《人工智能与Python程序设计》—— 一元线性回归

人工智能与Python程序设计 教研组

Python AI

一元线性回归

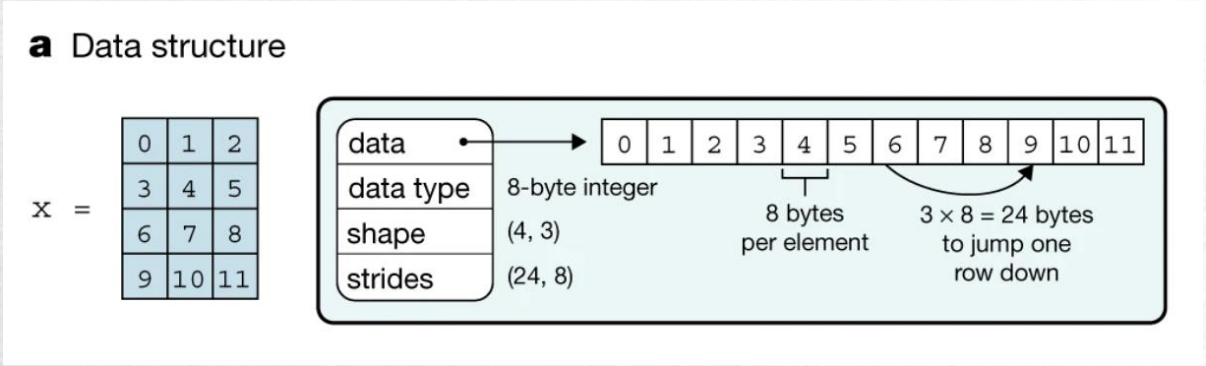
提纲

- numpy库
- 线性回归

Numpy数组

- Numpy数组（ndarray）是Numpy库中最核心的数据类型
  - 支持对多维数组（又叫张量（Tensor））的高效存储和访问
  - 其结构如下：

```
In [3]: import numpy as np
        x = np.arange(12).reshape((4,3))
        print(type(x))
        print(x)

        <class 'numpy.ndarray'>
        [[ 0  1  2]
         [ 3  4  5]
         [ 6  7  8]
         [ 9 10 11]]
```



(from Array programming with NumPy, Nature volume 585, pages357–362(2020))

  - 与python内置list不同：
    - 所有元素都是一个类型的，由data type(x.dtype)决定
    - 数据保存在一段连续的内存空间中（与C语言中数组类似）
    - 目的：节省存储空间，提升访问效率

numpy库数组属性

属性	描述
ndarray.ndim	数组轴的个数，也被称作秩
ndarray.shape	数组在每个维度上大小的整数元组
ndarray.size	数组元素的总个数
ndarray.dtype	数组元素的数据类型，dtype 类型可以用于创建数组中
ndarray.itemsize	数组中每个元素的字节大小
ndarray.data	包含实际数组元素的缓冲区地址
ndarray.flat	数组元素的迭代器

numpy库数组属性

- 数组属性

```
In [3]: x2 = np.ones((5,6))
        print(x2)
```

```
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

```
In [4]: x2.ndim
```

```
Out[4]: 2
```

```
In [5]: x2.shape
```

```
Out[5]: (5, 6)
```

```
In [6]: x2.dtype
```

```
Out[6]: dtype('float64')
```

numpy库数组形态操作方法

- 数组是ndarray类型对象，可以采用<a>.<b>()方式调用一些方法。
- 改变数组基础形态的操作方法：例如改变和调换数组维度等。
- np.flatten()函数用于数组降维，相当于平铺数组中数据，该功能在矩阵运算及图像处理中用处很大。

方法	描述
ndarray.reshape(n,m)	不改变数组 ndarray，返回一个维度为(n,m)的数组
ndarray.resize(new_shape)	与 reshape()作用相同，直接修改数组 ndarray
ndarray.swapaxes(ax1, ax2)	将数组 n 个维度中任意两个维度进行调换
ndarray. flatten()	对数组进行降维，返回一个折叠后的一维数组
ndarray.ravel()	作用同 np.flatten()，但是返回数组的一个视图

numpy库数组形态操作方法

```
In [7]: x2.flatten()

Out[7]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [11]: x3 = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], dtype=int)
         x4 = x3.reshape((3,4))
         print(x4)

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
In [12]: x3.resize((2,6))
         print(x3)

[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

ndarray 类的索引和切片方法

- 数组切片得到的是原始数组的视图，所有修改都会直接反映到源数组。如果需要得到的ndarray 切片的一 份 副 本， 需 要 进 行 复 制 操 作， 比 如arange[5:8].copy()

