**导语**：scikit-learn是Python中一个功能非常齐全的机器学习库，本篇文章将介绍如何用scikit-learn来进行kNN分类计算。

作者：肖睿

编辑：宏观经济算命师

本文由JoinQuant量化课堂推出，本文的难度属于进阶（上），深度为level-1。

阅读本文之前请掌握 [kNN](https://www.joinquant.com/post/2227?f=study&m=math)（**level-1**）的知识。  
建议读者掌握 [kd树](https://www.joinquant.com/post/2843?tag=new)（**level-1**）的知识。  
  
不费话，

from sklearn import neighbors

开始吧。

**功能详解**

本篇中，我们讲解的是 scikit-learn 库中的 neighbors.KNeighborsClassifier，翻译为 k 最近邻分类功能，也就是我们常说的 kNN，k-nearest neighbors。首先进行这个类初始化：

neighbors.KNeighborsClassifier(n\_neighbors=5, weights=’uniform’, algorithm=’auto’, leaf\_size=30, p=2, metric=’minkowski’, metric\_params=None, n-jobs=1)

好多啊参数呀，真是的。来，一个一个讲。

n\_neighbors 就是 kNN 里的 k，就是在做分类时，我们选取问题点最近的多少个最近邻。

weights 是在进行分类判断时给最近邻附上的加权，默认的 'uniform' 是等权加权，还有 'distance' 选项是按照距离的倒数进行加权，也可以使用用户自己设置的其他加权方法。举个例子：假如距离询问点最近的三个数据点中，有 1 个 A 类和 2 个 B 类，并且假设 A 类离询问点非常近，而两个 B 类距离则稍远。在等权加权中，3NN 会判断问题点为 B 类；而如果使用距离加权，那么 A 类有更高的权重（因为更近），如果它的权重高于两个 B 类的权重的总和，那么算法会判断问题点为 A 类。权重功能的选项应该视应用的场景而定。

algorithm 是分类时采取的算法，有 'brute'、'kd\_tree' 和 'ball\_tree'。kd\_tree 的算法在 kd 树文章中有详细介绍，而 ball\_tree 是另一种基于树状结构的 kNN 算法，brute 则是最直接的蛮力计算。根据样本量的大小和特征的维度数量，不同的算法有各自的优势。默认的 'auto' 选项会在学习时自动选择最合适的算法，所以一般来讲选择 auto 就可以。

leaf\_size 是 kd\_tree 或 ball\_tree 生成的树的树叶（树叶就是二叉树中没有分枝的节点）的大小。在 kd 树文章中我们所有的二叉树的叶子中都只有一个数据点，但实际上树叶中可以有多于一个的数据点，算法在达到叶子时在其中执行蛮力计算即可。对于很多使用场景来说，叶子的大小并不是很重要，我们设 leaf\_size=1 就好。

metric 和 p，是我们在 kNN 入门文章中介绍过的距离函数的选项，如果 metric ='minkowski' 并且 p=p 的话，计算两点之间的距离就是

d((x1,…,xn),(y1,…,yn))=(∑i=1n|xi−yi|p)1/p

d((x1,…,xn),(y1,…,yn))=(∑i=1n|xi−yi|p)1/pd((x1,…,xn),(y1,…,yn))=(∑i=1n|xi−yi|p)1/p

一般来讲，默认的 metric='minkowski'（默认）和 p=2（默认）就可以满足大部分需求。其他的 metric 选项可见[说明文档](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html)。metric\_params 是一些特殊 metric 选项需要的特定参数，默认是 None。

n\_jobs 是并行计算的线程数量，默认是 1，输入 -1 则设为 CPU 的内核数。

在创建了一个 KNeighborsClassifier 类之后，我们需要给它数据来进行学习。这时需要使用 fit() 拟合功能。

neighbors.KNeighborsClassifier.fit(X,y)

在这里：

X 是一个 list 或 array 的数据，每一组数据可以是 tuple 也可以是 list 或者一维 array，但要注意所有数据的长度必须一样（等同于特征的数量）。当然，也可以把 X 理解为一个矩阵，其中每一横行是一个样本的特征数据。

y 是一个和 X 长度相同的 list 或 array，其中每个元素是 X 中相对应的数据的分类标签。

KNeighborsClassifier 类在对训练数据执行 fit() 后会根据原先 algorithm 的选项，依据训练数据生成一个 kd\_tree 或者 ball\_tree。如果输入是 algorithm='brute'，则什么都不做。这些信息都会被保存在一个类中，我们可以用它进行预测和计算。几个常用的功能有：  
  
