

# ML(机器学习)

---

## ML(机器学习)

### 机器学习基础

数据

机器学习的基本任务

分类任务

回归任务

机器学习方法分类

监督学习

非监督学习

半监督学习

增强学习

机器学习的其他分类

批量学习和在线学习

批量学习

在线学习

参数学习和非参数学习

参数学习

非参数学习

### 工具简介

Jupyter Notebook

魔法命令

Numpy

创建Numpy数组

Numpy.array的基本操作

Numpy.array的数据访问

Numpy.array的合并与分割

Numpy.array中运算

Numpy中的聚合操作

Numpy中排序及其索引

Numpy中的Fancy Indexing

Numpy.array中的比较

Matplotlib

### kNN

kNN的训练预测流程

kNN的实现代码

自己实现的kNN

sklearn自带的kNN

训练集与测试集的划分

自己实现的train\_test\_split

sklearn自带的train\_test\_split

分类准确度

自己实现的计算分类准确度accuracy\_score

sklearn自带的计算分类准确度accuracy\_score

超参数

网格搜索 (Grid Search)

数据归一化

sklearn自带的Scaler

kNN算法的思考

### 线性回归法

简单线性回归

简单线性回归的目标——损失函数最小

简单线性回归的参数结论——最小二乘法

## 实现简单线性回归

自己实现的SimpleLinearRegression(无优化)

自己实现的SimpleLinearRegression(向量化运算)

回归算法的评价 (MSE,RMSE,MAE)

自己实现的MSE,RMSE和MAE

sklearn自带的MSE和MAE

最好的衡量线性回归法的指标: R Squared

## 多元线性回归

多元线性回归的正规方程解 (Normal Equation)

实现线性回归

自己实现的线性回归

sklearn自带的线性回归

## 梯度下降法

线性回归中的梯度下降法

梯度下降法的向量化和数据标准化

随机梯度下降法

sklearn自带的随机梯度下降法

关于梯度的调试

## PCA

PCA步骤

梯度上升法解决PCA问题

求数据的前n个主成分

高维数据向低维数据映射

sklearn中自带的PCA

## 多项式回归

sklearn自带的多项式回归和Pipeline

PolynomialFeatures

Pipeline

过拟合与欠拟合 (泛化能力)

测试数据集的意义

学习曲线

验证数据集与交叉验证

k-folds 交叉检验

留一法 LOO-CV

偏差方差权衡

模型泛化与岭回归 (模型正则化)

sklearn中自带的岭回归

LASSO

# 机器学习基础

## 数据

### 数据的基本概念和数学表示

- 数据整体叫做数据集 (data set)
  - 每一行数据称为一个样本 (sample)
  - 除最后一列表示种类, 每一列表达样本的一个特征 (feature) [属性、维度]
  - 最后一列, 称为标记 (label)
- 
- 最后一列之外的其他列构成样本特征矩阵  $\mathbf{x}$
  - 最后一列构成标记向量  $\mathbf{y}$

第  $i$  个样本的特征向量写作

$$X^{(i)}$$

第  $i$  个样本第  $j$  个特征值写作

$$X_j^{(i)}$$

第  $i$  个样本的标记写作

$$y^{(i)}$$

- 数据构成的多维空间称为特征空间 (feature space)
- 分类任务本质就是在特征空间切分
- 在高维空间同理

## 机器学习的基本任务

---

### 分类任务

- 二分类
  - 一些算法只支持完成二分类的任务
  - 但是多分类的任务可以转换成二分类的任务
  - 有一些算法天然可以完成多分类的任务
- 多标签分类

### 回归任务

- 结果是一个连续数字的值，而非一个类别
  - 有一些算法只能解决回归问题
  - 有一些算法只能解决分类问题
  - 有一些算法既可以解决回归问题，也可以解决分类问题
- 一些情况下，回归任务可以简化成分类任务

## 机器学习方法分类

---

### 监督学习

给机器的训练数据拥有“标记”或者“答案”

- k近邻
- 线性回归与多项式回归
- 逻辑回归
- SVM
- 决策树与随机森林

## 非监督学习

对没有“标记”的数据进行分类——聚类分析

- 对数据进行降维处理：方便可视化
  - 特征提取
  - 特征压缩：PCA
- 异常检测

## 半监督学习

一部分数据有“标记”或者“答案”，另一部分数据没有

通常都先使用无监督学习手段对数据做处理，之后使用监督学习手段做模型的训练和预测

## 增强学习

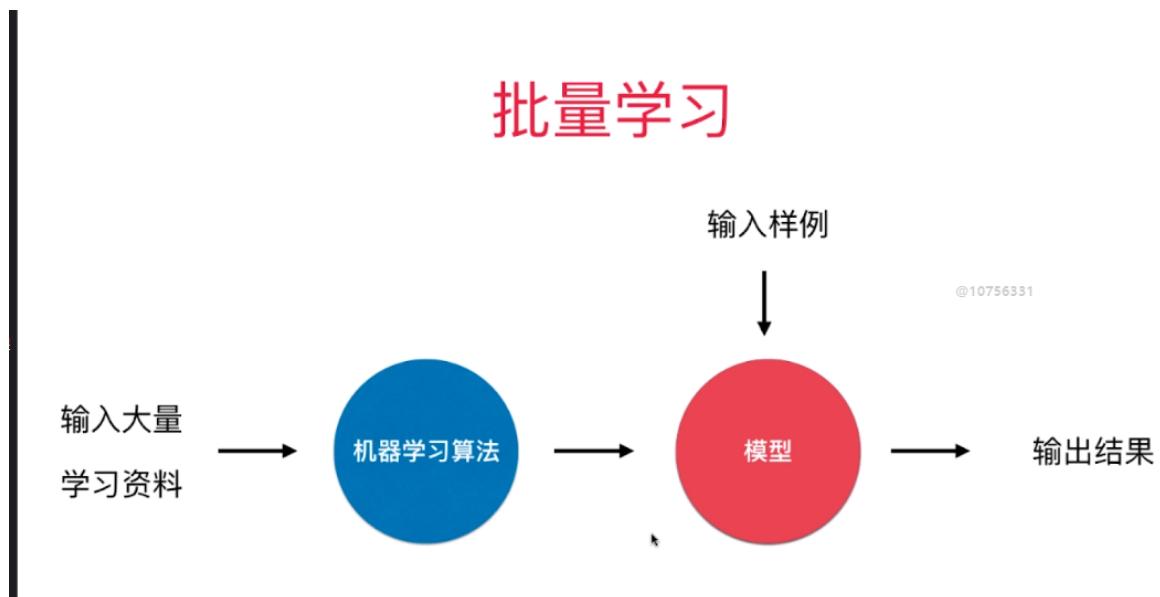
根据周围的情况，采取行动，更具采取行为的结果，学习行动方式

## 机器学习的其他分类

### 批量学习和在线学习

#### 批量学习

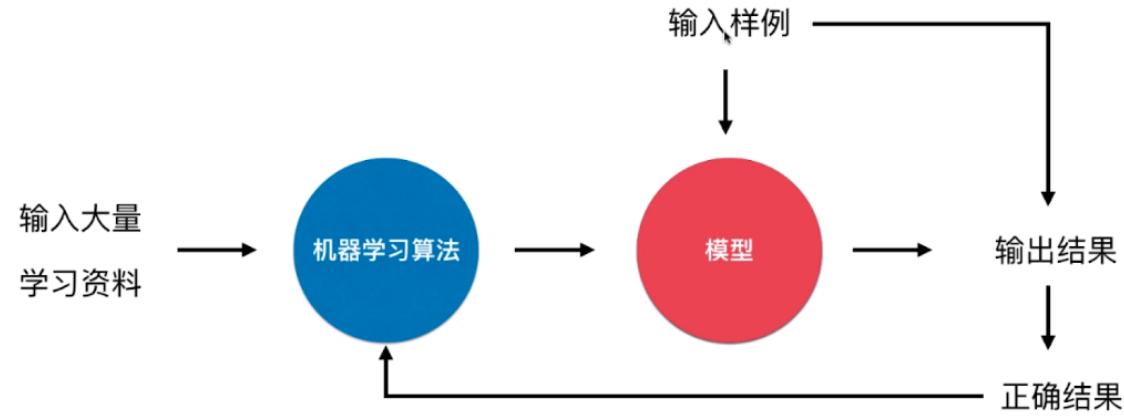
- 优点：简单
- 缺点：每次重新批量学习，运算量巨大



#### 在线学习

- 优点：及时反映新的环境的变化
- 问题：新的数据带来不好的变化？
  - 解决方案：需要加强对数据进行监控
- 其他：也适用于数据量巨大，完全无法批量学习的环境

# 在线学习



## 参数学习和非参数学习

### 参数学习

一旦学到了参数，就不再需要原有的数据集

### 非参数学习

不对模型进行过多假设

**非参数学习不等于没有参数！**

## 工具简介

### Jupyter Notebook

#### 魔法命令

##### %run 命令

```
%run Python脚本相较于Jupyter Notebook的位置  
%run ML_Demo/main.py
```

把Python脚本加载进Notebook (包括函数等内容)

当然也可以把该Python脚本用import进行引入

```
import ML_Demo.main  
from ML_Demo import main
```

**%timeit 命令** (测试单条指令的运行时间，多次运算后选择最快的几次取平均值)

```
%timeit L = [i**2 for i in range(1000000)]  
> 210 ms ± 1.75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

**%%timeit 命令** (测试指令块的运行时间, 多次运算后选择最快的几次取平均值)

```
%%timeit  
L = []  
for i in range(1000):  
    L.append(i ** 2)  
  
> 226 µs ± 1.19 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

**%time命令** (测试单条指令的运行时间, 只运行一次)

```
%time L = [i**2 for i in range(1000000)]  
  
> CPU times: total: 219 ms  
> Wall time: 223 ms
```

**%%time命令** (测试单条指令的运行时间, 只运行一次)

```
%%timeit  
L = []  
for i in range(1000):  
    L.append(i ** 2)  
  
> CPU times: total: 0 ns  
> Wall time: 9.01 ms
```

## Numpy

### 创建Numpy数组

```
import numpy as np  
  
# 构造多维数组(shape中的元组表示维度, dtype表示类型[默认为浮点型], fill_value为填充数组的值)  
np.zeros(10)  
> [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  
  
np.zeros(10, dtype=int)  
> [0 0 0 0 0 0 0 0 0 0]  
  
np.zeros(shape=(3, 4), dtype=int)  
> [[0 0 0 0]  
> [0 0 0 0]  
> [0 0 0 0]]
```

```

np.ones(shape=(3, 4))
> [[1. 1. 1. 1.]
> [1. 1. 1. 1.]
> [1. 1. 1. 1.]]

np.full(shape=(3, 4), fill_value=666)
> [[666 666 666 666]
> [666 666 666 666]
> [666 666 666 666]]


# 快速构造递增数组(和python3的range基本一致，但是步长可以为小数)
np.arange(0, 10)
> [0 1 2 3 4 5 6 7 8 9]

np.arange(0, 10, 0.5)
> [0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6. 6.5 7. 7.5 8. 8.5 9.
9.5]

np.arange(10, 0, -1)
> [10 9 8 7 6 5 4 3 2 1]

np.arange(10, 0, -1.5)
> [10. 8.5 7. 5.5 4. 2.5 1. ]


# 快速等分区间(和python3的range的参数基本一致，但是第三个参数是分成区间的个数)
np.linspace(0, 10, 5)
> [0. 2.5 5. 7.5 10. ]


# 快速生成随机整型向量(low下限, high上限, size生成的个数, dtype类型)
np.random.randint(low=0, high=10, size=1, dtype=int)
> [4]

np.random.randint(low=0, high=10, size=(2, 3), dtype=int)
> [[9 0 3]
> [2 5 0]]


# 可以修改随机数的种子(随机数的生成和种子是一一对应的关系)
np.random.seed = xxx


# 快速生成随机浮点型向量
np.random.random(10)
> [0.81430166 0.60066306 0.86614656 0.76750252 0.07353262 0.68018871 0.03070963
0.9244369 0.1550115 0.69099267]

np.random.random((3, 3))
> [[0.95475793 0.89230311 0.73095396]
> [0.47903551 0.34646541 0.49917552]
> [0.58268326 0.63736238 0.77798554]]


# 随机生成一个标准正态分布的值

