k-NN（k-近邻分类算法）

CREATED BY：彭奕伟

一、引言

kNN（k-Nearest Neighbor algorithm），又称为 k 近邻算法，是数据挖掘技术中原理最简单的算法之一。kNN的核心功能是解决有监督的分类问题，但也可以被用于回归问题中。作为惰性学习算法，kNN不产生模型，因此算法的准确性并不具备可推广性，但KNN能够快速高效地解决建立在特殊数据集上的预测分类问题，因此其具备非常广泛的使用情景。

kNN是一种**非概率模型**，从另一视角出发，它也是一种典型的**非参数模型**。

当我们无法假定存在一个在整个假设空间中有效的模型时，此时，我们只假定相似的输入具有相似的输出。相似的实例意味着相似的实物。正如我们都喜欢我们的"邻居"，因为他们太像我们。

于是，我们的算法使用合适的距离度量，从训练集中找出相似的实例，并且由它们插值，得到正确的输出。不同的非参数方法在定义相似性或由相似的训练实例插值方法方面不同。

不难看出，非参数的学习方法基于实例（instance-based）和记忆（memory-based），它们所要做的是把训练实例放在一个查找表中，并由它们进行插值。这意味需要存放所有的训练实例，而存放这些训练实例需要 N 存储量，而给定一个输入，需要找到相似的训练实例，这就需要 N 计算量，这种方法也称**惰性**学习方法。

非参数模型的一个很重要的特点就是：let the data speak for itself. 正因为如此，非参数模型的存储开销和计算开销都非常大。但是，由于不存在模型的错误假定问题，所以我们可以证明，当训练实例趋于无穷大的时候，非参数模型可以逼近任意复杂的真实模型。

二、k-NN算法

1. 算法原理

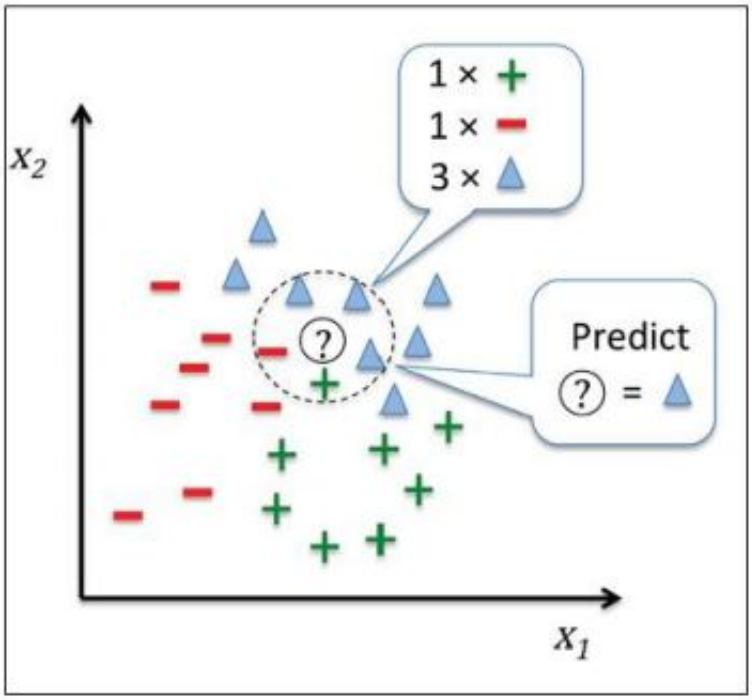
如下描述：一个数据集中存在多个已有标签的样本值，这些样本值共有的 n 个特征构成了一个多维空间 N 。当有一个需要预测 / 分类的样本 x 出现，我们把这个 x 放到多维空间 N 中，找到离其距离最近的 k 个样本，并将这些样本称为**近邻**（nearest neighbor)。对这 k 个近邻，查看他们的标签都属于何种类型，根据 ”少数服从多数，一点算一票” 的原则进行判断，数量最多的标签类别就是 x 的标签类型。其中涉及到的原理是 **“越相近越相似”**，这也是k**NN的基本假设**。

2. 核心思想

如果一个样本在特征空间中的 k 个最近邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。 kNN在类别决策时，只与极少量的相邻样本有关。由于kNN主要靠周围有限的邻近的样本，而不是靠判别类域的方法来确定所属类别的，因此对于类域的交叉或重叠较多的待

分样本集来说，kNN较其他方法更为适合。

3. 算法描述



- **算距离**

  给定测试对象 $Item$ ，计算它与训练集中每个对象的距离。

  依据公式计算 $Item$ 与 $D_1, D_2, \ldots D_j$ 之间的相似度，得到 $Sim(Item, D_1)$,$Sim(Item, D_2)$, ...,$Sim(Item, D_j)$.

- **找邻居**

  圈定距离最近的 k 个训练对象，作为测试对象的近邻。

  将$Sim(Item, D_1)$,$Sim(Item, D_2)$,...,$Sim(Item, D_j)$ 排序，若是超过相似度阈值 t ，则放入邻居集合 NN.

- **做分类**

  根据这 k 个近邻归属的主要类别，来对测试对象进行分类。

  自邻居集合 NN 中取出前 k 名，查看它们的标签，对这 k 个点的标签求和，以多数决，得到 $Item$ 可能类别。

三、k-NN模型

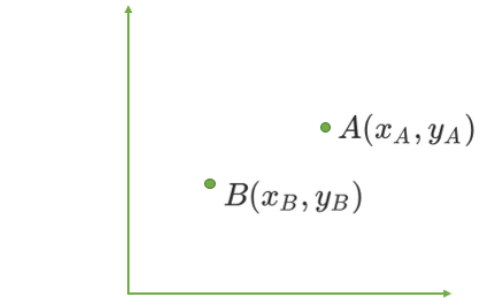
kNN模型的表示形式是整个数据集。除了对整个数据集进行存储之外，kNN没有其他模型。因此，kNN不具有显示的学习过程，在做分类时，对新的实例，根据其 k 个最近邻的训练实例的类别，通过多数表决等方式进行预测。k 近邻法实际上利用了训练数据集对特征向量空间进行划分，并作为其分类的 "模型" 。

