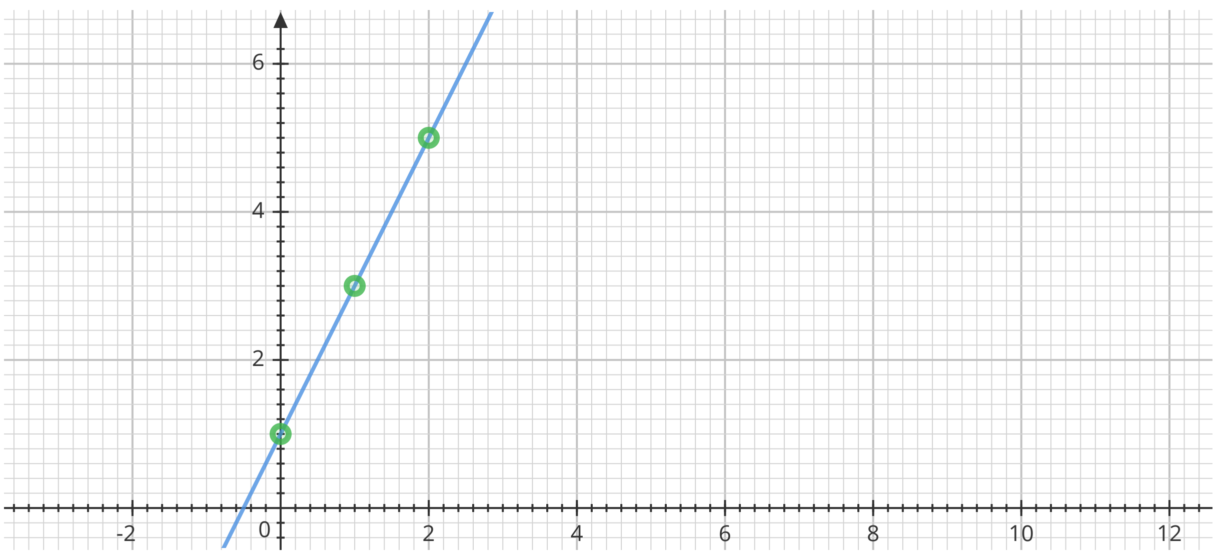
基本概念

假设我们已经测量到了一组数据：

序号	自变量x	因变量y
1	$x_1 = 0$	$y_1 = 1$
1	$x_2 = 1$	$y_2 = 3$
3	$x_3 = 2$	$y_3 = 5$
4	$x_4 = 3$	$y_4 = ?$

Table 1: 测量数据（1）

如何根据前三次的记录预测以及自变量x_4 来推算y_4 的值呢？我们可以根据测量数据做出散点图，如下：



继而假设一条直线$y = kx + b$ 表示点的变化趋势，要使直线尽可能地准确，就要求所有点到直线的距离之和$_$一个关于k,b 的二元函数$f(k,b)$ 最小，于是就能唯一地确定一组(k, b) 用以表示最终的直线。这就是最简单的线性规划。

推广一：高维形式

对于二维平面中的点我们可以用一维的线取拟合；对于三维空间中的点我们可以用二维的平面去拟合；对于n 维空间中的点，我们则可以用$n - 1$ 维的超平面去拟合。形式如下：