```

```
np.random.normal()
> 0.10850831101316975

# 随机生成一个自定义正态分布的值(第一个loc为均值, 第二个scale为方差, 最后一个size为生成数组的样式)
np.random.normal(10, 100)
> 112.9574899167099

np.random.normal(0, 1, size=(2, 3))
> [[-0.51506516 -0.31596675  0.61156275]
> [-0.66518676  0.46993595  0.8913645]]
```

## Numpy.array的基本操作

```
import numpy as np

x = np.arange(10)
xx = np.arange(15).reshape(3, 5)
print(x)
print(xx)

> [0 1 2 3 4 5 6 7 8 9]
> [[ 0  1  2  3  4]
> [ 5  6  7  8  9]
> [10 11 12 13 14]]

# 查询维度      ndim
x.ndim
> 1
xx.ndim
> 2

# 查询形状      shape
x.shape
> (10,)
xx.shape
> (4,5)

# 查询元素个数      size
x.size
> 10
xx.size
> 15
```

## Numpy.array的数据访问

```
# 下标直接访问(特别注意多位数组的访问方式)      x[_x]      xx[ _x, _y]
x[1]
> 1
x[-1]
> 9
```

```

xx[2][2] = xx[(2, 2)] = xx[2,2]
> 12

# 切片(三个参数和range基本一致,特别注意多位数组的切片)-----可以降维, 无法升维
x[2:5]
> [2 3 4]
x[:5]
> [0 1 2 3 4]
x[5:]
> [5 6 7 8 9]
x[::-2]
> [[0 2 4 6 8]]
x[::-1]
> [9 8 7 6 5 4 3 2 1 0]

xx[:2, :3]
> [[0 1 2]
> [5 6 7]]

xx[::-1, ::-1]
> [[14 13 12 11 10]
> [ 9  8  7  6  5]
> [ 4  3  2  1  0]]

# 深拷贝(如果简单的采用切片, 得到的是浅拷贝)
xxx = xx[:2, :3].copy()

# 数组重构形状      reshape-----无法降维, 可以升维
x.reshape(2, 5)
> [[0 1 2 3 4]
> [5 6 7 8 9]]

x.reshape(5, -1)    # 只关心行数
> [[0 1]
> [2 3]
> [4 5]
> [6 7]
> [8 9]]

x.reshape(-1, 5)    # 只关心列数
> [[0 1 2 3 4]
> [5 6 7 8 9]]

```

## Numpy.array的合并与分割

```

import numpy as np

x = np.arange(3)
xx = x[::-1].copy()
print(x)
print(xx)

> [0 1 2 ]
> [2 1 0]

```

```

# 合并
## 使用concatenate([])来进行合并矩阵，但是注意，必须是同一个ndim维度的
np.concatenate([x, xx])
> [0 1 2 2 1 0]

y = np.arange(6).reshape(2, 3)
> [[0 1 2]
> [3 4 5]]

np.concatenate([y, y])
> [[0 1 2]
> [3 4 5]
> [0 1 2]
> [3 4 5]]

## 用axis控制合并维度
np.concatenate([y, y], axis=1)
> [[0 1 2 0 1 2]
> [3 4 5 3 4 5]]

np.concatenate([x.reshape(1, -1), y]
> [[0 1 2]
> [0 1 2]
> [3 4 5]]

## 使用vstack([])来进行列向的堆叠，不一定需要是同一个ndim维度的，只需要列数相同即可
## 使用hstack([])来进行横向的堆叠，不一定需要是同一个ndim维度的，只需要行数相同即可
np.vstack([x, y])
> [[0 1 2]
> [0 1 2]
> [3 4 5]]

## 使用tile完成向量的堆叠成矩阵(vstack,hstack的合并使用)
np.tile(x, [y, z])    # 把x向量进行横向堆叠y次，纵向堆叠z次

# 分割
## 使用split(x, [y1, y2, ...])来进行分割，对x进行切割，已y1, y2...为切割点
x = np.arange(10)
np.split(x, [2, 7])
> [array([0, 1]), array([2, 3, 4, 5, 6]), array([7, 8, 9])]

## 同样用axis控制分割维度
## 使用vsplit(x, [])来进行列向的分割
## 使用hsplit(x, [])来进行横向的分割

```

## Numpy.array中运算

```

# 支持矩阵与数之间的算数运算(对应元素相乘)

## python3中对list进行乘法操作实际上是进行复制多次
## numpy可以直接对array进行乘法操作，而且比列表生成式要快的多的多
x = np.arange(10)
2 * x
> [ 0  2  4  6  8 10 12 14 16 18]

```

```
## 可以直接对numpy.array进行所有python3支持的算术运算，结果表现为对数组的每一个元素进行该运算
```

```
## numpy.array支持的特殊的运算
## np.abs(x)          x所有元素取绝对值
## np.sin(x)          x所有元素取正弦值
## np.cos(x)          x所有元素取余弦值
## np.tan(x)          x所有元素取正切值
## np.exp(x)          x所有元素取e的幂
## np.power(n,x)      x所有元素的n次幂
## np.log()           x所有元素取e的对数
## np.log2()          x所有元素取2的对数
## np.log10()         x所有元素取10的对数
```

```
# 支持矩阵之间的算数运算(对应元素相乘)
```

```
x = np.arange(1, 11).reshape((2, 5))
y = np.random.randint(2, 10, (2, 5))
x + y
x * y
```

```
# 支持向量与矩阵的运算(等价于该向量对于矩阵的每一个行或者列进行了算数运算)
```

```
v = np.array([1, 2])
a = np.arange(4).reshape((2, 2))
> [1 2]
> [[0 1]
> [2 3]]

v + a
> [[1 3]
> [3 5]]

v * q
> [[0 2]
> [2 6]]
```

```
# 矩阵的标准乘法
```

```
A.dot(B)
```

```
# 矩阵的转置
```

```
A.T
```

```
# 矩阵的逆
```

```
np.linalg.inv(A)
```

```
# 矩阵的伪逆矩阵
```

```
np.linalg.pinv(A)
```

## Numpy中的聚合操作

```
# 对数组进行求和  
np.sum(A)  
# 对数组进行求积  
np.prod(A)  
# 最大值最小值  
np.max(A)    np.min(A)  
# 平均值  
np.mean(A)  
# 中位数  
np.median(A)  
# 查看百分位的值(表示v中比x%大的最小值)  
np.percentile(v, x)  
# 方差  
np.var(A)  
# 标准差  
np.std(A)  
  
# 对于二维矩阵, 如果单独进行行列统计, 需要添加axis参数  
# axis=0: 表示沿着行对每个列进行操作  
# axis=1: 表示沿着列对每个行进行操作
```

## Numpy中排序及其索引

```
# 找到数组最小值的索引  
np.argmin()  
# 找到数组最大值的索引  
np.argmax()  
  
# 排序数组  
np.sort(A, axis=0)    # 对列进行排序(沿着行)  
np.sort(A, axis=1)    # 对行进行排序(沿着列)  
  
# 排序索引数组(返回的是排序之后元素在原先数组中的位置)  
np.argsort(A)  
  
# 利用快速排序的分割算法划分位置(左半边比x小, 右半边比x大)  
np.partition(A, x)  
# 利用快速排序的分割算法划分位置(左半边比x小, 右半边比x大)(返回的是划分之后元素在原先数组中的位置)  
np.argpartition(A, x)
```

## Numpy中的Fancy Indexing

```
# 利用下标数组间接获取下标位置的值并构造数组(最后矩阵的样式和下标数组的维数一致)  
v = np.arange(10)  
  
# 一维数组取点
```

```

idx = [3, 5, 8]
v[idx]
> [3 5 8]

# 二维数组取点
idx = [[1, 4], [5, 7]]
v[np.array(idx)]
> [[1 4]
> [5 7]]

# 横纵坐标取点
v = np.arange(100).reshape(10, -1)
row = np.array([3, 4, 5])
col = np.array([6, 7, 8])
v[row, col]
> [36 47 58]

## 可以通过控制布尔值来筛选想要的行和列
v = np.arange(16).reshape(4, -1)
col = np.array([True, False, True, False])
v[1:3, col]
> [[ 4  6]
> [ 8 10]]

```

## Numpy.array中的比较

```

# 直接使用大于小于等于号
v = np.arange(10)
v > 3
> [False False False False  True  True  True  True  True  True]

# 用 count_nonzero 统计有多少个不为零的数
np.count_nonzero(v<=3)

# 用 any 来判断是否有一个符合标准
np.any(v == 0)

# 用 all 来判断是否全部满足要求
np.all(v >= 0)

# 矩阵中的索引部分也可以使用布尔表达式来选择
v[v % 2 == 0]

```

## Matplotlib

```

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

x = np.linspace(0, 10, 100)

y = np.sin(x)

```

```
# 使用plot绘制横纵坐标的连续图  
plt.plot(x, y)  
  
# 使用scatter绘制纵坐标的散点图  
plt.scatter(x, y)  
  
# 在show之前可以在同一个图中多次调用plot来使多个函数在一张图中(color选定颜色, linestyle选择线的样式, label表示该函数的名称图示)  
plt.plot(x, y, color="red", linestyle="--", label="sin函数")  
  
# 使用xlim和ylim来控制横纵坐标范围  
  
# 使用axis同时对x和y的范围进行操作  
plt.axis([1,10,-2,2])  
  
# 使用 xlabel 和 ylabel 来设置横纵坐标的名称  
  
# 使用 title 来添加标题  
plt.title()  
  
# 使用 legend 来添加图示  
plt.legend()  
  
# 使用show来展示图片  
plt.show()
```

## kNN

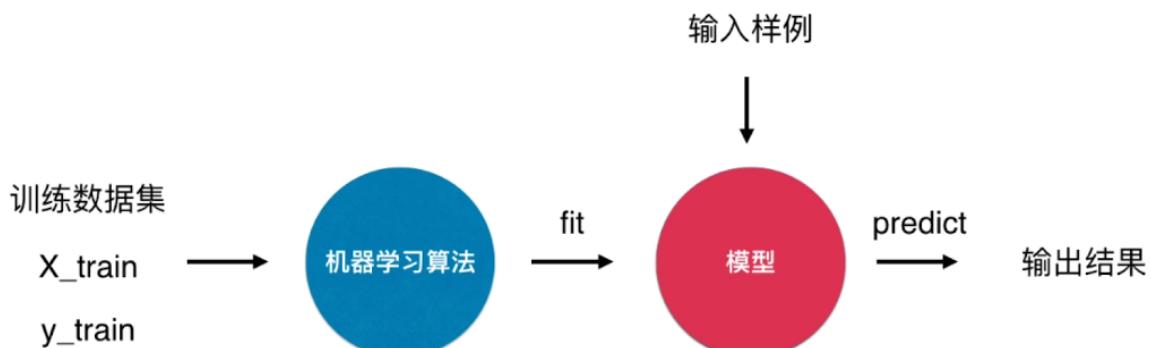
K近邻算法——k-Nearest Neighbors

KNN算法是一个不需要训练过程的算法

K近邻算法是非常特殊的，可以被认为是没有模型的算法

为了和其他算法统一，可以认为训练数据集本身就是模型本身

### kNN的训练预测流程



可以说kNN是一个不需要训练过程的算法

### kNN的实现代码

## 自己实现的kNN

```
import numpy as np
from math import sqrt
from collections import Counter

# 自己实现的(没有考虑距离的权重来解决平票问题)
def my_kNN_classifier(k, x_train, y_train, x):
    assert 1 <= k <= x_train.shape[0], "k must be valid"
    assert x_train.shape[0] == y_train.shape[0], \
        "the size of x_train must equal to the size of y_train"
    assert x_train.shape[1] == x.shape[0], \
        "the feature number of x must be equal to x_train"

    # 计算出 x 向量距离 x_train 中各个数据点的距离
    distances = [sqrt(np.sum((x - x_train) ** 2)) for x_train in x_train]
    # 对距离进行排序并得到其索引位置
    nearest = np.argsort(distances)
    # 找到 k 近邻的种类
    topK_y = [y_train[i] for i in nearest[:k]]
    # 进行种类统计
    votes = Counter(topK_y)
    # 返回统计数最多的一个种类
    return votes.most_common(1)[0][0]
```