1. 距离类模型中距离的确认

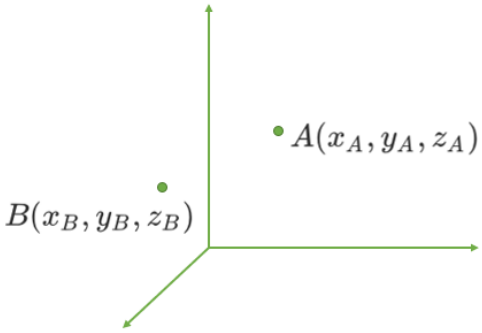
在距离类模型中，例如kNN中，有多种常见的距离衡量方法。大多数时候我们使用的是数据距离，而其中又以欧几里得距离为最常见。

1.1 欧几里得距离

对于数据之间的距离而言，在欧式空间中我们常常使用欧几里得距离，也就是我们常说的距离平方和开平方。回忆一下，一个平面直角坐标系上，如何计算两点之间的距离？一个立体直角坐标系上，又如何计算两点之间的距离？



$$d = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$



$$d = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2 + (z_A - z_B)^2}$$

在n维空间中，有两个点A和B，两点的坐标分别为：

$$A(x_{1A}, x_{2A}, x_{3A} \ldots\ldots x_{nA}), B(x_{1B}, x_{2B}, x_{3B} \ldots\ldots x_{nB})$$

则A和B两点之间的欧几里得距离的基本计算公式如下：

$$d(A,B)=\sqrt{(x_{1A}-x_{1B})^2+(x_{2A}-x_{2B})^2+\ldots+(x_{nA}-x_{nB})^2}=\sqrt{\sum_{i=1}^{n}((x_{iA}-x_{iB})^2)}$$

而在我们的机器学习中，坐标轴上的值 $x_1,x_2,\ldots x_n$ 正是我们样本数据上的n个特征。

1.2 曼哈顿距离

曼哈顿距离，也被称作街道距离。该距离用以标明两个点在标准坐标系上的绝对轴距总和，其计算方法相当于欧式距离的1次方，其基本计算公式如下：

$$d(A,B)=\sqrt[n]{\sum_{i=1}^{n}(|x_{iA}-y_{iB}|)^n}$$

当n=1时，就是曼哈顿距离；当n=2时，即为欧式距离；当n趋于无穷时，即为切比雪夫距离。

在kNN中，几乎默认使用欧几里得距离。

四、Python手动实现kNN

第一步：构建已经分类好的原始数据集

为了方便验证，这里使用Python的字典dict构建数据集，然后再将其转化成DataFrame格式。

```
import pandas as pd

rowdata={'电影名称':['无问西东','后来的我们','前任3','红海行动','唐人街探案','战狼2'],
         '打斗镜头':[1,5,12,108,112,115],
         '接吻镜头':[101,89,97,5,9,8],
         '电影类型':['爱情片','爱情片','爱情片','动作片','动作片','动作片']}
movie_data = pd.DataFrame(rowdata)

movie_data

# 探索数据，假如我们给与新数据[24,67]，你能够猜到这个新电影是什么类别么？
```

第二步：计算已知类别数据集中的点与当前点之间的距离

```
new_data = [24,67]
dist = list((((movie_data.iloc[:,1:3]-new_data)**2).sum(1))**0.5)
dist
```

第三步：将距离升序排列，然后选取距离最小的k个点

```
dist_l = pd.DataFrame({'dist': dist, 'labels': (movie_data.iloc[:, 3])})
dr = dist_l.sort_values(by = 'dist')[: 4]
dr
```

第四步：确定前k个点所在类别的加和

```
re = dr.loc[:,'labels'].value_counts()
re
```

第五步：选择出现次数最多的类别作为当前点的预测类别

```
result = []
result.append(re.index[0])
result
```

第六步：将上述过程封装成一个函数

```
import pandas as pd
"""
函数功能：KNN分类器
参数说明：
  new_data:需要预测分类的数据集
  dataSet:已知分类标签的数据集（训练集）
  k:k-近邻算法参数，选择距离最小的k个点
返回：
  result：分类结果
"""

def  classify0(inX,dataSet,k):
    result = []
    dist = list((((dataSet.iloc[:,1:3]-inX)**2).sum(1))**0.5)
    dist_l = pd.DataFrame({'dist':dist,'labels':(dataSet.iloc[:, 3])})
    dr = dist_l.sort_values(by = 'dist')[: k]
    re = dr.loc[:, 'labels'].value_counts()
```

```
    result.append(re.index[0])
    return result

# 测试函数的运行结果
inX = new_data
dataSet = movie_data
k = 3 # 试试看如果k=3?
classify0(inX,dataSet,k)
```

得到的结果与我们之前猜测的相似。在这个例子中，我们的数据仅有两个特征，因此计算非常简单，我们不用代码，即便手算也没有什么问题，然后当我们的数据有几十个甚至几百个特征的时候（这是工作中的常态，银行数据甚至动辄就上千个特征），我们就无法手动进行计算了，这时候代码的效力就体现出来。

五、使用scikit-learn实现kNN

我们已经手动实现了kNN算法，并且获得了意料之中的结果。但大家可能发现，在整个建模过程中，我们刚才的函数 只实现了算法原理这一小部分，还剩下一堆的：分训练集测试集，进行测试，评估模型等过程，都没有进行手写。实际上，如果这些代码全部都要手敲的话，整个kNN大概是需要200行左右的。作为机器学习中最简单的算法尚且如 此，更不要说原理复杂得多的算法们了，因此实际上，大多数时候我们是不会手敲代码的。相对的，我们使用自带一系列算法的算法库：Scikit-learn来帮助我们。