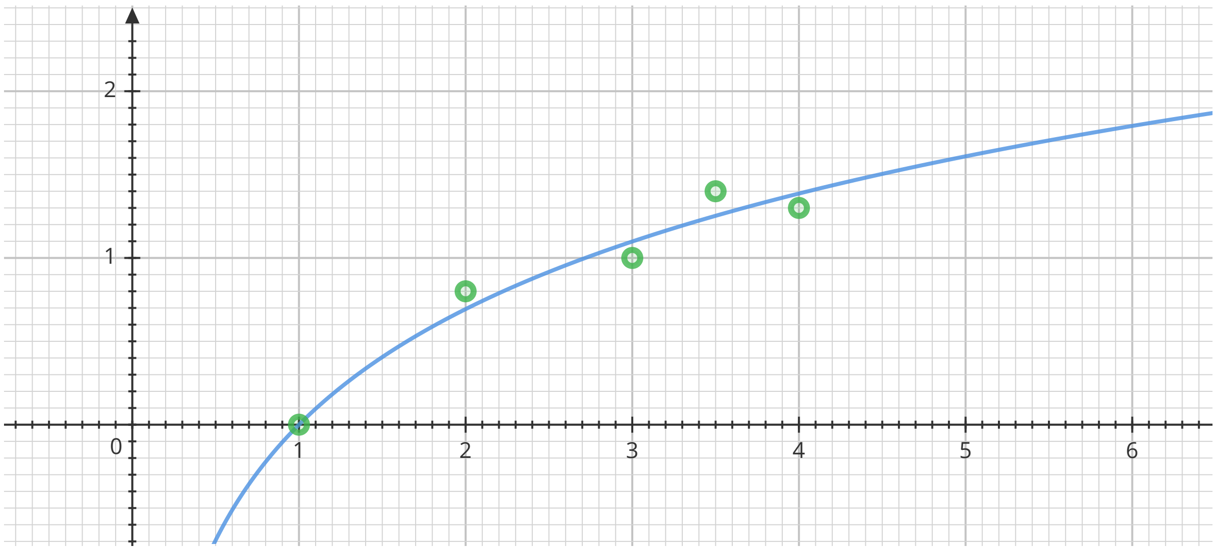
$$y = b + k_1x_1 + k_2x_2 + ... + k_nx_n \tag{1}$$

b 可以看作k_0x_0，$x_0 = 1$，所以上式也可写作：

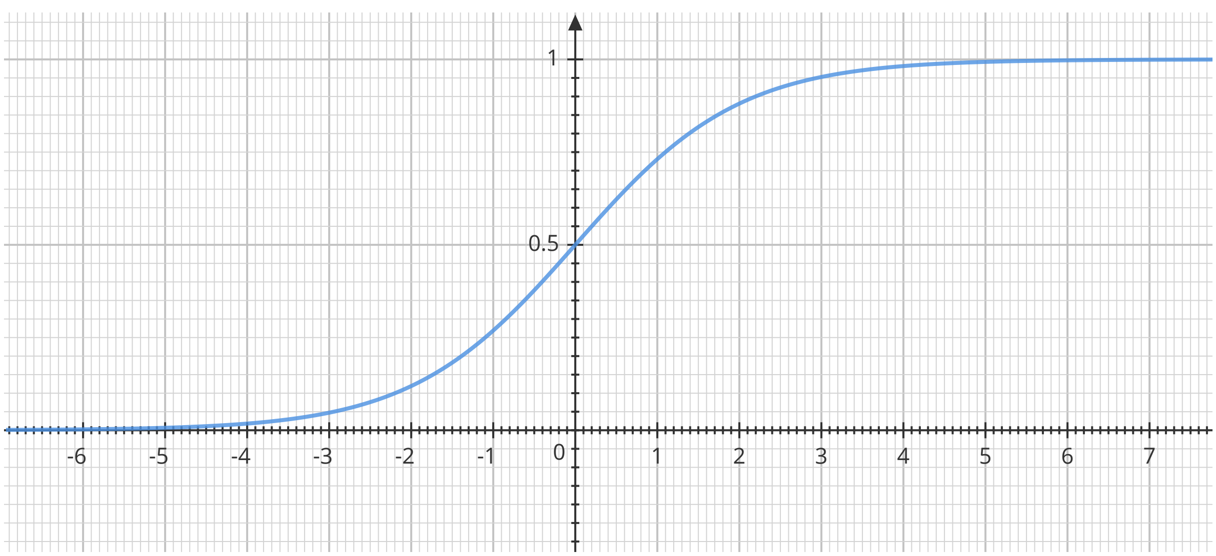
$$y = k_0x_0 + k_1x_1 + k_2x_2 + \dots + k_nx_n \tag{2}$$

