机器学习实践记录：从KNN算法到KD树

Author: Limzh

Section1. 序言

在模式识别中，k近邻算法是一个非参数统计学的的分类算法。KNN算法秉持着"物以类聚，人以群分"的自然产生的思想，不寻找数据的分布模式（这也是非参数的原因，无模型），而是通过数据集和预测点的距离关系输出预测点的标签。

官方文档的前言如下

> 最近邻方法背后的原理是从训练样本中找到与新点在距离上最近的预定数量的几个点，然后从这些点中预测标签。 这些点的数量可以是用户自定义的常量（K-最近邻学习）， 也可以根据不同的点的局部密度（基于半径的最近邻学习）确定。距离通常可以通过任何度量来衡量.Neighbors-based（基于邻居的）方法被称为 *非泛化* 机器学习方法， 因为它们只是简单地"记住"了其所有的训练数据（可能转换为一个快速索引结构，如 Ball Tree 或 KD Tree）。
>
> 尽管它简单，但最近邻算法已经成功地适用于很多的分类和回归问题，例如手写数字或卫星图像的场景。 作为一个 non-parametric（非参数化）方法，它经常成功地应用于决策边界非常不规则的分类情景下。

通过调用`sklearn`中的相关函数，我们可以直接窥得KNN算法的实现结果。

Section2. sklearn 中 KNN 算法

2.1 导入库介绍

```
import numpy as np
import pandas as pandas
import scipy
import matplotlib.pyplt as plt
from matplotlib.colors import ListedColormap
from scipy import stats
from sklearn import neighbors, datasets
from sklearn.datasets.samples_generator import make_classification
```

numpy，pandas，scipy 和 matplotlib 是数据处理和可视化的标准库. sklearn是python的机器学习算法库之一，全称scikit-learn。它集成了四大类机器学习算法，包括分类，回归，降维和聚类。同时它还支持生成算法需要的标数据集，数据的预处理以及数据的引入等功能。`make_classification`是数据生成器。`neighbors`包装了大量最近邻有关算法.

2.2 生成数据

```
X, y = make_classification(n_samples=200, n_features=2 ,n_redundant=0,
n_clusters_per_class=1, n_classes=3)
```

上面一行代码生成200个数据样例，每个样本有两个特征。标签值的取值范围为0，1，2，意味着所有点被分成三类。

2.3 创建待预测输入

```
# 2. generate to-be-classified data (predicted data)
# -- get the range of to-be-classified point
x_min, x_max = X[:,0].min() - 1, X[:,0].max() + 1
y_min, y_max = X[:,1].min() - 1, X[:,1].max() + 1
# -- mesh
h = .01 # step size for mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
# -- flatten and pair up
P = np.c_[xx.ravel(),yy.ravel()] # P is an vector of points
```

由于每个样本只具有两个特征，因此整个样本空间为二维平面，由于离样本集合太远的点没有太大的预测价值，因此我们仅关注在样本集合附近的点即可。故而，我们用`x_min, x_max, y_min, y_max`将样本集合所在的矩形圈定起来，在该矩形内的所有数据点构成的集合就是我们要预测的样本数据。在这里，我们使用`np.mesh()`函数生成网格点.（xx和yy的具体数值请自己尝试输出） 最后，得到P作为所有预测点的集合。P是一个一维数组，每一个元素是平面一点的坐标。至此，待预测数据已然完成。

2.4 导入分类器并拟合

```
# 3. generate classifier
clf = neighbors.KNeighborsClassifier(n_neighbors=15, weights='distance')
# 4. fit clf with trained-data
clf.fit(X, y)
```