自2007年发布以来，Scikit-learn已经成为Python中重要的机器学习库了。Scikit-learn，简称sklearn，支持了包括分类、回归、降维和聚类四大机器学习算法，以及特征提取、数据预处理和模型评估三大模块。

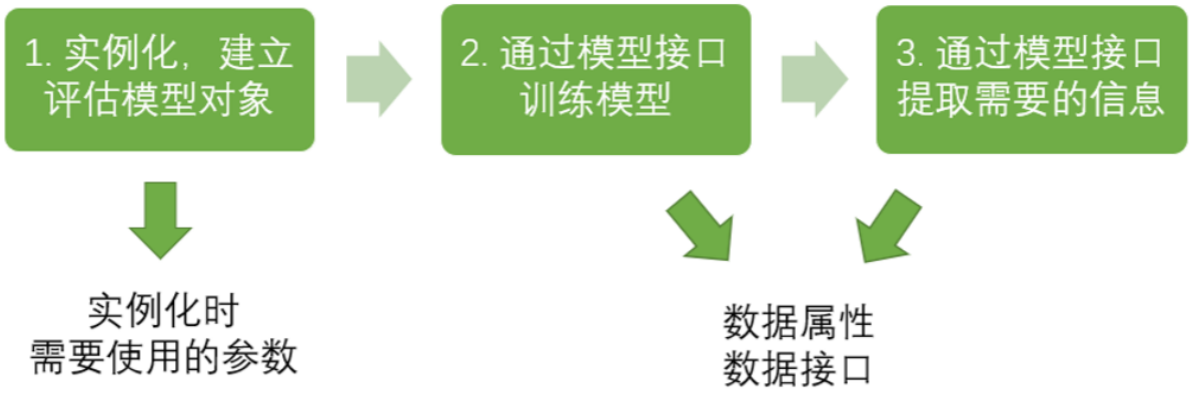
Scikit-learn 是基于 NumPy 与 SciPy 两大著名工具包，通常与 Pandas、Matplotlib 等开源数据处理框架合作，同时利用几大模块的优势，进行数据挖掘任务，大大提高了机器学习的效率。

http://scikit-learn.org/stable/index.html

在工程应用中，用Python手写代码来从头实现一个算法的可能性非常低，这样不仅耗时耗力，还不一定能够写出构架清晰，稳定性强的模型。更多情况下，是分析采集到的数据，根据数据特征选择适合的算法，在工具包中调用算法，调整算法的参数，获取需要的信息，从而实现算法效率和效果之间的平衡。而sklearn，正是这样一个可以帮助我们高效实现算法应用的工具包。 其实现代码的过程异常简单，就刚才我们所写的classify0函数，在sklearn中仅需要四行就可以完成。

1. sklearn的基本建模流程

我们先来了解一下sklearn建模的基本流程：



主要的设计原则如下：

- **一致性**：所有对象共享一个简单一致的界面(接口)。
  - 估算器：fit()方法。基于数据估算参数的任意对象，使用的参数是一个数据集(对应X, 有监督算法还需要一个y)，引导估算过程的任意其他参数称为超参数，必须被设置为实例变量。
  - 转换器：transform()方法。使用估算器转换数据集，转换过程依赖于学习参数。可以 使用便捷方式：fit_transform()，相当于先fit()再transform()。(fit_transform有时被优化过，速度更快)
  - 预测器：predict()方法。使用估算器预测新数据，返回包含预测结果的数据，还有 score()方法：用于度量给定测试集的预测效果的好坏。(连续y使用R方,分类y使用准确率accuracy)
- **监控**：检查所有参数，所有估算器的超参数可以通过公共实例变量访问，所有估算器的学习参数都可以通过有下划线后缀的公共实例变量访问。
- **防止类扩散**：对象类型固定，数据集被表示为Numpy数组或Scipy稀疏矩阵，超参是普通的Python字符或数字 。
- **合成**：现有的构件尽可能重用，可以轻松创建一个流水线Pipeline。
- **合理默认值**：大多数参数提供合理默认值，可以轻松搭建一个基本的工作系统。

Scikit-learn重视机器学习的逻辑性和系统性,所以有选择的忽略了某些算法中偏统计方 面的逻辑和结果,例如回归.

在这个流程下，kNN对应的代码是：

```
from sklearn.neighbors import KNeighborsClassifier  #导入需要的模块

clf = KNeighborsClassifier(n_neighbors=k)          #实例化
clf = clf.fit(X_train,y_train)                     #用训练集数据训练模型
result = clf.score(X_test,y_test)                  #导入测试集，从接口中调用需要的信息
```

具体来说，kNN算法在sklearn中通过下面这个类来实现：

class `sklearn.neighbors.KNeighborsClassifier`(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

其中参数n_neighbors就是我们的超参数k：最近的k个点。

我们可以试试看刚才的电影数据集：

```
from sklearn.neighbors import KNeighborsClassifier

# 假设爱情片为0，动作片为1
clf = KNeighborsClassifier(n_neighbors=3)
clf = clf.fit(movie_data.iloc[:,1:3],[0,0,0,1,1,1])
result = clf.predict([[24,67]]) # 获取结果

result

# 对模型进行一个评估，接口score返回预测的准确率
score = clf.score([[24,67]],[0])
```

让我们来试试更加复杂的数据集吧~

- **导入所需要的模块和库**

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

- **探索数据集**

```
data = load_breast_cancer()
data

X = data.data
y = data.target

X.shape
# 如果数据是一张表，就长这样 pd.DataFrame(X)

data.feature_names

name = ['平均半径','平均纹理','平均周长','平均面积',
        '平均光滑度','平均紧凑度','平均凹度',
        '平均凹点','平均对称','平均分形维数',
```

```
        '半径误差','纹理误差','周长误差','面积误差',
        '平滑度误差','紧凑度误差','凹度误差',
        '凹点误差','对称误差',
        '分形维数误差','最差半径','最差纹理',
        '最差的边界','最差的区域','最差的平滑度',
        '最差的紧凑性','最差的凹陷','最差的凹点',
        '最差的对称性','最差的分形维数']

y.shape

Xtrain,Xtest,Ytrain,Ytest = train_test_split(X,y,    # 特征和标签
                                             test_size=0.3)  # 测试集所占的比例
```