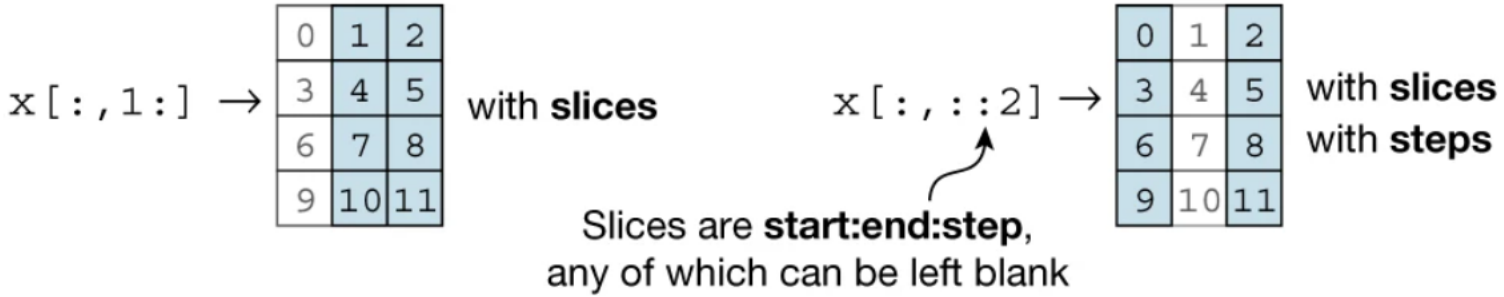
方法	描述
x[i]	索引第 i 个元素
x[-i]	从后向前索引第 i 个元素
x[n:m]	默认步长为 1，从前往后索引，不包含 m
x[-m:-n]	默认步长为 1，从后往前索引，结束位置为 n

索引Numpy数组

- Numpy数组支持比Python内置list更为丰富的索引方式
  - 索引一个元素：

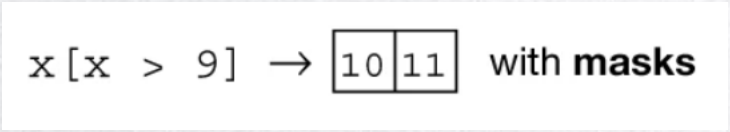
`x[1,2]` → `5` with **scalars**

  - 使用切片（slice）索引一块子区域：



索引Numpy数组

- Numpy数组支持比Python内置list更为丰富的索引方式
  - 按条件索引：

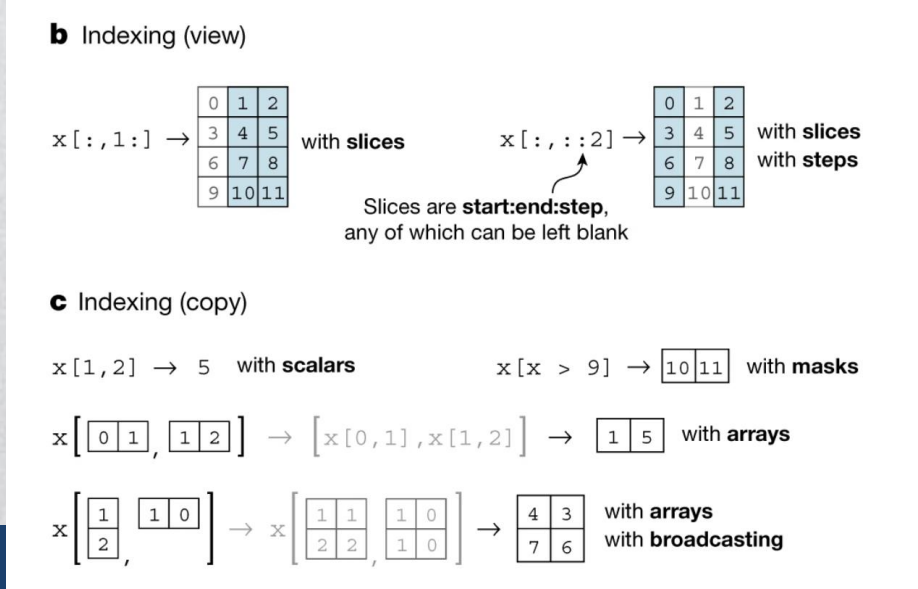


  - 用数组作为数组的索引：


```
# train / test split
shuffled_index = np.random.permutation(data_size)
x = x[shuffled_index]
y = y[shuffled_index]
```

索引Numpy数组

- 索引是否发生拷贝?
  - 设计思路：尽可能避免拷贝，以节省内存
  - 如果可能，索引操作会返回一个和原数组共享存储的视图（view）
  - 否则则拷贝数据生成一个新的Numpy数组对象




```
In [7]: x[:, ::2]

Out[7]: array([[ 0,  2],
               [ 3,  5],
               [ 6,  8],
               [ 9, 11]])

In [8]: x[:, ::2].strides

Out[8]: (24, 16)
```