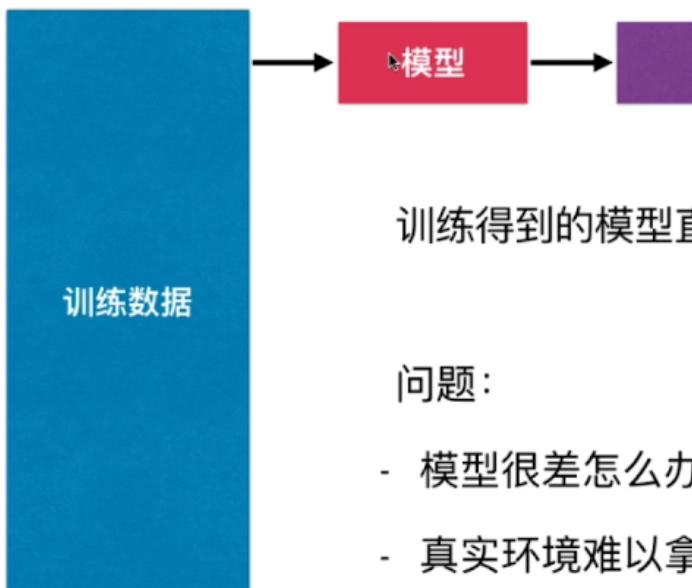
## sklearn自带的kNN

```
from sklearn.neighbors import KNeighborsClassifier

# 创建一个 kNN算法的分类器(没有考虑距离的权重来解决平票问题)
kNN_classifier = KNeighborsClassifier(n_neighbors=k)
# 开始进行拟合
kNN_classifier.fit(x_train, y_train)
# 进行预测
kNN_classifier.predict(x)
```

## 训练集与测试集的划分

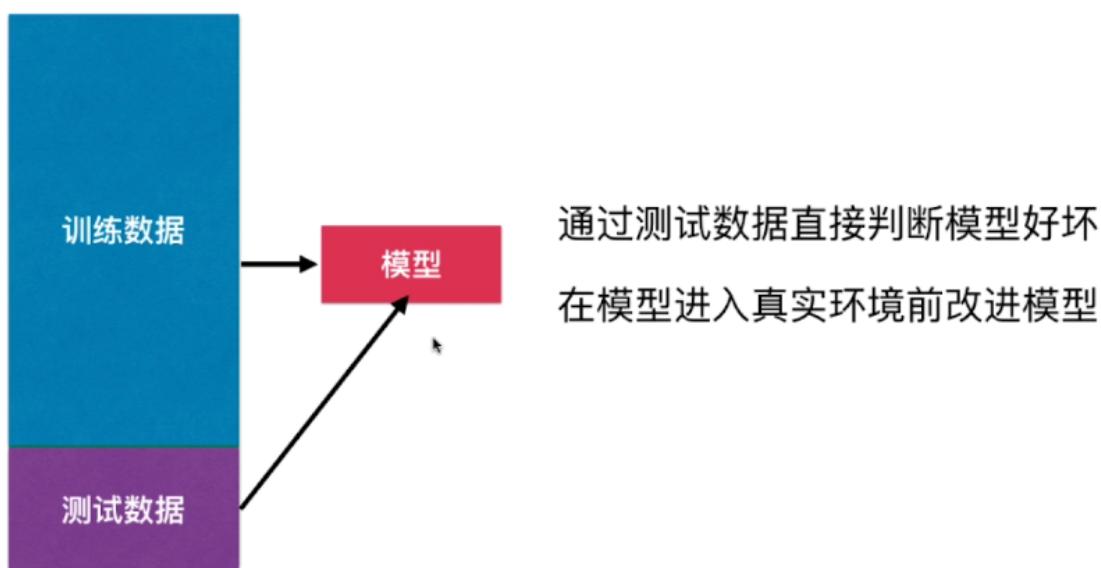
# 判断机器学习算法的性能



训练得到的模型直接在真实环境中使用。

问题：

- 模型很差怎么办？真实损失。
- 真实环境难以拿到真实label?



通过测试数据直接判断模型好坏  
在模型进入真实环境前改进模型

因此我们需要**训练数据集和测试数据集的分离**(train\_test\_split)

如何进行分离（保证样本和标签不会因为随机而混乱）：

- 要么就把样本和标签直接拼接在一起，随机打乱之后再进行拆分
- 要么直接乱序索引即可

## 自己实现的train\_test\_split

实现一个测试集和训练集分离的方法

```
import numpy as np
```

```

# 自己实现的
# 进行训练集和测试集的分离
# x 样本集, y 标签集, test_ratio 分割率, seed 随机化种子
def my_train_test_split(x, y, test_ratio=0.2, seed=None):
    assert x.shape[0] == y.shape[0], \
        "the size of x must be equal to the size of y"
    assert 0.0 <= test_ratio <= 1.0, \
        "test_ratio must be valid"
    if seed:
        np.random.seed(seed)
    # 进行索引随机化, permutation(x)可以获得 0~x 的随机排列
    shuffle_indexes = np.random.permutation(len(x))

    # 设置测试数据据的比例和大小
    test_size = int(len(x) * test_ratio)

    # 获得测试数据集和训练数据集的索引
    test_indexes = shuffle_indexes[:test_size]
    train_indexes = shuffle_indexes[test_size:]

    # 利用 Fancy Indexing 快速构建测试集和训练集
    x_train = x[train_indexes]
    y_train = y[train_indexes]
    # 利用 Fancy Indexing 快速构建测试集和训练集
    x_test = x[test_indexes]
    y_test = y[test_indexes]

    return x_train, y_train, x_test, y_test

```

## sklearn自带的train\_test\_split

```

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)

```

## 分类准确度

### 自己实现的计算分类准确度accuracy\_score

```

import numpy as np

# 自己实现的计算准确率的函数
def my_accuracy_score(y_true, y_predict):
    assert y_true.shape[0] == y_predict.shape[0], \
        "the size of y_true must be equal to the size of y_predict"
    return np.sum(y_true == y_predict) / len(y_true)

```

## sklearn自带的计算分类准确度accuracy\_score

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 计算准确率
shot_ratio = accuracy_score(y_test, y_predict)

# 如果不关心预测的结果，可以直接查看准确率，跳过得到y_predict的过程
kNN_classifier.score(X_test, y_test)
```

## 超参数

# 超参数和模型参数

- 超参数：在算法运行前需要决定的参数
- 模型参数：算法过程中学习的参数

kNN算法没有模型参数

kNN算法中的k是典型的超参数

寻找好的超参数：

- 领域知识
- 经验数值
- 实验搜索

kNN有三个超参数：

- k的大小 (n\_neighbors=5)
- 是否考虑距离的权重来解决平票问题 (weights="distance"/"uniform")

- 距离的计算选择 (曼哈顿距离, 欧拉距离, 明可夫斯基距离) (p=2)

# 距离

$$\left( \sum_{i=1}^n |X_i^{(a)} - X_i^{(b)}| \right)^{\frac{1}{1}}$$

$$\left( \sum_{i=1}^n |X_i^{(a)} - X_i^{(b)}|^2 \right)^{\frac{1}{2}}$$

$$\left( \sum_{i=1}^n |X_i^{(a)} - X_i^{(b)}|^p \right)^{\frac{1}{p}}$$

## 网格搜索 (Grid Search)

sklearn提供Grid Search为我们提供寻找超参数的好方法

```

import numpy as np
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV, train_test_split

# 加载数据
digits = load_digits()

# 获取样本矩阵和标签向量
X = digits.data
y = digits.target

# 进行训练集和测试集的分离
X_train, X_test, y_train, y_test = train_test_split(X, y)

# 定义超参数的网格
param_grid = [
    {
        'weights': ['uniform'],

```

```
'n_neighbors': [i for i in range(1, 11)]
},
{
    'weights': ['distance'],
    'n_neighbors': [i for i in range(1, 11)],
    'p': [i for i in range(1, 6)]
}
]

# 定义一个分类器
knn_clf = KNeighborsClassifier()

# 定义一个网格搜索器(knn_clf-原始分类器, param_grid-网格超参数, n_jobs-CPU运行颗数,
verbose-输出信息详细度)
grid_search = GridSearchCV(knn_clf, param_grid, n_jobs=-1, verbose=2)

# 进行超参数拟合
grid_search.fit(x_train, y_train)

# 获得网格搜索的最佳分类器
print(grid_search.best_estimator_)

# 获取网格搜索的最佳准确率
print(grid_search.best_score_)

# 获取网格搜索的最佳超参数
print(grid_search.best_params_)
```

kNN还有更多超参数的选择

## 更多的距离定义

- 向量空间余弦相似度 Cosine Similarity
- 调整余弦相似度 Adjusted Cosine Similarity
- 皮尔森相关系数 Pearson Correlation Coefficient
- Jaccard相似系数 Jaccard Coefficient

## 数据归一化

将所有的数据映射到同一尺度

**最值归一化**(normalization): 把所有数据映射到0-1之间

$$x_{scale} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

适用于分布**有明显边界**的情况；受outlier影响较大

**均值方差归一化**(standardization): 把所有数据归一到均值为0，方差为1的分布中

$$x_{scale} = \frac{x - x_{mean}}{s}$$

适用于数据分布**没有明显边界**的情况；有可能存在极端数据值

## 对测试数据集如何归一化？



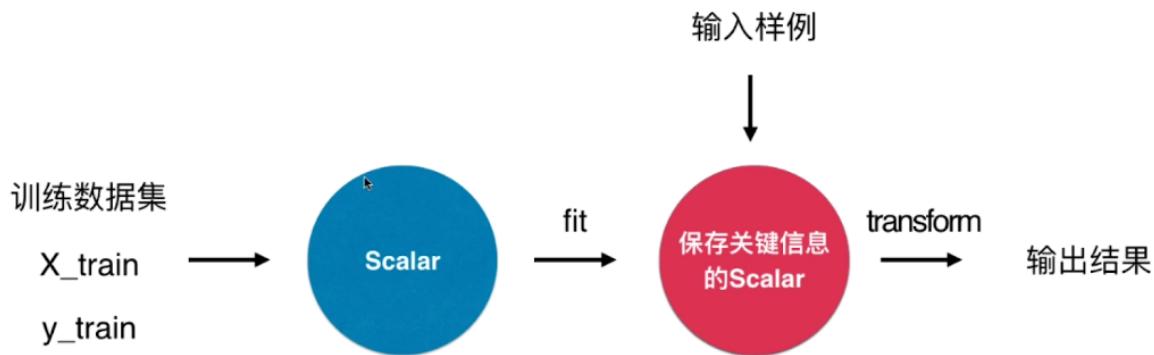
测试数据是模拟真实环境

- 真实环境很有可能无法得到所有测试数据的均值和方差
- 对数据的归一化也是算法的一部分

$$(x_{test} - \text{mean}_{\text{train}}) / \text{std}_{\text{train}}$$

## sklearn自带的Scaler

# 对测试数据集如何归一化？



```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 进行训练集和测试集的分离
X_train, X_test, y_train, y_test = train_test_split(x, y)

