# 大规模相似度计算

## 为什么计算相似度

如果你想找出不同用户和不同内容之间的相似之处，就必须进行相似度计算。通过相似度计算，我们可以找到跟我喜欢的物品相似的其他物品，对应的，也可以找到跟我志趣相投的其他用户（因为我喜欢的物品，他们也喜欢）。

那么如何定义相似度？如果我们将两个用户的相似度范围限定在0-1，0是完全不相似度，1为最相似。接下来，需要限定相似度的计算场景，比如在电影推荐中，用户之间的相似度指的是用户对于电影的口味或者是兴趣。比如两个用户都喜欢科幻电影，或者都喜欢某个明星参演的电影。但是，我们知道用户的兴趣是多元化的，喜欢科幻电影的人也可能会喜欢其他口味的电影类型。所以，加入其他补充信息，比如用户的年龄、性别等基础人口统计信息，得到的相似度结果会更加合理。

在数学上通常将相似度计算归结为距离计算，也就是将两个用户或者项目抽象为空间中的两个点，然后计算两个点之间的距离。距离越短，相似度越高。

## 相似度的度量方式

##### 杰卡尔德距离（Jaccard Distance）

1. 杰卡德相似系数  
   两个集合A和B的交集元素在A，B的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号J(A,B)表示。

(2) 杰卡德距离  
 与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。杰卡德距离可用如下公式表示：

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

假设有一个矩阵，横向代表item，包括row1、row2和row3，纵向代表user的购买情况，如果买过某个item，则对应数值为1，反之则为0。

user

item

通过python计算item两两之间的杰卡尔德相似度：

import scipy.spatial.distance as dist

v1=np.array([0,1,0,1,0,1])

v2=np.array([0,1,1,0,1,0])

v3=np.array([0,1,1,0,5,0])

matrix1=np.array([v1,v2])

matrix2=np.array([v1,v3])

print(matrix1)

print(matrix2)

ds1=dist.pdist(matrix1, 'jaccard')

ds2=dist.pdist(matrix2, 'jaccard')

print(ds1)

print(ds2)

其输出为：

[0.8]

[0,8]

上面代码中，v3相较于v2改变了第5个元素的评分，但是v1、v2和v1、v3之间的相似度保持不变。所以，杰卡尔德相似度只会判断是否购买，并不会计算具体评分。

##### 欧式距离（Euclidean Distance）

欧式距离是比较容易理解的一种距离计算方法，来源于空间中两点的距离公式。用向量表示两个n维向量与之间的欧式距离为：

假设一个矩阵，横向代表item，包括row1、row2和row3，纵向代表user的购买情况。矩阵中各个元素是user对item的具体评分，实际场景中评分是Double类型。

user

item

使用python计算row1和row2之间的欧式距离：

vector1 = np.array([0,2,0,4,0,1])

vector2 = np.array([0,1,1,4,1,0])

vector3 = np.array([0,1,3,4,1,0])

vector4 = np.array([1,0,1,4,1,0])

def EuclideanDistance(vector1, vector2):  
 #ds1=np.sqrt(np.sum(np.square(vector1-vector2)))   
 # np.linalg.norm 用于范数计算，默认是二范数，相当于平方和开根号  
 return 1.0/(1.0 + np.linalg.norm(vector1 - vector2))

ds1= EuclideanDistance(vector1, vector2)

ds2= EuclideanDistance(vector1, vector3)

ds3= EuclideanDistance(vector1, vector4)

**print**(ds1)

**print**(ds2)

print(ds3)

其输出为：

0.3333333333333333

0.2240092377397959

0.2612038749637414

##### 余弦相似度（Cosine）

几何中的夹角余弦可以用来衡量两个向量方向之间的差异，放在推荐系统中，可以用来衡量user或者item的行为向量之间的差异。两个n维样本点与用类似于夹角余弦的概念来衡量他们之间的相似度。

