Task

- 了解kNN算法的思想及其原理
- 使用python手动实现kNN算法，并在sklearn中调用kNN算法
- 了解监督学习和非监督学习的概念

---

kNN算法的思想及其原理

全称：kNN(k-NearestNeighbor)，即K个最"近"的邻居算法

算法思想：找到最"近"的k个邻居，然后按照他们的多数作为**分类标签**，或者按照他们的均值作为**回归**预测结果（"投票法"和"平均法"），一些情况会考虑结合距离进行加权投票和加权平均，距离越近的权重越大。

目标函数： 没有，没有显示学习过程，所以也不需要优化，只需要验证K值是否合理（交叉验证）

算法模型：实际上是特征空间的划分。模型由三个基本要素决定：

```
*   距离度量
*    k值
*    分类决策规则
```

算法流程：

1. 计算测试对象到训练集中每个对象的距离
2. 按照距离的远近排序
3. 选取与当前测试对象最近的k的训练对象，作为该测试对象的邻居
4. 统计这k个邻居的类别频次
5. k个邻居里频次最高的类别，即为测试对象的类别

> 在一个给定的**类别已知**的**训练样本集**中，已知样本集中每一个数据与所属分类的对应关系（标签）。
>
> 在输入不含有标签的新样本后，将新的数据的每个特征与样本集中数据对应的特征进行比较，
>
> 根据算法提取样本最相似的k个数据(最近邻)的分类标签。

> 通过多数表决等方式进行预测。即选择k个最相似数据中出现次数最多的分类，作为新数据的分类。

K值选择

K：临近数，即在预测目标点时取几个临近的点来预测。

如果**当K的取值过小**时，一旦有噪声的成分存在，将会对预测产生比较大影响； 例如取K值为1时，一旦最近的一个点是噪声，那么就会出现偏差， K值的减小就意味着整体模型变得复杂，容易发生过拟合；

如果**当K的取值过大**时，就相当于用较大邻域中的训练实例进行预测，学习的近似误差会增大； 这时与输入目标点较远实例也会对预测起作用，使预测发生错误； K值的增大就意味着整体的模型变得简单；

如果**K==N**的时候，那么就是取全部的实例，即为取实例中某分类下最多的点，就对预测没有什么实际的意义了；

K的取值尽量要取奇数，以保证在计算结果最后会产生一个较多的类别， 如果取偶数可能会产生相等的情况，不利于预测。

常用的方法是从k=1开始，使用检验集估计分类器的误差率。 重复该过程，每次K增值1，允许增加一个近邻。选取产生最小误差率的K。

一般k的取值不超过20，上限是n的开方，随着数据集的增大，K的值也要增大。

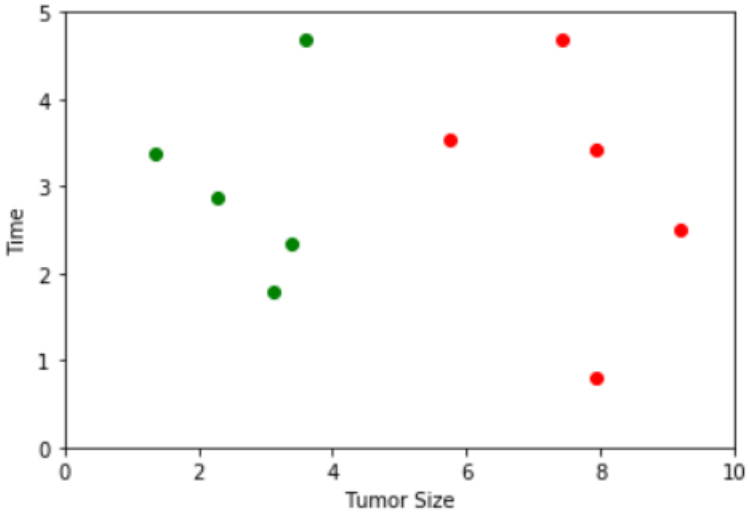
算法实现

1. 准备数据

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

# raw_data_x是特征，raw_data_y是标签，0为良性，1为恶性
raw_data_X = [[3.393533211, 2.331273381],
              [3.110073483, 1.781539638],
              [1.343853454, 3.368312451],
              [3.582294121, 4.679917921],
              [2.280362211, 2.866990212],
              [7.423436752, 4.685324231],
              [5.745231231, 3.532131321],
              [9.172112222, 2.511113104],
              [7.927841231, 3.421455345],
              [7.939831414, 0.791631213]
             ]
raw_data_y = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

# 设置训练组
X_train = np.array(raw_data_X)
y_train = np.array(raw_data_y)

# 将数据可视化
plt.scatter(X_train[y_train==0,0],X_train[y_train==0,1], color='g', label =
'Tumor Size')
plt.scatter(X_train[y_train==1,0],X_train[y_train==1,1], color='r', label =
'Time')
plt.xlabel('Tumor Size')
plt.ylabel('Time')
```

