# 一、第一次实验：算术编码

# 1. 算法描述

1. 主要功能：
2. 用户可以输入一组字符以及这些字符的概率。
3. 根据输入的字符和概率，构建一个概率字典。
4. 对用户输入的字符串进行算术编码，生成一个最短的二进制串。
5. 将生成的二进制串解码回原始的字符串。
6. 算法流程：

输入：一组字符，每个字符的概率，待编码的字符串。

输出：算术编码后的二进制串，以及解码后的字符串。

1. Step 1：获取用户输入

通过getChar函数获取用户输入的字符集合。

通过getProbability函数获取每个字符的概率。

通过getString函数获取需要编码的字符串。

1. Step 2：构建概率字典

使用createDict函数根据输入的字符和概率构建一个概率字典，该字典用于存储每个字符的累积概率区间。

1. Step 3：算术编码

使用arithEncode函数对输入的字符串进行算术编码，生成一个区间的上下界。

1. Step 4：生成最短二进制串

使用dec2Bin函数根据算术编码得到的区间生成最短的二进制串。

1. Step 5：解码

使用arithDecode函数将二进制串解码回原始的字符串。

1. Step 6：输出结果

打印出算术编码后的二进制串和对应的十进制浮点数。

打印出解码后的字符串。

# 核心代码

**构建概率字典函数：**

def createDict(char, prob):

# 初始化概率字典

probDict = {}

# 记总字符数为num

num = len(char)

# 构造字符概率字典

for i in range(num):

if (i == 0):

probDict[char[i]] = [Decimal('0.0'), prob[i]] # 第一个字符的概率字典下界起点为0.0，上界为0.0+该路

else:

probDict[char[i]] = [probDict[char[i - 1]][1], probDict[char[i - 1]][1] + prob[i]]

# 第二个数的下界是上一个字符的上界，上界是自己的下界+概率

return probDict

**算术编码函数：**

def arithEncode(string, probDict):

# 初始化上下界

lower\_bound = Decimal('0.0')

upper\_bound = Decimal('1.0')

for chr in string:

intervalLength = upper\_bound - lower\_bound # 区间长度

# 不断更新区间上下界，注意必须先更新上界，否则会导致上界更新错误（因为上界的计算用的是上一次的下界）

upper\_bound = lower\_bound + intervalLength \* probDict[chr][1]

lower\_bound = lower\_bound + intervalLength \* probDict[chr][0]

print(lower\_bound, upper\_bound)

# 返回最终区间的上下界

return lower\_bound, upper\_bound

**解码函数：**

def arithDecode(encodedBin, probDict, strLength):

decodedStr =''

# probDict\_copy存放原始的概率字典

probDict\_copy = deepcopy(probDict)

# 二进制串转十进制小数

encodedDec = bin2Dec(encodedBin)

# 因为要解码出strLength个字符，所以共循环strLength次

for \_ in range(strLength):

# 如果编码数字落在某个字符的概率区间内，则在结果中加入该字符

for chr, interval in probDict.items():

if ((encodedDec >= interval[0]) & (encodedDec < interval[1])):

decodedStr += chr

# 判断数字是否落在概率区间内，如果在，就继续对当前区间划分，看编码数字落在哪个字符对应的区间内

# 更新字典中各字符的概率区间

temp\_lower\_bound = interval[0] # 记录当前区间下界和区间长度

intervalLength = interval[1] - interval[0]

for chr in probDict.keys():

probDict[chr][0] = temp\_lower\_bound + intervalLength \* probDict\_copy[chr][0]

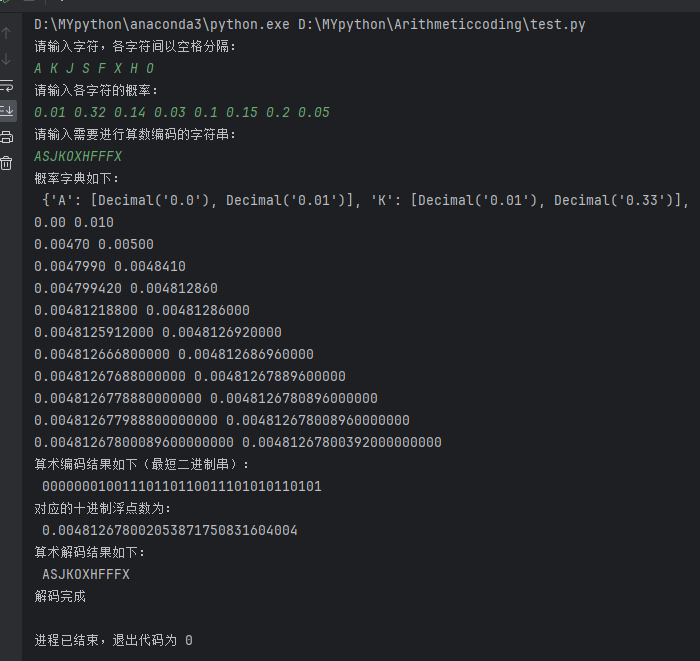
probDict[chr][1] = temp\_lower\_bound + intervalLength \* probDict\_copy[chr][1]

