12 | 局部敏感哈希：如何在常数时间内搜索Embedding最近邻？

你好，我是王喆。

在深度学习推荐系统中，我们经常采用 Embedding 召回这一准确又便捷的方法。但是，在面对百万甚至更高量级的候选集时，线性地逐一计算 Embedding 间的相似度，往往会造成极大的服务延迟。

这个时候，我们要解决的问题就是，**如何快速找到与一个 Embedding 最相似的 Embedding？**这直接决定了召回层的执行速度，进而会影响推荐服务器的响应延迟。

今天，我们就一起来学习一下业界解决近似 Embedding 搜索的主要方法，局部敏感哈希。

推荐系统中的“快速”Embedding 最近邻搜索问题

在深度学习推荐系统中，我们经常会使用 Embedding 方法对物品和用户进行向量化。在训练物品和用户的 Embedding 向量时，如果二者的 Embedding 在同一个向量空间内（如图 1），我们就可以通过内积、余弦、欧式距离等相似度计算方法，来计算它们之间的相似度，从而通过用户 - 物品相似度进行个性化推荐，或者通过物品 - 物品相似度进行相似物品查找。

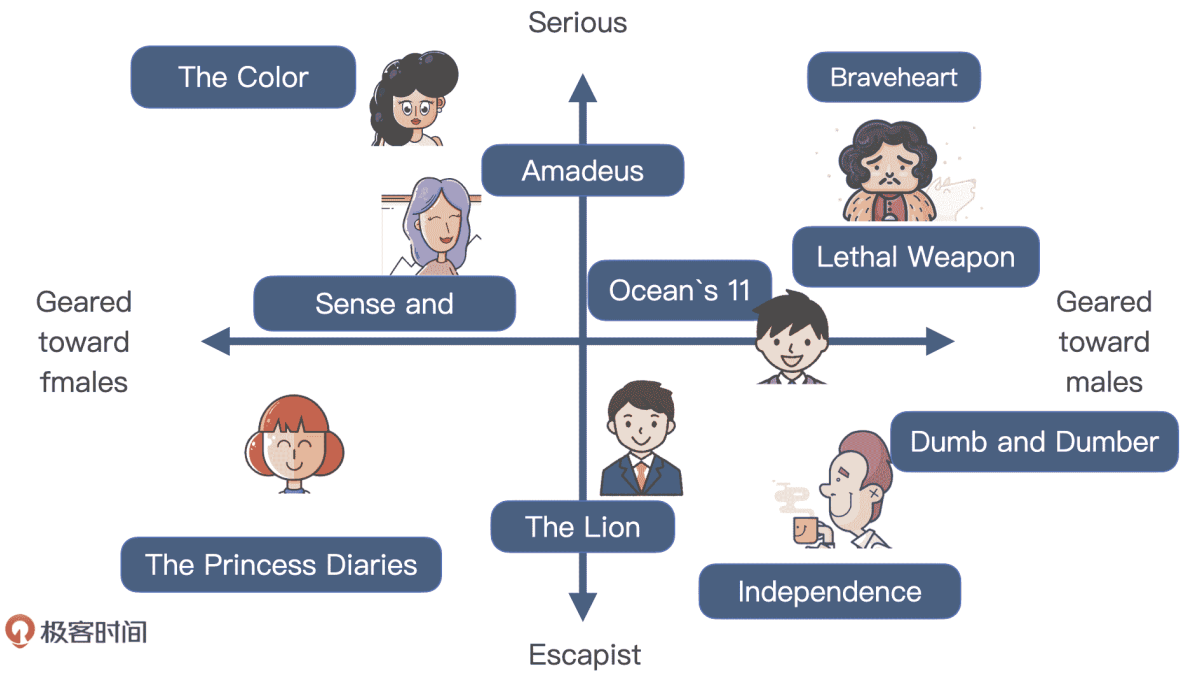


图1 用户和电影的Embedding向量空间

假设用户和物品的 Embeding 都在一个 k 维的 Embedding 空间中，物品总数为 n，那么遍历计算一个用户和所有物品向量相似度的时间复杂度是多少呢？不难算出是 O(k × n)。虽然这一复杂度是线性的，但物品总数 n 达到百万甚至千万量级时，线性的时间复杂度也是线上服务不能承受的。