推广二：非线性数据

对于非线性数据，例如：



很明显最终的曲线应该是一个对数$ln(x)$。如何去用线性的函数表示非线性的图像呢？
我们假设（构造）另一个函数$F = f(y) = e^y$，那么$F - x$ 就是线性变化的了。这里我们注意到$e^{()} \rightleftharpoons ln()$ 互为反函数。所以对于非线性的数据，我们可以根据数据变化的规律，来构造一个反函数，使其线性化。而对于实际情况来说，将线性的计算结果非线性化则更合理一些：逻辑回归中的$sigmoid()$ 函数就是其中一种，它可以将线性的结果非线性化成类似于逻辑值。



到这里，我们得到了第一个关键步骤：**【一】寻找反函数，将线性结果非线性化。**

对于已经线性化的问题来说，如果想得到一个尽可能准确的直线，那我们可以采取简单线性回归中的待定系数法确定$[k_0, k_1, \dots, k_n]$。也可以先假设一组系数$[k_0, k_1, \dots, k_n]$，根据这组系数得到一个不那么准确的结果$\hat{y}$，比较误差$\Delta(K) = Ł(y, \hat{y})$：

如果大于0，则将$[k_0, k_1, \dots, k_n]$，适当调小一点；

如果小于0，则将$[k_0, k_1, \dots, k_n]$，适当调大一点。

反复迭代之后，最终将误差控制在一个可以接受的范围之内，就算是找到了合适的拟合曲（直）线。

对于求误差函数最小值，我们可以采用梯度下降法。

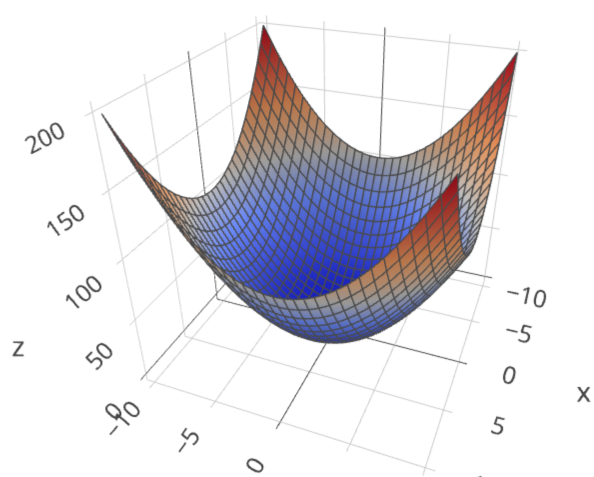
梯度下降

梯度是一个向量，而沿着梯度的方向就能最快达到误差函数的最小值：

$$\Delta(K) = f(k_1, k_2, k_3, \dots, k_n) \tag{3}$$

$$\nabla\Delta(K) = < \frac{\partial F}{\partial k_1}, \frac{\partial F}{\partial k_2}, \frac{\partial F}{\partial k_3}, \dots, \frac{\partial F}{\partial k_n} > \tag{4}$$

而梯度下降一次就是：$K^{[i+1]} = K^{[i]} - \alpha\nabla\Delta\left(K^{[i]}\right)$，其中$\alpha$ 称作学习率或者步长，其值的选择需要慎重考虑。而梯度下降能够生效的前提是误差函数是凸函数，简单来说是应该是个碗形，其实是对于任一自变量k_i，误差函数的导数都应该是单调递增的。否则会出现多个极点，造成无法收敛。



