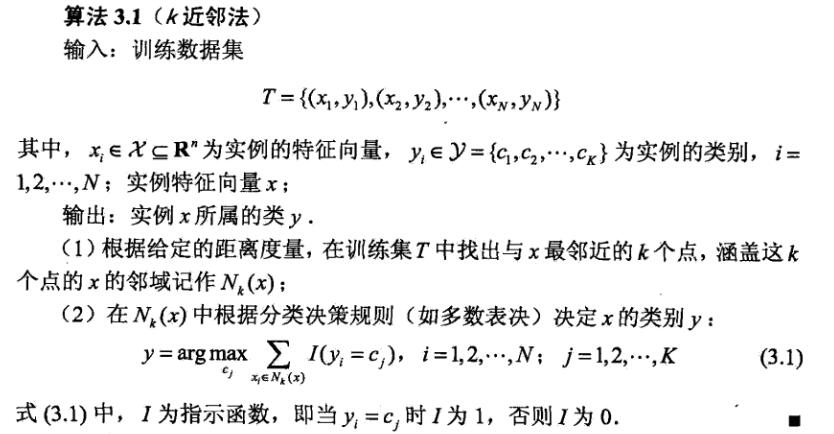
【K近邻算法】

一个少数服从多数的投票模型，可以用在多分类的情况。

算法描述：



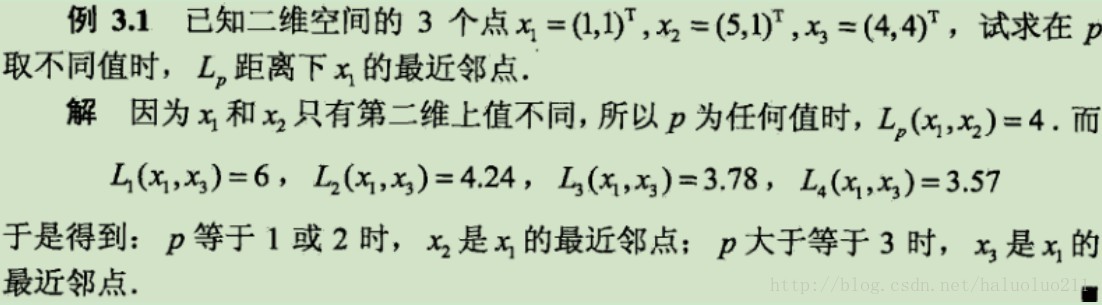
KNN是根据点与点之间的关系来进行判断，所以有些噪声可能会对结果有些影响。还有KNN并不需要训练，但需要遍历整个训练集，所以预测会比较慢。

【方法】

np.linalg.norm(a-b) #欧氏距离

np.dot(a,b) #按照矩阵乘法规则来运算

【算法举例】



【MNIST数据集】

图片是以字节的形式进行存储，我们需要把它们读取到 NumPy array 中，以便训练和测试算法。

import os

import struct

import numpy as np

def load\_mnist(path, kind='train'):

"""Load MNIST data from `path`"""

images\_path = os.path.join(path,

'%s-images-idx3-ubyte'

% kind)

labels\_path = os.path.join(path,

'%s-labels-idx1-ubyte'

% kind)

with open(images\_path, 'rb') as imgpath:

magic, num, rows, cols = struct.unpack('>IIII',

imgpath.read(16))

images = np.fromfile(imgpath,

dtype=np.uint8).reshape(len(labels), 784)

with open(labels\_path, 'rb') as lbpath:

magic, n = struct.unpack('>II',

lbpath.read(8))

labels = np.fromfile(lbpath,

dtype=np.uint8)

return images, labels

load\_mnist 函数返回两个数组，第一个是一个n x m维的 NumPy array(images)，这里的n是样本数(行数)，m 是特征数(列数)。训练数据集包含 60000 个样本，测试数据集包含 10000 样本。在 MNIST 数据集中的每张图片由 28 x 28 个像素点构成，每个像素点用一个灰度值表示。在这里，我们将 28 x 28 的像素展开为一个一维的行向量，这些行向量就是图片数组里的行(每行 784 个值，或者说每行就是代表了一张图片)。 load\_mnist 函数返回的第二个数组(labels) 包含了相应的目标变量，也就是手写数字的类标签(整数 0-9)。

通过执行下面的代码，我们将会从刚刚解压 MNIST 数据集后的 mnist 目录下加载 60000 个训练样本和10000个测试样本.

为了了解 MNIST 中的图片看起来到底是个啥，让我们来对它们进行可视化处理。从 feature matrix 中将784-像素值的向量 reshape为之前的 28\*28 的形状，然后通过 matplotlib 的 imshow 函数进行绘制:

import matplotlib.pyplot as plt

fig, ax = plt.subplots(

nrows=2,

ncols=5,

sharex=True,

sharey=True, )

ax = ax.flatten()

for i in range(10):

img = X\_train[y\_train == i][0].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

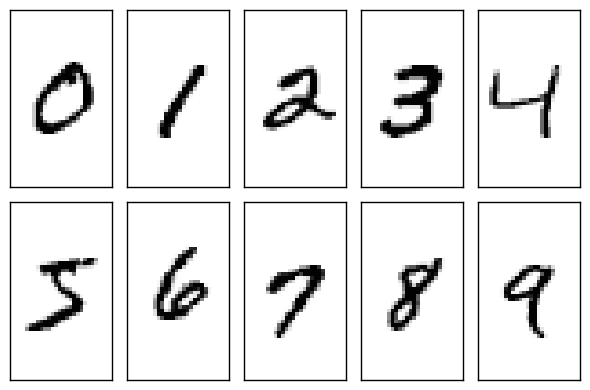
ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()