用python计算下列矩阵中row1和row2的相似度：

user

item

vector1 = np.array([0,2,0,2,0,2])

vector2 = np.array([0,1,1,4,1,0])

vector3 = np.array([0,1,3,4,1,0])

vector4 = np.array([0,4,0,4,0,4])

def cosine(vector1, vector2):

cos=np.dot(vector1,vector2)/(np.linalg.norm(vector1)\*(np.linalg.norm(vector2)))

return cos

cos1=cosine(vector1,vector2)

cos2=cosine(vector1,vector3)

cos3=cosine(vector1,vector4)

print(cos1)

print(cos2)

print(cos3)

输出为：

0.6622661785325219

0.5555555555555556

1.0000000000000002

##### 皮尔逊相关系数（Pearson Correlation Coefficient）

皮尔逊相关系数公式实际上就是在计算夹角余弦之前将两个向量减去各个样本的平均值，达到中心化的目的。从知友的回答可以明白，皮尔逊相关函数是余弦相似度在维度缺失上面的一种改进方法。

利用pytho-numpy实现皮尔逊相关系数计算：

def Pearson(dataA,dataB):

# 皮尔逊相关系数的取值范围(-1 ~ 1),0.5 + 0.5 \* result 归一化(0 ~ 1)

return 0.5 + 0.5 \* np.corrcoef(dataA,dataB,rowvar = 0)[0][1]

##### 修正余弦相似度（Adjusted Cosine）

为什么需要在余弦相似度的基础上使用修正余弦相似度。比如在推荐系统中，X和Y两个用户对两个内容的评分分别为（1,2）和（4,5），使用余弦相似度得到的结果是0.98，两者极为相似。但从评分上看X似乎不喜欢2这个 内容，而Y则比较喜欢，余弦相似度对数值的不敏感导致了结果的误差，需要修正这种不合理性

利用python实现修正余弦相似度：

# 修正cosine 减去的是对item i打过分的每个user u，其打分的均值

data = np.mat([[1,2,3],[3,4,5]])

avg = np.mean(data[:,0]) # 下标0表示正在打分的用户

def AdjustedCosine(dataA,dataB,avg):

sumData = (dataA - avg) \* (dataB - avg).T # 若列为向量则为 dataA.T \* dataB

denom = np.linalg.norm(dataA - avg) \* np.linalg.norm(dataB - avg)

return 0.5 + 0.5 \* (sumData / denom)

print(AdjustedCosine(data[0,:],data[1,:],avg))

##### 汉明距离（Hamming Distance）

两个等长字符串s1与s2之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为2。

v1=np.array([1,1,0,1,0,1,0,0,1])

v2=np.array([0,1,1,0,0,0,1,1,1])

smstr=np.nonzero(v1-v2)

print(smstr) # 不为0 的元素的下标

sm= np.shape(smstr[0])[0]

print( sm )

#输出

#(array([0, 2, 3, 5, 6, 7]),)

#6

##### 曼哈顿距离（Manhatten Distance）

从名字就可以猜出这种距离的计算方法了。想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。实际驾驶距离就是这个“曼哈顿距离”(L1范数)。而这也是曼哈顿距离名称的来源，曼哈顿距离也称为城市街区距离(City Block distance)。

两个n维向量与之间的曼哈顿距离为：

利用python代码计算曼哈顿距离：

vector1 = np.array([1,2,3])

vector2 = np.array([4,5,6])

op3=np.sum(np.abs(vector1-vector2))

op4=np.linalg.norm(vector1-vector2,ord=1)

#输出

#9

#9.0

##### 对比分析

简单来看，每一个相似度度量指标都不是完美的，比如杰卡尔德无法将具体评分带入公式中，只是判断有和没有。余弦相似度在两个向量方向一致时相似度为1，这很显然是不完全正确的，比如上文中[0,2,0,2,0,2]和[0,4,0,4,0,4]这两个item的评分向量就相差很多，相似度为1 很显然不正确。