# 预处理：进行均值方差归一化
# 创建 Scalar 对象
standardScaler = StandardScaler()
# 进行拟合(传递值)
standardScaler.fit(X_train)

# 查看各个维度的均值
print(standardScaler.mean_)

# 查看各个维度的方差
print(standardScaler.scale_)

# 进行归一化处理训练集
X_train = standardScaler.transform(X_train)

# 利用训练集的均值方差归一化的scalar对测试集进行归一化
X_test_standard = standardScaler.transform(X_test)
```

## kNN算法的思考

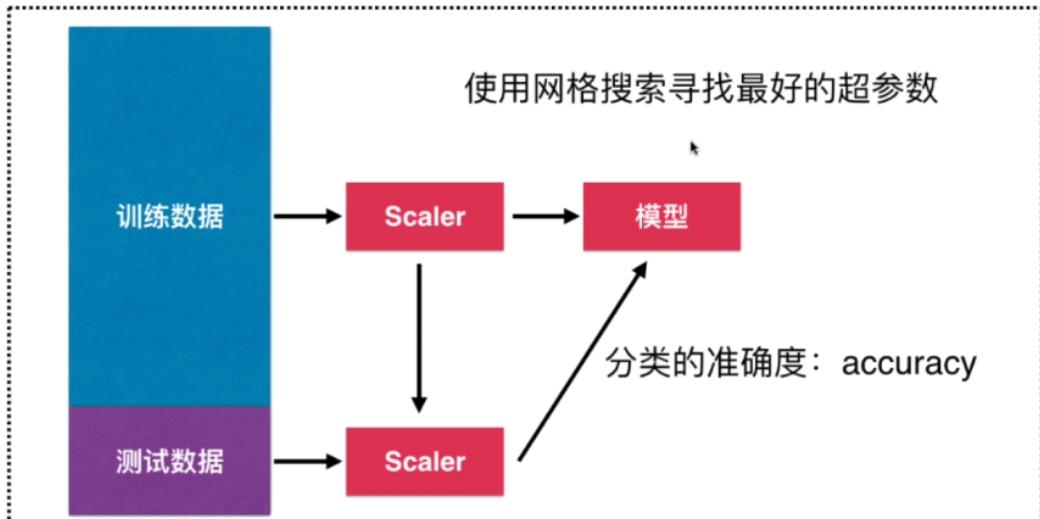
优点：

- 可以解决分类问题
- 天然可以解决多分类问题
- 可以使用k近邻算法解决回归问题

缺点：

- 效率低下
- 高度数据相关
- 预测的结果不具有可解释性
- 维数灾难（随着维数的增加，“看似相近”的两个点之间的距离越来越大）

# 机器学习流程回顾

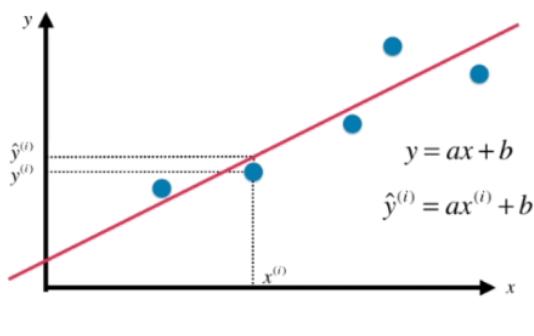


## 线性回归法

LR (Linear Regression)

### 简单线性回归

## 简单线性回归



假设我们找到了最佳拟合的直线方程：

$$y = ax + b$$

则对于每一个样本点  $x^{(i)}$

根据我们的直线方程，预测值为：

$$\hat{y}^{(i)} = ax^{(i)} + b$$

真值为：  $y^{(i)}$

# 简单线性回归

假设我们找到了最佳拟合的直线方程：

$$y = ax + b$$

则对于每一个样本点  $x^{(i)}$

根据我们的直线方程，预测值为：

$$\hat{y}^{(i)} = ax^{(i)} + b$$

真值为：  $y^{(i)}$

我们希望  $y^{(i)}$  和  $\hat{y}^{(i)}$  的差距尽量小

表达  $y^{(i)}$  和  $\hat{y}^{(i)}$  的差距：

$$\underline{\underline{|y^{(i)} - \hat{y}^{(i)}|}}$$

$$\underline{\underline{(y^{(i)} - \hat{y}^{(i)})^2}}$$

考虑所有样本：  $\sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$

## 简单线性回归的目标——损失函数最小

# 简单线性回归

目标：使  $\sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$  尽可能小

$$\hat{y}^{(i)} = ax^{(i)} + b$$

目标：找到a和b，使得  $\sum_{i=1}^m (y^{(i)} - ax^{(i)} - b)^2$  尽可能小

# 一类机器学习算法的基本思路

目标：找到 $a$ 和 $b$ ，使得  $\sum_{i=1}^m (y^{(i)} - ax^{(i)} - b)^2$  尽可能小



损失函数(loss function) 效用函数(utility function)

通过分析问题，确定问题的损失函数或者效用函数；  
通过最优化损失函数或者效用函数，获得机器学习的模型。

## 简单线性回归的参数结论——最小二乘法

# 简单线性回归

目标：找到 $a$ 和 $b$ ，使得  $\sum_{i=1}^m (y^{(i)} - ax^{(i)} - b)^2$  尽可能小

典型的最小二乘法问题：最小化误差的平方

$$a = \frac{\sum_{i=1}^m (x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^m (x^{(i)} - \bar{x})^2} \quad , \quad b = \bar{y} - a\bar{x}$$

## 实现简单线性回归

## 自己实现的SimpleLinearRegression(无优化)

```
import numpy as np

class SimpleLinearRegression:
    def __init__(self):
        self.a_ = None
        self.b_ = None

    def fit(self, x_train, y_train):
        assert x_train.ndim == 1, \
            "Simple Linear Regressor can only solve single feature training \
data"
        assert len(x_train) == len(y_train), \
            "the size of x_train must be equal to the size of y_train"

        # 获取平均值
        x_mean = np.mean(x_train)
        y_mean = np.mean(y_train)

        # 初始化分子分母
        num = 0.0
        d = 0.0
        # 利用最小二乘法的公式求出参数 a_ 和 b_
        for x, y in zip(x_train, y_train):
            num += (x - x_mean) * (y - y_mean)
            d += (x - x_mean) ** 2
        self.a_ = num / d
        self.b_ = y_mean - self.a_ * x_mean
        return self

    def predict(self, x_predict):
        return np.array(self.a_ * x_predict + self.b_)
```

## 自己实现的SimpleLinearRegression(向量化运算)

不需要使用for循环，直接转换为向量点乘

# 向量化运算

$$\sum_{i=1}^m w^{(i)} \cdot v^{(i)} \quad w = (w^{(1)}, w^{(2)}, \dots, w^m)$$



$$v = (v^{(1)}, v^{(2)}, \dots, v^m)$$



@10756331

$$w \cdot v$$

```
import numpy as np

class SimpleLinearRegression:
    def __init__(self):
        self.a_ = None
        self.b_ = None

    def fit(self, x_train, y_train):
        assert x_train.ndim == 1, \
            "Simple Linear Regressor can only solve single feature training \
data"
        assert len(x_train) == len(y_train), \
            "the size of x_train must be equal to the size of y_train"

        # 获取平均值
        x_mean = np.mean(x_train)
        y_mean = np.mean(y_train)

        # 利用最小二乘法的公式求出参数 a_ 和 b_ (采用向量化运算)
        num = (x_train - x_mean).dot(y_train - y_mean)
        d = (x_train - x_mean).dot(x_train - x_mean)
        self.a_ = num / d
        self.b_ = y_mean - self.a_ * x_mean
        return self

    def predict(self, x_predict):
        return np.array(self.a_ * x_predict + self.b_)
```

回归算法的评价 (MSE,RMSE,MAE)

# 线性回归算法的评测

$$\frac{1}{m} \sum_{i=1}^m (y_{test}^{(i)} - \hat{y}_{test}^{(i)})^2$$

均方误差 MSE  
(Mean Squared Error)

# 线性回归算法的评测

$$\sqrt{\frac{1}{m} \sum_{i=1}^m (y_{test}^{(i)} - \hat{y}_{test}^{(i)})^2} = \sqrt{MSE_{test}}$$

均方根误差 RMSE  
(Root Mean Squared Error)

# 线性回归算法的评测

$$\frac{1}{m} \sum_{i=1}^m |y_{test}^{(i)} - \hat{y}_{test}^{(i)}|$$

平均绝对误差 MAE  
(Mean Absolute Error)

## 自己实现的MSE, RMSE和MAE

```
from math import sqrt

import numpy as np
from sklearn.datasets import load_boston
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# 加载数据
from my_sklearn.simpleLinearRegression import SimpleLinearRegression

boston = load_boston()

# 暂时只使用波士顿房价数据的 RM(房间的数量) 这一列
x = boston.data[:, 5]
y = boston.target

# 去除不确定的点
max_y = np.max(y)
x = x[y < max_y]
y = y[y < max_y]

# 划分训练集和测试集
x_train, x_test, y_train, y_test = train_test_split(x, y)

# 创建简单线性回归模型
reg = SimpleLinearRegression()

# 进行拟合(利用最小二乘法计算a,b)
reg.fit(x_train, y_train)
```

```
# 得到预测结果
y_predict = reg.predict(x_test)

# 绘制拟合图像
plt.scatter(x_train, y_train)
plt.plot(x_train, reg.predict(x_train), color="red")
plt.show()

# MSE
mse_test = np.sum((y_test - y_predict) ** 2) / len(y_test)
print(mse_test)

# RMSE
rmse_test = sqrt(np.sum((y_test - y_predict) ** 2) / len(y_test))
print(rmse_test)

# MAE
mae_test = np.sum(np.abs(y_test - y_predict)) / len(y_test)
print(mae_test)
```

## sklearn自带的MSE和MAE

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# MSE
mse_test = mean_squared_error(y_test,y_predict)
print(mse_test)

# MAE
mae_test = mean_absolute_error(y_test,y_predict)
print(mae_test)
```

## 最好的衡量线性回归法的指标：R Squared

# R Squared

$$R^2 = 1 - \frac{SS_{residual}}{SS_{total}}$$

(Residual Sum of Squares)  
(Total Sum of Squares)

$$R^2 = 1 - \frac{\sum_i (\hat{y}^{(i)} - y^{(i)})^2}{\sum_i (\bar{y} - y^{(i)})^2}$$

# R Squared

$$R^2 = 1 - \frac{\sum_i (\hat{y}^{(i)} - y^{(i)})^2}{\sum_i (\bar{y} - y^{(i)})^2}$$

- $R^2 \leq 1$
- $R^2$  越大越好。当我们的预测模型不犯任何错误时， $R^2$ 得到最大值1
- 当我们的模型等于基准模型时， $R^2$ 为0
- 如果  $R^2 < 0$ ，说明我们学到的模型还不如基准模型。此时，很有可能我们的数据不存在任何线性关系。

# R Squared

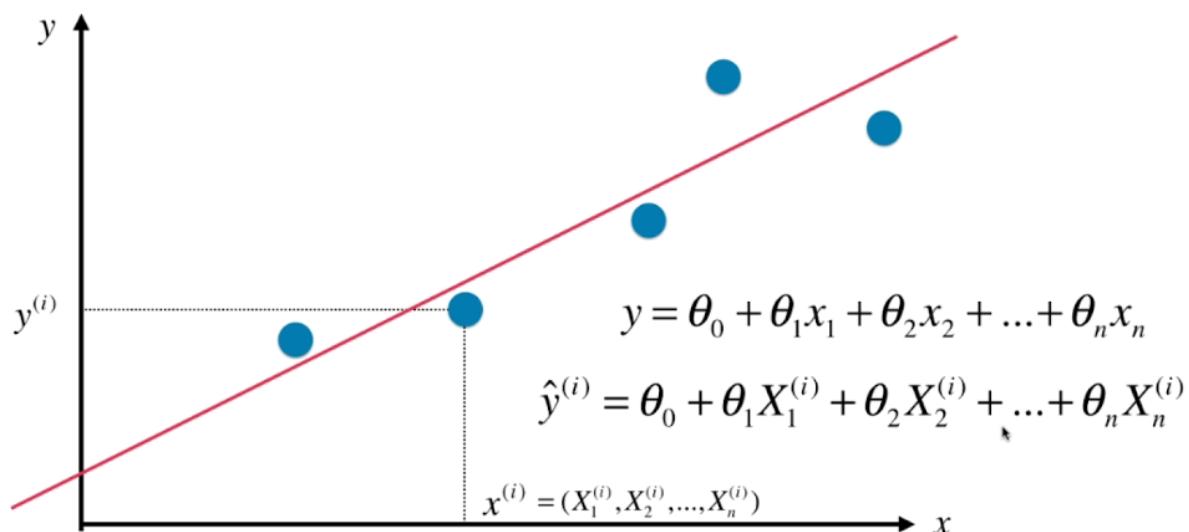
$$R^2 = 1 - \frac{\sum_i (\hat{y}^{(i)} - y^{(i)})^2}{\sum_i (\bar{y} - y^{(i)})^2} = 1 - \frac{(\sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2) / m}{(\sum_{i=1}^m (y^{(i)} - \bar{y})^2) / m}$$
$$= 1 - \frac{MSE(\hat{y}, y)}{Var(y)}$$

```
from sklearn.metrics import r2_score  
  
# R Square  
R_Square_test = r2_score(y_test, y_predict)
```