换一个角度思考这个问题，由于用户和物品的 Embedding 同处一个向量空间内，因此**召回与用户向量最相似的物品 Embedding 向量这一问题，其实就是在向量空间内搜索最近邻的过程**。如果我们能够找到高维空间快速搜索最近邻点的方法，那么相似 Embedding 的快速搜索问题就迎刃而解了。

使用“聚类”还是“索引”来搜索最近邻？

遇到最近邻搜索的问题，我想大部分同学直觉上肯定会想到两种解决方案，**一种是聚类**，我们把相似的点聚类到一起，不就可以快速地找

到彼此间的最近邻了吗？**另一种是索引**，比如，我们通过某种数据结构建立基于向量距离的索引，在查找最近邻的时候，通过索引快速缩小范围来降低复杂度。这两种想法可不可行呢？我们一一尝试一下。

对于聚类问题，我想最经典的算法当属 K-means。它完成聚类的过程主要有以下几步：

1. 随机指定 k 个中心点；
2. 每个中心点代表一个类，把所有的点按照距离的远近指定给距离最近的中心点代表的类；
3. 计算每个类包含点的平均值作为新的中心点位置；
4. 确定好新的中心点位置后，迭代进入第 2 步，直到中心点位置收敛，不再移动。

到这里，整个 K-means 的迭代更新过程就完成了，你可以看下图 2。

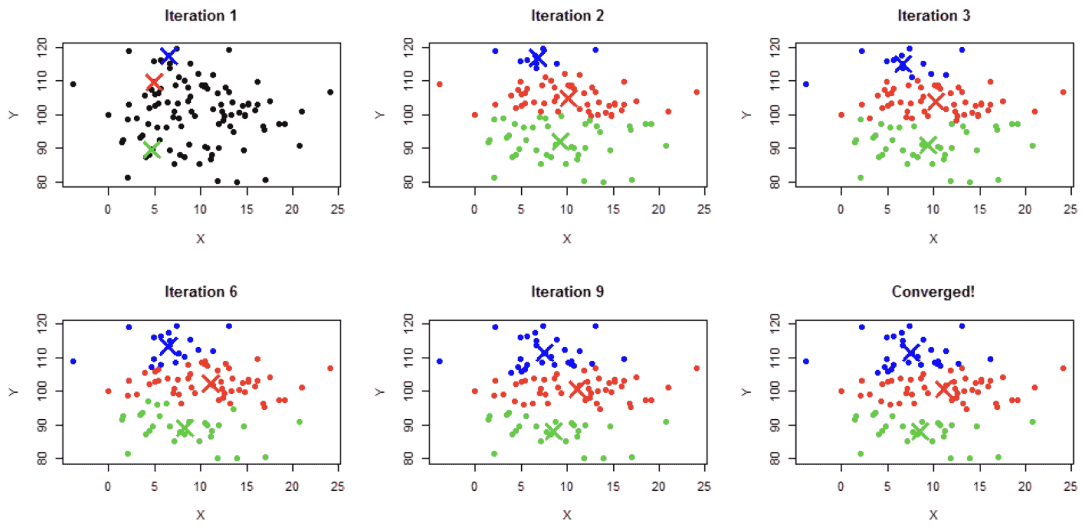


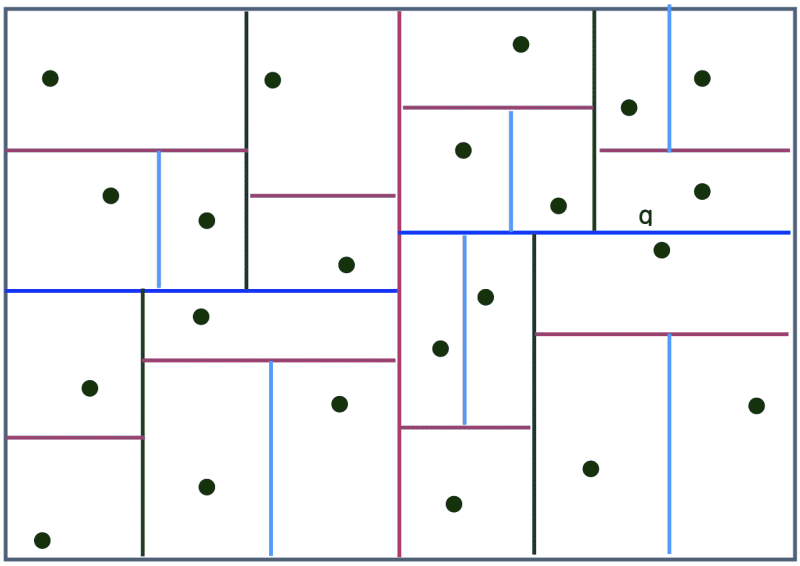
图2 三中心点的K-means算法迭代过程

如果我们能够在离线计算好每个 Embedding 向量的类别，在线上我们只需要在同一个类别内的 Embedding 向量中搜索就可以了，这会大大缩小了 Embedding 的搜索范围，时间复杂度自然就下降了。