**k 最近邻**

neighbors.KNeighborsClassifier.kneighbors(X=None, n\_neighbors=None, return\_distance= True)

这里 X 是一 list 或 array 的坐标，如果不提供，则默认输入训练时的样本数据。

n\_neighbors 是指定搜寻最近的样本数据的数量，如果不提供，则以初始化 kNeighborsClassifier 时的 n\_neighbors 为准。

这个功能输出的结果是 (dist=array[array[float]], index=array[array[int]])。index 的长度和 X 相同，index[i] 是长度为 n\_neighbors 的一 array 的整数；假设训练数据是 fit(X\_train, y\_train)，那么 X\_train(index[i][j]) 是在训练数据（X\_train）中离 X[i] 第 j 近的元素，并且 dist[i][j] 是它们之间的距离。

输入的 return\_distance 是是否输出距离，如果选择 False，那么功能的输出会只有 index 而没有 dist。  
  
**预测**

neighbors.kNeighborsClassifier.predict(X)

也许是最常用的预测功能。输入 X 是一 list 或 array 的坐标，输出y是一个长度相同的 array，y[i] 是通过 kNN 分类对 X[i] 所预测的分类标签。  
  
**概率预测**

neighbors.kNeighborsClassifier.predict\_proba(X)

输入和上面的相同，输出 p 是 array[array[float]]，p[i][j] 是通过概率 kNN 判断 X[i] 属于第 j 类的概率。这里类别的排序是按照词典排序；举例来说，如果训练用的分类标签里有 (1,'1','a')三种，那么1就是第0类，'1' 是第1类，'a' 是第2类，因为在 Python 中 1<'1'<'a'。  
  
**正确率打分**

neighbors.KNeighborsClassifier.score(X, y, sample\_weight=None)

这是用来评估一次 kNN 学习的准确率的方法。很多可能会因为样本特征的选择不当或者 k 值得选择不当而出现过拟合或者偏差过大的问题。为了保证训练方法的准确性，一般我们会将已经带有分类标签的样本数据分成两组，一组进行学习，一组进行测试。这个 score() 就是在学习之后进行测试的功能。同 fit() 一样，这里的 X 是特征坐标，y 是样本的分类标签；sample\_weight 是对样本的加权，长度等于 sample 的数量。返回的是正确率的百分比。

**实际例子**

好，举例子了。

除了sklearn.neighbors，还需要导入numpy和matplotlib画图。

import random

from sklearn import neighbors

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

我们随机生成 6 组 200 个的正态分布

x1 = random.normal(50, 6, 200)

y1 = random.normal(5, 0.5, 200)

x2 = random.normal(30,6,200)

y2 = random.normal(4,0.5,200)

x3 = random.normal(45,6,200)

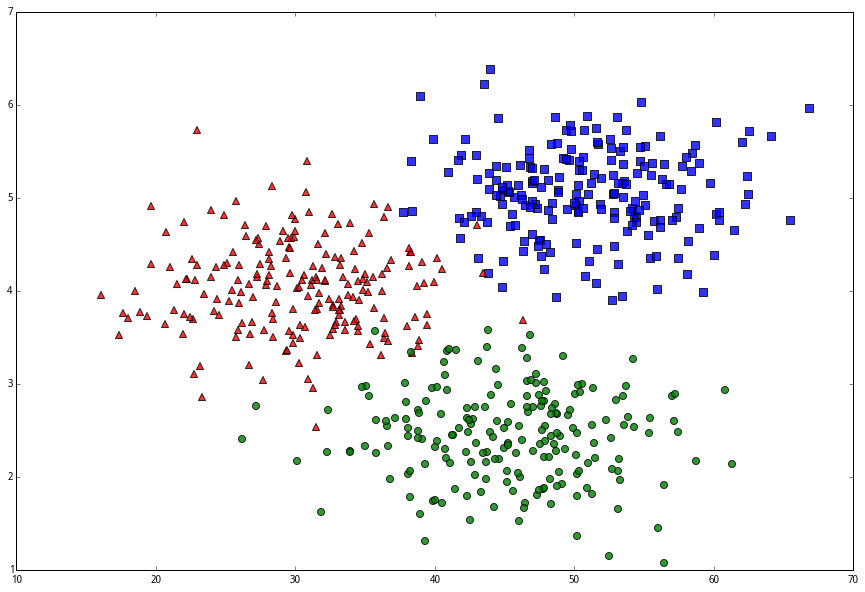
y3 = random.normal(2.5, 0.5, 200)