- **建立模型&评估模型**

```
clf = KNeighborsClassifier(n_neighbors=4)
clf = clf.fit(Xtrain,Ytrain)
score = clf.score(Xtest,Ytest)

score

# 找出一个数据点的最近邻（返回索引）
clf.kneighbors(Xtest[[30,20],:]
               ,return_distance=True) # 是否返回距离
```

如何实现一个算法？源码与sklearn之争

在机器学习的世界，总有着源码与调包之争。

调包的人不需要对算法理解得太过深刻，对背后的数学原理长久不用可能也会忘记，只需要调用他人写好的代来达成自己的目的，因此被写源码的人嘲笑嫌弃，业界甚至流传着“调包侠”的恶名，指那些只会调用其他人写好的代码，自己不写代码的人。

然而，这其实只是一种假象——真正自己写的，只有超巨大厂BAT，头条，小米等等这些公司，他们拥有巨大的技术团队，超强硬件支撑，并且拥有过于细致的需求，以至于市面上以“大而全”为目标的算法库无法满足他们的需要。而这些公司中真正有价值的机器学习工程师，每天都在忙于改变世界，根本不会混到初学者的圈子里来。

世界上几乎90%的算法公司都使用别人写好的包，因为自己创造算法不是那么简单的事儿。且不说如果自己写算法能否写得比市面上已经存在得包还快还稳定，一个算法不仅仅是要实现对结果的预测，它最终要最终部署到整个系统中，实现在产品上，是有非常长的一段路需要走的，其中包括了从后端数据库到前端产品的一系列过程，如果你的算法是自己写的，你的兼容性很难保证，除非公司的所有技术接口都是自己写的一套程序！而这对于世界上的大部分公司来说根本不可能，没那么闲，没那么多资源和时间。

2. 选取最优 k 值

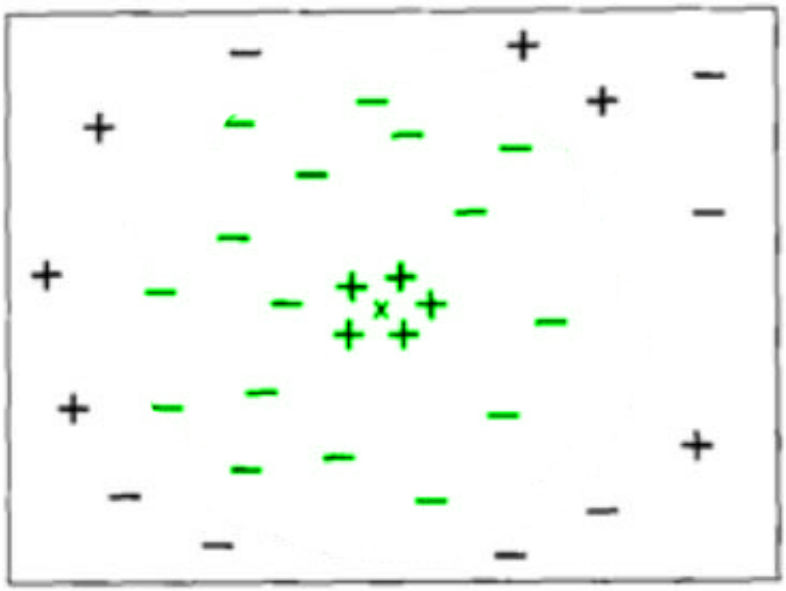
2.1 学习曲线

kNN中的 k 是一个超参数，所谓 "超参数"，就是需要人为输入，算法不能通过直接计算得出的参数。

kNN中的 k 代表的是距离需要分类的测试点 x 最近的 k 个样本点，如果不输入这个值，那么算法中重要部分 **"选出 k 个最近邻"** 就无法实现。从 kNN的原理中可见，是否能够确认合适的 k 值对算法有极大的影响。

如果选择的 k 值较小，就相当于较小的邻域中的训练实例进行预测，这时候只有与输入实例较近的（相似的）训练实例才会对预测结果起作用，但缺点是预测结果会对近邻的实例点非常敏感。如果邻近的实例点恰好是噪声，预测就会出错。换句话说，k 值的减小意味着整体模型变得复杂，容易发生过拟合。

相反地，如果选择的 k 值较大，就相当于较大的邻域中的训练实例进行预测。这时与输入实例较远的（不相似的）训练实例也会对预测起作用，使预测发生错误。k 值的增大意味着整体模型变得简单（如下图，k近邻由绿色表示）。因此，超参数 k 的选定是 kNN的头号问题。



- K值较小，则模型复杂度较高，容易发生过拟合，学习的估计误差会增大，预测结果对近邻的实例点非常敏感。
- K值较大可以减少学习的估计误差，但是学习的近似误差会增大，与输入实例较远的训练实例也会对预测起作用，使预测发生错误，k值增大模型的复杂度会下降。

在应用中，k值一般取一个比较小的值，通常采用交叉验证法来来选取最优的K值。

那我们怎样选择一个最佳的 k 呢？在这里我们要使用机器学习中的神器：**参数学习曲线**。参数学习曲线是一条以不同的参数取值为横坐标，不同参数取值下的模型结果为纵坐标的曲线，我们往往选择模型表现最佳点的参数取值作为这个参数的取值。比如我们的乳腺癌数据集：

```
# 更换不同的n_neighbors参数的取值，观察结果的变化
clf = KNeighborsClassifier(n_neighbors=4)
clf = clf.fit(Xtrain,Ytrain)
score = clf.score(Xtest,Ytest)

score
```

绘制学习曲线：

```
import matplotlib.pyplot as plt

score = []
krange = range(1,20) # 为什么这里是从1开始?

for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i)
    clf = clf.fit(Xtrain,Ytrain)
    score.append(clf.score(Xtest,Ytest))
plt.plot(krange,score);
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])
```