break

return decodedStr

# 实验结果

根据实验输入要求，模拟用户输入字符和字符概率，用来构建概率字典。输入要进行算术编码的字符串，运行程序，结果如图所示：



为了方便用户了解算术编码的过程，在实验结果展示时，将概率字典、区间上下界、算术编码结果、浮点数转换和解码结果都打印出来。

# 亮点、分析与总结

亮点：

在设计函数时，因为需要用户提供概率和需要编码的字符串，所以我在编写函数时，尽可能多地添加了用户交互的功能代码。代码通过函数getChar、getProbability和getString与用户进行交互，获取必要的输入数据，使得算法具有较好的灵活性和易用性。

另外，我是使用Decimal类处理浮点数运算，避免了传统浮点数运算中的精度问题，确保了概率计算的准确性。

因为是用户输入，所以在输入阶段，我要保证数据的有效性，通过循环和条件判断确保了输入数据的有效性，比如字符的个数、概率的总和以及待编码字符串的合法性。

模块化设计：算法被分解为多个函数，如createDict、arithEncode、dec2Bin和arithDecode，每个函数负责一个特定的任务，提高了代码的可读性和可维护性。

概率字典构建：通过createDict函数构建的概率字典为算术编码和解码提供了基础，这种数据结构的设计是实现有效编码的关键。

算法改进：

优化性能：对于长字符串，当前的编码和解码过程可能效率不高。可以考虑实现更高效的数据结构或算法来处理大数据量。

错误处理：增加更多的错误处理逻辑，比如输入数据不合法时提供更清晰的错误信息。

关键步骤分析：

构建概率字典：这是整个算法的基础，直接影响到编码和解码的准确性和效率。

算术编码：通过不断更新区间上下界来实现数据的压缩，是算法的核心步骤。

二进制串生成：将编码区间转换为二进制串，这一步的效率直接影响到编码的最终长度。

解码过程：正确地从二进制串恢复原始数据，验证了编码算法的可逆性和实用性。

# 二、第二次实验：实现PCA主成分分析

# 1. 算法描述

1. 主要功能：
2. 读取数据集，并计算数据集的列数。
3. 对数据集执行主成分分析，以减少数据的维度。
4. 计算PCA前后数据的方差，以评估降维的效果。
5. 算法流程：

输入：数据集，需要保留的主成分个数。

输出：PCA前数据方差，降维后的数据，PCA后数据方差。

1. Step 1：读取数据集并获取列数

使用get\_asc\_file\_dimensions函数通过给定的文件路径读取数据集，并确定数据集中的列数。

1. Step 2：数据均值中心化

从数据集中减去每列的均值，实现数据的均值中心化。

1. Step 3：计算协方差矩阵

利用中心化后的数据计算协方差矩阵，这是PCA中的关键步骤，用于衡量数据特征之间的相关性。

1. Step 4：计算特征值和特征向量

使用NumPy的np.linalg.eig函数计算协方差矩阵的特征值和特征向量。

1. Step 5：排序特征值和特征向量

根据特征值的大小对特征向量进行排序，选择最大的num\_components个特征向量作为主成分。

1. Step 6：数据投影到主成分上

将原始数据投影到选定的主成分上，得到降维后的数据。

1. Step 7：计算PCA前后的方差

分别计算PCA前后数据的方差，以评估PCA的效果。

1. Step 8：输出结果

打印PCA前后数据的方差，以及降维后的数据。

# 核心代码

**自定义PCA函数：**

def custom\_pca(dataset, num\_components):

# 均值中心化

centered\_data = dataset - np.mean(dataset, axis=0)