## 多元线性回归

### 多元线性回归的正规方程解 (Normal Equation)

## 多元线性回归



# 多元线性回归

$$\hat{y}^{(i)} = \theta_0 + \theta_1 X_1^{(i)} + \theta_2 X_2^{(i)} + \dots + \theta_n X_n^{(i)}$$

$$\theta = (\theta_0, \theta_1, \theta_2, \dots, \theta_n)^T$$

$$\hat{y}^{(i)} = \theta_0 X_0^{(i)} + \theta_1 X_1^{(i)} + \theta_2 X_2^{(i)} + \dots + \theta_n X_n^{(i)}, X_0^{(i)} \equiv 1$$

$$X^{(i)} = (X_0^{(i)}, X_1^{(i)}, X_2^{(i)}, \dots, X_n^{(i)})$$

$$\hat{y}^{(i)} = X^{(i)} \cdot \theta$$

# 多元线性回归

$$\hat{y} = X_b \cdot \theta$$

目标：使  $\sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$  尽可能小



目标：使  $(y - X_b \cdot \theta)^T (y - X_b \cdot \theta)$  尽可能小

# 多元线性回归的正规方程解 (Normal Equation)

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

问题：时间复杂度高： $O(n^3)$  (优化 $O(n^{2.4})$ )

优点：不需要对数据做归一化处理

## 实现线性回归

### 自己实现的线性回归

```
import numpy as np
from sklearn.metrics import r2_score

class LinearRegression:

    def __init__(self):
        # 系数
        self.coef_ = None
        # 截距
        self.interception_ = None
        # 整体theta向量(系数+截距)
        self._theta = None

    def __repr__(self):
        return "LinearRegression()"

    def fit_normal(self, x_train, y_train):
        x_b = np.hstack([np.ones(shape=(len(x_train), 1)), x_train])
        self._theta = np.linalg.inv(x_b.T.dot(x_b)).dot(x_b.T).dot(y_train)
        self.interception_ = self._theta[0]
        self.coef_ = self._theta[1:]
        return self

    def predict(self, x_predict):
        x_b = np.hstack([np.ones(shape=(len(x_predict), 1)), x_predict])
        return x_b.dot(self._theta)

    def score(self, x_test, y_test):
        y_predict = self.predict(x_test)
        return r2_score(y_test, y_predict)
```

## sklearn自带的线性回归

```
from sklearn.linear_model import LinearRegression

# 创建线性回归模型
reg = LinearRegression()

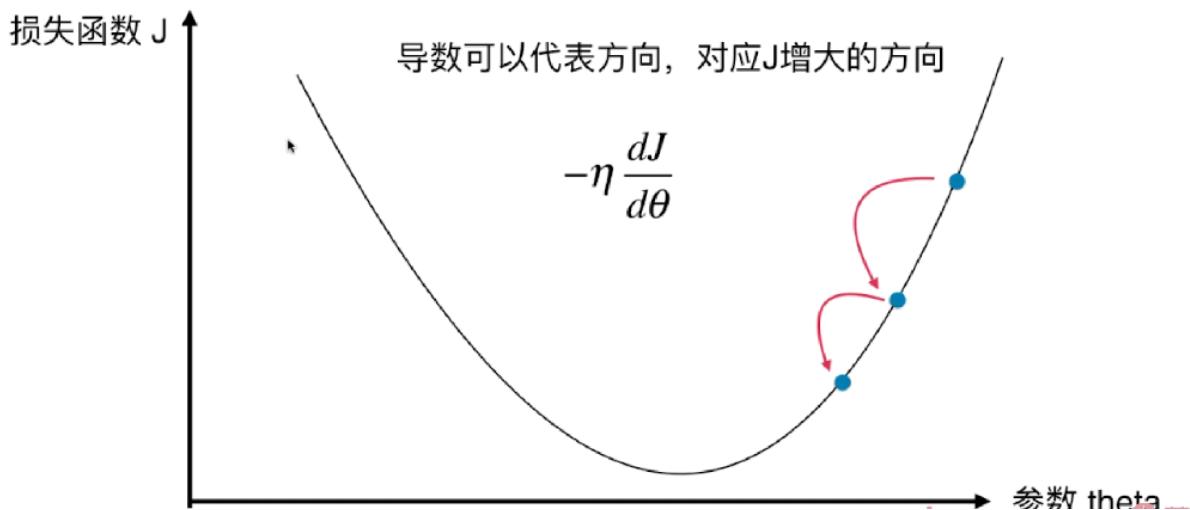
# 进行拟合(利用最小二乘法计算a,b)
reg.fit(x_train, y_train)

print(reg.coef_)
print(reg.score(x_test, y_test))
```

## 梯度下降法

- 不是一个机器学习算法
- 是一种基于搜索的最优化方法
- 作用：最小化一个损失函数
- 梯度上升法：最大化一个效用函数

## 梯度下降法



# 梯度下降法

- $\eta$  称为学习率(learning rate)
- $\eta$  的取值影响获得最优解的速度
- $\eta$  取值不合适，甚至得不到最优解
- $\eta$  是梯度下降法的一个超参数

## 梯度下降法

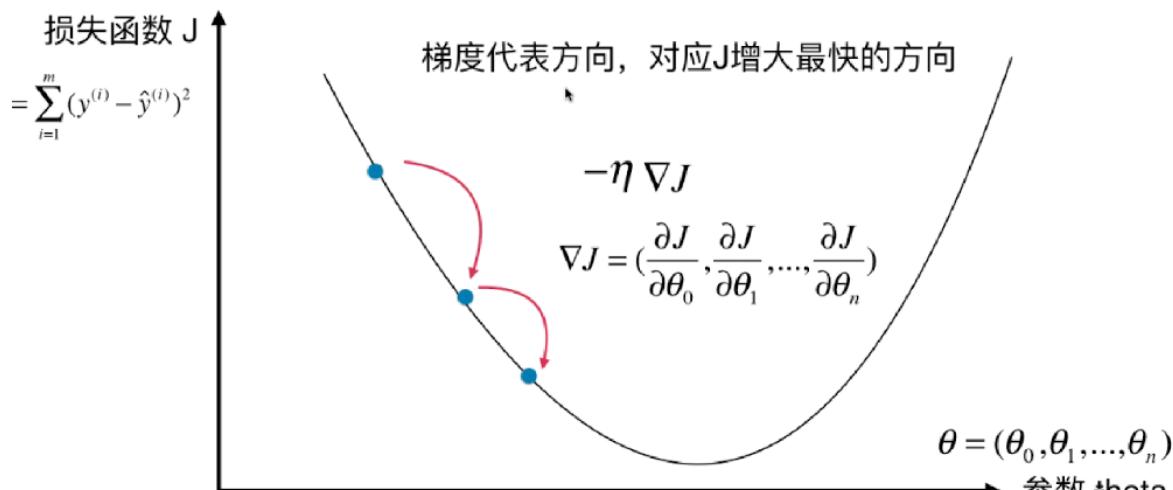
- 并不是所有函数都有唯一的极值点

解决方案：

- 多次运行，随机化初始点
- 梯度下降法的初始点也是一个超参数

## 线性回归中的梯度下降法

### 梯度下降法



目标：使  $\sum_{i=1}^m (y^{(i)} - \theta_0 - \theta_1 X_1^{(i)} - \theta_2 X_2^{(i)} - \dots - \theta_n X_n^{(i)})^2$  尽可能小

$$\nabla J(\theta) = \begin{pmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^m 2(y^{(i)} - X_b^{(i)} \theta) \cdot (-1) \\ \sum_{i=1}^m 2(y^{(i)} - X_b^{(i)} \theta) \cdot (-X_1^{(i)}) \\ \sum_{i=1}^m 2(y^{(i)} - X_b^{(i)} \theta) \cdot (-X_2^{(i)}) \\ \vdots \\ \sum_{i=1}^m 2(y^{(i)} - X_b^{(i)} \theta) \cdot (-X_n^{(i)}) \end{pmatrix} = 2 \cdot \begin{pmatrix} \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_1^{(i)} \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_2^{(i)} \\ \vdots \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_n^{(i)} \end{pmatrix}$$

### 线性回归中使用梯度下降法

目标：使  $\frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$  尽可能小

$$J(\theta) = MSE(y, \hat{y})$$

$$\text{有时取: } J(\theta) = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$$

$$\nabla J(\theta) = \begin{pmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{pmatrix} = \frac{2}{m} \cdot \begin{pmatrix} \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_1^{(i)} \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_2^{(i)} \\ \vdots \\ \sum_{i=1}^m (X_b^{(i)} \theta - y^{(i)}) \cdot X_n^{(i)} \end{pmatrix}$$

## 梯度下降法的向量化和数据标准化

### 线性回归中使用梯度下降法

$$\frac{2}{m} \cdot \begin{pmatrix} \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_0^{(i)} \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_1^{(i)} \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_2^{(i)} \\ \dots \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_n^{(i)} \end{pmatrix} \cdot \begin{pmatrix} X_0^{(1)} & X_1^{(1)} & X_2^{(1)} & \dots & X_n^{(1)} \\ X_0^{(2)} & X_1^{(2)} & X_2^{(2)} & \dots & X_n^{(2)} \\ X_0^{(3)} & X_1^{(3)} & X_2^{(3)} & \dots & X_n^{(3)} \\ \dots & \dots & \dots & \dots & \dots \\ X_0^{(m)} & X_1^{(m)} & X_2^{(m)} & \dots & X_n^{(m)} \end{pmatrix} = \frac{2}{m} \cdot (X_b\theta - y)^T \cdot X_b$$
$$\frac{2}{m} \cdot X_b^T \cdot (X_b\theta - y)$$

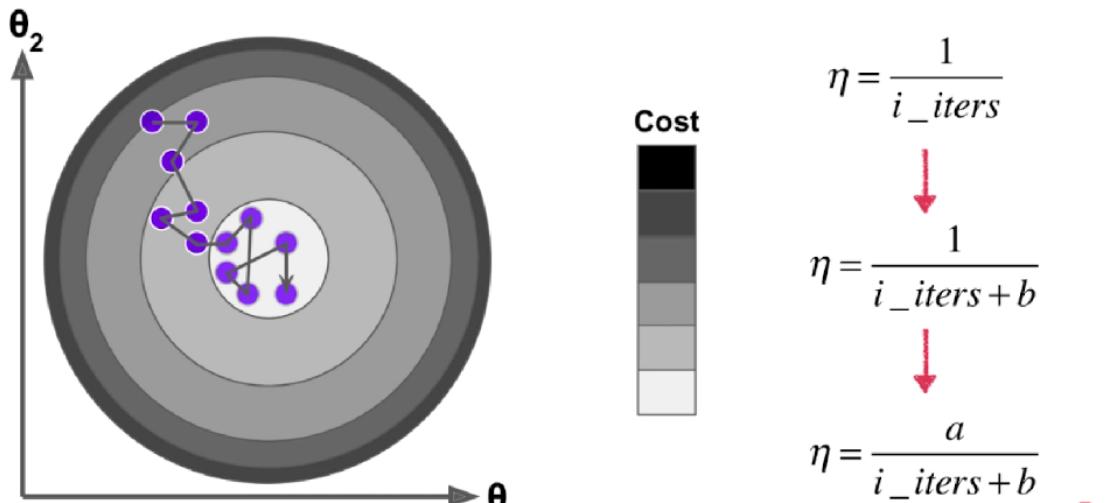
由于各个指标的度量可能不一样，导致有一些变量影响太大，我们可以采用**数据归一化**来处理整个流程，这样不仅可以加快速度，而且可以加快准确度

## 随机梯度下降法

### 随机梯度下降法 Stochastic Gradient Descent

$$\frac{2}{m} \cdot \begin{pmatrix} \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_0^{(i)} \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_1^{(i)} \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_2^{(i)} \\ \dots \\ \sum_{i=1}^m (X_b^{(i)}\theta - y^{(i)}) \cdot X_n^{(i)} \end{pmatrix} \cdot 2 \cdot \begin{pmatrix} (X_b^{(i)}\theta - y^{(i)}) \cdot X_0^{(i)} \\ (X_b^{(i)}\theta - y^{(i)}) \cdot X_1^{(i)} \\ (X_b^{(i)}\theta - y^{(i)}) \cdot X_2^{(i)} \\ \dots \\ (X_b^{(i)}\theta - y^{(i)}) \cdot X_n^{(i)} \end{pmatrix} = 2 \cdot (X_b^{(i)})^T \cdot (X_b^{(i)}\theta - y^{(i)})$$