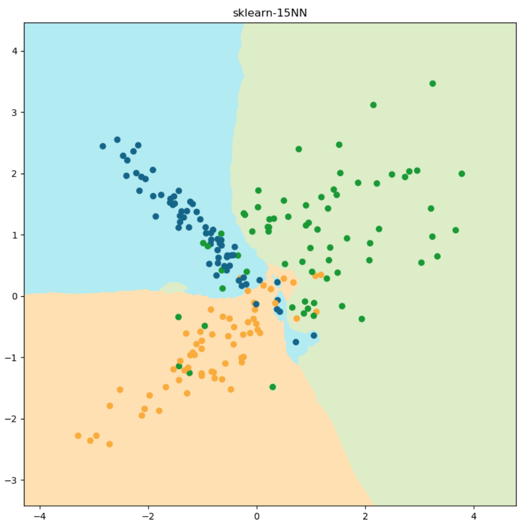
2.5 预测并将图绘制

```

# 5. predict P with fitted classifier and get the result Z full of prediected
value
Z = clf.predict(P) # note that it requires P is a vector of 1 dimension

# 6. plot the result。
# -- for scattered points (trained data)
color_bold = ListedColormap(['#156589', '#199934', '#F9AB3B'])

# -- for the predicted plane (to-be-classified data)
color_light = ListedColormap(['#B2EBF2','#DCEDC8','#FFE0B2'])


# note that the backgroud(color_light should be printed before to avoid
overlapping)
plt.pcolormesh(xx, yy, Z.reshape(xx.shape), cmap = color_light)
plt.scatter(X[:,0], X[:,1], c=y, cmap=color_bold)


# add title
plt.title('sklearn-15NN')
plt.show()
```

2.6 完整代码

```
#!/usr/bin/env python
# -*- coding:UTF-8 -*-
# AUTHOR: Minzhang Li
# FILE: F:\MyGithubs\Machine-Learning\Supervised-Learning\Nearest-
Neighbors\code\KNN_sklearn.py
# DATE: 2020/08/01 Sat
# TIME: 11:01:58

# DESCRIPTION: This file offers an example demonstrating how sklearn lib works.

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.datasets import  make_classification
from sklearn import  neighbors, datasets
from tqdm import tqdm

def main():
    # 1. generate samples. 200 samples for which are of two dims, 3-classes
labels. note that dont mess up 'class' with 'cluster'
    X, y = make_classification(n_samples=200, n_features=2 ,n_redundant=0,
n_clusters_per_class=1, n_classes=3)
    # 2. generate to-be-classified data (predicted data)
    # -- get the range of to-be-classified point
    x_min, x_max = X[:,0].min() - 1, X[:,0].max() + 1
    y_min, y_max = X[:,1].min() - 1, X[:,1].max() + 1
    # -- mesh
    h = .01 # step size for mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # -- flatten and pair up
    P = np.c_[xx.ravel(),yy.ravel()] # P is an vector of points
    # 3. generate classifier
    clf = neighbors.KNeighborsClassifier(n_neighbors=15, weights='distance')
    # 4. fit clf with trained-data
    clf.fit(X, y)
    # 5. predict P with fitted classifier and get the result Z full of
prediected value
    Z = clf.predict(P) # note that it requires P is a vector of 1 dimension
    # 6. plot the result.
    # -- for scattered points (trained data)
    color_bold = ListedColormap(['#156589', '#199934', '#F9AB3B'])
    # -- for the predicted plane (to-be-classified data)
    color_light = ListedColormap(['#B2EBF2','#DCEDC8','#FFE0B2'])
    # note that the backgroud(color_light should be printed before to avoid
overlapping)
    plt.pcolormesh(xx, yy, Z.reshape(xx.shape), cmap = color_light)
    plt.scatter(X[:,0], X[:,1], c=y, cmap=color_bold)
    # add title
    plt.title('sklearn-15NN')
    plt.show()


if __name__ == '__main__':
    main()
```

2.7 结果



Section3. 性能评估与超参数优化

在以上的实践中，我们做到了使用数据D在分类任务T中获得性能结果，但我们需要对该性能进行评估，并依照评估结果优化超参数K。因为当K过小，对噪音敏感，过拟合；K过大，则欠拟合。选取合适的K值对于分类器性能的好坏至关重要。