```
plt.axis([0,10,0,5])
plt.show()
```

数据可视化后生成的图片如下图所示。 其中横轴是肿块大小，纵轴是发现时间。 每个病人的肿块大小和发病时间构成了二维平面特征中的一个点。 对于每个点，我们通过label明确是恶性肿瘤（绿色）、良性肿瘤（红色）。



2. 求距离

求点x到数据集中每个点的距离（欧式距离：$\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$ ）

```
from math import sqrt

distances = []  # 用来记录x到样本数据集中每个点的距离
for x_train in X_train:
    d = sqrt(np.sum((x_train - x) ** 2))
    distances.append(d)

# 使用列表生成器，一行就能搞定，对于X_train中的每一个元素x_train都进行前面的运算，把结果生成一个列表
distances = [sqrt(np.sum((x_train - x) ** 2)) for x_train in X_train]

distances
```

输出：

```
[5.611968000921151,
 6.011747706769277,
 7.565483059418645,
 5.486753308891268,
 6.647709180746875,
 1.9872648870854204,
 3.168477291709152,
 0.8941051007010301,
 0.9830754144862234,
 2.7506238644678445]
```

在求出距离列表之后，我们要找到最小的距离，需要进行一次**排序操作**。其实不是简单的排序，因为我们把只将距离排大小是没有意义的，我们要知道距离最小的k个点是在样本集中的位置。

使用函数： `np.argsort(array)` 对一个数组进行排序，返回的是相应的排序后结果的索引

3. 选k值

然后我们选择k值，这里暂定为6，那就找出最近的6个点（top 6），并记录他们的标签值（y）

```
k = 6
topK_y = [y_train[i] for i in nearest[:k]]
topK_y

输出：
[1, 1, 1, 1, 1, 0]
```

4. 决策规则

找到与测试样本点最近的6个训练样本点的标签y是什么。可以查不同类别的点有多少个。

将数组中的元素和元素出现的频次进行统计

```
from collections import Counter
votes = Counter(topK_y)
votes

输出：一个字典，原数组中值为0的个数为1，值为1的个数为5
Counter({0:1, 1:5})

# Counter.most_common(n) 找出票数最多的n个元素，返回的是一个列表，列表中的每个元素是一个元组，元组中第一个元素是对应的元素是谁，第二个元素是频次
votes.most_common(1)

输出：
[(1,5)]

predict_y = votes.most_common(1)[0][0]
predict_y

输出：
1
```

得到预测的y值是1

完整代码

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

class kNNClassifier:

    def __init__(self, k):
        """初始化分类器"""
        assert k >= 1, "k must be valid"
        self.k = k
        self._X_train = None
        self._y_train = None
```

```
    def fit(self, X_train, y_train):
        """根据训练数据集X_train和y_train训练kNN分类器"""
        assert X_train.shape[0] == y_train.shape[0], \
            "the size of X_train must be equal to the size of y_train"
        assert self.k <= X_train.shape[0], \
            "the size of X_train must be at least k"
        self._X_train = X_train
        self._y_train = y_train
        return self

    def predict(self,X_predict):
        """给定待预测数据集X_predict，返回表示X_predict结果的向量"""
        assert self._X_train is not None and self._y_train is not None, \
            "must fit before predict!"
        assert X_predict.shape[1] == self._X_train.shape[1], \
            "the feature number of X_predict must be equal to X_train"
        y_predict = [self._predict(x) for x in X_predict]
        return np.array(y_predict)

    def _predict(self, x):
        distances = [sqrt(np.sum((x_train - x) ** 2)) for x_train in
self._X_train]
        nearest = np.argsort(distances)
        topK_y = [self._y_train[i] for i in nearest]
        votes = Counter(topK_y)
        return votes.most_common(1)[0][0]

    def __repr__(self):
        return "kNN(k=%d)" % self.k
```

当我们写完定义好自己的kNN代码之后，可以在jupyter notebook中使用魔法命令进行调用：

```
%run myAlgorithm/kNN.py

knn_clf = kNNClassifier(k=6)
knn_clf.fit(X_train, y_train)
X_predict = x.reshape(1,-1)
y_predict = knn_clf.predict(X_predict)
y_predict

输出：
array([1])
```

调用sklearn中的kNN

对于机器学习来说，其流程是：训练数据集 -> 机器学习算法 -fit-> 模型 输入样例 -> 模型 -predict-> 输出结果

kNN算法没有模型，模型其实就是训练数据集，predict的过程就是求k近邻的过程。

```
from sklearn.neighbors import KNeighborsClassifier

# 创建kNN_classifier实例
kNN_classifier = KNeighborsClassifier(n_neighbors=6)

# kNN_classifier做一遍fit(拟合)的过程，没有返回值，模型就存储在kNN_classifier实例中
```

```
kNN_classifier.fit(X_train, y_train)

# kNN进行预测predict，需要传入一个矩阵，而不能是一个数组。reshape()成一个二维数组，第一个参数是1表示只有一个数据，第二个参数-1，numpy自动决定第二维度有多少
y_predict = kNN_classifier.predict(x.reshape(1,-1))
y_predict

输出：
array([1])
```