但这个过程还是存在着一些边界情况。比如，聚类边缘的点的最近邻往往会包括相邻聚类的点，如果我们只在类别内搜索，就会遗漏这些近似点。此外，中心点的数量 k 也不那么好确定，k 选的太大，离线迭代的过程就会非常慢，k 选的太小，在线搜索的范围还是很大，并没有减少太多搜索时间。所以基于聚类的搜索还是有一定局限性的，解决上面的问题也会增加过多冗余过程，得不偿失。

既然聚类有局限性，那索引能不能奏效呢？我们这里可以尝试一下经典的向量空间索引方法 kd-tree（k-dimension tree）。与聚类不同，它是为空间中的点 / 向量建立一个索引。这该怎么理解呢？

举个例子，你可以看下图 3 中的点云，我们先用红色的线把点云一分为二，再用深蓝色的线把各自片区的点云一分为二，以此类推，直到每个片区只剩下一个点，这就完成了空间索引的构建。如果我们能够把这套索引“搬”到线上，就可以利用二叉树的结构快速找到邻接点。比如，希望找到点 q 的 m 个邻接点，我们就可以先搜索它相邻子树下的点，如果数量不够，我们可以向上回退一个层级，搜索它父片区下的其他点，直到数量凑够 m 个为止。



极客时间

图3 kd-tree索引

听上去 kd-tree 索引似乎是一个完美的方案，但它还是无法完全解决边缘点最近邻的问题。对于点 q 来说，它的邻接片区是右上角的片区，但是它的最近邻点却是深蓝色切分线下方的那个点。所以按照 kd-tree 的索引方法，我们还是会遗漏掉最近邻点，它只能保证快速搜索到近似的最近邻点集合。而且 kd-tree 索引的结构并不简单，离线和在线维护的过程也相对复杂，这些都是它的弊端。

那有没有更“完美”的解决方法呢？

局部敏感哈希的基本原理及多桶策略

为了“拯救”我们推荐系统的召回层，“局部敏感哈希”（Locality Sensitive Hashing，LSH）这一方法横空出世，它用简洁而高效的方法几乎完美地解决了这一问题。那它是怎么做到的呢？我们先来看看局部敏感哈希的基本原理。