用这样的方式我们就找出了最佳的 k 值——这体现了机器学习当中，一切以 "模型效果" 为导向的性质，我们可以使用模型效果来帮助我们选择参数。

2.2 交叉验证

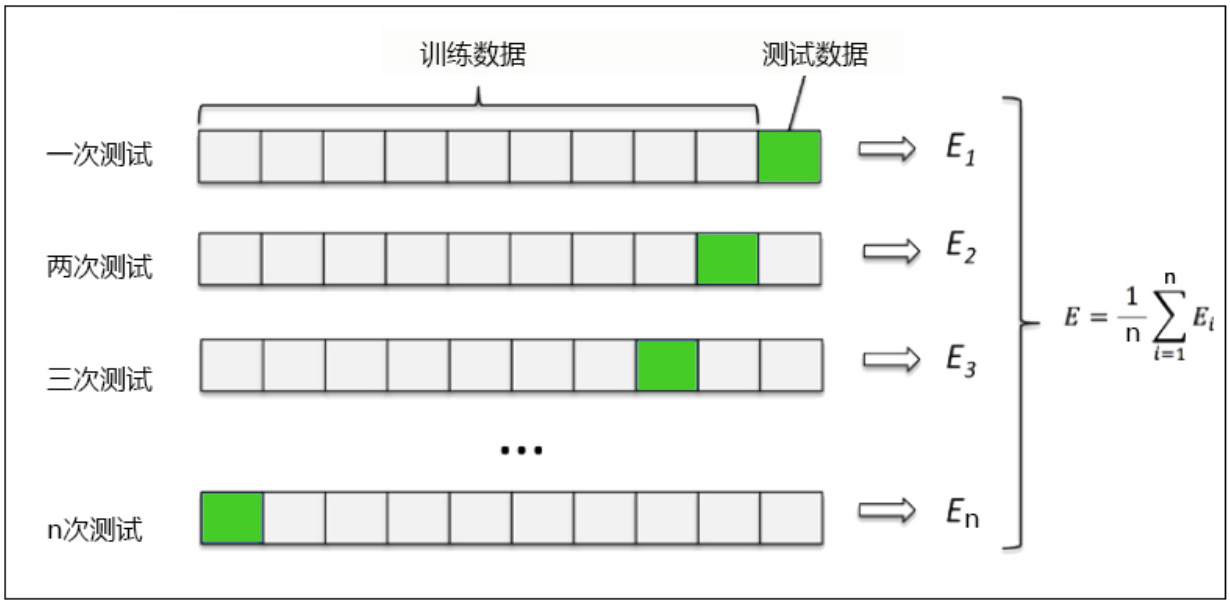
确定了 k 之后，我们还能够发现一件事，来看看这一段代码：

```
Xtrain,Xtest,Ytrain,Ytest = train_test_split(X,y,test_size=0.3)

score = []
krange = range(1,20)

for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i)
    clf = clf.fit(Xtrain,Ytrain)
    score.append(clf.score(Xtest,Ytest))
plt.plot(krange,score);
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])
```

你发现了什么？每次运行的时候学习曲线都在变化，模型的效果时好时坏，这是为什么呢？实际上，这是由于训练集合测试集的划分不同造成的。模型每次都使用不同的训练集进行训练，不同的测试集进行测试，自然也就会有不同的模型结果。在业务当中，我们的训练数据往往是已有的历史数据，但我们的测试数据却是新进入系统的一系列还没有标签的未知数据。我们的确追求模型的效果，但我们追求的是**模型在未知数据集上的效果，在陌生数据集上表现优秀的能力被称为泛化能力，即我们追求的是模型的泛化能力。**

我们认为，如果模型在一套训练集和数据集上表现优秀，那说明不了问题，只有在众多不同的训练集和测试集上都表现优秀，模型才是一个稳定的模型，模型才具有真正意义上的泛化能力。为此，机器学习领域有着发挥神作用的技能：**交叉验证**，来帮助我们认识模型。

最简单的交叉验证是k 折交叉验证。我们知道训练集和测试集的划分会干扰模型的结果，因此用交叉验证 n 次的结果求出的均值，是对模型效果的一个更好的度量。



在sklearn中，我们可以直接使用类`cross_val_score`来返回交叉验证得到的结果。

```
from sklearn.model_selection import cross_val_score as CVS

clf = KNeighborsClassifier(n_neighbors=3)
cvresult = CVS(clf                              # 首先输入已经实例化好的，还没有进行训练的模型
               ,X,y                             # 输入没有进行分割的数据集
               ,cv=5                            # 输入交叉验证的折数
              )

cvresult

# 均值：查看模型的平均效果
cvresult.mean()

# 方差：查看模型是否稳定
cvresult.var()

# 一个模型，要平均效果好，并且又稳定，才能有较高的泛化能力
```

绘制带交叉验证的学习曲线：

```
score = []
var_ = []
krange = range(1,20)

for i in krange:
```

```
    clf = KNeighborsClassifier(n_neighbors=i)
    cvresult = CVS(clf,X,y,cv=5)
    score.append(cvresult.mean())
    var_.append(cvresult.var())
plt.plot(krange,score,color="k")
plt.plot(krange,np.array(score)+np.array(var_)*2,c="red",linestyle="--")
plt.plot(krange,np.array(score)-np.array(var_)*2,c="red",linestyle="--")
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])
```

对于带交叉验证的学习曲线，我们需要观察的就不仅仅是最高的准确率了，而是**准确率高，方差还相对较小**的点，这样的点泛化能力才是最强的。在交叉验证+学习曲线的作用下，我们选出的超参数能够保证更好的泛化能力。