对比欧式距离的计算和夹角余弦的计算结果发现，欧式距离的相似度比较小，而夹角余弦相似度比较大，即夹角余弦更能反映两者之间的变动趋势，两者有很高的变化趋势相似度，而欧式距离较大是因为两者数值有很大的区别，即两者拥有很高的数值差异。

欧氏距离和余弦距离各自有不同的计算方式和衡量特征，因此它们适用于不同的数据分析模型。欧氏距离能够体现个体数值特征的绝对差异，所以更多的用于需要从维度的数值大小中体现差异的分析，如使用用户行为指标分析用户价值的相似度或差异。

余弦距离更多的是从方向上区分差异，而对绝对的数值不敏感，更多的用于使用用户对内容评分来区分兴趣的相似度和差异，同时修正了用户间可能存在的度量标准不统一的问题（因为余弦距离对绝对数值不敏感）。

## 基于评分的相似度计算工程实现

在推荐系统中，如果有user对item的评分，那就可以就行相似度计算了，其中包括item相似度与user相似度。但是，真实场景中用户点击或者购买某一商品时并不会对商品打分，这时可以通过一些方法进行构造式评分。

|  |  |
| --- | --- |
| **行为** | 评**分** |
| 点击 | 2.0 |
| 浏览详情 | 3.0 |
| 收藏 | 3.5 |
| 加购 | 4.0 |
| 下单 | 4.5 |

另外，有些推荐场景并没有用户点击、加购等复杂的操作行为，只有购买行为。这时可以用0和1表示用户的购买情况，0表示未买过，1表示买过。下图就是将购买行为转换成0和1 的矩阵。

|  |  |  |  |
| --- | --- | --- | --- |
| U\i | Item1 | Item2 | Item3 |
| User1 | 1 | 0 | 1 |
| User2 | 0 | 1 | 0 |
| User3 | 1 | 0 | 0 |

如果要得到标准的1-5的评分矩阵，需要将用户购买物品的次数转换成评分。数据归一化到[a,b]区间范围的方法：

（1）首先找到样本数据X的最小值Min及最大值Max  
（2）计算系数为：k=（b-a)/(Max-Min)  
（3）得到归一化到[a,b]区间的数据：norY=a+k(X-Min)

用python实现归一化操作为：  
def Normalization(x,a,b):  
 k = (b-a)/float(max(x)-min(x))  
 vec = [a+k\*(float(i)-min(x))/float(max(x)-min(x)) for i in x]  
 return [round(i, 2) for i in vec]

有上述方法转换得到的评分再取小数点后两位得到评分矩阵：

|  |  |  |  |
| --- | --- | --- | --- |
| User | Item | count | Rating |
| User1 | Item1 | 4 | 1.44 |
| User2 | Item2 | 10 | 1.15 |
| User3 | Item3 | 1 | 1.0 |

在MovieLens电影数据ml-20m中，计算所有元素的笛卡尔积，得到所有元素两两计算的相似度。这种常见的计算方式比较原始，也是非常耗时的。比如ratings.csv文件中Item去重后个数为2万多个，用户的维度有13万。如果计算Item的相似度，需要计算4亿次，时间复杂度较高。另外，由于用户user维度更大，两两计算相似度，那总共就要算次相似度，计算效率是个亟需解决的问题。存储方面，计算user相似度时，假设每次计算保存4个Byte，总共需要70G的数据量，空间存储也是问题。

### RDD分布式计算

当前代码中实现的相似度计算是通过笛卡尔积的方式计算RDD中两两匹配的相似度。RDD作为分布式计算数据集，可以很容易的实现多线程、多节点的进行相似度计算，比单机单进程的计算方式要快很多。一般业务场景下item数量要远远大于user数据量。所以，在大数据量下，即时单机可以容忍itemCF的计算时间，userCF则不行。最简单的RDD优化方式，可以通过广播变量的方式提高对比计算的效率。

### 向量化计算

为什么要进行向量化。向量化计算是机器学习领域常用的方法，也就是把循环遍历转换成向量计算，由此提高计算效率。在前面章节中介绍了Spark MLlib中的向量数据类型，我们用分布式矩阵CoordinateMatrix实现一个真实场景的Item相似度计算案例。