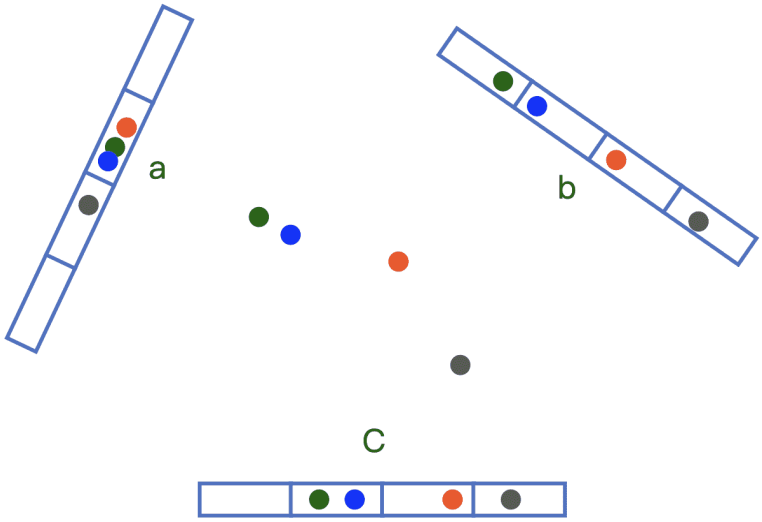
1. 局部敏感哈希的基本原理

局部敏感哈希的基本思想是希望让相邻的点落入同一个“桶”，这样在进行最近邻搜索时，我们仅需要在一个桶内，或相邻的几个桶内的元素中进行搜索即可。如果保持每个桶中的元素个数在一个常数附近，我们就可以把最近邻搜索的时间复杂度降低到常数级别。

那么，如何构建局部敏感哈希中的“桶”呢？下面，我们以基于欧式距离的最近邻搜索为例，来解释构建局部敏感哈希“桶”的过程。

首先，我们要弄清楚一个问题，如果将高维空间中的点向低维空间进行映射，其欧式相对距离是不是会保持不变呢？以图 4 为例，图 4 中间的彩色点处在二维空间中，当我们把二维空间中的点通过不同角度映射到 a、b、c 这三个一维空间时，可以看到原本相近的点，在一维空间中都保持着相近的距离。而原本远离的绿色点和红色点在一维空间 a 中处于接近的位置，却在空间 b 中处于远离的位置。

因此我们可以得出一个定性的结论：**欧式空间中，将高维空间的点映射到低维空间，原本接近的点在低维空间中肯定依然接近，但原本远离的点则有一定概率变成接近的点。**



极客时间

图4 高维空间点向低维空间映射

利用低维空间可以保留高维空间相近距离关系的性质，我们就可以构造局部敏感哈希“桶”。对于 Embedding 向量来说，由于 Embedding 大量使用内积操作计算相似度，因此我们也可以用内积操作来构建局部敏感哈希桶。假设 v 是高维空间中的 k 维 Embedding 向量，x 是随机生成的 k 维映射向量。那我们利用内积操作可以将 v 映射到一维空间，得到数值 $h(v) = v \cdot x$。

而且，我们刚刚说了，一维空间也会部分保存高维空间的近似距离信息。因此，我们可以使用哈希函数 h(v) 进行分桶，公式为：$h_{x,b}(v) = \lfloor wx \cdot v+b \rfloor$。其中，$\lfloor \rfloor$ 是向下取整操作，w 是分桶宽度，b 是 0 到 w 间的一个均匀分布随机变量，避免分桶边界固化。

不过，映射操作会损失部分距离信息，如果我们仅采用一个哈希函数进行分桶，必然存在相近点误判的情况，因此，我们可以采用 m 个哈希函数同时进行分桶。如果两个点同时掉进了 m 个桶，那它们是相似

点的概率将大大增加。通过分桶找到相邻点的候选集合后，我们就可以在有限的候选集合中通过遍历找到目标点真正的 K 近邻了。

刚才我们讲的哈希策略是基于内积操作来制定的，内积相似度也是我们经常使用的相似度度量方法，事实上距离的定义有很多种，比如“曼哈顿距离”“切比雪夫距离”“汉明距离”等等。针对不同的距离定义，分桶函数的定义也有所不同，但局部敏感哈希通过分桶方式保留部分距离信息，大规模降低近邻点候选集的本质思想是通用的。

2. 局部敏感哈希的多桶策略

刚才我们讲到了可以使用多个分桶函数的方式来增加找到相似点的概率。那你可能有疑问，如果有多个分桶函数的话，具体应该如何处理不同桶之间的关系呢？这就涉及局部敏感哈希的多桶策略。

