**《算法导论》综合实验报告**

|  |  |
| --- | --- |
| 姓名 | **周萌** |
| 学号 | **19350008** |
| 邮箱 | **zhoumeng819@foxmail.com** |
| 手机 | **19821808568** |

## 1、问题描述

用python实现并验证《基于LSH的时间序列DTW相似性查询》（李敏）。

定义1.（时间序列）: 时间序列是按相等的时间采样的数据点构成的序列，是任意的实数，为时间序列的长度。

定义2.（时间序列数据库）: 时间序列数据库是条时间序列的集合，是一条长度为的时间序列。

定义3.（时间序列相似性查询）：给定一条查询序列、时间序列数据库、相似性度量函数和阈值，时间序列相似性查询是从时间序列数据库找到和查询序列相似程度大于等于的所有时间序列。即

动态时间弯曲首次被应用是在语音识别领域中，是一种准确性较高的时间序列距离度量方法。该方法区别于传统的欧氏距离，可以通过弯曲时间轴，实现一点对应多点的匹配，从而可以度量两条不等长的时间序列。

定义4.（动态时间弯曲）：给定两条时间序列和，它们的动态时间弯曲距离定义为:

表示空序列，，，表示和之间的距离。 动态时间弯曲距离就是利用动态规划的思想寻找一条具有最小弯曲代价的弯曲路径，时间复杂度为。 局部敏感哈希是一种应用于高维空间中的近似最近邻查询方法。它的基本思想是将原本相近的对象以很大的概率hash到同一个桶中，而原本较远的对象hash到同一个桶中的概率会很低。

定义5.（局部敏感哈希）：令对象的集合，为距离函数，对于哈希家族，如果任意两个对象、满足如下两个条件，则认为是敏感的： 如果，则的概率至少为； 如果，则的概率至少为； 表示和之间的距离，，和表示对和进行hash变换，只有时才有意义。该论文采用汉明距离下的哈希函数：

是敏感的。

## 2、数据结构与算法描述

为能使用汉明距离度量时间序列的相似度，首先需将实数形式的时间序列数据转化成字符串。为了避免平移、缩放等外界因素对衡量时间序列相似度带来的影响，需要对时序数据进行标准化，一般用标准化：

然后对每一维的数据分配ID，实数形式的数据即被整型的ID所代替，减少存储代价，同时也降维，为用汉明距离度量提供了方便。ID计算如下：

ID=\*COLUMN\_NUM+

COLUMN\_NUM为桶的个数，遍历原始序列数据找到其最大值和最小值，根据给定的 COLUMN\_NUM，确定每个桶的宽度，然后就可以将原始的序列分配到每一个桶中，并为其分配ID。实数值的时间序列则映射为 ID 序列。算法如下：

算法1． TimeSeriesTranstoIDSequence

Input:  
 S: time series   
 COLUMN\_NUM : parameter   
 ROW\_NUM : parameter  
Output:  
 S': ID sequence presentation of S  
tempS←TimeSeriesNormalization(S); # 对时间序列进行标准化  
MaxMin←findMaxMin(tempS); # 查找出时间序列中的最大值与最小值  
S'←;   
length←(() / ROW\_NUM) \* 1.01; # 计算桶的长度  
width←((Length(S)-1) / COLUMN\_NUM) \* 1.01; #计算桶的宽度  
for every point of S do   
 row = + 1;  
 col = + 1;  
 = (row - 1) \* COLUMN\_NUM + col; #确定ID  
 S'←S'∪;  
end  
return S';

算法2为ID序列投影到局部敏感哈希空间，映射的方法就是将不同长度的ID序列抻长为原始最大长度的w倍，每一个ID可能随机抻长一次或者多次，并且将每次抻长的参数保存，以便对查询序列采取相同的操作。

算法2． StrecthIDSequence

Input:  
 S': ID sequence   
Output:  
 StretchS: stretched representation of S'  
SeriesLength←findLongestSeries(S); # 寻找时间序列最大的长度  
rand←;   
for i←0 to 3\* SeriesLength   
 ←generaterandomnumber( 0 or 1); #生成随机数  
 rand←rand∪;  
end;  
StretchS←;  
for every point of S' do   
 k←0; j←0;  
 while j<3\* SeriesLength #拉伸  
 if(k<length(S'))  
 StretchS←StretchS∪{};  
 k←k+;  
 else  
 StretchS←StretchS∪{0};  
 j←j+1;  
 end  
end   
return StretchS;

算法3建立候选集并验证，将数据库中抻长后的ID序列中随机选取某几个位置进行哈希，将哈希过程中的参数保存，对每次的哈希家族构建前缀索引。判断查询序列的哈希族和数据库序列的哈希族是否相同。数据库中的序列只要满足有一次的哈希家族和查询序列的相匹配，我们就将该序列作为候选序列。然后计算该哈希族对应的原始序列和查询序列之间的DTW距离，如果二者的DTW距离小于或者等于阈值，则认为二者是相似的。对一条序列的哈希算法如下：

算法3． HashIDSequence

Input:  
 S': StretchS  
 m: hash location number  
Output:  
HashS: hashed representation of S'  
hash←;  
for i←0 to m  
 $ ← random number(0~Length(S')); #产生范围1- Length(S')随机数  
 hash ← hash∪{};  
 for j←0 to i  
 while == # 如果产生的随机数相同，再重新生成  
 ←random number(0~Length(S'));  
 end  
 hash←hash∪{}; # 替代原先的随机数  
 end  
end  
HashS←;  
for i←0 to m  
 HashS←HashS∪{}; # 产生哈希族  
end  
return HashS;