//利用Spark分布式矩阵计算Item相似度

def itemVectorSimilarity(spark: SparkSession,  
 trainData: Dataset[UserItem],  
 topN: Int) = {  
 //计算相似度矩阵  
 def standardCosine(matrix: CoordinateMatrix): RDD[MatrixEntry] = {  
 val similarity = matrix.toIndexedRowMatrix().columnSimilarities()  
 val sim = similarity.entries  
 sim  
 }  
  
 val sim = standardCosine(parseToMatrix(trainData)).map {  
 case MatrixEntry(user1, user2, sim) => (user1, (user2, sim))

}.groupByKey().

flatMapValues {x =>  
 val sim\_users = x.toList.sortBy(-\_.\_2).take(topN)  
 sim\_users  
 }.map(x => (x.\_1, x.\_2.\_1, x.\_2.\_2))  
}

### 近似最近邻方法ANN

ANN并非人工神经网络，而是近似邻居算法（Approximate Nearest Neighbor）的缩写。根据之前的方法，计算笛卡尔积属于蛮力（brute-force）搜索的方式，也就是在全空间进行搜索，为了加快查找的速度，几乎所有的ANN方法都是通过对全空间分割，将其分割成很多小的子空间，在搜索的时候，通过某种方式，快速锁定在某一（几）子空间，然后在该（几个）子空间里做遍历。可以看到，正是因为缩减了遍历的空间大小范围，从而使得ANN能够处理大规模数据的索引[[1]](#footnote-1)。关于ANN的算法实现有很多，比较典型的实现有Annoy、nmslib和SPTAG等等。几乎所有的ANN算法都是对于全空间的划分，大多数使用的是树模型。

1. LSH

局部敏感哈希（Locality-Sensitive Hashing）简称LSH是一种用于求解高维空间中的近似近邻搜索的方法。LSH保证了高维空间相近的点映射到低维空间相近的概率很高，所以这也是一个降维的过程。

1. K-Means Tree

K-Means Tree实际就是对数据做了多层K-means，每一层到当前的划分“叶子节点”包含样本数都小于1个。

1. K-D Tree

是对数据点在k维空间中划分的一种数据结构。k-d tree实际上是一种二叉树。

#### Annoy介绍

Annoy是近似最近邻算法实现的一个 C++ 库，它是（Approximate Nearest Neighbors Oh Yeah)的缩写，其本身带有 Python接口。用于搜索空间中给定查询项目的的近似项目。它也创建了大型只读文件的数据结构，这些数据结构被mmap[[2]](#footnote-2)到内存中，以便许多进程可以共享相同的数据。

**安装**

安装很简单，只需通过pip安装：

pip install --user annopy

这行命令会从PyPI上拉下Annoy最新版本。

对于 C++版本，只需从github中克隆项目，并插入声明#include "annoylib.h“即可。

**背景**

还有一些其他库可以实现最近的邻域搜索。Annoy几乎是执行速度最快的那一个（下面会做一些性能对比），但实际上还有另一个使得 Annoy 与众不同的功能：它能够使用静态文件作为索引。这意味着您可以跨进程共享索引。Annoy 还会将创建索引与加载索引分离，以便您可以将索引作为文件传递并快速映射到内存中。Annoy 的另一个好处是，它尝试最小化内存占用，因此索引非常小。

这个功能相当好用，如果要查找最近邻，并且有许多 CPU，则只需生成一次索引。您还可以传递和分发静态文件，以用于生产环境、Hadoop 作业等。任何进程都能够将索引加载（mmap）到内存中，并能够立即进行查找。

该项目在 Spotify[[3]](#footnote-3)上使用并进行音乐推荐。在运行矩阵分解算法后，每个用户/项目都可以在 f 维空间中表示为向量。Annoy帮助系统搜索相似的user/item，这是一个超过百万维度的空间，因此内存使用是首要问题。