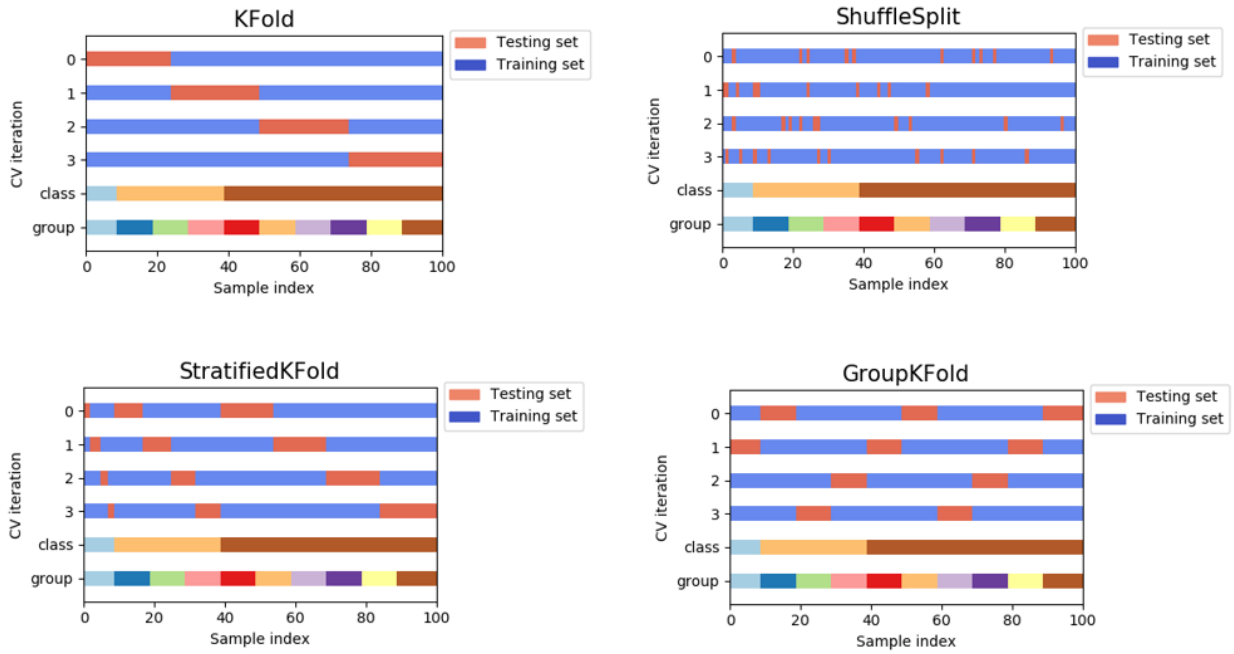
然而交叉验证却没有这么简单。交叉验证有许多坑大家可能会踩进去，在这里给大家列举出来：

1）最标准，最严谨的交叉验证应该有三组数据：训练集、验证集和测试集。当我们获取一组数据后，我们会先将数据集分成整体的训练集和测试集，然后我们把训练集放入交叉验证中，从训练集中分割更小的训练集（$k-1$ 份）和验证集（1份），此时我们返回的交叉验证结果其实是验证集上的结果。我们使用验证集寻找最佳参数，确认一个我们认为泛化能力最佳的模型，然后我们将这个模型使用在测试集上，观察模型的表现。

通常来说，我们认为经过验证集找出最终参数后的模型的泛化能力是增强了的，因此模型在未知数据（测试集）上的效果会更好，但尴尬的是，模型经过交叉验证在验证集上的调参之后，在测试集上的结果没有变好的情况时有发生。原因其实是：我们自己分的训练集和测试集，会影响模型的效果；同时，交叉验证后的模型的泛化能力增强了，表示它在未知数据集上方差更小，平均水平更高，但却无法保证它在现在分出来的测试集上预测能力最强。如此说来，是否有测试集的存在，其实意义不大了。

如果我们相信交叉验证的调整结果是增强了模型的泛化能力的，那即便测试集上的测试结果并没有变好（甚至变坏了），我们也认为模型是成功的。如果我们不相信交叉验证的调整结果能够增强模型的泛化能力，而一定要依赖测试集来进行判断，我们完全没有进行交叉验证的必要，直接用测试集上的结果来跑学习曲线就好了。所以，究竟是否需要验证集，其实是存在争议的，在严谨的情况下，大家还是使用有验证集的方式。

2）交叉验证的方法不止 "k 折"一种，分割训练集和测试集的方法也不止一种，分门别类的交叉验证占据了sklearn中非常长的一章：https://scikit-learn.org/stable/modules/cross_validation.html。所有的交叉验证都是在分割训练集和测试集，只不过侧重的方向不同，像 "k 折"就是按顺序取训练集和测试集，ShuffleSplit就侧重于让测试集分布在数据的全方位之内，StratifiedKFold则是认为训练数据和测试数据必须在每个标签分类中占有相同的比例。各类交叉验证的原理繁琐，大家在机器学习道路上一定会逐渐遇到更难的交叉验证，但是万变不离其宗：本质上交叉验证是为了解决训练集和测试集的划分对模型带来的影响，同时检测模型的泛化能力的。



3）k 折交叉验证对数据的分割方式是按顺序的，因此在使用交叉验证之前需要排查数据的标签本身是否有顺序，如果有顺序则需要打乱原有的顺序，或者更换交叉验证方法，像ShuffleSplit就完全不在意数据本身是否是有顺序的。

4）交叉验证的折数不可太大，因为折数越大抽出来的数据集越小，训练数据所带的信息量会越小，模型会越来越不稳定。如果你发现不使用交叉验证的时候模型表现很好，一使用交叉验证模型的效果就骤降，一定要查看你的标签是否有顺序，然后就是查看你的数据量是否太小，折数是否太高。

```
# 比如，100折交叉验证

score = []
var_ = []
krange = range(1,20)

for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i)
    cvresult = CVS(clf,X,y,cv=100)
    score.append(cvresult.mean())
    var_.append(cvresult.var())
plt.plot(krange,score,color="k")
plt.plot(krange,np.array(score)+np.array(var_)*2,c="red",linestyle="--")
plt.plot(krange,np.array(score)-np.array(var_)*2,c="red",linestyle="--")
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])

cvresult
```