# 计算协方差矩阵

# C = (1 / (n-1)) \* X\_centered^T \* X\_centered

# 利用公式来计算协方差矩阵 C，其中 X\_centered^T 表示 X\_centered 的转置。

# 获取维度(列数)

ncols = get\_asc\_file\_dimensions(file\_path)

covariance\_matrix =(1 / (ncols-1)) \* np.dot(centered\_data.T, centered\_data)

# 计算特征值和特征向量

eigenvalues, eigenvectors = np.linalg.eig(covariance\_matrix)

# 对特征值和特征向量进行排序

sorted\_indices = np.argsort(eigenvalues)[::-1]

sorted\_eigenvalues = eigenvalues[sorted\_indices]

sorted\_eigenvectors = eigenvectors[:, sorted\_indices]

# 选择前num\_components个主成分

selected\_eigenvectors = sorted\_eigenvectors[:, :num\_components]

# 将数据投影到选定的主成分上

projected\_data = np.dot(centered\_data, selected\_eigenvectors)

# 计算PCA之前的数据方差

variance\_before\_pca = np.var(dataset, axis=0)

# 计算PCA之后的数据方差

variance\_after\_pca = np.var(projected\_data, axis=0)

return variance\_before\_pca, projected\_data, variance\_after\_pca

# 实验结果

# 

读取数据集，进行PCA处理后，分别打印出处理前和处理后的数据，作为对比，方便用户观察。

# 亮点、分析与总结

亮点：

虽然可以调库直接实现，但是要充分理解PCA的原理和过程，我自定义了一个函数来实现，这样不依赖于现成的库函数，有助于我深入理解PCA的工作原理。另外，经过一定程度的学习了解，我尽可能地保持算法流程清晰，代码简洁，易于理解和学习。并在程序最后进行

数据方差分析：计算并输出PCA前后的数据方差，有助于评估PCA的效果和数据降维的质量。

算法改进：

我可以添加可视化代码：提供数据降维前后的可视化，帮助用户更直观地理解PCA的效果。

关键步骤：

数据读取与列数获取：读取数据集并获取其列数，为后续计算做准备。

均值中心化：将数据集中心化，使其具有零均值，这是PCA的前提条件。

协方差矩阵计算：计算数据的协方差矩阵，反映数据特征间的相关性。

特征值和特征向量计算：求解协方差矩阵的特征值和特征向量，这些是找到主成分的关键。

特征向量排序与选择：根据特征值的大小对特征向量进行排序，并选择前num\_components个特征向量作为主成分。

数据投影：将原始数据投影到选定的主成分上，实现数据的降维。

方差计算：计算PCA前后数据的方差，评估PCA的效果。

总结： 我通过自定义函数的方式，使得PCA的每个步骤都清晰可见。算法的实现考虑了数据的均值中心化、协方差矩阵的计算、特征值和特征向量的求解等关键步骤，并通过方差的计算来评估PCA的效果。尽管代码已经相当简洁和直观，但仍有改进空间，特别是在参数传递、异常处理和算法扩展性方面。

# 三、第三次实验：BOF提取图片特征与检索

# 1. 算法描述

1. 主要功能：
2. 从图像中提取SIFT特征描述符。
3. 使用k-means算法对特征描述符进行聚类，以生成视觉词汇。
4. 利用视觉词汇将图像特征向量化，形成图像的直方图表示。
5. 计算逆文档频率（IDF）并用于加权图像直方图，增强稀有特征的重要性。
6. 通过余弦相似度或欧几里得距离度量图像之间的相似性，并检索相似图像。
7. 算法流程：

输入：图像集，聚类中心数（k），采样率。

输出：聚类中心（视觉词汇），图像特征直方图，相似图像检索结果。

1. Step 1：提取SIFT特征

使用OpenCV库从图像中提取SIFT特征描述符。

1. Step 2：初始化描述符矩阵

将所有图像的SIFT特征描述符堆叠成一个矩阵。

1. Step 3：k-means聚类

实现自定义的k-means算法，用于确定特征描述符的聚类中心（视觉词汇）。

1. Step 4：计算图像直方图

将每个图像的SIFT特征描述符投影到聚类中心上，形成图像的直方图表示。

1. Step 5：计算IDF并加权直方图

计算各视觉词汇的逆文档频率，并用于加权图像直方图。

1. Step 6：图像相似性检索

通过余弦相似度或欧几里得距离检索与输入图像相似的图像。

1. Step 7：显示结果

显示输入图像和检索到的相似图像。

# 核心代码

**K-means函数：**

def kmeans(X, k):

n = X.shape[0] # 数据点总数

epoch = 0 # 迭代次数

idxs = [random.randint(0,k) for \_ in range(k)]