## 3、程序实现

# TimeSeriesTranstoIDSequence  
def divCell(data):  
 seqID = np.zeros(data.shape, dtype = int)  
 for i in range(len(data)):  
 mean = np.mean(data[i])  
 std = np.std(data[i])  
 data[i] = (data[i]-mean)/std # 对序列进行标准化  
 dmax = data[i].max() # 找到序列的最大最小值  
 dmin = data[i].min()  
 length = np.size(data,1) # 序列的长度  
 cellLength = (dmax-dmin)/rowNum\*1.01 #计算桶的长度  
 cellWidth = (length-1)/colNum\*1.01 #计算桶的宽度  
 for j in range(length):  
 row = np.floor((data[i][j]-dmin)/cellLength)+1  
 col = np.floor(j/cellWidth)+1  
 if col>colNum:  
 col = colNum  
 seqID[i][j]=(row-1)\*colNum+col # 确定ID  
 return seqID

# StrecthIDSequence  
def Embedding(seqID,seed):  
 length = np.size(seqID,1) # 序列的长度   
 N = 3\*length  
 np.random.seed(seed) # 确定随机数种子，使每次生成的数一致  
 rand = np.random.randint(0,2,N)  
 embedID = np.zeros([len(seqID),N], dtype = int)  
 for i in range(len(seqID)):  
 j = 0  
 k = 0  
 while(j<N): # 拉伸  
 if k<length:  
 embedID[i][j]=seqID[i][k]  
 k += rand[j]  
 else:  
 embedID[i][j]=0  
 j += 1  
 return embedID

# HashStrechIDSequence  
def Lsh(embedID,m,seed):  
 length = np.size(embedID,1)  
 np.random.seed(seed)   
 # 产生范围1- Length不重复随机数m个  
 hashPara = np.random.choice(length, size=m, replace=False, p=None)  
 hashBucket = np.zeros([len(embedID),m], dtype = int)  
 for i in range(len(embedID)):  
 for j in range(m):  
 hashBucket[i][j] = embedID[i][hashPara[j]]  
 return hashBucket

4、程序测试过程与结果分析

time\_start = time.time() # 开始时间  
pointID = divCell(database)  
seqID = divCell(query)   
# 进行多次哈希  
hashbucket1 = Lsh(pointID,m,1) ;hashBucket1 = Lsh(seqID,m,1)   
hashbucket2 = Lsh(pointID,m,2) ;hashBucket2 = Lsh(seqID,m,2)   
hashbucket3 = Lsh(pointID,m,3) ;hashBucket3 = Lsh(seqID,m,3)   
resultlist = []  
for i in range(len(hashBucket1)): # 遍历查询序列  
 value = ""   
 for j in range(len(hashbucket1)): # 遍历数据库序列  
 flag1 = (hashBucket1[i]==hashbucket1[j]).all() # 哈希族是否相等  
 flag2 = (hashBucket2[i]==hashbucket2[j]).all()  
 flag3 = (hashBucket3[i]==hashbucket3[j]).all()  
 if flag1 or flag2 or flag3 : # 只要有哈希族匹配就计算dtw optimal\_path, dtw\_score = dtw\_path(query[i], database[j])   
 if dtw\_score <= threshold: # DTW小于阈值则相似序列  
 value += str(j)+" "  
 resultlist.append(value)  
time\_end = time.time() #结束时间  
time\_avg = (time\_end - time\_start)/len(query) #每次查询的平均时间

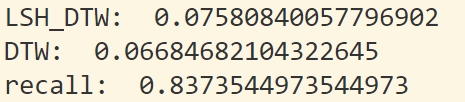
数据集描述

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | DataBase | Length | DTW阈值 |
| Plane | 105 | 144 | 10.0 |
| Symbols | 995 | 398 | 23.0 |
| CBF | 900 | 128 | 41.0 |
| TwoPatterns | 4000 | 128 | 25.0 |

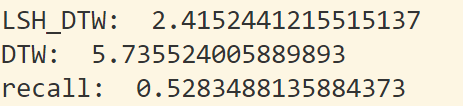
以上是对数据集的描述，分别为名称，数据库序列数量，时间序列长度以及DTW阈值。数据集来自UCR公开数据集。 使用tslearn库计算DTW（这样效率更高）。

运行在一台CPU为锐龙 5 3550H 3.7GHz，内存为16GB的笔记本上，操作系统为win10。各数据集的查询一次的平均时间如下：

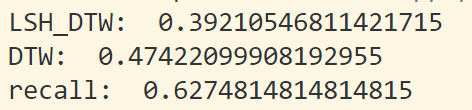
Plane数据集结果：



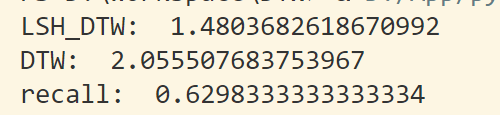
Symbols数据集结果：



CBF数据集结果：



TwoPatterns数据集结果：



发现LSH\_DTW的运行时间比DTW短，但牺牲了一定的准确率。实验中发现超参数对于结果的影响比较大，要取得理想结果还需要不断调整超参数。

## 5、后续可以创新的点

对于时序数据流的分析，目前大多数算法是建立在静态数据库的基础上，无法解决持续大量的数据流问题。时序数据流上的相似性研究可能是一个趋势。时序数据具有高维性，许多算法具有较高的时间开销和空间开销，对于资源有限的情况下如何优化，如何在储存空间有限时快速高效实现算法也是一大问题。

分工：算法1、算法3程序编写、报告撰写 