x1、x2、x3 作为 x 坐标，y1、y2、y3 作为 y 坐标，两两配对。(x1,y1) 标为 1 类，(x2, y2) 标为 2 类，(x3, y3)是 3 类。将它们画出得到下图，1 类是蓝色，2 类红色，3 类绿色。

plt.scatter(x1,y1,c='b',marker='s',s=50,alpha=0.8)

plt.scatter(x2,y2,c='r', marker='^', s=50, alpha=0.8)

plt.scatter(x3,y3, c='g', s=50, alpha=0.8)



我们把所有的 x 坐标和 y 坐标放在一起

x\_val = np.concatenate((x1,x2,x3))

y\_val = np.concatenate((y1,y2,y3))

记得计算距离的归一化问题吗？我们求出 x 值的最大差还有 y 值的最大差。

x\_diff = max(x\_val)-min(x\_val)

y\_diff = max(y\_val)-min(y\_val)

将坐标除以这个差以归一化，再将 x 和 y 值两两配对。

x\_normalized = [x/(x\_diff) for x in x\_val]

y\_normalized = [y/(y\_diff) for y in y\_val]

xy\_normalized = zip(x\_normalized,y\_normalized)

训练使用的特征数据已经准备好了，还需要生成相应的分类标签。生成一个长度600的list，前200个是1，中间200个是2，最后200个是3，对应三种标签。

labels = [1]\*200+[2]\*200+[3]\*200

然后，就要生成 sklearn 的最近 k 邻分类功能了。参数中，n\_neighbors 设为 30，其他的都使用默认值即可。

clf = neighbors.KNeighborsClassifier(30)

（注意我们是从sklearn里导入了neighbors。如果是直接导入了sklearn，应该输入sklearn.neighbors.KNeighborsClassifier()）

下面就要进行拟合了。归一化的数据是 xy\_normalized，分类标签是 labels，

clf.fit(xy\_normalized, labels)

就这么简单。下面我们来实现一些功能。  
  
**k 最近邻**

首先，我们想知道 (50,5) 和 (30,3) 两个点附近最近的 5 个样本分别都是什么。啊，坐标别忘了除以 x\_diff 和 y\_diff 来归一化。

nearests = clf.kneighbors([(50/x\_diff, 5/y\_diff),(30/x\_diff, 3/y\_diff)], 10, False)

nearests

得到

array([[ 97, 134, 177, 144, 10], [278, 569, 242, 324, 504]])

也就是说训练数据中的第 97、134、177、144、10 个离 (50,5) 最近，第 278、569、242、324、504 个离 (30,3) 最近。  
  
**预测**

还是上面那两个点，我们通过 30NN 来判断它们属于什么类别。

prediction = clf.predict([(50/x\_diff, 5/y\_diff),(30/x\_diff, 3/y\_diff)])

prediction

得到

array([1, 2])

也就是说 (50,5) 判断为 1 类，而 (30,3) 是 2 类。  
  
**概率预测**

那么这两个点的分类的概率都是多少呢？

prediction\_proba = clf.predict\_proba([(50/x\_diff, 5/y\_diff),(30/x\_diff, 3/y\_diff)])

prediction\_proba

得到

array([[ 1. , 0. , 0. ], [ 0. , 0.8, 0.2]])