Centroids = X[idxs] # 构造初始的中心点矩阵（1000x128）

while(1):

epoch += 1

cls\_pos = {}

# 对于每个点，计算它到各中心的距离，得到它所属的类

cls, \_ = vq(X, Centroids)

for i in range(n):

cls\_pos.setdefault(cls[i],[]).append(X[i]) # cls\_pos：以类为key，其value为一个数组，保存属于该类的点坐标

for j in range(k):

if j not in cls\_pos.keys():

cls\_pos.setdefault(j,[]).append(Centroids[j]) # 若某类没有点被分配进来，则该类只有类中心点一个点

# 所有点均已分到相应的类中。下求新的聚类中心坐标

Centroids\_new = calcNewCentroids(cls\_pos)

# 当聚类中心不再变化时，停止迭代

if (Centroids\_new == Centroids).all():

break

# 更新聚类中心为新的聚类中心

Centroids = Centroids\_new

# 返回聚类中心坐标及迭代次数

return epoch, Centroids

**直方图处理代码：**

epoch, voc = kmeans(descriptors[::sampling\_rate,:], k) # 1000 1min11s

# 计算一幅图的直方图向量

def project(descriptors, words\_num, voc): # 描述子向量，视觉词汇数，视觉词典

imhist = np.zeros(words\_num) # 初始化直方图向量

cls, \_ = vq(descriptors, voc) # 将该图的各个描述子分配到离它最近的聚类中心代表的类中

for c in cls:

imhist[c] += 1

return imhist

imhists = np.zeros((image\_nums, words\_num))

for i in range(image\_nums):

imhists[i] = project(desc[i], words\_num, voc)

occurence\_num = np.sum(imhists > 0, axis=0) # 计算各视觉词汇在1000张图片中出现了几次，1x1000

IDF = np.log((image\_nums) / (occurence\_num + 1)) # 逆文档频率 = log(总图片数/(出现次数+1))，1x1000

for i in range(image\_nums):

imhists[i] = imhists[i] / np.sum(imhists[i]) \* IDF # imhists[i] / np.sum(imhists[i])是TF，即各视觉词汇在一幅图中出现的频率

def cosin\_dist(x, imhists):

dists = []

for i in range(image\_nums):

dists.append(np.dot(x, imhists[i]) / (np.linalg.norm(x, ord=2) \* np.linalg.norm(imhists[i], ord=2)))

return np.sort(dists)[::-1], np.argsort(dists)[::-1]

def euclid\_dist(x, imhists):

dists = []

for i in range(image\_nums):

dists.append(np.sqrt(np.sum((x - imhists[i]) \*\* 2)))

return np.sort(dists), np.argsort(dists)