假设有 A、B、C、D、E 五个点，有 h1和 h2两个分桶函数。使用 h1来分桶时，A 和 B 掉到了一个桶里，C、D、E 掉到了一个桶里；使用 h2来分桶时，A、C、D 掉到了一个桶里，B、E 在一个桶。那么请问如果我们想找点 C 的最近邻点，应该怎么利用两个分桶结果来计算呢？

如果我们用“且”（And）操作来处理两个分桶结果之间的关系，那么结果是这样的，找到与点 C 在 h1函数下同一个桶的点，且在 h2函数下同一个桶的点，作为最近邻候选点。我们可以看到，满足条件的点只有一个，那就是点 D。也就是说，点 D 最有可能是点 C 的最近邻点。因此，用且操作作为多桶策略，可以最大程度地减少候选点数量。但是，由于哈希分桶函数不是一个绝对精确的操作，点 D 也只是最有可能的最近邻点，却不一定是最近邻点，因此，“且”操作其实也增大了漏掉最近邻点的概率。

那如果我们采用“或”（Or）操作作为多桶策略，又会是什么情况呢？具体操作就是，我们找到与点 C 在 h1函数下同一个桶的点，或在 h2 函数下同一个桶的点。这个时候，我们可以看到候选集中会有三个点，分别是 A、D、E。这样一来，虽然我们增大了候选集的规模，减少了漏掉最近邻点的可能性，但增大了后续计算的开销。

当然，局部敏感哈希的多桶策略还可以更加复杂，比如使用 3 个分桶函数分桶，把同时落入两个桶的点作为最近邻候选点等等。

那么，我们到底应该选择“且”操作还是“或”操作，以及到底该选择使用几个分桶函数，每个分桶函数分几个桶，这还是一个工程上的权衡问题。我虽然不能给出具体的最佳数值，但可以给你一些取值的建议：

1. 点数越多，我们越应该增加每个分桶函数中桶的个数；相反，点数越少，我们越应该减少桶的个数；
2. Embedding 向量的维度越大，我们越应该增加哈希函数的数量，尽量采用且的方式作为多桶策略；相反，Embedding 向量维度越小，我们越应该减少哈希函数的数量，多采用或的方式作为分桶策略。

最后，我们再回头来解决课程开头提出的问题，局部敏感哈希能在常数时间得到最近邻的结果吗？答案是可以的，如果我们能够精确的控制每个桶内的点的规模是 C，假设每个 Embedding 的维度是 N，那么找到最近邻点的时间开销将永远在 O(C·N) 量级。采用多桶策略之后，假设分桶函数数量是 K，那么时间开销也在 O(K·C·N) 量级，这仍然是一个常数。

局部敏感哈希实践

现在，我们已经知道了局部敏感哈希的基本原理和多桶策略，接下来我们一起进入实践环节，利用 SparrowRecsys 训练好的物品 Embedding，来实现局部敏感哈希的快速搜索吧。为了保证跟 Embedding 部分的平台统一，这一次我们继续使用 Spark MLlib 完成 LSH 的实现。

在将电影 Embedding 数据转换成 dense Vector 的形式之后，我们使用 Spark MLlib 自带的 LSH 分桶模型 BucketedRandomProjectionLSH（我们简称 LSH 模型）来进行 LSH 分桶。其中最关键的部分是设定 LSH 模型中的 BucketLength 和 NumHashTables 这两个参数。其中，BucketLength 指的就是公式 2 中的分桶宽度 w，NumHashTables 指的是多桶策略中的分桶次数。