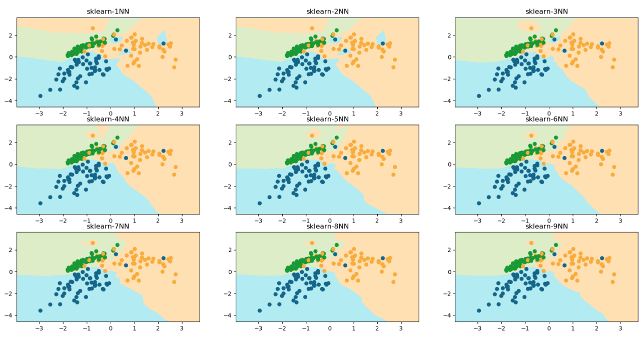
因此我们需要性能评估机制，引入测试集的概念进行超参数优化。

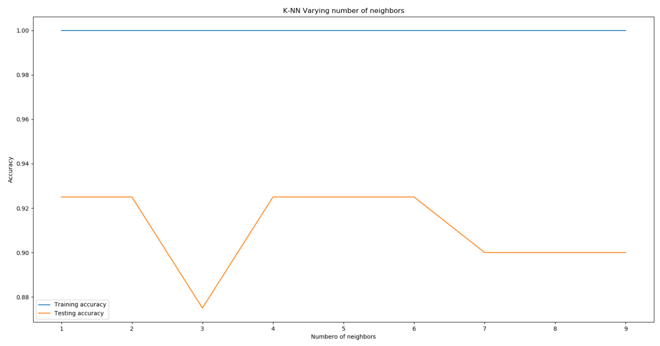
3.1 对不同K的模型的准确度进行评价

```
    # To find the best K we should test the result at test set:
    K = np.arange(1,10) # [1,10)
    test_accuracy = np.zeros(len(K))
    train_accuracy = np.zeros(len(K))
    res = []
    for i, k in enumerate(tqdm(K)):
        # 3. generate classifier
        clf = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance')
        # 4. fit clf with trained-data
        clf.fit(x_train, y_train)
        # 5. predict P with fitted classifier and get the result Z full of
prediected value
        Z = clf.predict(P) # note that it requires P is a vector of 1 dimension
        # push in res list
        res.append(Z)
        # evaluate
        train_accuracy[i] = clf.score(x_train, y_train)
        test_accuracy[i] = clf.score(x_test, y_test)
```

注: `model.score()` 的含义参见附录

3.2 结果





Section4. 动手实践: 蛮力版KNN算法

4.1 仿sklearn接口一览

我们需要实现一个分类器类，并模仿sklearn的风格为其创建接口。这样做是为了增加分类器的拓展性。

```
class KNeighborsClassifier_simple(object):
    def __init__(self, n_neighbors =15, weights = 'distance'):
        pass
    def predict(self, P):
        pass
    def fit(self, X, y):
        pass
    def score(X, y):
        pass
```

初始化的时候，需要传入权重属性和K两个超参数。`fit(X, y)`接受训练集的样本数据及其标签集， 在fit中进行拟合，学习模型。`predict(P)`接受待预测点集合P， 输出一个$1 \times len(P)$的标签集合作为结果. `score(X, y)`接受测试集，对相应超参数的情形下模型的精度进行评估。

除此之外，KNN算法的三大元素： 距离算法， 权重算法（分类决策规则）， K的选择中，距离算法和权重算法需要从`predict(self, P)`中剥离出来，以方便替换. 而 K的选择则通过简单交叉验证在主函数中得到即可。

故而，最后我们得到的完整接口列表是：

```
class KNeighborsClassifier_simple(object):
    def __init__(self, n_neighbors =15, weights = 'distance'): pass
    def __output(self, neighbors): pass
    def __distance(self, x, p): pass
    def __predict_point(self, p): pass
    def predict(self, P): pass
    def fit(self, X, y): pass
    def score(X, y): pass
    def score(X, y, Z): pass
```

在蛮力版KNN算法中，距离算法选用欧氏距离，分类决策规则使用多数投票规则加距离权重。而最重要的，在`__predict_point(self, p)`中，k neighbors的寻找采用蛮力排序算法。

4.2 完整代码

```
class KNeighborsClassifier_simple(object):
    def __init__(self, n_neighbors = 15, weights = 'distance'):
        self.n_neighbors = n_neighbors
        self.weights = weights
        # original data
        self.X = None
        self.y = None
        # combination of samples
        self.samples = None
        self.dim = None

    def __distance(self, x, p):
        '''distance algorithm'''
        res = 0
        for i in range(self.dim):
            res += (x[i] - p[i]) ** 2
        # n root
        return np.power(res, 1 / self.dim)

    def __output(self, neighbors):
        '''when K neighbors array already, output labels'''
        # given neigbors array, output predicted label
        n_labels = len(set(self.y))
        labels = np.zeros(n_labels)
        # count num based on distance
        for i in neighbors:
            if i[-1] != 0: labels[int(i[-2])] += 1/ i[-1]
            else: labels[int(i[-2])] += 1e9
        return np.argmax(labels)

    def __predict_point(self, p):
        '''predict one point'''
        # brute-forcely calculate distance
        for i in range(len(self.samples)):
            self.samples[i][-1] = self.__distance(self.samples[i], p)
        # sort and find k neighbors
        ord = list(self.samples.copy())
        ord.sort(key = lambda x: x[-1])
        ord = np.array(ord)
        neighbors = ord[:self.n_neighbors]
        # predict the res
        return self.__output(neighbors)

    def fit(self, X, y):
        assert len(X) == len(y) and len(X) != 0
        # X, y
        self.X = X
        self.y = y
        # get dim
        self.dim = len(X[0])
        self.samples = np.c_[X, y]
        distances = np.zeros(len(self.samples), dtype = float)
        # add one more dim in samples for distance
        self.samples = np.c_[self.samples, distances]
    def predict(self, P):
        res = np.zeros(len(P))
        for i, p in enumerate(P):
            res[i] = self.__predict_point(p)
        return res
    def score(self, X, y):
        '''score without predicted data'''
        y_predicted = self.predict(X)
        res = np.sum(y_predicted == y ) / len(y)
        return res
    def score(self, X, y, Z):
        '''score with predicted data'''
        y_predicted = Z
        res = np.sum(y_predicted == y) / len(y)
        return res
```