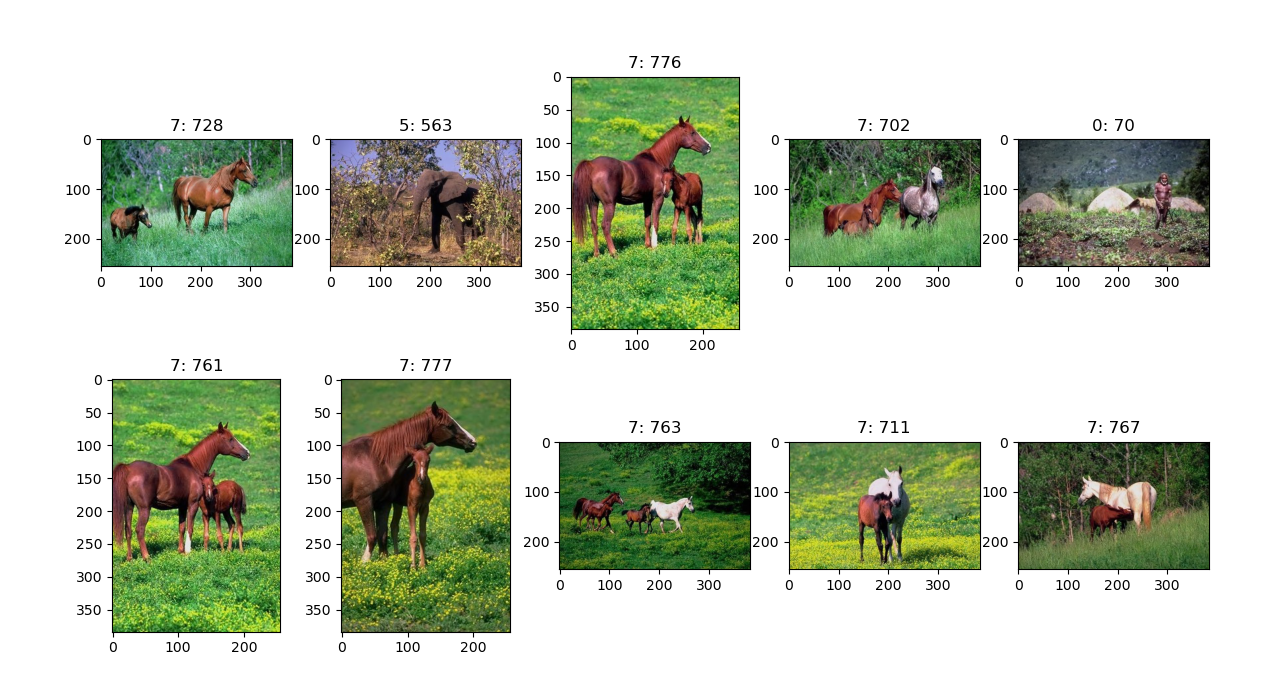
# 实验结果

在测试实验运行时，我设定图片id为7/720。在运行程序输出结果。









根据输入图像，程序对图像特征进行特征提取识别和索引，找到相似度最高的10幅图，从结果进行观察比较，可以发现，原图是草地上的马，在检索得到的10幅图中，有8张图最符合原图的特征，另有1张大象，1张野人。

# 亮点、分析与总结

亮点：

我自定义了k-means实现，并没有直接调用库函数，这样对算法的实现有了更清晰的展示，有助于深入理解聚类过程。

特征提取：使用SIFT算法提取图像的尺度不变特征，增强了特征的鲁棒性。

另外，我添加了IDF加权部分，用来提高稀有特征的权重，增强检索结果的相关性。

图像检索：实现了基于视觉词汇的图像相似性检索，可以应用于图像识别和检索任务。

可视化：通过Matplotlib库可视化输入图像和检索结果，提高了用户体验。

关键步骤：

SIFT特征提取：提取图像的关键特征，为后续的聚类和检索提供基础。

k-means聚类：通过聚类生成视觉词汇，将高维特征映射到低维空间。

直方图计算：将图像特征转换为直方图表示，便于后续的相似性度量。

相似性度量：使用余弦相似度或欧几里得距离度量图像之间的相似性。

程序改进：

程序的输入数据是提前设定的，可以将输入改为满足用户交互的方式，由用户输入。

总结： 这次实验要求实现一个基于SIFT特征和k-means聚类的BOF图像检索系统。我通过调库函数提取图像的SIFT特征，并使用自定义的聚类算法生成视觉词汇，然后利用这些词汇将图像特征向量化，并通过相似性度量检索相似图像。算法的实现考虑了特征提取、聚类、特征向量化和相似性检索等关键步骤，并通过可视化增强了用户体验。但是并没有过多的考虑用户交互的功能，实际应用中可能需要进一步优化和改进，以进一步提高用户体；另外，算法并不是最优的，因为自定义函数，这个程序相对于检索不同的图像得出结果所需的时间是大不相同的，以实验为例，程序运行需要大致3-4分钟才能输出结果，而直接调用函数的话用时不到1分钟，说明自定义的函数还是有较多问题以待优化和解决的。