发现了么？虽然对于现在的数据集而言，模型的效果没有下降，但是模型的方差却变得非常巨大。随意拆开一个交叉验证结果，你会发现模型的效果非常不稳定，根据数据点的不同从100%到70%的准确率都有可能，在实际使用时千万避免把折数设得太大。

使用交叉验证和学习曲线，可以帮助我们找到机器学习中大部分算法的最佳状态，它们也是机器学习算法调参中最基础的部分。

六、kNN中距离的相关讨论

在乳腺癌数据集上我们发现，模型的结果最好也就是大约93%左右，如果我们希望继续提升模型效果，怎么做呢？来看下面这一段代码：

```
from sklearn.preprocessing import MinMaxScaler as mms
X__ = mms().fit_transform(X)

score = []
var_ = []
krange = range(1,20)

for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i)
    cvresult = CVS(clf,X__,y,cv=5)
    score.append(cvresult.mean())
    var_.append(cvresult.var())
plt.plot(krange,score,color="k")
plt.plot(krange,np.array(score)+np.array(var_)*2,c="red",linestyle="--")
plt.plot(krange,np.array(score)-np.array(var_)*2,c="red",linestyle="--")
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])
```

你发现了什么？模型的效果很快就上升了，而这里只增加了两行代码：

```
from sklearn.preprocessing import MinMaxScaler as mms     # sklearn中专职归一化的类
X__ = mms().fit_transform(X)                              # 将数据集归一化
```

1. 距离类模型的归一化需求

什么是归一化呢？让我们把X放到数据框中来看一眼：

```
X_ = pd.DataFrame(X,columns=name)

X_.head()

X_.describe().T
```

你是否观察到，每个特征的均值差异很大？有的特征数值很大，有的特征数值很小，这种现象在机器学习中被称为 **"量纲不统一"**。kNN是距离类模型，欧氏距离的计算公式中存在着特征上的平方和：

$$d(A,B)=\sqrt{(x_{1A}-x_{1B})^2+(x_{2A}-x_{2B})^2+\ldots+(x_{nA}-x_{nB})^2}=\sqrt{\sum_{i=1}^{n}((x_{iA}-x_{iB})^2)}$$

试想看看，如果某个特征 x_i 的取值非常大，其他特征的取值和它比起来都不算什么，那距离的大小很大程度上都会由这个巨大特征 x_i 来决定，其他的特征之间的距离可能就无法对 $d(A,B)$ 的大小产生什么影响了，这种现象会让 kNN这样的距离类模型的效果大打折扣。然而在实际分析情景当中，绝大多数数据集都会存在各特征值量纲不同的情况，此时若要使用 kNN分类器，则需要先对数据集进行归一化处理，即是将所有的数据压缩都同一个范围内。

- **preprocessing.MinMaxScaler**

当数据（x）按照最小值中心化后，再按极差（最大值-最小值）缩放，数据移动了最小值个单位，并且会被收敛到[0，1]之间，而这个过程，就称作**数据归一化**（Normalization，又称Min-Max Scaling）。注意，Normalization是归一化，不是正则化，真正的正则化是regularization，不是数据预处理的一种手段。0-1归一化的公式如下：

$$x^*=\frac{x-min(x)}{max(x)-min(x)}$$

在sklearn当中，我们使用**preprocessing.MinMaxScaler**来实现这个功能。MinMaxScaler有一个重要参数，feature_range，控制我们希望把数据压缩到的范围，默认是[0，1]。

```
from sklearn.preprocessing import MinMaxScaler

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

# 如果换成表是什么样子?
import pandas as pd
pd.DataFrame(data)

# 实现归一化
scaler = MinMaxScaler()                     # 实例化
scaler = scaler.fit(data)                   # fit，在这里本质是生成min(x)和max(x)
result = scaler.transform(data)             # 通过接口导出结果
result

result_ = scaler.fit_transform(data)        # 训练和导出结果一步达成

# 也可以直接通过numpy来实现
```

```
import numpy as np
X = np.array([[-1, 2], [-0.5, 6], [0, 10], [1, 18]])

# 归一化
X_nor = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_nor
```

学会了归一化的类之后，来看看正确的归一化流程。最初的时候，为了让大家能够最快地看到模型的效果变化，这里直接在全数据集X上进行了归一化，然后放入交叉验证绘制学习曲线，这种做法是错误的，只是为了教学目的方便才这样操作。**真正正确的方式是，先分训练集和测试集，再归一化！**

为什么呢？想想看归一化的处理手段，我们是使用数据中的最小值和极差在对数据进行压缩处理，如果我们在全数据集上进行归一化，那最小值和极差的选取是会参考测试集中的数据的状况的。因此，当我们归一化后，无论我们如何分割数据，都会由一部分测试集的信息被 "泄露" 给训练集（当然，也有部分训练集的信息被泄露给了测试集，但我们不关心这个），这会使得我们的模型效果被高估。

在现实业务中，我们只知道训练集的数据，不了解测试集究竟会长什么样，所以我们要利用训练集上的最小值和极差来归一化测试集。正确的操作是这样的：

```
from sklearn.preprocessing import MinMaxScaler as mms

Xtrain,Xtest,Ytrain,Ytest =
train_test_split(X_,y,test_size=0.3,random_state=420)

MMS = mms().fit(Xtrain)                    # 这一步是在学习训练集，生成训练集上的极小值
和极差

Xtest_ = MMS.transform(Xtest)              # 用训练集上的极小值和极差归一化测试集
Xtrain_ = MMS.transform(Xtrain)            # 用训练集上的极小值和极差归一化训练集

score = []
var_ = []
krange = range(1,20)

# 严谨地使用了验证集
for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i)
    cvresult = CVS(clf,Xtrain_,Ytrain,cv=5)
    score.append(cvresult.mean())
    var_.append(cvresult.var())
plt.plot(krange,score,color="k")
plt.plot(krange,np.array(score)+np.array(var_)*2,c="red",linestyle="--")
plt.plot(krange,np.array(score)-np.array(var_)*2,c="red",linestyle="--")
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])

# 测试模型效果
```

```
clf = KNeighborsClassifier(n_neighbors=8).fit(Xtrain_,Ytrain)
score = clf.score(Xtest_,Ytest)

score
```