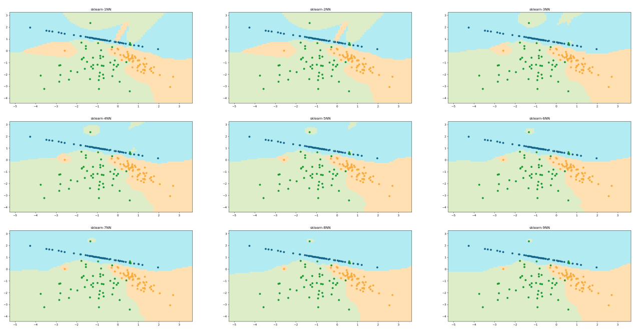
4.3 复杂度分析

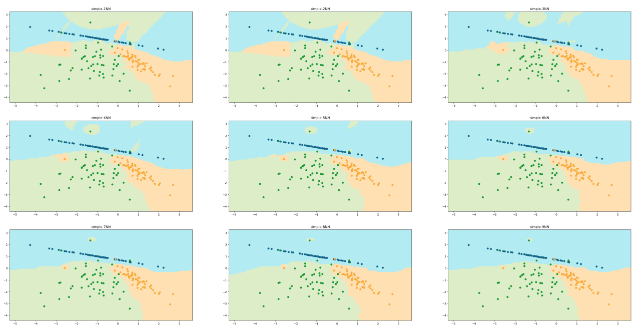
假设样本空间的维度是 r, 数据集的大小是 p, 预测输入的大小是 n. 那么对于预测输入的每一个预测点，都要对数据集的所有点进行一次距离计算，这意味着 $O(r \cdot n \cdot p)$的时间复杂度, 而对于每一个点，计算完和每一个数据集点的距离之后都要进行一次排序来找到k个邻居，这意味着 $O(n \cdot logn)$的时间复杂度。最后，决策分类规则要对K个邻居的标签进行处理，需要$O(n \cdot k)$的时间复杂度。所以，蛮力算法的时间复杂度为 $O(n \cdot (r \cdot p + n \cdot logn + k))$. 这意味着，至少需要 $O(n^2 logn)$ 。