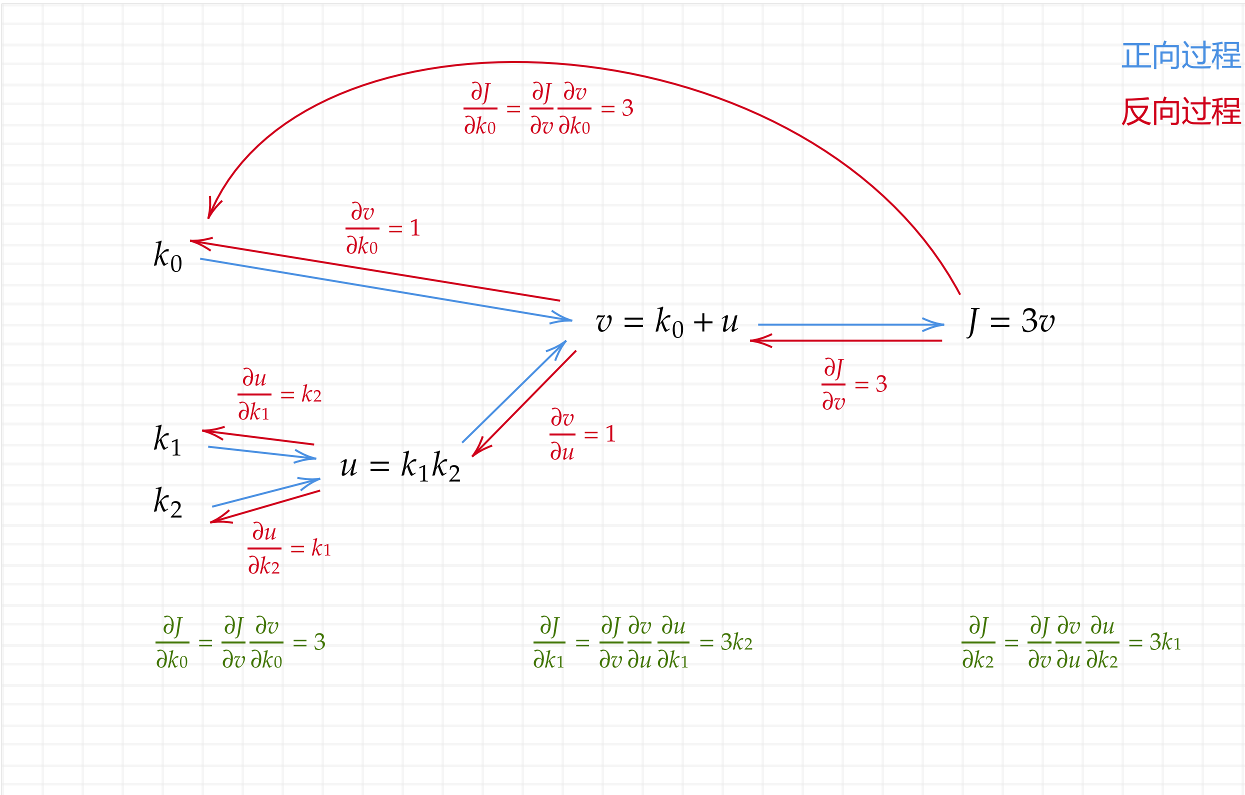
而对于有m 组测量数据时，一般取其平均值：$J(K)=\frac{1}{m}\sum_{i=0}^{m}\Delta(K_i)$，也叫成本函数。

所以另外两个个关键步骤：

【二】寻找合适的凸函数作为误差函数（成本函数）

【三】确定合适的学习率

那么应该如何确定梯度呢？这里我们可以依赖链式求导法则实现：



于是第四个关键步骤：**【四】根据链式求导法则，计算每个变量对应成本函数的偏微分**

接下来的事情就是让机器反复迭代了，以确定最合适的系数组合K。

逻辑回归的实现
例子来自于【哔哩哔哩-大话神经网络，10行代码不调包，听不懂你打我！】，讲的是去不去陪妹子看电影的故事。
假设我们有这么一组数据，A, B, C, D 四个人以前一起看电影的记录，来推测当A 下次去看电影的时候，D 会不会一起去。

A	B	C	D
0	0	1	0
1	1	1	1
1	0	1	1
0	1	1	0
1	0	0	?