ndarray 类的索引和切片方法

```
In  [13]: x = np.random.rand(6,3)
          x[2]

Out[13]: array([0.71340748, 0.37877357, 0.02473368])

In  [14]: x[2:4]

Out[14]: array([[0.71340748, 0.37877357, 0.02473368],
                [0.84463945, 0.50083296, 0.76809101]])

In  [15]: x[-5:-2:2]

Out[15]: array([[0.14797886, 0.71228022, 0.0721306 ],
                [0.84463945, 0.50083296, 0.76809101]])
```

ndarray 类的算术运算函数

- 加减乘除等算术运算函数

函数	描述
np.add(x1, x2 [, y])	y = x1 + x2
np.subtract(x1, x2 [, y])	y = x1 - x2
np.multiply(x1, x2 [, y])	y = x1 * x2
np.divide(x1, x2 [, y])	y = x1 / x2
np floor_divide(x1, x2 [, y])	y = x1 // x2，返回值取整
np.negative(x [,y])	y = -x
np.power(x1, x2 [, y])	y = x1**x2
np.remainder(x1, x2 [, y])	y = x1 % x2

ndarray 类的算术运算函数

- 这些函数中，输出参数y 可选，如果没有指定，将创建并返回一个新的数组保存计算结果；如果指定参数，则将结果保存到参数中。例如，两个数组相加可以简单地写为a+b，而np.add(a,b,a)则表示a+=b。

In [4]:
```
import numpy as np
x = np.random.rand(3,2)
y = np.ones((3,2))
np.add(x,y,x)
print(x)
```

```
[[1.68169165 1.54105006]
 [1.66019779 1.07333519]
 [1.2191459  1.74021582]]
```

In [5]:
```
z = x + y
print(z)
```

```
[[2.68169165 2.54105006]
 [2.66019779 2.07333519]
 [2.2191459  2.74021582]]
```

ndarray 类的比较运算函数

- 比较运算函数：返回一个布尔数组，包含两个数组中对应元素值的比较结果

函数	符号描述
np. equal(x1, x2 [, y])	y = x1 == x2
np. not_equal(x1, x2 [, y])	y = x1 != x2
np. less(x1, x2, [, y])	y = x1 < x2
np. less_equal(x1, x2, [, y])	y = x1 <= x2
np. greater(x1, x2, [, y])	y = x1 > x2
np. greater_equal(x1, x2, [, y])	y = x1 >= x2
np.where(condition[x,y])	根据给出的条件判断输出 x 还是 y

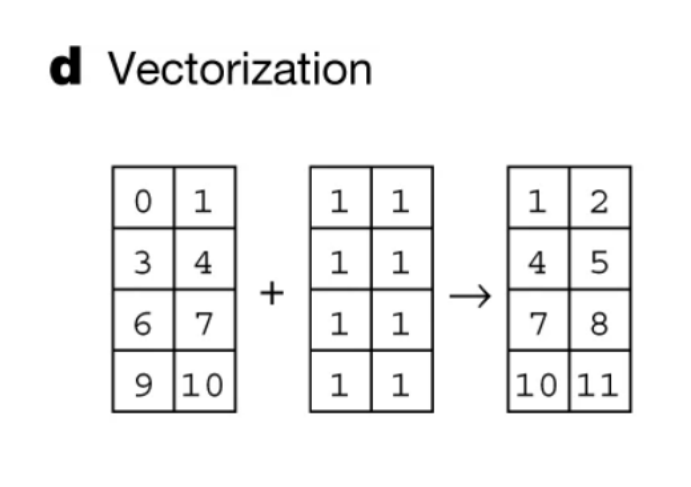
ndarray 类的比较运算函数

- where()函数是三元表达式x if condition else y 的矢量版本

```
In  [6]: np.less(z, 2.5)

Out[6]: array([[False, False],
               [False,  True],
               [ True, False]])

In  [7]: np.less(z, [[2.5, 2.5], [2.5, 2.5], [2.5, 2.5]])

Out[7]: array([[False, False],
               [False,  True],
               [ True, False]])

In  [9]: np.where(z>2.5, 0, z)

Out[9]: array([[0.        , 0.        ],
               [0.        , 2.07333519],
               [2.2191459 , 0.        ]])
```