**功能摘要：**

1. 欧几里德距离，曼哈顿距离，余弦距离，汉明距离(Hamming Distance)或点（内）积距离。

2. 余弦距离相当于归一化向量的欧几里德距离= sqrt（2-2 \* cos（u，v））

3. 如果维度很小（比如小于<100维），效果会更好，但即使是1,000维也能表现出色

4. 内存使用量小

5. 允许您在多个进程之间共享内存

6. 索引创建与查找分开（特别是在创建树后，您无法添加更多项目）

7. Python支持2.7,3.6和3.7。

8. 如果数据量很大，可以不用放到内存中，在磁盘上构建索引再使用

利用python调用计算item相似度的案例：

from annoy import AnnoyIndex  
class AnnoyMovie:  
 def \_\_init\_\_(self, dataFile):  
 self.datafile = dataFile  
 self.data = self.loadData()  
 self.ann, self.userNum = self.addItem()  
  
 def loadData(self):  
 print("Load data...")  
 data = []  
 for line in open(self.datafile):  
 itemid, vector = line.strip("\n").split(",")  
 data.append((itemid, vector))  
 return data  
  
 def addItem(self):  
 userNum = len(self.data.pop()[1].split(" "))  
 print("user num", userNum)  
 ann = AnnoyIndex(userNum, 'angular')  
 for itemId, vector in self.data:  
 ann.add\_item(int(itemId), map(float,vector.split(" ")))  
 ann.build(1000)  
 return ann, userNum  
  
 def getNear(self, index, itemId, simNum):  
 print(index.get\_nns\_by\_item(itemId, simNum))  
  
 def save(self):  
 print("保存索引文件...")  
 self.ann.save('movie.ann')  
  
 def load(self):  
 print("加载索引文件...")  
 u = AnnoyIndex(self.userNum, 'angular')  
 u.load('movie.ann')  
 return u

注意：Annoy只接受整数作为Item的标识符。请注意，它将为max（id）+1个Item分配内存，因为它假设您的Item编号为0 ... n-1。

**部分python API接口：**

AnnoyIndex（f，metric）

返回一个新的读写索引并存储f维度的向量。相似度计算可以是“angular”（类似于cosine距离），“欧几里德”，“曼哈顿”，“汉明”或“点积”。

a.add\_item（i，v）

用向量v添加项i（任何非负整数）。注意它将为max（i）+1项分配内存。

a.build（n\_trees）

构建一个n\_trees树的森林。查询时，更多树提供更高的精度。调用build后，不能再添加任何项目。

a.save（fn，prefault = False）将索引保存到磁盘并加载它（参见下一个函数）。保存后，不能再添加任何项目。

a.load（fn，prefault = False）

从磁盘加载索引。如果prefault设置为True，它将预读取整个文件到内存中。默认值为False。

a.unload（）为卸载。

a.get\_nns\_by\_item（i，n，search\_k = -1，include\_distances = False）

返回n个最接近的项目。在查询期间，它将检查最多search\_k节点，如果没有提供，则默认为n\_trees \* n。 search\_k为您提供更好的准确性和速度之间的运行时权衡。如果将include\_distances设置为True，它将返回一个包含两个列表的2元素元组：第二个元素包含所有相应的距离。

get\_nns\_by\_vector（v，n，search\_k = -1，include\_distances = False）

相同但是按向量v查询。 a.get\_item\_vector（i）返回先前添加的项目i的向量。 a.get\_distance（i，j）返回项i和j之间的距离。

#### hnswlib介绍[[4]](#footnote-4)

在说hnslib之前先说nmslib。非度量空间库（nmslib）是一个高效的跨平台的相似性搜索库，也是用于评估相似性搜索方法的工具包。其核心库没有任何第三方依赖。Nmslib是一个可扩展的库，这意味着可以添加新的搜索方法和距离函数。Nmslib可以直接在C++和python中调用。另外，还可以构建一个查询服务器，可以用Java调用。