4.4 结果比较

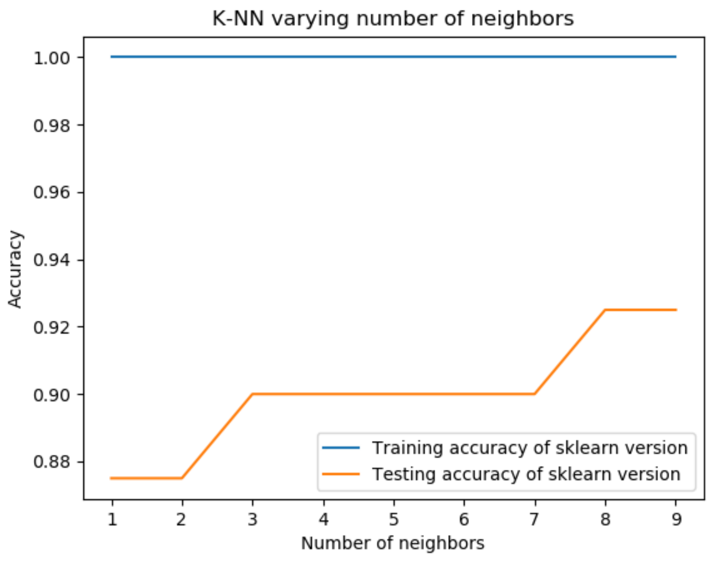
选取数据集 200 x 2， 待预测点集合规模10000左右, 有以下结果。

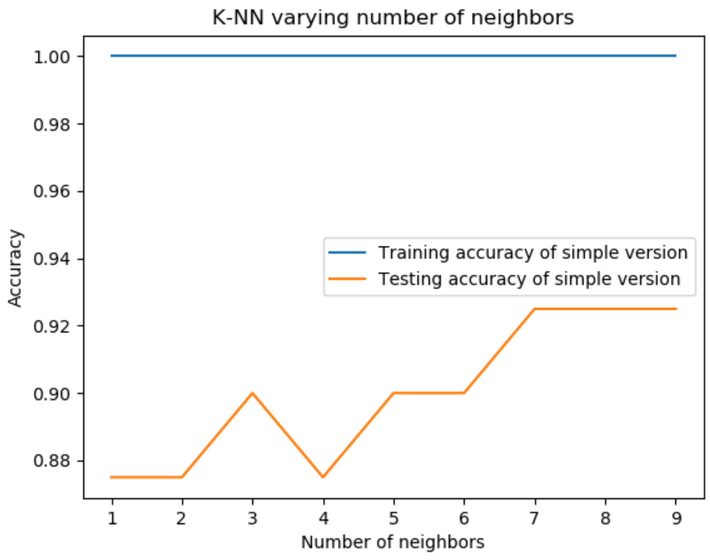
4.4.1 图


sklearn-1NN
sklearn-2NN
sklearn-3NN
sklearn-4NN
sklearn-5NN
sklearn-6NN
sklearn-7NN
sklearn-8NN
sklearn-9NN


simple-1NN
simple-2NN
simple-3NN
simple-4NN
simple-5NN
simple-6NN
simple-7NN
simple-8NN
simple-9NN

4.4.2 准确度





4.4.3 运行时间

版本	数据规模	时间
sklearn	数据集，200 x 2. 待预测集, 10000. K： 1 - 9	0:00:01.44
simple	数据集，200 x 2. 待预测集, 10000. K： 1 - 9	0:01:09.12

不难看出，simple版本的效率远远慢于sklearn. 其原因主要是对K邻居的检索和强制性地，对待预测集中每一个点都要进行的距离计算。

Section5. 动手实践： kdTree版KNN算法

实现k近邻法时，我们主要考虑的问题是如何对训练数据进行进行快速k近邻搜索。“快速”这点在特征空间的维数极大或者数据容量大时尤其必要。所以，为了提高k近邻搜索的效率，我们可以考虑使用特殊的结构存储训练数据，以减少计算距离的次数。这样的方法有很多，我们这里来介绍一下**kd树**（kd tree）方法。kdTree的核心思路是通过维度轴的划分，使得样本数据在样本空间中层次地被存储，这种顺序性使得局部的空间搜索成为可能。 构造kd树相当于不断地用垂直于坐标轴的超平面将k维空间切分，构成一系列的k维超矩形区域（在这里我们可以看到我们似乎可以看到决策树的影子，事实上二者确实很相像）。

构造kd树的方法如下：

1. 首先构造根节点，使根节点对应于k维空间中包含所有实例点的超矩形区域
2. 通过下面所说的递归方法，不断地对k维空间进行划分，生成子节点：
3. 先在超矩形区域（根节点）上选择一个坐标轴和在此坐标轴上的一个切分点，确定一个**超平面**（就是能够将数据集分开的，在多维空间上的平面，公式为wx+b=0wx+b=0w是超平面的法向量，b是超平面的截距），这个超平面通过选定的切分点并垂直于选定的坐标轴，将当前超矩形区域切分为左右两个子区域（子节点）
4. 将实例划分到两个子区域后，继续上述过程，直到子区域内没有实例，终止之后的节点为叶节点。在此过程中，将实例保存在相应的节点上。