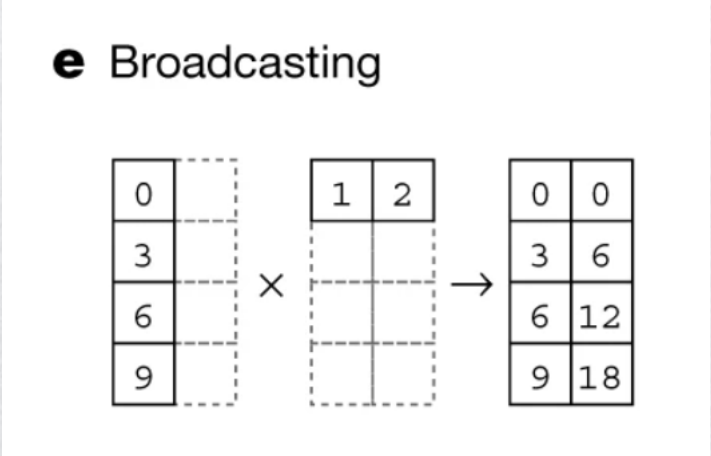
Numpy数组

- 对Numpy数组进行的操作和运算会自动的“向量化”（Vectorization）
  - 分别作用于Numpy数组中的每个元素



Numpy数组

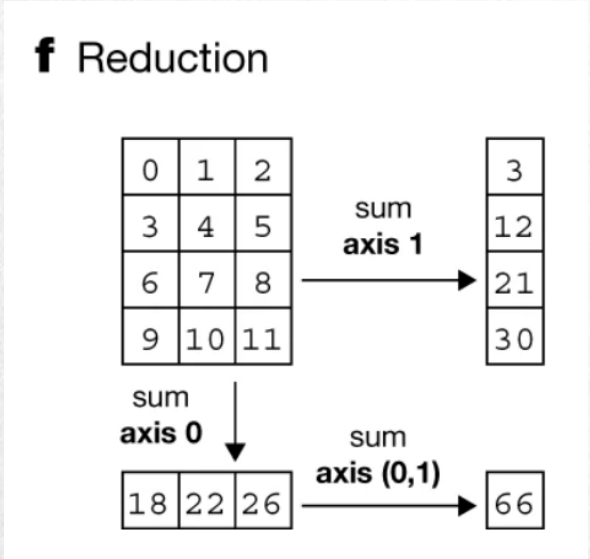
- 广播（Broadcasting)
  - 通过“复制”d次，将1维变成d维
    - 计算效率高于使用for循环依次计算
  - 以满足向量化计算的要求



Numpy数组

- 归约（Reduction）
  - 通过求和、求平均数等运算，将d维缩减为1维
    - 以求和函数np.sum为例
    - 可以通过axis可选参数决定按照哪个维度进行计算
    - 默认axis=None，将所有元素求和

```
In [16]: np.sum(x)
Out[16]: 66

In [17]: np.sum(x, axis=0)
Out[17]: array([18, 22, 26])

In [18]: np.sum(x, axis=1)
Out[18]: array([ 3, 12, 21, 30])
```



numpy库矩阵运算

- 矩阵乘法、转置、逆

```
import numpy as np
a1 = np.array([[1,2,3],[4,5,6]]) # a1为2*3矩阵
a2 = np.array([[1,2],[3,4],[5,6]]) # a2为3*2矩阵

print(a1.dot(a2))
b = np.dot(a1,a2)
print(b)

c = np.matmul(a1,a2)
print(c)
print(c.transpose())
```

```
[[22 28]
 [49 64]]
[[22 28]
 [49 64]]
[[22 28]
 [49 64]]
[[22 49]
 [28 64]]
```

```
import numpy.linalg as lg
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(lg.inv(a))

a = np.eye(3) # 3阶单位矩阵
print(lg.inv(a)) # 单位矩阵的逆为他本身

c = a.tolist()
print(c)
print(type(c))
```

```
[[ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]
 [-6.30503948e+15  1.26100790e+16 -6.30503948e+15]
 [ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]]
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
<class 'list'>
```

numpy库矩阵运算

- 乘法*

```
a1 = np.array([[1,2,3],[4,5,6]])
a2 = np.array([[3,2,1],[-1,-2,1]])
a3 = 3
print(a1*a2)
print(a3*a1)
```

```
[[  3   4   3]
 [ -4 -10   6]]
[[ 3  6  9]
 [12 15 18]]
```

numpy库其它运算函数

- numpy 库还包括三角运算函数、傅里叶变换、随机和概率分布、基本数值统计、位运算等非常丰富的功能

函数	描述
np.abs(x)	计算基于元素的整形，浮点或复数的绝对值。
np.sqrt(x)	计算每个元素的平方根
np.squre(x)	计算每个元素的平方
np.sign(x)	计算每个元素的符号：1(+)，0， -1(-)
np.ceil(x)	计算大于或等于每个元素的最小值
np.floor(x)	计算小于或等于每个元素的最大值
np.rint (x[, out])	圆整,取每个元素为最近的整数,保留数据类型
np.exp(x[, out])	计算每个元素指数值
np.log(x), np.log10(x), np.log2(x)	计算自然对数(e),基于 10,2 的对数,log(1 + x)

numpy库数据存储

- numpy能够读写磁盘上的文本数据或二进制数据
- 以二进制格式将数组保存到磁盘：np.save，默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为.npy的文件中
- np.load从磁盘读取.npy文件

```
In [13]: np.save('C:/RUC/课程/PythonAI/课程课件/testz.npy',z)
         y = np.load('C:/RUC/课程/PythonAI/课程课件/testz.npy')
         print(y)
```

```
[[2.68169165 2.54105006]
 [2.66019779 2.07333519]
 [2.2191459  2.74021582]]
```