# 随机梯度下降法 Stochastic Gradient Descent



## sklearn自带的随机梯度下降法

```
import numpy as np
from sklearn.datasets import load_boston
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 加载数据
boston = load_boston()

# 使用波士顿房价数据
X = boston.data
y = boston.target

# 去除不确定的点
max_y = np.max(y)
X = X[y < max_y]
y = y[y < max_y]

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)

# 预处理: 进行均值方差归一化
# 创建 Scaler 对象
standardScaler = StandardScaler()
# 进行拟合(传递值)
standardScaler.fit(X_train)

X_train_standard = standardScaler.transform(X_train)
X_test_standard = standardScaler.transform(X_test)

# 创建随机梯度下降法的线性回归模型
reg = SGDRegressor()

reg.fit(X_train_standard, y_train)
```

```
print(reg.score(x_test_standard, y_test))
```

## 关于梯度的调试

可以先计算好，有一个大致的目标和方向，之后再推导公式进行精细运算

## 关于梯度的调试

10756331

$$\theta = (\theta_0, \theta_1, \theta_2, \dots, \theta_n)$$
$$\theta_1^+ = (\theta_0, \theta_1 + \varepsilon, \theta_2, \dots, \theta_n)$$
$$\theta_1^- = (\theta_0, \theta_1 - \varepsilon, \theta_2, \dots, \theta_n)$$
$$\frac{\partial J}{\partial \theta} = \left( \frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots, \frac{\partial J}{\partial \theta_n} \right)$$
$$\frac{\partial J}{\partial \theta_1} = \frac{J(\theta_1^+) - J(\theta_1^-)}{2\varepsilon}$$

```
def dj_debug(theta, x_b, y, epsilon=0.01):
    res = np.empty(len(theta));
    for i in range(len(theta)):
        theta_1 = theta.copy()
        theta_1[i] += epsilon
        theta_2 = theta.copy()
        theta_2[i] += epsilon
        res[i] = (j(theta_1, x_b, y) - j(theta_2, x_b, y)) / (2 * epsilon)
    return res
```

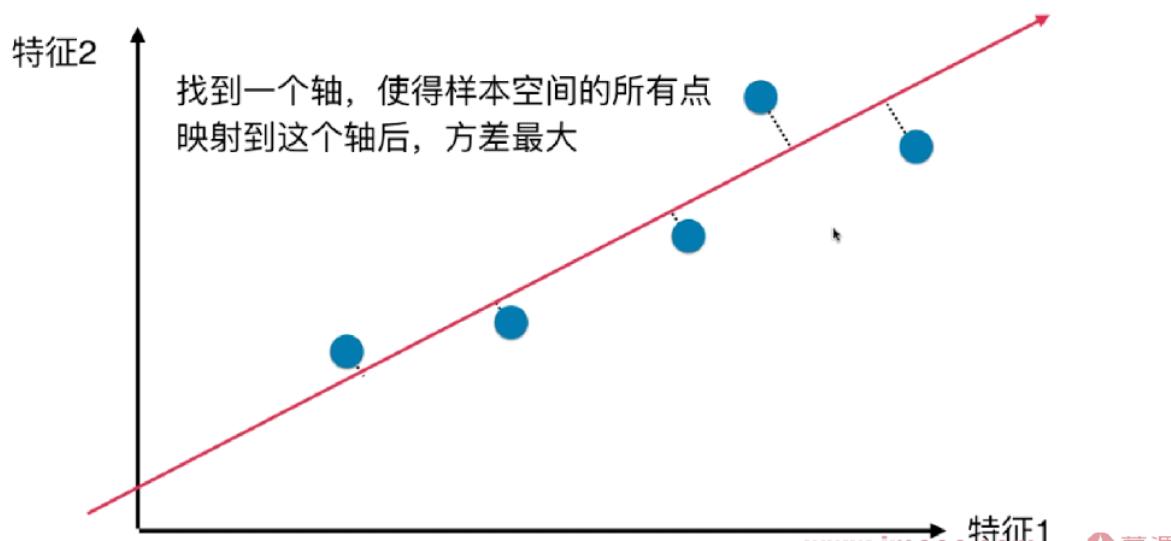
## PCA

Principal Component Analysis ----- 主成分分析

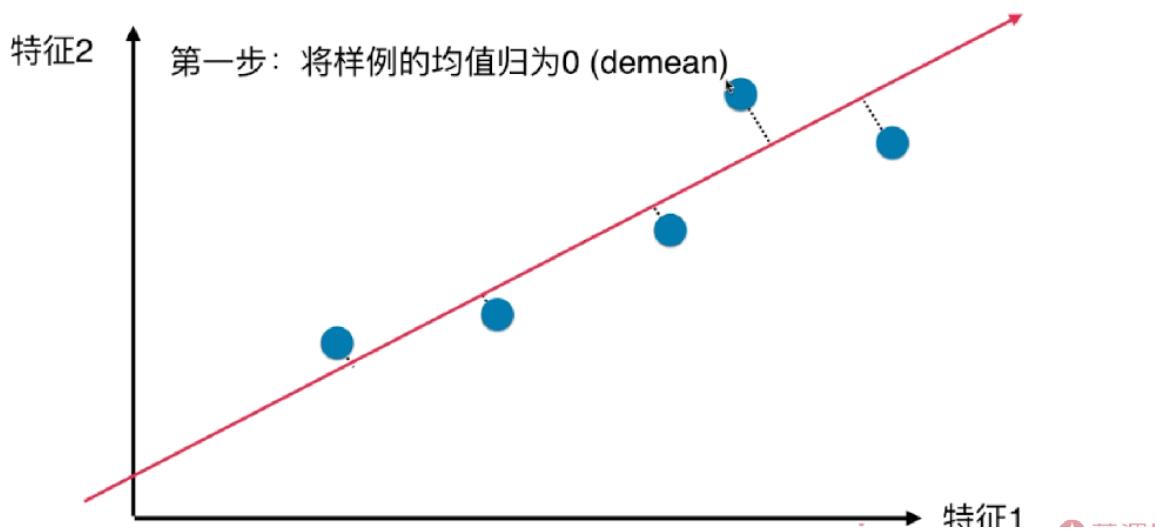
- 一个非监督的机器学习算法
- 主要用于数据的降维
- 通过降维，可以发现更便于人类理解的特征
- 其他应用：可视化；去噪

## PCA步骤

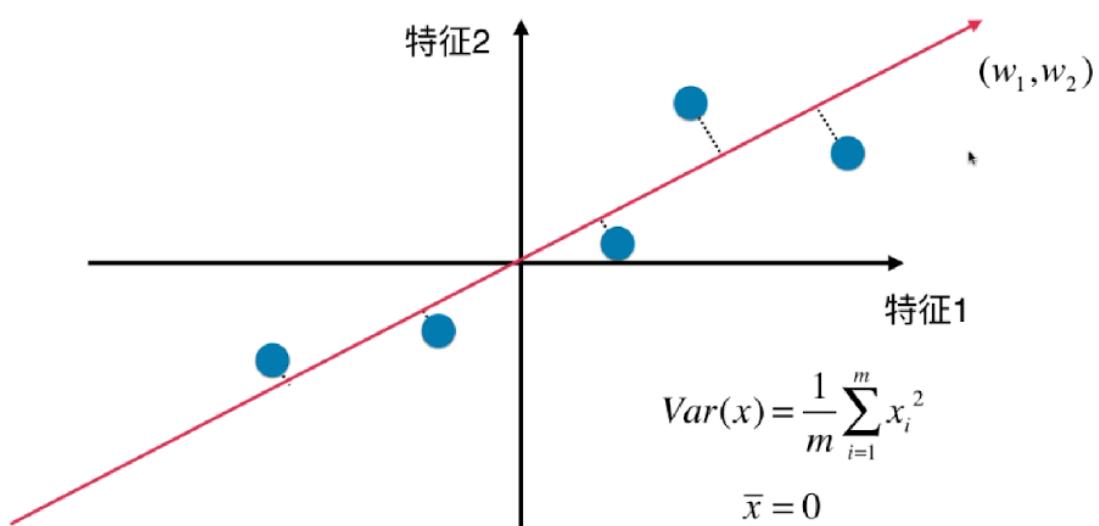
# 主成分分析



# 主成分分析



# 主成分分析



# 主成分分析

对所有的样本进行demean处理

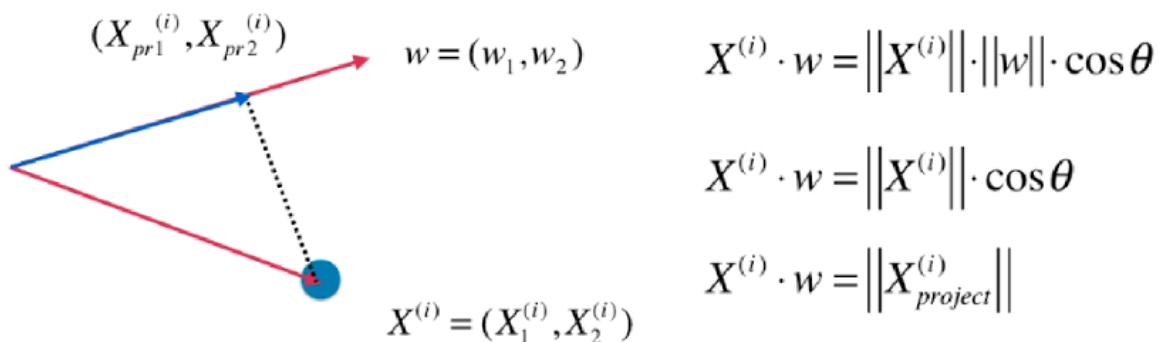
我们想要求一个轴的方向  $w = (w_1, w_2)$

使得我们所有的样本，映射到w以后，有：

$$Var(X_{project}) = \frac{1}{m} \sum_{i=1}^m \|X_{project}^{(i)} - \bar{X}_{project}\|^2 \text{ 最大}$$

# 主成分分析

$$Var(X_{project}) = \frac{1}{m} \sum_{i=1}^m \|X^{(i)} \cdot w\|^2 \text{ 最大}$$



# 主成分分析

目标：求 $w$ , 使得  $Var(X_{project}) = \frac{1}{m} \sum_{i=1}^m (X^{(i)} \cdot w)^2$  最大

$$Var(X_{project}) = \frac{1}{m} \sum_{i=1}^m (X_1^{(i)}w_1 + X_2^{(i)}w_2 + \dots + X_n^{(i)}w_n)^2$$

$$Var(X_{project}) = \frac{1}{m} \sum_{i=1}^m \left( \sum_{j=1}^n X_j^{(i)}w_j \right)^2$$

一个目标函数的最优化问题，使用梯度上升法解决

## 梯度上升法解决PCA问题

# 主成分分析

目标：求 $w$ , 使得  $f(X) = \frac{1}{m} \sum_{i=1}^m (X_1^{(i)}w_1 + X_2^{(i)}w_2 + \dots + X_n^{(i)}w_n)^2$  最大

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \dots \\ \frac{\partial f}{\partial w_n} \end{pmatrix} = \frac{2}{m} \begin{pmatrix} \sum_{i=1}^m (X^{(i)}w)X_1^{(i)} \\ \sum_{i=1}^m (X^{(i)}w)X_2^{(i)} \\ \dots \\ \sum_{i=1}^m (X^{(i)}w)X_n^{(i)} \end{pmatrix}$$

# 主成分分析

$$= \frac{2}{m} \begin{pmatrix} \sum_{i=1}^m (X^{(i)}w) X_1^{(i)} \\ \sum_{i=1}^m (X^{(i)}w) X_2^{(i)} \\ \dots \\ \sum_{i=1}^m (X^{(i)}w) X_n^{(i)} \end{pmatrix} \cdot \begin{pmatrix} \frac{2}{m} \cdot (X^{(1)}w, X^{(2)}w, X^{(3)}w, \dots, X^{(m)}w) \\ \begin{pmatrix} X_1^{(1)} & X_2^{(1)} & X_3^{(1)} & \dots & X_n^{(1)} \\ X_1^{(2)} & X_2^{(2)} & X_3^{(2)} & \dots & X_n^{(2)} \\ X_1^{(3)} & X_2^{(3)} & X_3^{(3)} & \dots & X_n^{(3)} \\ \dots & \dots & \dots & \dots & \dots \\ X_1^{(m)} & X_2^{(m)} & X_3^{(m)} & \dots & X_n^{(m)} \end{pmatrix} = \frac{2}{m} \cdot (Xw)^T \cdot X \\ \frac{2}{m} \cdot X^T (Xw) \end{pmatrix}$$