清楚了模型中的关键参数，执行的过程就跟我们讲过的其他 Spark MLlib 模型一样了，都是先调用 fit 函数训练模型，再调用 transform 函数完成分桶的过程，具体的实现你可以参考下面的代码。

```
def embeddingLSH(spark:SparkSession, movieEmbMap:M
  //将电影embedding数据转换成dense Vector的形式，便于之
  val movieEmbSeq = movieEmbMap.toSeq.map(item =>
  val movieEmbDF = spark.createDataFrame(movieEmbS


  //利用Spark MLlib创建LSH分桶模型
  val bucketProjectionLSH = new BucketedRandomProj
    .setBucketLength(0.1)
    .setNumHashTables(3)
    .setInputCol("emb")
    .setOutputCol("bucketId")
  //训练LSH分桶模型
  val bucketModel = bucketProjectionLSH.fit(movieE
  //进行分桶
```

```
  val embBucketResult = bucketModel.transform(movi

  //打印分桶结果
  println("movieId, emb, bucketId schema:")
  embBucketResult.printSchema()
  println("movieId, emb, bucketId data result:")
  embBucketResult.show(10, truncate = false)

  //尝试对一个示例Embedding查找最近邻
  println("Approximately searching for 5 nearest n
  val sampleEmb = Vectors.dense(0.795,0.583,1.120,
  bucketModel.approxNearestNeighbors(movieEmbDF, s
}
```

为了帮助你更加直观的看到分桶操作的效果，我把使用 LSH 模型对电影 Embedding 进行分桶得到的五个结果打印了出来，如下所示：

```
+-------+------------------------------+-----------
|movieId|emb                           |bucketId
+-------+------------------------------+-----------
|710    |[0.04211471602320671,..]      |[[-2.0], [1
|205    |[0.6645985841751099,...]      |[[-4.0], [3
|45     |[0.4899883568286896,...]      |[[-6.0], [-
|515    |[0.6064003705978394,...]      |[[-3.0], [-
|574    |[0.5780771970748901,...]      |[[-5.0], [2
+-------+------------------------------+-----------
```

你可一看到在 BucketId 这一列，因为我们之前设置了 NumHashTables 参数为 3，所以每一个 Embedding 对应了 3 个 BucketId。在实际的最近邻搜索过程中，我们就可以利用刚才讲的多桶策略进行搜索了。

事实上，在一些超大规模的最近邻搜索问题中，索引、分桶的策略还能进一步复杂。如果你有兴趣深入学习，我推荐你去了解一下 Facebook 的开源向量最近邻搜索库 FAISS，这是一个在业界广泛应用的开源解决方案。

小结

本节课，我们一起解决了“Embedding 最近邻搜索”问题。

事实上，对于推荐系统来说，我们可以把召回最相似物品 Embedding 的问题，看成是在高维的向量空间内搜索最近邻点的过程。遇到最近邻问题，我们一般会采用聚类和索引这两种方法。但是聚类和索引都无法完全解决边缘点最近邻的问题，并且对于聚类来说，中心点的数量 k 也并不好确定，而对于 kd-tree 索引来说，kd-tree 索引的结构并不简单，离线和在线维护的过程也相对复杂。

因此，解决最近邻问题最“完美”的办法就是使用局部敏感哈希，在每个桶内点的数量接近时，它能够把最近邻查找的时间控制在常数级别。为了进一步提高最近邻搜索的效率或召回率，我们还可以采用多桶策略，首先是基于“且”操作的多桶策略能够进一步减少候选集规模，增加计算效率，其次是基于“或”操作的多桶策略则能够提高召回率，减少漏掉最近邻点的可能性。

最后，我在下面列出了各种方法的优缺点，希望能帮助你做一个快速的复盘。

方法	优点	缺点
线性计算	简单直接	候选集规模太大时，效率过低
K-means	通过聚类缩小候选集范围	离线收敛过慢，中心点数不好确定， 存在边缘点问题
kd-tree	通过建立kd-tree索引快速找 到近邻点	结构较复杂，维护难度大， 存在边缘点问题
局部敏感哈希	简单，快速	单桶局部敏感哈希无法确保 找到的点就是最近邻点
多桶局部敏感哈希	简单、快速、准确	没有明显缺点

课后思考

如果让你在推荐服务器内部的召回层实现最近邻搜索过程，你会怎样存储和使用我们在离线产生的分桶数据，以及怎样设计线上的搜索过程呢？

欢迎你在留言区写出你的答案，更欢迎你把这一过程的实现提交 Pull Request 到 SparrowResys 项目，如果能够被采纳，你将成为这一开源项目的贡献者之一。我们下节课再见！