numpy库数据存储

- numpy.savez函数可以将多个数组保存到同一个文件中：第一个参数是文件名，其后的参数都是需要保存的数组
- 可以使用关键字参数为数组起一个名字，非关键字参数传递的数组会自动起名为arr_0, arr_1, ...
- 输出的是一个压缩文件(扩展名为npz)，其中每个文件都是一个save函数保存的npy文件，文件名对应于数组名
- load函数自动识别npz文件，并且返回一个类似于字典的对象，可以通过数组名作为关键字获取数组的内容

numpy库数据存储

In [14]:
```
np.savez('C:/RUC/课程/PythonAI/课程课件/testarray.npz',x,y,res_z=z)
Res = np.load('C:/RUC/课程/PythonAI/课程课件/testarray.npz')
print(Res['arr_0'])
```

```
[[1.68169165 1.54105006]
 [1.66019779 1.07333519]
 [1.2191459  1.74021582]]
```

In [15]:
```
print(Res['arr_1'])
```

```
[[2.68169165 2.54105006]
 [2.66019779 2.07333519]
 [2.2191459  2.74021582]]
```

In [16]:
```
print(Res['res_z'])
```

```
[[2.68169165 2.54105006]
 [2.66019779 2.07333519]
 [2.2191459  2.74021582]]
```

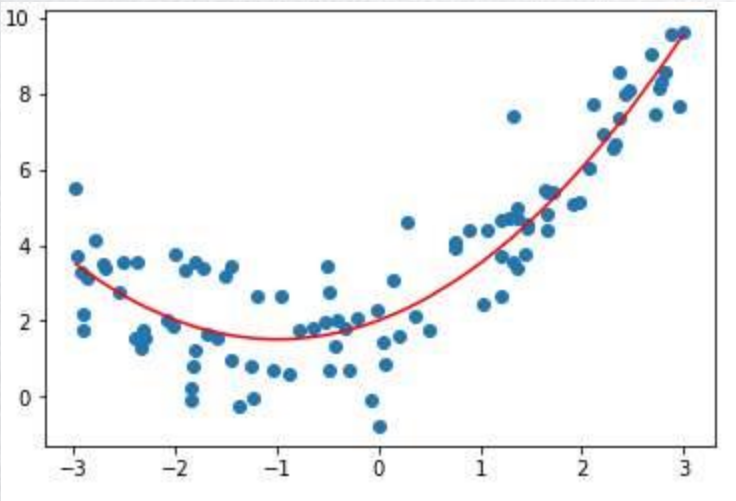
Python AI
Numpy与科学计算

提纲

- 第三方库安装
- numpy库
- **线性回归**

回归分析

- 回归分析是一种预测性的建模技术
  - 因变量（目标）和自变量（特征）之间的关系
  - 用于预测分析、时间序列模型以及发现变量之间的因果关系。
- 通常使用曲线/线来拟合数据点，目标是使数据点到曲线的距离差异最小。