获取数据
一般传感器采集到的数据会被以文本的形式保存在SD 卡中，这里以$.csv$ 文件为例：

```
'''data.csv
A,B,C,D  # title
0,0,1,0
1,1,1,1
1,0,1,1
0,1,1,0
'''

import numpy as np

def loadData():
  org_data = np.loadtxt(
    './data.csv',  # 文件名
    delimiter=',',  # 设置分隔符
    usecols=(0, 1, 2, 3),  # 需要用到的数据范围
    skiprows=1  # 跳过标题行
  )
  exp_data = orgData[:, 0:3]  # A,B,C 样本数据列
  res_data = orgData[:, 3:4]  # D 结果数据列
```

```
    exp_data = np.insert(exp_data, 3, 1, axis=1)
    # 在样本数据后附加一列常数项，值为1
    return exp_data, res_data

'''
exp_data=[
  [0. 0. 1.]
  [1. 1. 1.]
  [1. 0. 1.]
  [0. 1. 1.]
]
res_data=[
  [0.]
  [1.]
  [1.]
  [0.]
]
# 附加常数项后
exp_data=[
  [0. 0. 1. 1.]
  [1. 1. 1. 1.]
  [1. 0. 1. 1.]
  [0. 1. 1. 1.]
]
'''
```

初始化权重

给权重$w_0, w_1, w_2, \ldots, w_n$ 设定一组合适的初始值，可以有效地减少迭代的次数，更快地找到答案，这里我们设置一组随机数。

```
def initWeight(shape):  # 给定输入数据的维度n
  rows, cols = shape
  np.random.seed(1)
  weight = 2 * np.random.random((cols, 1))-1  # 生成一个nx1 的随机数矩阵，元素数值介于[-1,1]
  return weight
```

对于第i 次测量，有：

$$z_i = x_{0i}w_0 + x_{1i}w_1 + \ldots + x_{ni}w_n = np.dot(exp\_data[i],\ weight)$$

关键点一：正则化函数
上文提到，逻辑回归中我们采用$Sigmoid()$ 函数

$$a = \frac{1}{1+e^{-z}} \tag{5}$$

来将线性结果z_i 非线性化，使其尽量贴近测量数据res_data。

```
def Sigmoid(z):
  return 1/(1+np.exp(-z))
# numpy 中的“广播”特性可以很容易地将数值运算扩展到矩阵运算中
```