2. 以距离作为惩罚因子的优化

用最近邻点距离远近修正在对未知分类过程中，"一点一票" 的规则是 kNN模型优化的一个重要步骤。也就是说，对于原始分类模型而言，在选取最近的 k 个元素之后，将参考这些点的所属类别，并对其进行简单计数，而在计数的过程中这些点 "一点一票"，**这些点每个点对分类目标点的分类过程中影响效力相同。**但这实际上是不公平的，就算是最近邻的 k 个点，每个点的分类目标点的距离仍然有远近之别，而近的点往往和目标分类点有更大的可能性属于同一类别（该假设也是 kNN分类模型的基本假设）。因此，我们可以选择合适的惩罚因子，让入选的 k 个点在最终判别目标点属于某类别过程发挥的作用不相同，即让相对较远的点判别效力更弱，而相对较近的点判别效力更强。这一点也可以减少 kNN算法对 k 取值的敏感度。

关于惩罚因子的选取有很多种方法，最常用的就是根据每个最近邻 x_i 距离的不同对其作加权，加权方法为设置 w_i 权重，该权重计算公式为：

$$w_i = \frac{1}{d(x', x_i)}$$

这里需要注意的是，关于模型的优化方法只是在理论上而言进行优化会提升模型判别效力，但实际应用过程中最终能否发挥作用，本质上还是取决于优化方法和实际数据情况的契合程度，如果数据本身存在大量异常值点，则采用距离远近作为惩罚因子则会有较好的效果，反之则不然。因此在实际我们进行模型优化的过程当中，是否起到优化效果还是要以最终模型运行结果为准。

在sklearn中，我们可以通过参数`weights`来控制是否适用距离作为惩罚因子：

重要参数：weights

用于决定是否使用距离作为惩罚因子的参数，默认是 "uniform"
可能输入的值有：
"uniform"：表示一点一票
"distance"：表示以每个点到测试点的距离的倒数计算该点的距离所占的权重，使得距离测试点更近的样本点比离测试点更远的样本点具有更大的影响力

```
# 加上参数weights

score = []
var_ = []
krange = range(1,20)

for i in krange:
    clf = KNeighborsClassifier(n_neighbors=i,weights="distance")
    cvresult = CVS(clf,Xtrain_,Ytrain,cv=5)
```

```
    score.append(cvresult.mean())
    var_.append(cvresult.var())
plt.plot(krange,score,color="k")
plt.plot(krange,np.array(score)+np.array(var_)*2,c="red",linestyle="--")
plt.plot(krange,np.array(score)-np.array(var_)*2,c="red",linestyle="--")
bestindex = krange[score.index(max(score))]-1
print(bestindex)
print(score[bestindex])

# 测试模型效果
clf = KNeighborsClassifier(n_neighbors=8
                           ,weights="distance"
                          ).fit(Xtrain_,Ytrain)
score = clf.score(Xtest_,Ytest)

score
```

到这里，能够对 kNN进行的全部优化就已经完成了。kNN代表着"投票类"的算法，一直广泛受到业界的欢迎。不过kNN也有自己的缺点，那就是它的计算非常缓慢，因为kNN必须对每一个测试点来计算到每一个训练数据点的距离，并且这些距离点涉及到所有的特征，当数据的维度很大，数据量也很大的时候，kNN的计算会成为诅咒，大概几万数据就足够让kNN跑几个小时了。

k - NN	
算法功能	分类（核心），回归
算法类型	有监督学习，距离类模型
数据输入	包含数据标签y，且特征空间中至少包含 k 个训练样本 (k>=1) 作为距离类模型，需要对数据进行归一化处理 自定义的超参数 k (k>=1)
模型输出	在kNN分类中，输入是标签中的某个类别 在kNN回归中，输出的是对象的属性值，该值是距离输入的数据最近的k个训练样本的标签的平均值

七、模型评价

根据算法基本执行流程，我们可总结最近邻分类器的特点如下：

- 应用广泛：

  最近邻分类属于一类更广泛的技术，这种技术称为基于实例的学习，它使用具体的训练实例进行预测，而不必维护源自数据的抽象（或模型）。基于实例的学习算法需要邻近性度量来确定实例间的相似性或距离，还需要分类函数根据测试实例与其他实例的邻近性返回测试实例的预测类标号。

- 计算效率低，耗费计算资源较大：

  像最近邻分类器这样的消极学习方法不需要建立模型，所以，学习的开销很大，因为需要逐个计算测试样例和训练样例之间的相似度。相反，积极学习方法通常花费大量计算资源来建立模型，模型一旦建立，分类测试样例就会非常快。

- 抗噪性较弱，对噪声数据（异常值）较为敏感：

  最近邻分类器基于局部信息进行预测，而决策树和基于规则的分类器则试图找到一个拟合整个输入空间的全局模型。正是因为这样的局部分类决策，最近邻分类器（k很小时）对噪声非常敏感。

- 模型不稳定，可重复性较弱：

  最近邻分类器可以生成任意形状的决策边界，这样的决策边界与决策树和基于规则的分类器通常所局限的直线决策边界相比，能提供更加灵活的模型表示。最近邻分类器的决策边界还有很高的可变性，因为它们依赖于训练样例的组合。增加最近邻的数目可以降低这种可变性。

- 需要进行归一化处理：

  除非采用适当的邻近性度量和数据预处理，否则最近邻分类器可能做出错误的预测。例如，我们想根据身高（以米为单位）和体重（以磅为单位）等属性来对一群人分类。属性高度的可变性很小，从1.5米到1.85米，而体重范围则可能是从90磅到250磅。如果不考虑属性值的单位，那么邻近性度量可能就会被人的体重差异所左右。