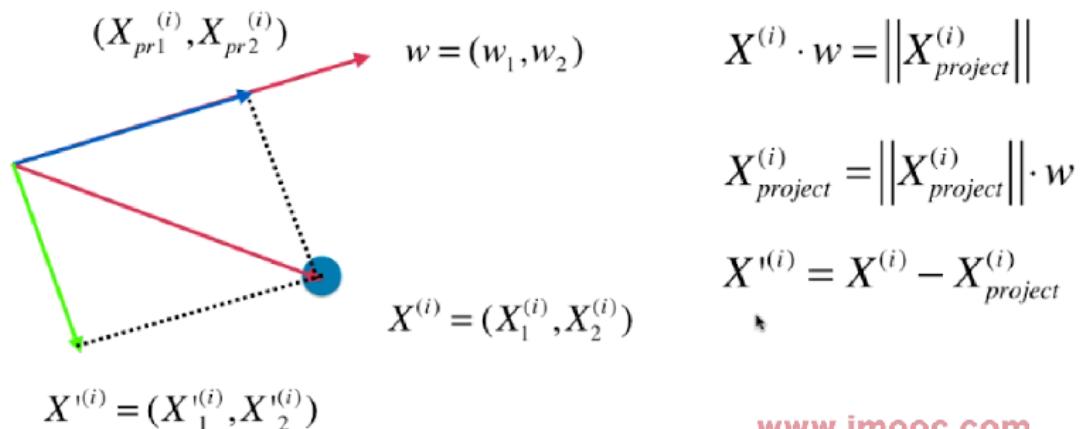
## 求数据的前n个主成分

求出第一主成分之后，如何求出下一个主成分

答：数据进行改变，将数据在第一个主成分上的分量去掉

# 主成分分析

数据进行改变，将数据在第一个主成分上的分量去掉



求出第一主成分以后，如何求出下一个主成分？

数据进行改变，将数据在第一个主成分上的分量去掉

在新的数据上求第一主成分

## 高维数据向低维数据映射

### 高维数据向低维数据映射

$$X = \begin{pmatrix} X_1^{(1)} & X_2^{(1)} & \dots & X_n^{(1)} \\ X_1^{(2)} & X_2^{(2)} & \dots & X_n^{(2)} \\ \dots & \dots & \dots & \dots \\ X_1^{(m)} & X_2^{(m)} & \dots & X_n^{(m)} \end{pmatrix} \quad W_k = \begin{pmatrix} W_1^{(1)} & W_2^{(1)} & \dots & W_n^{(1)} \\ W_1^{(2)} & W_2^{(2)} & \dots & W_n^{(2)} \\ \dots & \dots & \dots & \dots \\ W_1^{(k)} & X_2^{(k)} & \dots & X_n^{(k)} \end{pmatrix}$$

$$\underset{m \times n}{X} \cdot \underset{n \times k}{W_k^T} = \underset{m \times k}{X_k}$$

$$X_k = \begin{pmatrix} X_1^{(1)} & X_2^{(1)} & \dots & X_k^{(1)} \\ X_1^{(2)} & X_2^{(2)} & \dots & X_k^{(2)} \\ \dots & \dots & \dots & \dots \\ X_1^{(m)} & X_2^{(m)} & \dots & X_k^{(m)} \end{pmatrix} \quad W_k = \begin{pmatrix} W_1^{(1)} & W_2^{(1)} & \dots & W_n^{(1)} \\ W_1^{(2)} & W_2^{(2)} & \dots & W_n^{(2)} \\ \dots & \dots & \dots & \dots \\ W_1^{(k)} & X_2^{(k)} & \dots & X_n^{(k)} \end{pmatrix}$$

$$\underset{m \times k}{X_k} \cdot \underset{k \times n}{W_k} = \underset{m \times n}{X_m}$$

## sklearn中自带的PCA

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

# 加载数据
digits = load_digits()

# 获取样本矩阵和标签向量
X = digits.data
y = digits.target

# 进行训练集和测试集的分离
X_train, X_test, y_train, y_test = train_test_split(X, y)

# # 利用 explained_variance_ratio_ 查看降维的情况以此来选择降维数
# # 查看所有维度的降维覆盖率
# pca = PCA(n_components=X_train.shape[1])
# pca.fit(X_train)
# print(pca.explained_variance_ratio_)
#
# # 画出覆盖率曲线图
# plt.plot([i for i in range(X_train.shape[1])],
#          [np.sum(pca.explained_variance_ratio_[:i + 1])
#           for i in range(X_train.shape[1])])
# plt.show()

# 初始化 PCA 对象，其中 n_components 的值表示最后降维后的主成分个数
# pca = PCA(n_components=50)
# 初始化 PCA 对象，其中的小数表示方差覆盖的比率
pca = PCA(0.99)
# print(pca.n_components_)

# 进行拟合(先求出前n-2个主成分，然后把n维转化为2维，获得降维矩阵)
# 1. 梯度上升求一个主成分(demean均值归零，求方差最大)
# 2. 把数据在该主成分上的分量去掉(向量相减)
# 3. 直到剩余 n_components 个维度，获得降维矩阵
pca.fit(X_train)

# 获取降维结果(利用输入矩阵和降维矩阵进行点乘)
X_train_reduction = pca.transform(X_train)

# 获得测试数据的降维结果
X_test_reduction = pca.transform(X_test)

# 创建一个 kNN算法的分类器
kNN_classifier = KNeighborsClassifier()

# 开始进行拟合
kNN_classifier.fit(X_train_reduction, y_train)

# 计算准确率
print(kNN_classifier.score(X_test_reduction, y_test))
```

# 多项式回归

## sklearn自带的多项式回归和Pipeline

### PolynomialFeatures

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

x = np.random.uniform(-3, 3, size=100)
# 调整为列向量
X = x.reshape(-1, 1)
y = 0.5 * x ** 2 + x + 2 + np.random.normal(0, 1, 100)

# 进行多项式升维处理,degree表示升至的维度
poly = PolynomialFeatures(degree=2)
# 拟合
poly.fit(X)
# 调整至指定维数
X2 = poly.transform(X)

# 采用线性回归
reg = LinearRegression()

reg.fit(X2, y)

print(reg.coef_)
```

### Pipeline

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline

x = np.random.uniform(-3, 3, size=100)
# 调整为列向量
X = x.reshape(-1, 1)
y = 0.5 * x ** 2 + x + 2 + np.random.normal(0, 1, 100)

# 定义一个管道, 该管道内部为多项式回归执行的步骤
poly_reg = Pipeline([
    ("poly", PolynomialFeatures(degree=2)),
    ("std_scaler", StandardScaler()),
    ("lin_reg", LinearRegression())
])