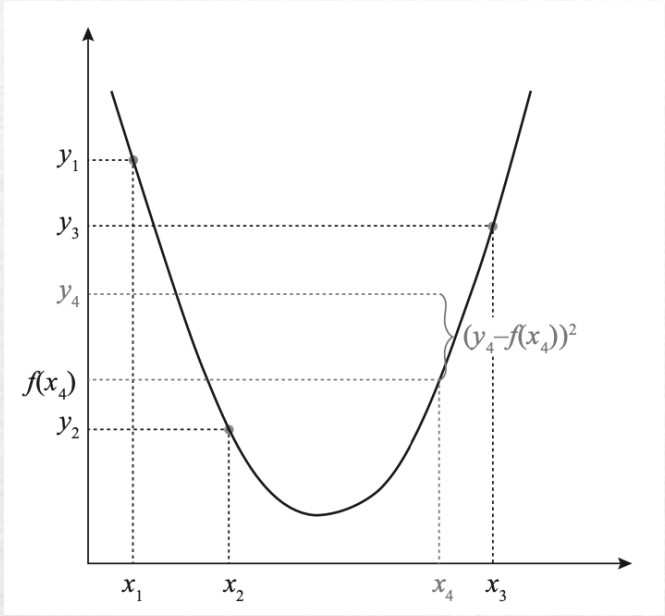
训练集/测试集

- 训练集
  - 给定样本的特征和目标值，用来训练或者拟合模型，获得模型参数。
- 测试集
  - 用来检验训练得到的模型的性能，评估模型的泛化能力。

- 给定数据集，首先需要做训练/测试集划分。

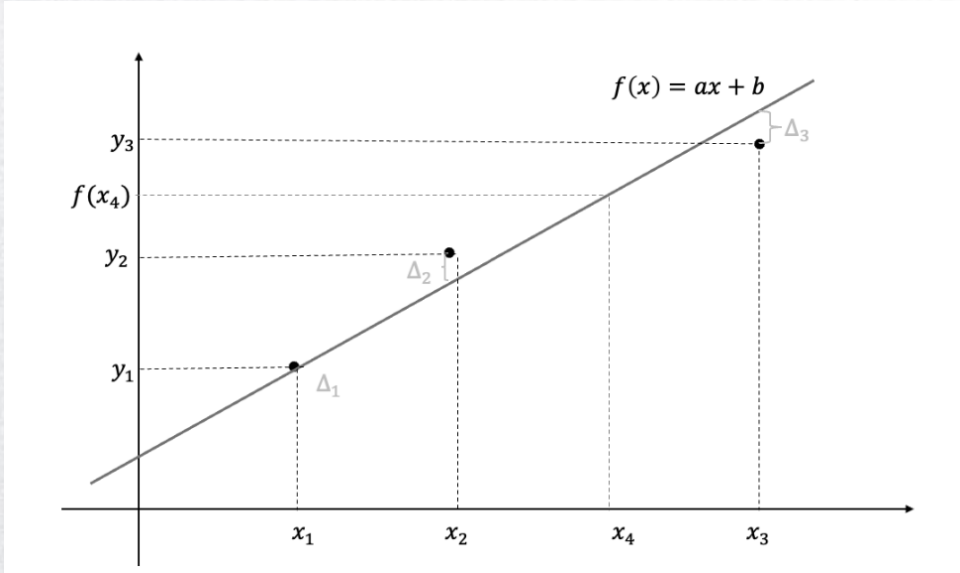
一元回归举例

- 在坐标纸上给定三个点$(x_1, y_1), (x_2, y_2), (x_3, y_3)$（训练集合），用平滑的曲线把这三个点连接起来
  - 回归模型：平滑的曲线$f(x)$
- 给定一个新的位置x_4（测试数据），在坐标纸上标出$f(x_4)$的值（对x_4点对应值进行预测）
  - 回归预测
- x_4处真实的值是y_4，因此预测误差为$\left(y_4 - f(x_4)\right)^2$
  - 回归误差



一元线性回归

- 在坐标纸上给定三个点的训练集合
  - $(x_1, y_1), (x_2, y_2), (x_3, y_3)$
- 并且规定$f(x)$为线性函数（直线）
  - $f(x) = \mathrm{w}x + b$
- 三个点，一条直线
  - 可能无法满足这条直线完美穿越所有的点（训练集合上的误差）
- 训练误差的计算
  - $\frac{1}{3}\left((f(x_1) - y_1)^2 + (f(x_2) - y_2)^2 + (f(x_3) - y_3)^2\right)$



- 模型训练要解决的问题：找到使得训练误差最小的参数组合(w, b)

损失函数

- 使用任何一组参数都可以得到一组预测值$\hat{y}$，需要一个标准来对预测结果进行度量：定量化一个目标函数式度量预测结果的好坏。
- MSE(mean square error)均方误差：线性模型应用于训练样本x_i，得到一组预测值$\hat{y}_i$，将预测值和真实值 y_i之间的平均的平方距离定义为损失函数：

$$L = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

- N为样本总数

损失函数

- 将线性预测函数式代入损失函数，将需要求解的参数w和b看做是损失函数L的自变量：

$$L(w,b)=\frac{1}{N}\sum_{i=1}^{N}(wx_i+b-y_i)^2$$

- 通过最小化损失函数求解最优参数：

$$(w^*,b^*)=\arg\min_{w,b}\frac{1}{N}\sum_{i=1}^{N}(wx_i+b-y_i)^2$$

优化

$$(w^*, b^*) = \arg\min_{w,b} \frac{1}{N} \sum_{i=1}^{N} (wx_i + b - y_i)^2$$

- 对w求导:

$$\frac{\partial L(w,b)}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} 2(wx_i + b - y_i)x_i$$

优化

$$(w^*, b^*) = \arg\min_{w,b} \frac{1}{N} \sum_{i=1}^{N} (wx_i + b - y_i)^2$$

- 对b求导:

$$\frac{\partial L(w,b)}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} 2(wx_i + b - y_i)$$

优化：梯度下降法

- 1. 随机选择一组初始参数(w^0, b^0)
- 2. 对于可微函数，梯度方向时函数增长速度最快的方向，梯度的反方向是函数减少最快的方向。计算损失函数在当前参数点处的梯度。
- 3. 从当前参数点向梯度相反的方向移动，移动步长为:

$$w^{t+1} = w^t - lr\frac{\partial L(w,b)}{\partial w}(w^t)$$

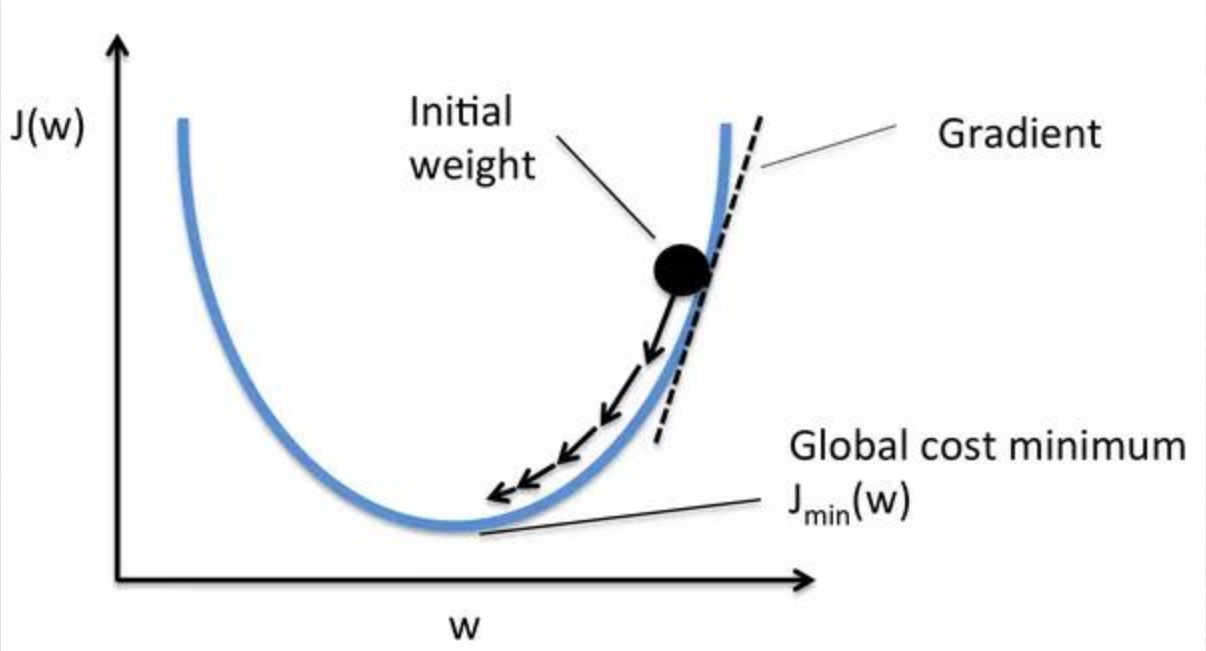
$$b^{t+1} = b^t - lr\frac{\partial L(w,b)}{\partial b}(b^t)$$

学习率

- 4. 循环迭代步骤 3， 直到前后两次迭代得到的(w^t, b^t)差值足够小，即参数基本不再变化，说明此时损失函数已经达到局部最小值。
- 5. 输出(w^t, b^t)，即为使得损失函数最小时的参数取值 。

梯度下降法

- 一元线性回归



示例：梯度下降法

```
class LinearRegression(object):

    def __init__(self, learning_rate=0.01, max_iter=100, seed=None):
        '''
        一元线性回归类的构造函数：
        参数 学习率：learning_rate
        参数 最大迭代次数：max_iter
        参数 seed：产生随机数的种子
        从正态分布中采样w和b的初始值
        '''
        np.random.seed(seed)
        self.lr = learning_rate
        self.max_iter = max_iter
        self.w = np.random.normal(1, 0.1)
        self.b = np.random.normal(1, 0.1)
        self.loss_arr = []
```

示例：梯度下降法

```
def fit(self, x, y):
    '''
    类的方法：训练函数
    参数 自变量：x
    参数 因变量：y
    返回每一次迭代后的损失函数
    '''
    for i in range(self.max_iter):
        self.__train_step(x,y)
        y_pred = self.predict(x)
        self.loss_arr.append(self.loss(y, y_pred))

def __f(self, x, w, b):
    '''
    类的方法：计算一元线性回归函数在x处的值
    '''
    return x * w + b
```

```
def predict(self, x):
    '''
    类的方法：预测函数
    参数：自变量：x
    返回：对x的回归值
    '''
    y_pred = self.__f(x, self.w, self.b)
    return y_pred

def loss(self, y_true, y_pred):
    '''
    类的方法：计算损失
    参数 真实因变量：y_true
    参数 预测因变量：y_pred
    返回：MSE损失
    '''
    return np.mean((y_true - y_pred)**2)
```

示例：梯度下降法

```
def __calc_gradient(self, x, y):
    '''
    类的方法：分别计算对w和b的梯度
    '''
    d_w = np.mean(2* (x * self.w + self.b - y) * x)
    d_b = np.mean(2*(x * self.w + self.b - y))
    return d_w, d_b

def __train_step(self,x,y):
    '''
    类的方法：单步迭代，即一次迭代中对梯度进行更新
    '''
    d_w, d_b = self.__calc_gradient(x,y)
    self.w = self.w - self.lr * d_w
    self.b = self.b - self.lr * d_b
    return self.w, self.b
```

示例：梯度下降法

In [36]:

