, X[:, 0].max()+0.5, 100),
    np.linspace(X[:, 1].min()-0.5, X[:, 1].max()+0.5, 100))
for i in range(2):
    C = Cs[i]
    clf = LinearSVC(C=C)
    clf.fit(X, y)
    print(clf.predict(X[:10]))

    ax[i].plot(X[y==0][:, 0], X[y==0][:, 1], 'x', color='red')
    ax[i].plot(X[y==1][:, 0], X[y==1][:, 1], 'o', color='blue')
    ax[i].contour(x1, x2, clf.decision_function(
        np.array([x1, x2]).transpose(1, 2, 0).reshape(10000, 2)
        ).reshape(100, 100), [0])
    ax[i].set_xlabel('x1')
```

```
    ax[i].set_ylabel('x2')
    ax[i].set_title('C={0}'.format(C))
    ax[i].axis('square')

plt.show()
```



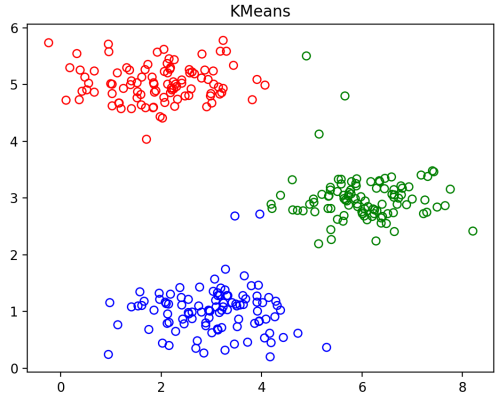

```
"""
class sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale',
coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200,
class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr',
break_ties=False, random_state=None)
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.svm import SVC

data = loadmat('data/ex6data2.mat')
X = data['X']
y = data['y'].flatten()

Cs, gammas = [1, 100], [1, 10, 30]
fig, ax = plt.subplots(2, 3)
fig.suptitle('SVC')
x1, x2 = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
    np.linspace(X[:, 1].min(), X[:, 1].max(), 100))
for i in range(2):
    for j in range(3):
        C, gamma = Cs[i], gammas[j]
        clf = SVC(C=C, kernel='rbf', probability=True, gamma=gamma)
        clf.fit(X, y)

        ax[i][j].plot(X[y==0][:, 0], X[y==0][:, 1], 'x', color='red', alpha=0.5)
        ax[i][j].plot(X[y==1][:, 0], X[y==1][:, 1], 'o', color='blue', markerfacecolor='none', alpha
            =0.5)
        ax[i][j].contour(x1, x2, clf.decision_function(
            np.array([x1, x2]).transpose(1, 2, 0).reshape(10000, 2)
            ).reshape(100, 100), [0])
```

```
        ax[i][j].set_title('C={0}, gamma={1}'.format(C, gamma))

plt.show()
```



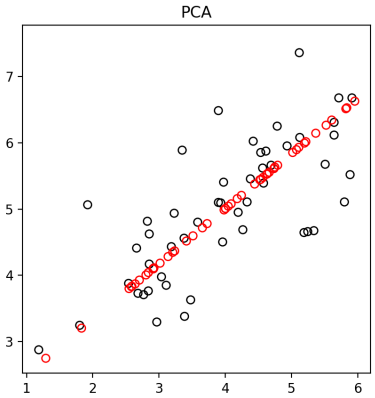
K-Means 使用 KMeans 对二维数据进行聚类，并将不同簇及其聚类中心可视化。

```
"""
class sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++', n_init=10,
max_iter=300, tol=0.0001, precompute_distances='deprecated', verbose=0,
random_state=None, copy_x=True, n_jobs='deprecated', algorithm='auto')
"""

import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

data = loadmat('data/ex7data2.mat')
X = data['X']

clt = KMeans(n_clusters=3)
belong = clt.fit_predict(X)