下面我们来举一个构造平衡kd树的例子。

例：构造平衡kd树

> 输入：k维空间数据集T=x1,x2,…,xNT=x1,x2,…,xN
>
> 其中，
> $xi = (x(1)i, x(2)i, \ldots, x(k)i)T, i = 1, \ldots, Nxi = (xi(1), xi(2), \ldots, xi(k))T, i = 1, \ldots, N$
>
> 输出：kd树
>
> 过程：
>
> 1. 开始：构造根节点，根节点对应于包含T的k维空间的超矩形区域： 选择x(1)x(1)为将与超平面垂直的坐标轴，以T（训练样本集合）中所有样本的x(1)x(1)坐标的中位数为切分点，将根节点对应的超矩形区域切分为两个子区域。切分由通过切分点并与坐标轴x(1)x(1)垂直的超平面实现。 由根节点生成深度为1的左右子节点（子区域）：左子节点对应坐标x(1)x(1)小于切分点的子区域，右子节点对应坐标x(1)x(1)大于切分点的子区域。 落在切分超平面上的样本保存在根节点。
> 2. 重复：对深度为j的节点，选择x(l)x(l)为切分的坐标轴，$l = j(modk) + 1l = j(modk) + 1(xmody)$表示求余运算，即求x整除y的余数），以该节点的区域中所有样本的x(l)x(l)的中位数作为切分点，将该节点对应的矩形区域切分为两个子区域。切分由通过切分点并与坐标轴x(l)x(l)垂直的超平面实现： 由该节点生成的深度为j+1的左右子节点：左子节点对应坐标x(l)x(l)小于切分点的子区域，有直接点对应坐标x(l)x(l)大于切分点的子区域。
> 3. 直到两个区域没有实例存在时停止，从而形成对kd树区域的划分。

参考: 机器学习 第七章 k近邻

5.1 构造平衡kd树

```
class Node(object):
    def __init__(self, feature = None, father = None, l_node = None, r_node =
None, div = 0, visited = False):
        self.feature = feature
        self.father = father
        self.l_node = l_node
        self.r_node = r_node
        self.div = div
        self.visited = visited

class kdTree(object):
    def __init__(self, X, dim):
        assert X is not None and len(X) != 0
        self.root = Node(div = 0)
        self.dim = dim
        self.samples = X
        self.__build(self.root, list(self.samples))
    def __build(self, node, X):
        # return conditions
        if len(X) == 1: node.feature = X[0]; return
        X.sort(key = lambda x: x[node.div]) # note: X is a list before __build 
calls.
        # update feature
        node.feature = X[len(X) // 2]
        # create nodes: leaves do not have children
        if len(X[:len(X)//2]):
            node.l_node = Node(father = node, div = (node.div+1) % self.dim)
            self.__build(node.l_node, X[:len(X)//2])
        if len(X[len(X)//2 + 1:]):
            node.r_node = Node(father = node, div = (node.div+1) % self.dim)
            self.__build(node.r_node, X[len(X)//2 + 1:])
        return
    def search(self, p, node):
        if p[node.div] < node.feature[node.div]:
            if node.l_node is not None: return self.search(p, node.l_node)
            return node
        if p[node.div] >= node.feature[node.div] :
            if node.r_node is not None: return self.search(p, node.r_node)
            return node
```

5.2 kd树版最近邻（递归）

利用kd树的最近邻搜索

输入：已构造的kd树；测试样本点x；

输出：x的最近邻

1. 在kd树中找到包含测试样本点x的叶节点： 从根节点出发，递归向下访问kd树。若目标点当前维的坐标小于切分点的坐标，则移动到左子节点，否则移动到右子节点。直到子节点为叶节点为止
2. 将搜索到的叶节点作为“当前最近点”
3. 递归地向上回退，并在每个节点上进行以下操作： a. 如果该节点保存的训练样本点比当前最近点距离测试样本点还近，则将该训练样本点作为“当前最近点” b. 当前最近点一定存在于该节点一个子节点对应的区域。检查该子节点的父节点的另一个子节点对应的区域是否有更近的点。具体地说，就是当前区域是否与以测试样本点为球心，测试样本点与“当前最近点”的距离为半径的超球体相交。 如果相交，则可能在另一个子节点对应的区域内存在距离测试样本点更近的点，算法移动到另一个子节点，进行最近邻搜索 如不相交，则算法向上回退
4. 回退到根节点时，搜索结束，最后的“当前最近点”即为x的最近邻点。

如果，训练样本点是随机分布的，那么kd树搜索的平均计算复杂度就是O(logN)O(logN)。kd树更适用于训练样本数远大于空间维数时的k近邻搜索。当空间维数接近训练样本数时，它的效率会迅速下降，几乎接近线性扫描。