而$Sigmoid()$ 函数地导函数为

$$a' = a(1-a) \tag{6}$$

关键点二：误差函数
依然如上文所述，我们需要构造一个w_i 的凸函数来作为误差函数，这里我们选择：

$$L = loss(a, y) = -(ylog(a) + (1-y)log(1-a)) \tag{7}$$

注意，在$numpy$ 中$log()$ 默认是以e 为底的，也就是说：

$$L = loss(a, y) = -(y*ln(a) + (1-y)ln(1-a))$$

更改对数的底并不影响其凸函数的性质，其导函数为：

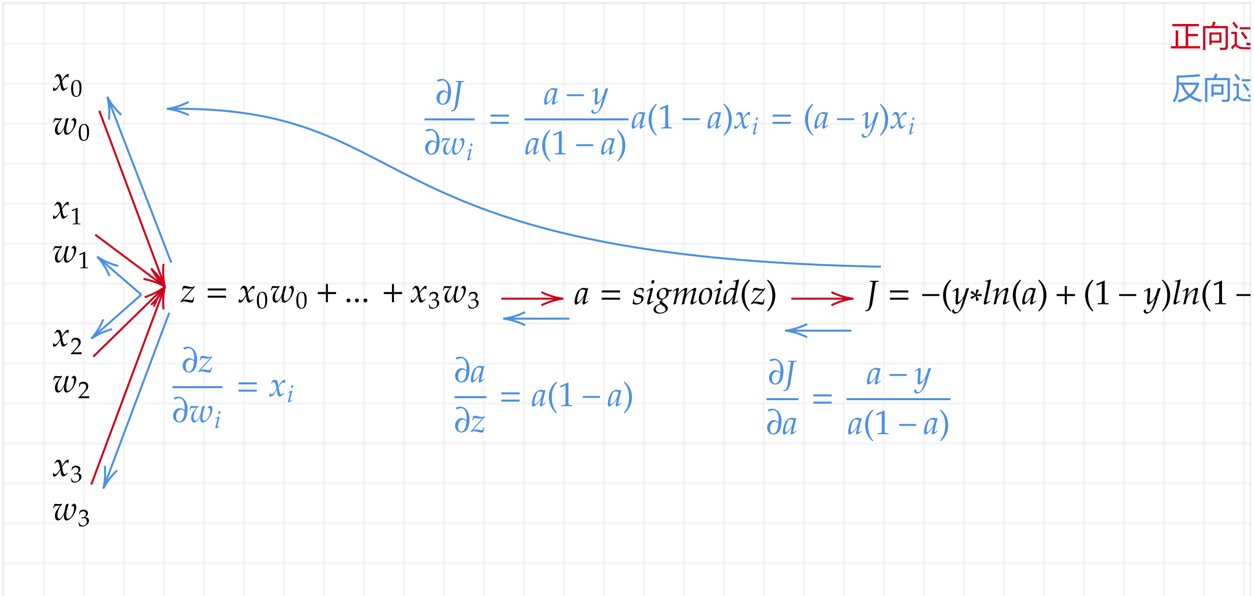
$$L' = -\left(\frac{y}{a} + \frac{-(1-y)}{(1-a)}\right) = -\frac{(1-a)y - a(1-y)}{a(1-a)} = \frac{a-y}{a(1-a)} \tag{8}$$

```
def Loss():
  return -(y*np.log(a))-(1-y)*np.log(1-a)

def Cost(loss):  # cost 去误差函数的平均值，函数在实际计算中的用处不大
  return np.mean(loss, axis=0)
```

关键点三：确定学习率α
α 一般取$0.1, 0.3, 0.01, 0.03, ...$

关键点四：计算偏微分
对于每一次的采样数据，都有：



而对于全部的采样数据有：

$$\frac{\partial J}{\partial w_i} = \frac{1}{m}\sum_{j=0}^{m}(a_j - y_j)x_i \tag{9}$$

对应的代码实现为：

```
def Gradient(exp_data, res_data, a):
  return np.dot(exp_data.T, a - res_data)/len(exp_data)
'''
# 这里需要注意一下dot 函数的作用
exp_data=[
  [0. 0. 1. 1.]
  [1. 1. 1. 1.]
  [1. 0. 1. 1.]
  [0. 1. 1. 1.]
]

exp_data.T=[
  [0. 1. 1. 0.]
  [0. 1. 0. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]
]

```

```
a=[
  [0.19859385]
  [0.24593466]
  [0.1734943 ]
  [0.27798934]
]

y=[
  [0.]
  [1.]
  [1.]
  [0.]
]

a-y=[
  [ 0.19859385]
  [-0.75406534]
  [-0.8265057 ]
  [ 0.27798934]
]

np.dot(exp_data.T, a - res_data) = [ # 刚好对应着误差函数与x_i 的乘积之和
  [-1.58057104]
  [-0.47607599]
  [-1.10398785]
  [-1.10398785]
]
'''
```

综上可得到具体的流程:

```
def main():
    alpha = 0.3
    exp_data, res_data = loadData()
    weight = initWeight(exp_data.shape)

    for i in range(1000):
```

```
            a = Sigmoid(np.dot(exp_data, weight))
            e = Loss(a, res_data)  # 没啥用
            grad = Gradient(exp_data, res_data, a)
            weight -= grad*alpha

    # 验证数据，注意补上常数项1
    print(1/(1+np.exp(-np.dot([[1, 0, 0,1]], weight))))
    pass


if __name__ == "__main__":
    main()

'''
输出结果：
[[0.99872545]]
这里不是概率，只表示去的可能性比较高
'''
```