告诉我们，(50, 5) 有 100% 的可能性是 1 类，而 (30,3) 有 80% 是 2 类，20% 是3类。  
  
**准确率打分**

我们再用同样的均值和标准差生成一些正态分布点，以此检测预测的准确性。

x1\_test = random.normal(50, 6, 100)

y1\_test = random.normal(5, 0.5, 100)

x2\_test = random.normal(30,6,100)

y2\_test = random.normal(4,0.5,100)

x3\_test = random.normal(45,6,100)

y3\_test = random.normal(2.5, 0.5, 100)

xy\_test\_normalized = zip(np.concatenate((x1\_test,x2\_test,x3\_test))/x\_diff,\

np.concatenate((y1\_test,y2\_test,y3\_test))/y\_diff)

labels\_test = [1]\*100+[2]\*100+[3]\*100

测试数据生成完毕，下面进行测试

score = clf.score(xy\_test\_normalized, labels\_test)

score

得到预测的正确率是 97% 还是很不错的。

再看一下，如果使用 1NN 分类，会出现过拟合的现象，那么准确率的平分就变为…

clf1 = neighbors.KNeighborsClassifier(1)  
clf1.fit(xy\_normalized, labels)  
clf1.score(xy\_test\_normalized, labels\_test)

95%，的确是降低了。我们还应该注意，这里的预测准确率很高是因为训练和测试的数据都是人为按照正态分布生成的，在实际使用的很多场景中（比如，涨跌预测）是很难达到这个精度的。

**生成些漂亮的图**

说到 kNN 那当然离不开分类图，不过这一般是为了教学用的，毕竟只能展示两个维度的数据，超过三个特征的话就画不出来了。所以这部分内容只是本篇的附加部分，有兴趣的读者可以向下阅读。

首先我们需要生成一个区域里大量的坐标点。这要用到 np.meshgrid() 函数。给定两个 array，比如 x=[1,2,3] 和 y=[4,5]，np.meshgrid(x,y) 会输出两个矩阵

[112233]

[112233][123123]

和

[454545]

[454545][444555]

这两个叠加到一起得到六个坐标，

[(1,4)(1,5)(2,4)(2,5)(3,4)(3,5)]

[(1,4)(1,5)(2,4)(2,5)(3,4)(3,5)][(1,4)(2,4)(3,4)(1,5)(2,5)(3,5)]

就是以 [1,2,3] 为横轴，[4,5] 为竖轴所得到的长方形区间内的所有坐标点。

好，我们现在要生成 [1,80]x[1,7] 的区间里的坐标点，横轴要每 0.1 一跳，竖轴每 0.01 一跳。于是

xx,yy = np.meshgrid(np.arange(1,70.1,0.1), np.arange(1,7.01,0.01))

于是 xx 和 yy 都是 601 乘 691 的矩阵。还有，不要忘了除以 x\_diff 和 y\_diff 来将坐标归一化。

xx\_normalized = xx/x\_diff

yy\_normalized = yy/y\_diff

下面，np.ndarray.ravel() 功能可以把一个矩阵抻直成一个一维 array，把

[112233]

[112233][123123]

变成

[123123]

[123123][123123]

np.c\_() 又把两个 array 粘起来（类似于 zip），输入

[123123]

[123123][123123]

和

[444555]

[444555][444555]

输出

[142434152535]

[142434152535][123123444555]

或者理解为

{(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)}

{(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)}{(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)}

于是

coords = np.c\_[xx\_normalized.ravel(), yy\_normalized.ravel()]

得到一个 array 的坐标。下面就可以进行预测

Z = clf.predict(coords)

当然，Z 是一个一维 array，为了和 xx 还有 yy 相对应，要把Z的形状再转换回矩阵

Z = Z.reshape(xx.shape)

下面用 pcolormesh 画出背景颜色。这里，ListedColormap 是自己生成 colormap 的功能，#rrggbb 颜色的 rgb 代码。pcolormesh 会根据 Z 的值（1、2、3）选择 colormap 里相对应的颜色。pcolormesh 和 ListedColormap 的具体使用方法会在未来关于画图的文章中细讲。

light\_rgb = ListedColormap([ '#AAAAFF', '#FFAAAA','#AAFFAA'])