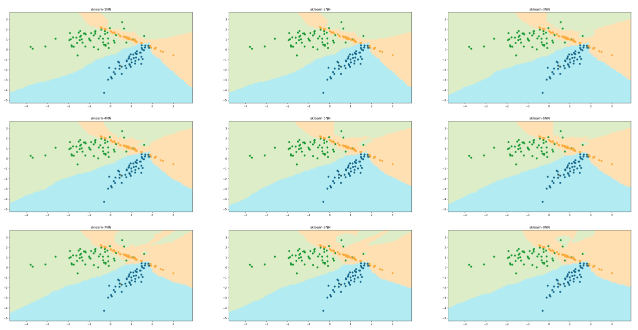
5.3 完整代码

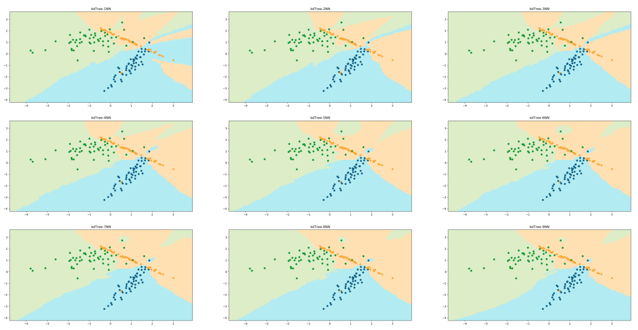
```
class KNeighborsClassifier(object):
    def __init__(self, n_neighbors, weights = 'distance'):
        self.samples = None
        self.dim = None
        self.tree = None
        self.K = n_neighbors
        self.weights = weights
        # original data
        self.X = None
        self.y = None
        # k neighbors array
        self.neighbors = []
    def __decision_rules(self, neighbors):
        '''For a to-be-classified point, this function decides its class based
on k-neighbors of it.
        return the label predicted.'''
        labels = np.zeros(len(set(self.y)))
        for i, neighbor in enumerate(neighbors):
            if neighbor[-1] == 0: return neighbor[-2]
            labels[int(neighbor[-2])] += 1 / neighbor[-1]
        return np.argmax(labels)
    def __distance(self, x, p):
        # it is required the x and p are type of ndarray
        res = np.sum((x - p)**2)
        return np.power(res, 1 / self.dim)
    def __find_k_neighbors(self, node, neighbors, p):
        # note that, neighbors should be in order now.

        # Given a node, we technically do three things:
        # 1. check whether we should involve this node in
        # 2. figure out in which node we forward our recursion, node.father or
node.child?
        # 3. check ending condition

        # firstly, set node visited.
        node.visited = True
        node.feature[-1] = self.__distance(node.feature[:-2], p)
        # 1. neighbors updates
        # if k < self.K, we just add it into neighbors
        if len(self.neighbors) < self.K: self.neighbors.append(node.feature)
        # else, check whether involve it in neighbors
        elif node.feature[-1] < self.neighbors[-1][-1]:
            self.neighbors[-1] = node.feature
            self.neighbors.sort(key = lambda x: x[-1])
        # 2. node to be forwarded towards
        # if neighbors is full
        # enter node father
        if abs(node.feature[node.div] - p[node.div]) >= self.neighbors[-1][-1]
and len(self.neighbors) >= self.K:
            if node.father is not None and node.father.visited is False:
                self.__find_k_neighbors(node.father, self.neighbors, p)
            node.visited = False
            return
        else:
            if node.l_node is not None and node.l_node.visited is False:
                self.__find_k_neighbors(node.l_node, neighbors, p)
            if node.r_node is not None and node.r_node.visited is False:
                self.__find_k_neighbors(node.r_node, neighbors, p)
            if node.father is not None and node.father.visited is False:
                self.__find_k_neighbors(node.father, neighbors, p)
            # note: before return, set visited false
            node.visited = False
```

```python
            return
    def __predict_point(self, p):
        '''P is just a point, only one data to be classified'''
        # note: neighbors should always be a list.
        self.neighbors = []
        node = self.tree.search(p, self.tree.root)
        self.__find_k_neighbors(node, self.neighbors, p)
        # print(self.neighbors)
        return self.__decision_rules(self.neighbors)
    def fit(self, X, y):
        assert len(X) == len(y) and len(X)
        # combinate X and y, add one more dim for distance and done.
        # note: this operation can avoid the error that X is one dimensional
        self.samples = np.c_[X, y]
        self.samples = np.c_[self.samples, np.zeros(len(self.samples))]
        # dim
        self.dim = len(X[0])
        # kd-Tree
        self.tree = kdTree(self.samples, self.dim)
        # original data
        self.X = X
        self.y = y
        return
    def predict(self, P):
        '''Given P, output prediected result Z.'''
        # note that P is required to be one-dimensional vector
        # the result initialization
        res = np.zeros(len(P))
        for i, p in enumerate(P):
            res[i] = self.__predict_point(p)
        return res
    def score(self, X, y):
        y_predicted = self.predict(X)
        return np.sum(y_predicted == y) / len(y)
```