快速近似邻居搜索，Hnswlib与nmslib来自于同一作者，而hnswlib更快，占用内存更小。

明显优势：

1. 体积小，调用方便，C++11开发且没有其他任何依赖

2. 接口包含C++，python和R

3. 支持新增索引，同时支持元素删除

4. 可以自定义距离计算

5. 与nmslib相比，明显减少了内存占用，同时缩短了构建时间

利用python调用hnswlib实现item相似度计算：

import hnswlib  
import numpy as np  
import sys  
  
class HnmslibMovie:  
 def \_\_init\_\_(self, dataFile):  
 self.datafile = dataFile  
 self.data, self.data\_labels = self.loadData()  
 self.ann = self.addItem()  
  
 #导入评分矩阵文件  
 def loadData(self):  
 print("Load data...")  
 data = []  
 data\_labels = []  
 for line in open(self.datafile):  
 index, vector = line.strip("\n").split(",")  
 rating\_list = vector.split(" ")  
 data\_labels.append(index)  
 data.append(rating\_list)  
 data = np.float32(data)  
 data\_labels = np.int32(data\_labels)  
 return data, data\_labels  
  
 def addItem(self):  
 self.dim = len(self.data[0])  
 self.num\_elements = len(self.data\_labels)  
 print("dimension num", self.dim)  
 #声明索引  
 p = hnswlib.Index(space='cosine', dim= self.dim) #space设置为l2, cosine 或 ip  
 #初始化索引，应事先知道最大数量的元素  
 p.init\_index(max\_elements= self.num\_elements, ef\_construction=200, M=16)  
 #插入元素（可以被调用多次）  
 p.add\_items(self.data, self.data\_labels)  
 #设置ef，控制召回  
 p.set\_ef(50) # ef设置必须要大于k  
 return p  
  
 def getNear(self, p, index, simNum):  
 #查询近邻，k=最近元素的个数（返回两个numpy数组）  
 labels, distances = p.knn\_query(self.data[np.where(self.data\_labels==index)[0][0]], k=simNum)  
 print("label:",labels, "distance: ", distances)  
 return labels, distances  
  
 def save(self):  
 print("保存索引文件...")  
 self.ann.save\_index('movie.ann')  
  
 def load(self):  
 print("加载索引文件...")  
 p = hnswlib.Index(space='cosine', dim=self.dim)  
 p.load\_index('movie.ann', max\_elements = self.num\_elements)  
 return p  
  
 def localQuery(self):  
 self.save()  
 p = self.load()  
 # 制定itemid 和 相似度的个数simNum  
 self.getNear(p, 1, 10)

#### Spark-MinHashLSH

Spark实现LSH（局部敏感哈希）如：BucketedRandomProjectionLSH 和 MinHash。

Spark-LSH实现Item相似度计算，Scala代码如下：

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.ml.linalg.{SparseVector, Vectors}  
import org.apache.spark.ml.feature.{BucketedRandomProjectionLSH, MinHashLSH, MinHashLSHModel}  
  
val user\_count = data\_rating.map(\_.\_1.toInt).max  
val data\_rating\_matrix = data\_rating.map (x => (x.\_2.toInt, (x.\_1.toInt, x.\_3))).  
 groupByKey().mapValues { x =>  
 val features = x.toList.map {  
 case (feature\_index, value) =>  
 (feature\_index, value)  
 }.sortBy(\_.\_1)  
 Vectors.sparse(user\_count - 1, features)  
}.sortByKey()  
val data\_rating\_df = spark.createDataFrame(data\_rating\_matrix).toDF("id", "features")

val mh = new MinHashLSH().  
 setNumHashTables(5).  
 setInputCol("features").  
 setOutputCol("hashes")  
  
val itemModel = mh.fit(data\_rating\_df)  
val transformed\_data\_rating = itemModel.transform(data\_rating\_df)  
transformed\_data\_rating.show(false)

val itemSimRDD = data\_rating\_matrix.collect().  
 map(x=> (x.\_1, itemModel.approxNearestNeighbors(data\_rating\_df, x.\_2, 10)))