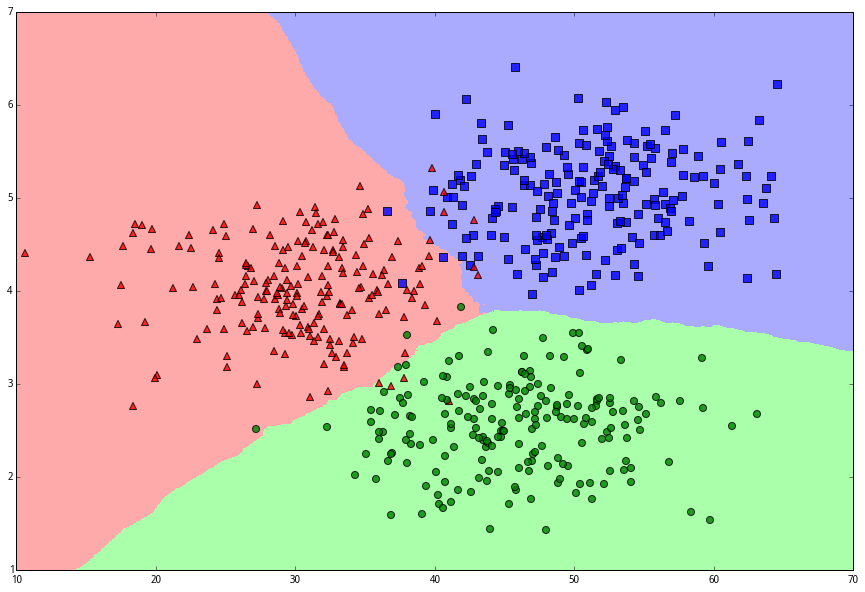
plt.pcolormesh(xx, yy,Z, cmap=light\_rgb)

plt.scatter(x1,y1,c='b',marker='s',s=50,alpha=0.8)

plt.scatter(x2,y2,c='r', marker='^', s=50, alpha=0.8)

plt.scatter(x3,y3, c='g', s=50, alpha=0.8)

plt.axis((10, 70,1,7))



下面再进行概率预测，使用

Z\_proba = clf.predict\_proba(coords)

得到每个坐标点的分类概率值。假设我们想画出红色的概率，那么提取所有坐标的 2 类概率，转换成矩阵形状

Z\_proba\_reds = Z\_proba[:,1].reshape(xx.shape)

再选一个预设好的红色调 cmap 画出来

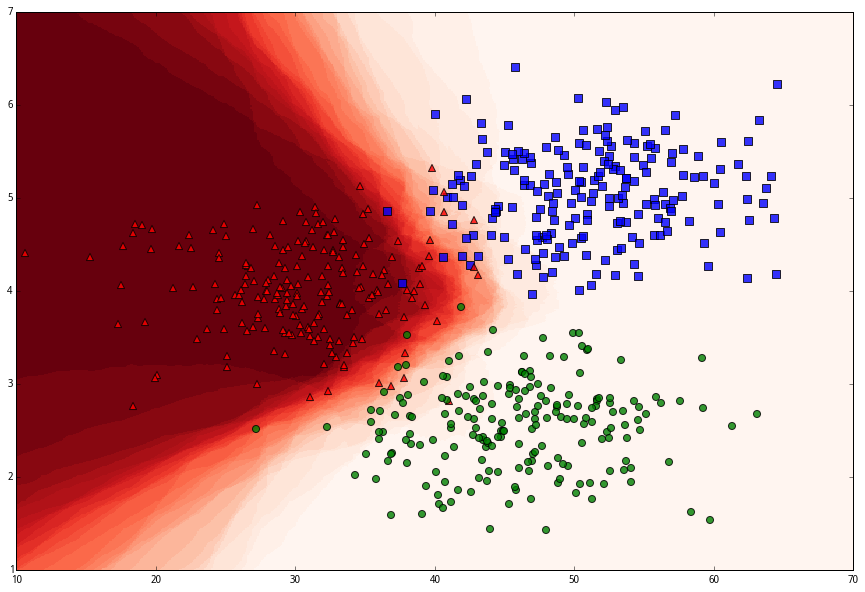
plt.pcolormesh(xx, yy,Z\_proba\_reds, cmap='Reds')

plt.scatter(x1,y1,c='b',marker='s',s=50,alpha=0.8)

plt.scatter(x2,y2,c='r', marker='^', s=50, alpha=0.8)

plt.scatter(x3,y3, c='g', s=50, alpha=0.8)

plt.axis((10, 70,1,7))



**结语**

scikit-learn 包的功能非常齐全，使用 kNN 分类进行预测也简单易懂。使用的难点在于将数据整理成函数可以处理的格式的过程偏于繁琐，从输出中读取结论可能也有些麻烦。本文细致地讲解了包中函数的输入、输出以及处理方法，希望读者可以轻松地将这些功能运用在实际应用中。