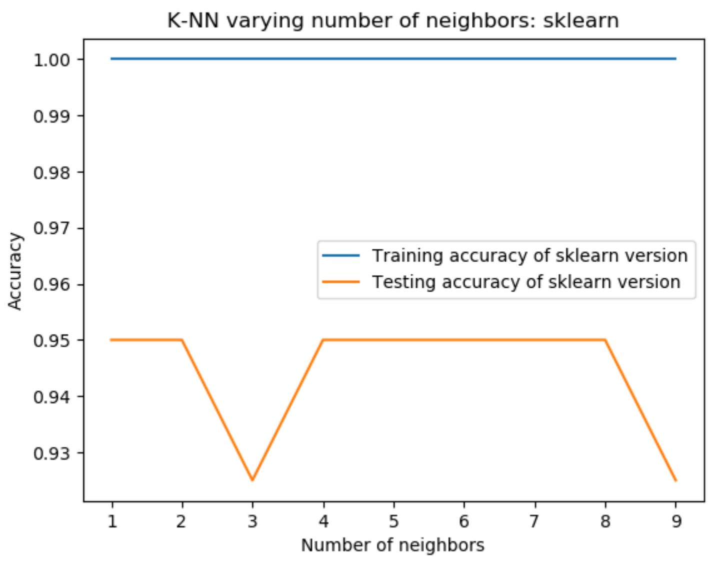
5.4 结果比较

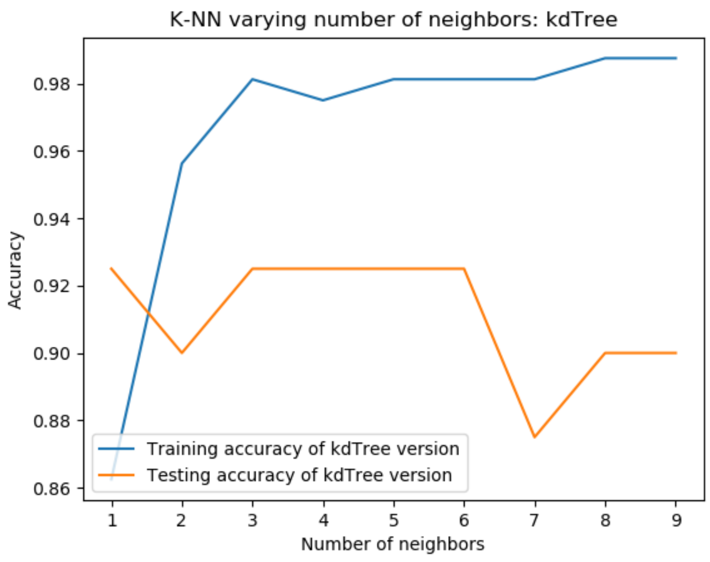
5.4.1 图


sklearn-1NN
sklearn-2NN
sklearn-3NN
sklearn-4NN
sklearn-5NN
sklearn-6NN
sklearn-7NN
sklearn-8NN
sklearn-9NN


kdTree-1NN
kdTree-2NN
kdTree-3NN
kdTree-4NN
kdTree-5NN
kdTree-6NN
kdTree-7NN
kdTree-8NN
kdTree-9NN

5.4.2 准确率





5.4.3 时间复杂度

版本	数据规模	时间
sklearn	数据集，200 x 2. 待预测集, 10000, K： 1 - 9	0:00:01.44
simple	数据集，200 x 2. 待预测集, 10000. K： 1 - 9	0:01:09.12
kdTree	数据集，200 x 2. 待预测集, 10000. K： 1 - 9	0:00:30.59

附录

1. sklearn库的结构

2. model.score()的含义