我们现在应该可以看到一个 2\*5 的图片, 里面分别是 0-9 单个数字的图片。



此外, 我们还可以绘制某一数字的多个样本图片, 来看一下这些手写样本到底有多不同:

fig, ax = plt.subplots(

nrows=5,

ncols=5,

sharex=True,

sharey=True, )

ax = ax.flatten()

for i in range(25):

img = X\_train[y\_train == 7][i].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

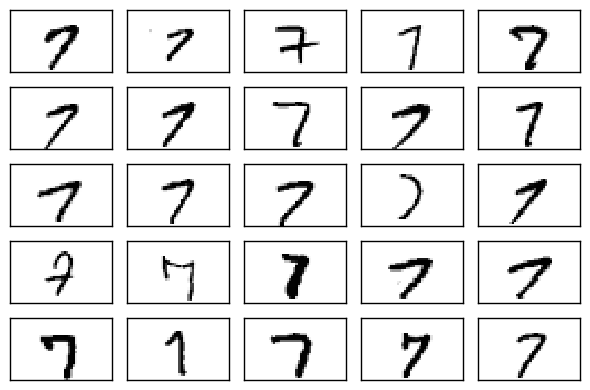
ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()

执行上面的代码后, 我们应该看到数字 7 的 25 个不同形态:



【代码】

链接：

http://blog.csdn.net/wds2006sdo/article/details/51933044

<http://blog.csdn.net/haluoluo211/article/details/78177510>

【KD-树】

是对数据点在k维空间（如二维(x，y)，三维(x，y，z)，k维(x1，y，z..)）中划分的一种数据结构，主要应用于多维空间关键数据的搜索（如：范围搜索和最近邻搜索）。就是把整个空间划分为特定的几个部分，然后在特定空间的部分内进行相关搜索操作。

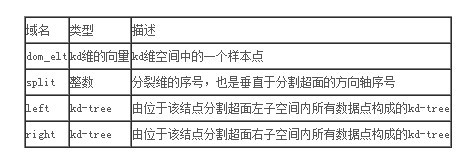
【例子】

6个二维数据点{(2,3)，(5,4)，(9,6)，(4,7)，(8,1)，(7,2)}构建kd树的具体步骤为：

* 确定：split域=x。具体是：6个数据点在x，y维度上的数据方差分别为39，28.63，所以在x轴上方差更大，故split域值为x；
* 确定：Node-data = （7,2）。具体是：根据x维上的值将数据排序，6个数据的中值(所谓中值，即中间大小的值)为7，所以Node-data域位数据点（7,2）。这样，该节点的分割超平面就是通过（7,2）并垂直于：split=x轴的直线x=7；
* 确定：左子空间和右子空间。具体是：分割超平面x=7将整个空间分为两部分：x<=7的部分为左子空间，包含3个节点={(2,3),(5,4),(4,7)}；另一部分为右子空间，包含2个节点={(9,6)，(8,1)}；

【基本算法】

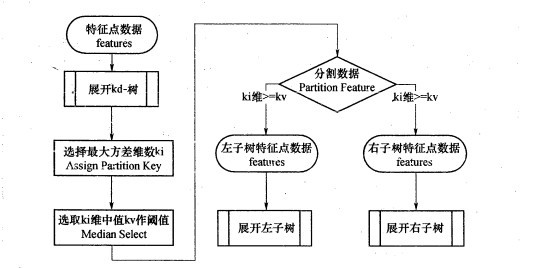
**A. k-d树的数据结构** # class KD\_node



**B.划分维度的原理** # computeVariance(arr)

（基于方差的选择方法）尽可能将相似的点放在一颗子树里面，所以kd-tree采取的思想就是计算所有数据点在每个维度上的数值的方差，然后方差最大的维度就作为当前节点的划分维度。

**C.k-d树的创建** # createKDTree(root,data\_list)



**D.利用kd-tree进行最近邻的查找** #findNN(root,query)

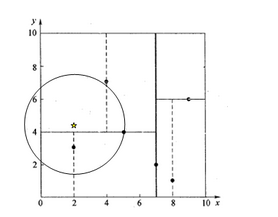
1.二叉查找：

从根节点开始进行查找，直到叶子节点；在这个过程中，记录最短的距离，和对应的数据点；同时维护一个栈，用来存储经过的节点

2.回溯查找：

通过计算查找点到分割平面的距离(这个距离比较的是分割维度上的值的差，并不是分割节点到分割平面上的距离，虽然两者的值是相等的)与当前最短距离进行比较，决定是否需要进入节点的相邻子空间进行查找

e.g



已知：

* 图中的黑点为kd-tree中的数据点，五角星为查询点；
* 通过kd-tree的分支决策会将它分到坐上角的那部分空间，但并不是意味着它到那个空间中的点的距离最近

步骤：

1）首先扫描到叶子节点，扫描过程中记录的最近点为p（5，4），最短距离为d；

2）现在开始回溯，假设分割的维度为ss，其实回溯的过程就是确定是否有必要进入相邻子空间进行搜索，确定的依据就是当前点到最近点的距离d是否大于当前点到分割面（在二维空间中实际上就是一条线）的距离L，如果d < L,那么说明完全没有必要进入到另一个子空间进行搜索，直接继续向上一层回溯；如果有d > L,那么说明相邻子空间中可能有距查询点更近的点

【代码】

http://blog.csdn.net/yan456jie/article/details/52074141

http://www.cnblogs.com/v-July-v/archive/2012/11/20/3125419.html