itemSimRDD

LSH的具体参数：

Tables： 该参数越大，可以降低误报率（false negative rate）；参数越小，可以提高运行的效率。

### 实验对比与最佳方式

对比上述几种相似度计算方式，由于user比item相似度计算更加耗时，且userID无法进行抽样检查。这里选择item进行相似度对比。

分别用单机和集群两种资源配置方式进行测试。

单机配置：CPU-20cores，memory-30G

集群配置：8workers，CPU-160cores，Executor-19G

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data | 计算方式 | 加载并查询一次(Top10) | 加载并查询1千次(Top10) | 加载并查询1万次(Top10) |
| ml-1m | Annoy | 6.15s | 19.631s | / |
| hnswlib | 11.27s | 64.57s | / |
| SPTAG |  |  | / |
| CRM | Annoy |  |  |  |
| hnswlib |  |  |  |
| SPTAG |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Data | 计算方式 | Local | Cluster |
| CRM | RDD | 60min | 9min |
| Spark-Vectorization | 25.39min | 2.6min |
| Spark-LSH | 14.11min | 3.8min |

当前最佳方案： ANN + TopN，无需计算笛卡尔积，只要查找最近邻TopN即可。如果是计算笛卡尔积，时间复杂度至少 O（2^n）, 如果变成TopN查找问题，时间复杂度立马下降到O（n）。 由此，在真实推荐中，可以通过近似最近邻方式快速找到相似（接近）的用户和物品。

由此，大规模高纬度的相似度计算问题，实际上可以通过实时查询的方式快速找到相似度对象。

影响相似度计算时间的因素：

1. User和Item的数量
2. 用户产生的行为数量，这直接影响数据集的稀疏度
3. 硬件资源，理论上将节点越多，计算速度越快

### 常见问题与解决方法

**稀疏向量的压缩**

实际生产环境中，生成的用户向量都是非常稀疏的，构成的矩阵也是非常稀疏的，直白来说就是很多值都是0，有一些存储稀疏矩阵的格式。比如 CSR 或者 COO。

CSR：CSR是一个整体编码方式，由三部分构成，数值、列号和行偏移。

COO：COO每个元素用一个三元组表示（行号，列号，数值），只存储有值的元素，缺失值不存储。

这些存储格式，在常见的框架中都已经实现，比如 Python-scipy和Spark-MLlib。

**矩阵降维**

首先，相似度计算本身并不需要计算全量数据，比如超过3个月的行为数据就不应该作为计算数据，因为用户的行为习惯是变化的，太老的数据已经不能反映客户当前偏好和习惯。

协同过滤在实际场景中itemCF效果会好于UserCF，一般情况下只需要计算Item相似度即可。User相似计算一般会在计算用户特征相似时使用，找出新客户最相似的老客户就可以推荐老客户购买过的商品，这一方法可以解决用户冷启动的问题。

1. <https://yongyuan.name/blog/ann-search.html> [↑](#footnote-ref-1)
2. mmapped解释：在计算中，mmap是一个符合POSIX标准的Unix系统调用，它将文件或设备映射到内存中。它是一种内存映射文件I / O的方法。它实现了请求分页，因为文件内容不是直接从磁盘读取的，最初根本不使用物理RAM。在访问特定位置之后，以“懒惰”方式执行来自磁盘的实际读取。在不再需要内存之后，重要的是munmap（2）指向它的指针。可以使用mprotect管理保护信息，并且可以使用madvise强制执行特殊处理。 在Linux，Mac OS X和BSD中，mmap可以创建多种类型的映射。其他操作系统可能仅支持这些操作系统的子集，例如，共享映射在没有全局VFS或I / O高速缓存的操作系统中可能不实用。 [↑](#footnote-ref-2)
3. 一个正版流媒体音乐服务平台，2008年10月在瑞典首都斯德哥尔摩正式上线。 [↑](#footnote-ref-3)
4. <https://github.com/nmslib/hnswlib> [↑](#footnote-ref-4)