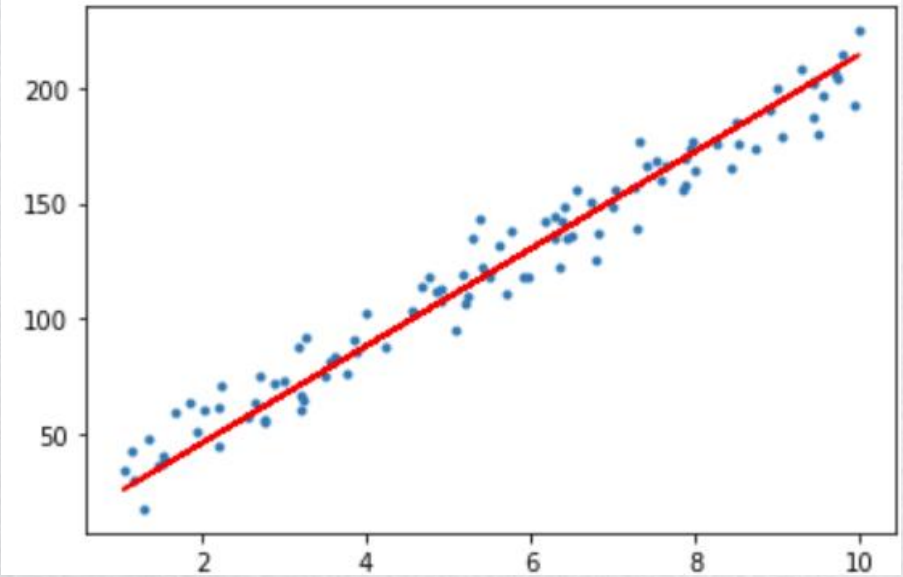
```
import numpy as np
import matplotlib.pyplot as plt

def show_data(x, y, w=None, b=None):
    plt.scatter(x, y, marker='.')
    if w is not None and b is not None:
        plt.plot(x, w*x+b, c='red')
    plt.show()

# data generation
np.random.seed(272)
data_size = 100
x = np.random.uniform(low=1.0, high=10.0, size=data_size)
y = x * 20 + 10 + np.random.normal(loc=0.0, scale=10.0, size=data_size)
```

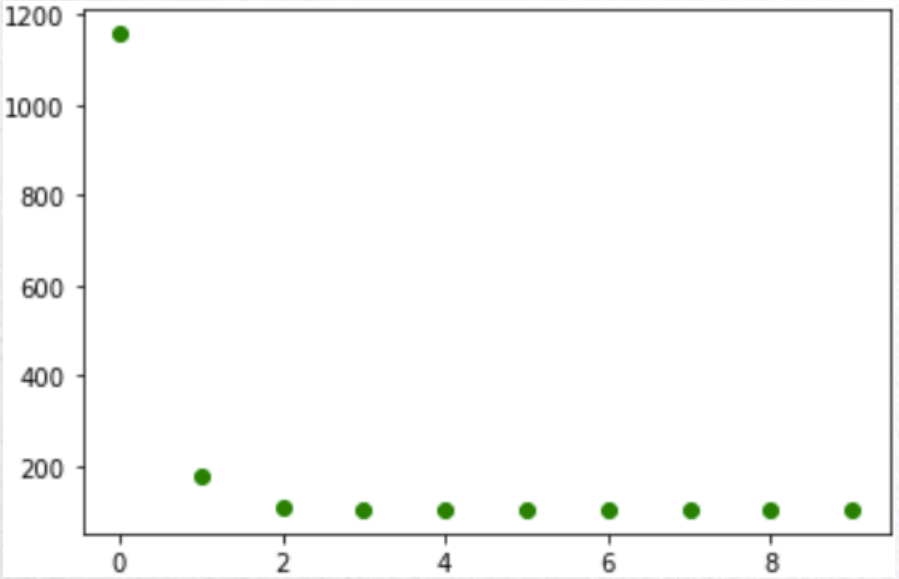
示例：梯度下降法

```
# train / test split
shuffled_index = np.random.permutation(data_size)
x = x[shuffled_index]
y = y[shuffled_index]
split_index = int(data_size * 0.7)
x_train = x[:split_index]
y_train = y[:split_index]
x_test = x[split_index:]
y_test = y[split_index:]

# train the liner regression model
regr = LinearRegression(learning_rate=0.01, max_iter=10, seed=314)
regr.fit(x_train, y_train)
print('w: \t{:.3}'.format(regr.w))
print('b: \t{:.3}'.format(regr.b))
show_data(x, y, regr.w, regr.b)

# plot the evolution of cost
plt.scatter(np.arange(len(regr.loss_arr)), regr.loss_arr, marker='o', c='green')
plt.show()
```

示例：梯度下降法

```
w:      21.0
b:      4.41
```





思考

- 多元线性回归：
  - 自变量x是向量，因变量y是数字
  - 自变量x是向量，因变量y也是向量

- 怎样用梯度下降法训练多元线性回归模型？

谢谢！