在`kNN_classifier.fit(X_train, y_train)`这行代码后其实会有一个输出：

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=6, p=2,
           weights='uniform')
```

参数

```
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)
```

- `n_neighbors`: int，可选参数(默认为 5)。用于kneighbors查询的默认邻居的数量
- `weights`: 权重， str or callable(自定义类型), 可选参数(默认为 'uniform')。用于预测的权重参数，可选参数如下：
  - `uniform`: 统一的权重. 在每一个邻居区域里的点的权重都是一样的。
  - `distance`: 权重点等于他们距离的倒数。使用此函数，更近的邻居对于所预测的点的影响更大。
  - `[callable]`: 一个用户自定义的方法，此方法接收一个距离的数组，然后返回一个相同形状并且包含权重的数组。

- `algorithm`: {'auto', 'ball_tree', 'kd_tree', 'brute'}, 可选参数（默认为 'auto'）。计算最近邻居用的算法：
  - `balltree`
  - `kd_tree`
  - `brute`
  - `auto` 会基于传入fit方法的内容，选择最合适的算法。注意：如果传入fit方法的输入是稀疏的，将会重载参数设置，直接使用暴力搜索。

- `leaf_size`: int, 可选参数(默认为 30)。传入BallTree或者KDTree算法的叶子数量。此参数会影响构建、查询BallTree或者KDTree的速度，以及存储BallTree或者KDTree所需要的内存大小。此可选参数根据是否是问题所需选择性使用。

- `p` integer, 可选参数(默认为 2)。用于Minkowski metric（闵可夫斯基空间）的超参数。p = 1, 相当于使用曼哈顿距离，p = 2, 相当于使用欧几里得距离]，对于任何 p ，使用的是闵可夫斯基空间。

- `metric` string or callable, 默认为 'minkowski'。用于树的距离矩阵。默认为闵可夫斯基空间，如果和p=2一块使用相当于使用标准欧几里得矩阵. 所有可用的矩阵列表请查询 DistanceMetric 的文档。

- `metric_params` dict, 可选参数(默认为 None)。给矩阵方法使用的其他的关键词参数。

- `n_jobs` int, 可选参数(默认为 1)。用于搜索邻居的，可并行运行的任务数量。如果为-1, 任务数量设置为CPU核的数量。不会影响fit

`KneighborsClassifier`方法:

方法名	含义
`fit(X, y)`	使用X作为训练数据，y作为目标值（类似于标签）来拟合模型。
`get_params([deep])`	获取估值器的参数。
`neighbors([X,neighbors,return_distance])`	查找一个或几个点的K个邻居。
`kneighbors_graph([X,n_neighbors,mode])`	计算在X数组中每个点的k邻居的（权重）图。
`predict(X)`	给提供的数据预测对应的标签。
`predict_proba(X)`	返回测试数据X的概率估值。
`score(X,y[,sample_weight])`	返回给定测试数据和标签的平均准确值。
`set_params(**params)`	设置估值器的参数。

有监督学习与无监督学习

	有监督学习	无监督学习
样本	必须要有训练集与测试样本。在训练集中找规律，而对测试样本使用这种规律。	没有训练集，只有一组数据，在该组数据集内寻找规律。
目标	方法是识别事物，识别的结果表现在给待识别数据加上了标签。因此训练样本集必须由带标签的样本组成。	方法只有要分析的数据集的本身，预先没有什么标签。 如果发现数据集呈现某种聚集性，则可按自然的聚集性分类，但不予以某种预先分类标签对上号为目的。

总结

KNN算法是最简单有效的分类算法，简单且容易实现。当训练数据集很大时，需要大量的存储空间，而且需要计算待测样本和训练数据集中所有样本的距离，所以非常耗时

KNN对于随机分布的数据集分类效果较差，对于类内间距小，类间间距大的数据集分类效果好，而且对于边界不规则的数据效果好于线性分类器。

KNN对于样本不均衡的数据效果不好，需要进行改进。改进的方法时对k个近邻数据赋予权重，比如距离测试样本越近，权重越大。

KNN很耗时，时间复杂度为O(n)，一般适用于样本数较少的数据集，当数据量大时，可以将数据以树的形式呈现，能提高速度，常用的有kd-tree和ball-tree。

参考链接

机器学习的敲门砖：初探kNN算法

kNN算法：K最近邻(kNN，k-NearestNeighbor)分类算法

基于scikit-learn包实现机器学习之KNN(K近邻)