# 四、第四次实验：LSH索引

# 1. 算法描述

1. 主要功能：
2. 快速在大规模数据集中找到查询点的近似最近邻。
3. 评估近似最近邻搜索的性能，包括查准率、召回率和准确率。
4. 算法流程：

输入：数据集，桶的数量，桶宽a，偏移量b。

输出：近似最近邻列表，性能评估指标，包括查准率、召回率和准确率。

1. Step 1：数据预处理：

加载数据集和预先计算的真实近邻索引。

1. Step 2：初始化LSH参数：

生成随机的哈希矩阵R，桶宽a和偏移量b。

1. Step 3：构建哈希表：

使用hash\_and\_fill函数将数据点映射到不同的桶中。

1. Step 4：LSH搜索：

对于每个查询点，使用find函数通过哈希表找到近似最近邻。

1. Step 5：性能评估：

计算查准率、召回率和准确率来评估搜索结果的质量。

1. Step 6：结果输出：

打印出近似搜索的耗时和性能评估指标。

# 核心代码

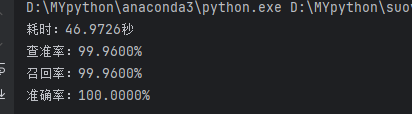
**映射索引函数：**

def hash\_and\_fill(inputs, R, b, a):  
 # 初始化空的hash\_table  
 buckets = [{} for \_ in range(bucket\_num)]  
 mapped\_idxes = floor((dot(inputs, R) + b) / a) # 每一行是这个点在所有桶中的哈希值  
 for i, hash\_keys in enumerate(mapped\_idxes):  
 # 遍历每个数据点的哈希键。  
 for j, hash\_key in enumerate(hash\_keys):  
 # 对于每个数据点，遍历其映射到的每个桶的索引  
 # 每个桶是一个字典，其中的所有key对应 该桶的所有索引键值，每个key对应的value是一个list，里面存放映射到该桶、该索引键值的所有点在原数据集的idx  
 buckets[j].setdefault(hash\_key, []).append(i)  
 return buckets  
# 数据点的索引添加到对应桶和索引键的列表

**最近索引查询代码：**

def find(q, k, R, b, a, buckets):  
 global candi\_set  
 hash\_keys = np.floor((dot(q, R) + b) / a)[0] # 取[0]转为数组  
 # 遍历q点的索引键列表，找各桶中与其索引键值相等的点  
 for i, hash\_key in enumerate(hash\_keys):  
 if i == 0:  
 candi\_set = set(buckets[0][hash\_key])  
 else:  
 candi\_set = candi\_set.union(buckets[i][hash\_key]) # 候选集  
 candi\_set = list(candi\_set) # 转为list便于遍历  
 # 遍历候选集，求出离q最近的k个点并返回  
 dist = [calc\_dist(data[i], q) for i in candi\_set]  
  
 set\_idxes = argsort(dist)[1: k + 1] # set\_idxes是近邻点在候选集中的索引，要将其转为在原数据集中的索引  
 res = [candi\_set[i] for i in set\_idxes]  
 return res

# 实验结果



实验因为要求计算评估指标，所以要提前准备好验证集，但是实验资料里是没有验证集（标准正确答案）的，所以我预先使用sklearn.neighbors库里的NearestNeighbors函数暴力求解，并将最近邻结果保留在true\_indices.csv文件里，作为评估时的验证集。实验运行结果如上。

# 亮点、分析与总结

亮点：

对于欧式距离的计算，我使用了Numba库加速：通过@nb.jit(nopython=True)装饰器，利用Numba库对计算欧氏距离的函数进行加速，将python代码计算转化为机器计算，极大地提高算法的执行效率。在添加修饰前，耗时接近160秒，处理后缩短到了47秒。

多桶哈希映射：通过hash\_and\_fill函数将数据点映射到多个哈希桶中，并且使用随机矩阵R和偏移量b，为每个桶提供不同的哈希函数，增加了搜索的灵活性。

自定义LSH搜索：实现了自定义的LSH搜索逻辑，包括哈希映射和候选集的构建。

评估指标计算：计算了查准率、召回率和准确率，全面评估了LSH搜索算法的性能。

关键步骤：

哈希函数构建：使用随机矩阵和偏移量构建哈希函数，决定数据点如何映射到桶。

哈希表构建：将数据点映射到哈希桶中，并记录每个桶的索引键值和对应的数据点索引。

候选集生成：对于查询点，通过多个桶的交集生成候选集。

近似搜索：在候选集中找到距离查询点最近的k个点。

性能评估：计算并输出查准率、召回率和准确率。

总结： 这次实验，因为Idistance索引的b+树不好处理，我最终选择了LSH索引，实验程序尽可能地完成了LSH近似最近邻搜索流程，包括哈希表的构建、搜索算法的实现和性能评估。另外，我还通过使用Numba库和多桶哈希策略，使得算法在效率和搜索质量之间取得了平衡。但是程序并不完美，搜索总时长达47秒，相对于Idistance索引的搜索耗时（10秒左右，甚至5秒以下）还是相当长的。欧氏距离的计算分配和哈希桶的索引搜索都是耗时较多的部分，但是LSH算法相对于KNN算法来说，根据桶索引选择一部分点进行欧式距离计算，在一定程度上节省了较多的时间。