ax = plt.subplot(1, 1, 1)
ax.plot(X[belong==0][:, 0], X[belong==0][:, 1], 'o', color='blue', markerfacecolor='none')
ax.plot(X[belong==1][:, 0], X[belong==1][:, 1], 'o', color='red', markerfacecolor='none')
ax.plot(X[belong==2][:, 0], X[belong==2][:, 1], 'o', color='green', markerfacecolor='none')
ax.set_title('KMeans')
plt.show()
```

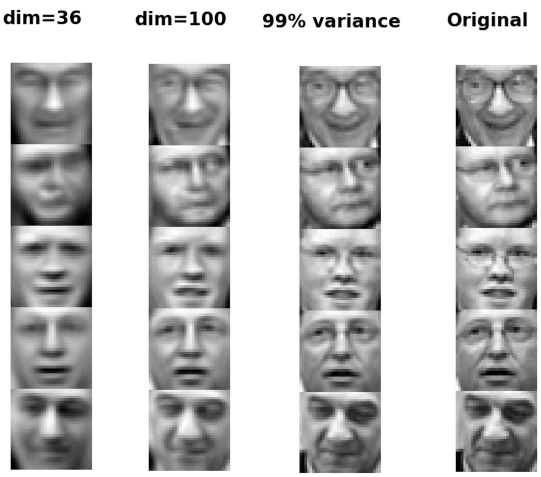


主成分分析 使用 PCA 完成数据降维，并比较低维投影与重构结果。

```
"""
class sklearn.decomposition.PCA(n_components=None, *, copy=True, whiten=False,
svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)
"""

import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

data = loadmat('data/ex7data1.mat')
X = data['X']

pca = PCA(n_components=1)
pca.fit(X)
redX = pca.transform(X) # reduced X
recX = pca.inverse_transform(redX) # recovered X

ax = plt.subplot(1, 1, 1)
ax.plot(X[:, 0], X[:, 1], 'o', color='black', markerfacecolor='none')
ax.plot(recX[:, 0], recX[:, 1], 'o', color='red', markerfacecolor='none')
ax.set_title('PCA')
ax.axis('square')
plt.show()
```



```
"""
class sklearn.decomposition.PCA(n_components=None, *, copy=True, whiten=False,
svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)
"""

import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import matplotlib
from sklearn.decomposition import PCA

X = loadmat('data/ex7faces.mat')['X']
X = np.transpose(X.reshape((5000, 32, 32)), [0, 2, 1]).reshape(5000, 1024)
X = -X

pca = PCA(n_components=0.99)
pca.fit(X)
redX = pca.transform(X) # reduced X
recX = pca.inverse_transform(redX) # recovered X

def show_a_face(face, ax):
    """
    face.shape: (1024, )
    """
    ax.matshow(face.reshape((32, 32)), cmap=matplotlib.cm.binary)
    ax.axis('off')

fig, ax = plt.subplots(5)
fig.suptitle('99% variance', fontweight='bold')
fig.subplots_adjust(hspace=0, wspace=None)
for i in range(5):
    show_a_face(recX[i], ax[i])
plt.show()
```



异常检测 使用协方差估计与异常检测接口拟合正常数据分布，并识别低概率样本。

```
"""
class sklearn.covariance.EllipticEnvelope(*, store_precision=True, assume_centered=False,
support_fraction=None, contamination=0.1, random_state=None)
```

```
sklearn.metrics.f1_score(y_true, y_pred, *, labels=None, pos_label=1,
average='binary', sample_weight=None, zero_division='warn')
"""

import numpy as np
from scipy.io import loadmat
from sklearn.covariance import EllipticEnvelope
from sklearn.metrics import f1_score
import matplotlib.pyplot as plt

data = loadmat('data/ex8data1.mat')
X, Xval, yval = data['X'], data['Xval'], data['yval'].flatten()

bestc, bestf1 = 0, 0
for c in np.linspace(0, 0.1, 100):
    ano = EllipticEnvelope(contamination=c)
    ano.fit(X)
    pred = ano.predict(X)
    pred[pred==1] = 0
    pred[pred==-1] = 1
    f1 = f1_score(yval, pred)
    if f1 > bestf1:
        bestc, bestf1 = c, f1

ano = EllipticEnvelope(contamination=bestc)
print(bestc)
ano.fit(X)
pred = ano.predict(X)

ax = plt.subplot(1, 1, 1)
ax.set_xlabel('Latency (ms)')
ax.set_ylabel('Throughput (mb/s)')
ax.plot(X[:, 0], X[:, 1], 'x', alpha=0.5, color='blue')
ax.plot(X[pred==-1][:, 0], X[pred==-1][:, 1], 'o', color='red', markerfacecolor='none', ms=10)
plt.show()
```