# 进行拟合
poly_reg.fit(X, y)
```

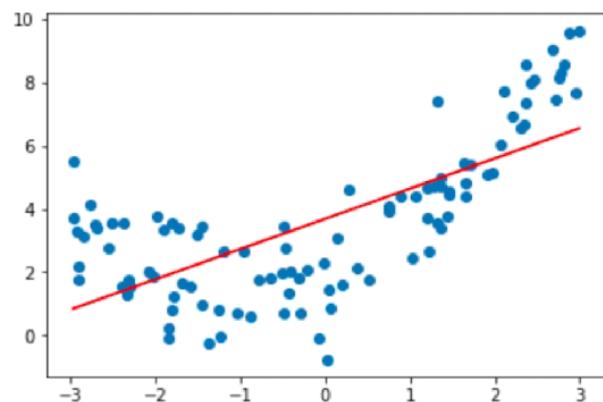
```
# 进行预测  
y_predict = poly_reg.predict(x)  
  
# 进行绘图  
plt.scatter(x, y)  
plt.plot(np.sort(x), y_predict[np.argsort(x)])  
plt.show()
```

## 过拟合与欠拟合（泛化能力）

### 欠拟合和过拟合

欠拟合 underfitting

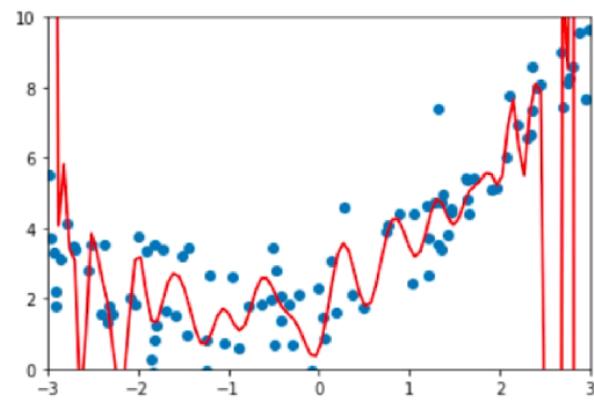
算法所训练的模型不能完整表述数据关系



### 欠拟合和过拟合

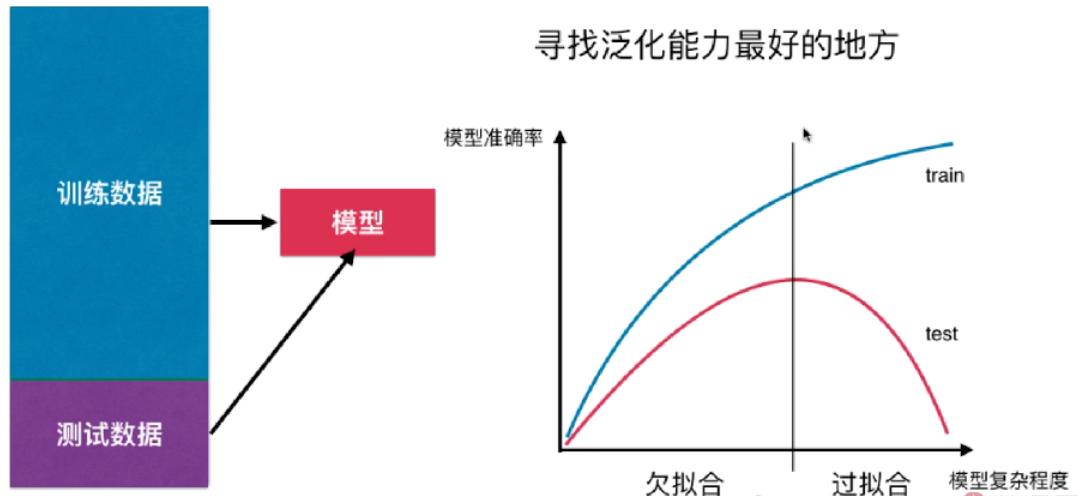
过拟合 overfitting

算法所训练的模型过多地表达了数据间的噪音关系



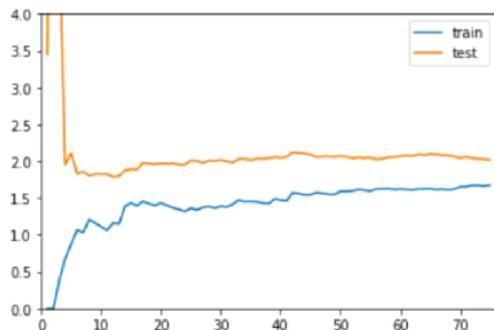
## 测试数据集的意义

# 模型复杂度曲线

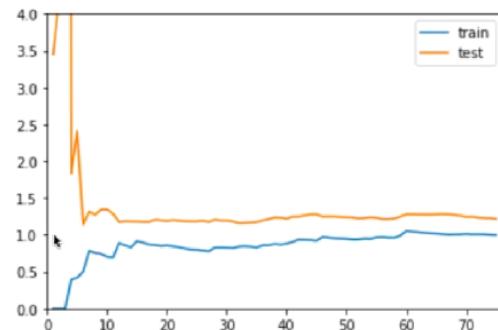


## 学习曲线

### 学习曲线

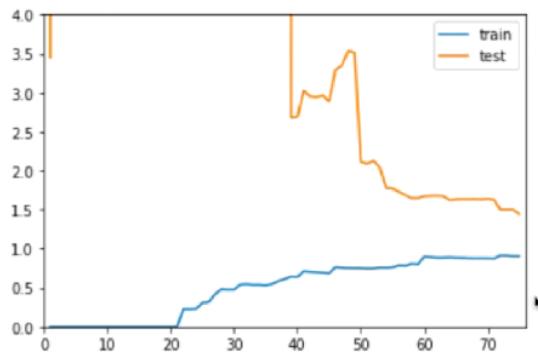


欠拟合

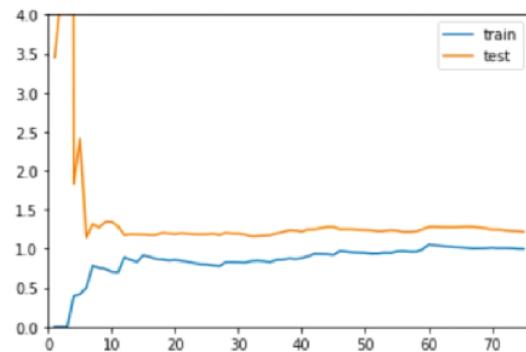


最佳

# 学习曲线



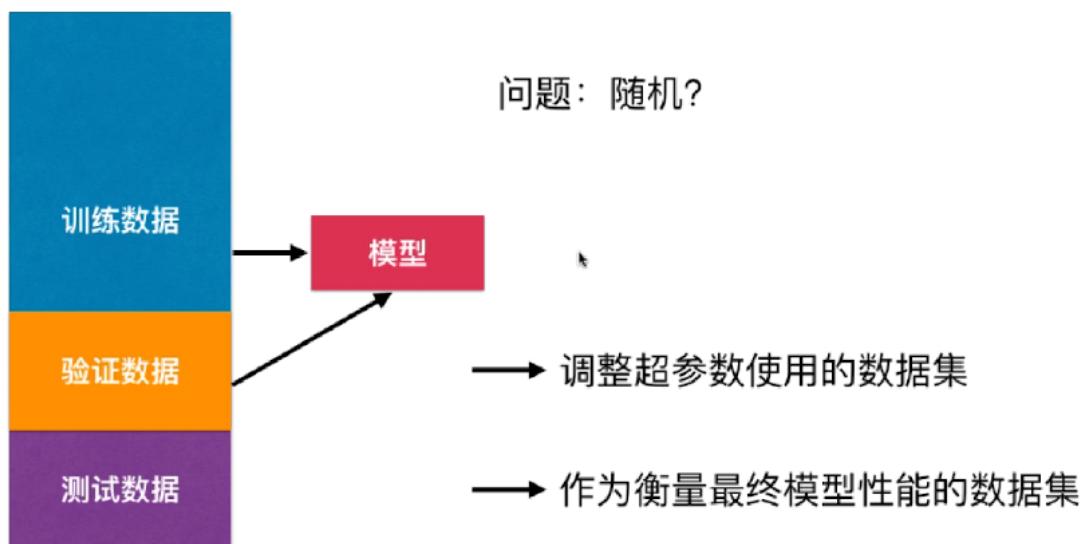
过拟合



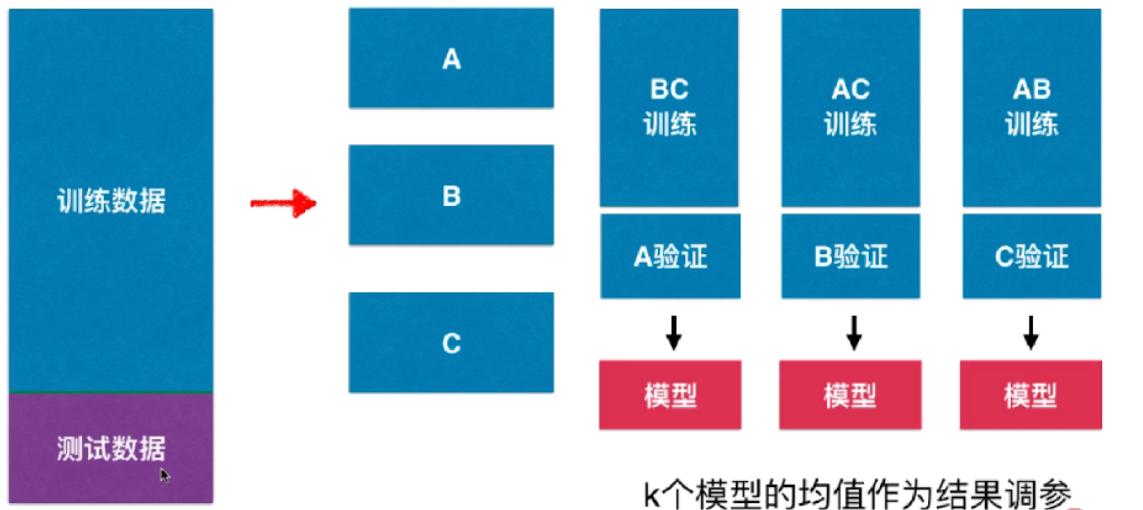
最佳

## 验证数据集与交叉验证

## 测试数据集的意义



# 交叉验证 Cross Validation



```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

# 加载数据
digits = load_digits()

# 获取样本矩阵和标签向量
X = digits.data
y = digits.target

# 进行训练集和测试集的分离
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)

# 使用交叉验证(其实和网格搜索中的GridSearchCV很接近)
best_score, best_p, best_k = 0, 0, 0
for k in range(2, 11):
    for p in range(1, 6):
        knn_clf = KNeighborsClassifier(weights="distance", n_neighbors=k, p=p)
        # 进行交叉验证(cv表示分组的数量)
        scores = cross_val_score(knn_clf, X_train, y_train, cv=5)
        # 取平均值作为最后结果
        score = np.mean(scores)
        if score > best_score:
            best_score, best_p, best_k = score, p, k
print(best_score, best_p, best_k)
```

## k-folds 交叉检验

# k-folds 交叉验证

把训练数据集分成k份，

称为k-folds cross validation

缺点，每次训练k个模型，相当于整体性能慢了k倍

## 留一法 LOO-CV

# 留一法 LOO-CV

把训练数据集分成m份，称为留一法

Leave-One-Out Cross Validation

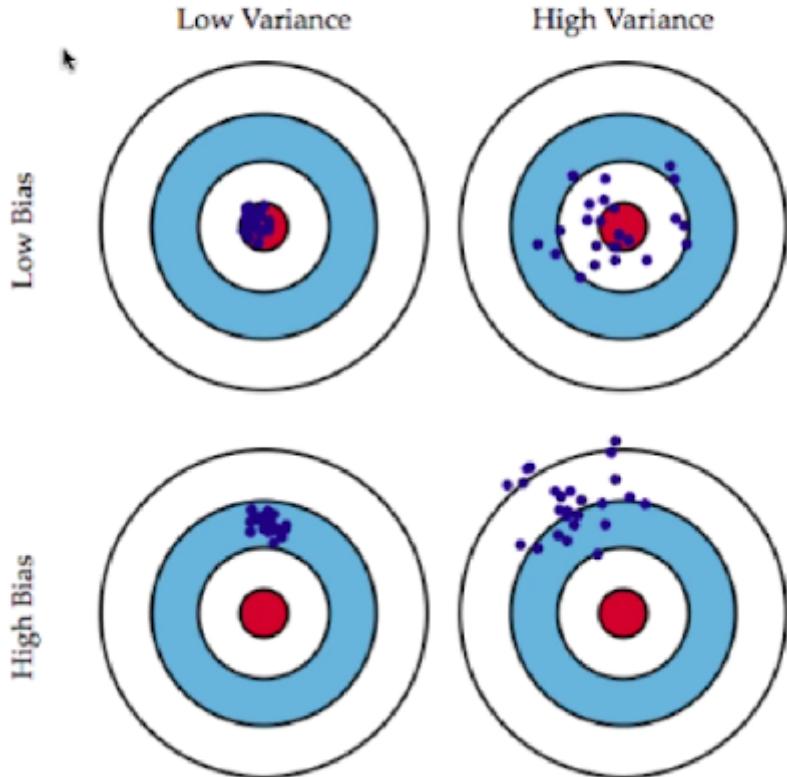
完全不受随机的影响，最接近模型真正的性能指标

缺点：计算量巨大

## 偏差方差权衡

Bias Variance Trade off

# 偏差和方差



## 偏差 (Bias)

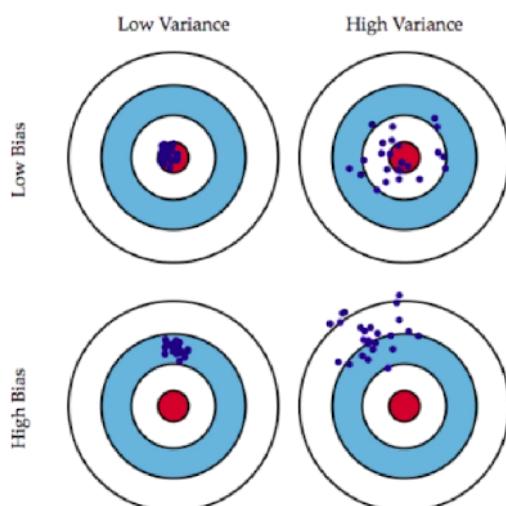
导致偏差的主要原因：

对问题本身的假设不正确！

如：非线性数据使用线性回归

i6331

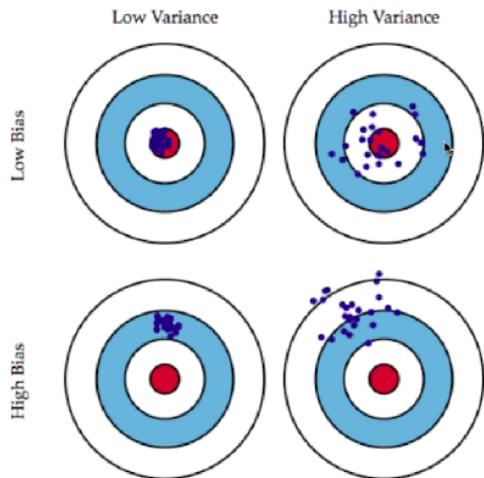
欠拟合 underfitting



# 方差 (Variance)

数据的一点点扰动都会  
较大地影响模型。

通常原因，使用的模型太复杂。  
如高阶多项式回归。



## 偏差和方差

有一些算法天生是高方差的算法。如kNN。

非参数学习通常都是高方差算法。因为不对数据进行任何假设

有一些算法天生是高偏差算法。如线性回归。

参数学习通常都是高偏差算法。因为堆数据具有极强的假设

# 偏差和方差

偏差和方差通常是矛盾的。

降低偏差，会提高方差。

降低方差，会提高偏差。

## 方差

机器学习的主要挑战，来自于方差！

解决高方差的通常手段：

- 1 降低模型复杂度
- 2 减少数据维度；降噪
- 3 增加样本数
- 4 使用验证集

# 模型泛化与岭回归（模型正则化）

岭回归：Ridge Regression

模型正则化：简化系数的大小

## 模型正则化 Regularization

目标：使  $\sum_{i=1}^m (y^{(i)} - \theta_0 - \theta_1 X_1^{(i)} - \theta_2 X_2^{(i)} - \dots - \theta_n X_n^{(i)})^2$  尽可能小

目标：使  $J(\theta) = MSE(y, \hat{y}; \theta)$  尽可能小



加入模型正则化，目标：使  $J(\theta) = MSE(y, \hat{y}; \theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$  尽可能小

### sklearn中自带的岭回归

```
import numpy as np
from matplotlib import pyplot as plt
from skimage.metrics import mean_squared_error
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline

x = np.random.uniform(-3, 3, size=100)
# 调整为列向量
X = x.reshape(-1, 1)
y = 0.5 * x ** 2 + x + 2 + np.random.normal(0, 1, 100)

# 定义一个岭回归
ridge_reg = Pipeline([
    ("poly", PolynomialFeatures(degree=20)),
    ("std_scaler", StandardScaler()),
    ("ridge_reg", Ridge(alpha=0.001))
])

# 进行拟合
ridge_reg.fit(x, y)

# 进行预测
y_predict = ridge_reg.predict(x)

# 查看方差
print(mean_squared_error(y, y_predict))

# 进行绘图
plt.scatter(x, y)
plt.plot(np.sort(x), y_predict[np.argsort(x)])
plt.show()
```

# LASSO

---