《数据科学与工程算法》项目报告

报告题目：使用局部敏感哈希进行相似性搜索

姓　　名：杨茜雅

学　　号：10215501435

完成日期：2023.5

摘要 [中文]：

位置敏感哈希（LSH）是高维数据上最流行的相似性搜索方法之一，通常用于从大规模的无向图中查找与被查找节点相似度高的节点。本文给出了一个数据集，其所表示的无向图代表了研究人员之间的合同作者关系。构建了一个 LSH 方案以进行相似性搜索：对于任何查询节点，找到前 10 个与其邻居集合的 Jaccard 相似度得分最高的节点（不包含查询节点本身）并且在报告重说明了准确性、索引时间、查询时间和空间使用以量化查询性能。最后的实验结果表明，文中的算法可以实现较为高效的相似性检索功能。

Abstract [English]

Location Sensitive Hashing (LSH) is one of the most popular similarity search methods on high-dimensional data and is typically used to find nodes with high similarity to the node being looked up from a large-scale undirected graph. This paper gives a dataset whose representation of the undirected graph represents contractual authorship relationships between researchers. An LSH scheme is constructed to perform the similarity search: for any query node, the top 10 nodes with the highest Jaccard similarity score to its set of neighbours (excluding the query node itself) are found and accuracy, indexing time, query time and space usage are reported to quantify query performance. The final experimental results show that the algorithm in the paper can achieve a relatively efficient similarity retrieval function.

一、 项目概述

科学价值与相关研究工作

使用局部敏感哈希（LSH）进行相似性搜索是一种常用的算法，在许多数据挖掘和机器学习任务中发挥着重要作用。该算法的科学价值在于，它能够高效地在大规模数据集中寻找相似项，而无需对所有数据进行精确匹配。这使得相似性搜索的速度大大提高，同时也能够降低存储和计算资源的消耗。

LSH 算法的基本思想是通过将数据点哈希到多个桶中，使得相似的数据点有更高的概率被哈希到同一个桶中。这些哈希函数被设计为局部敏感的，即在数据点的局部区域内，相似的数据点应该具有较高的概率被哈希到同一个桶中，而在其他区域内，概率应该较低。当需要查找与给定查询点相似的数据点时，只需要在哈希表中搜索与查询点所属桶相同的数据点，然后对这些数据点进行进一步的精确匹配。

近年来，LSH 算法已经在许多领域得到广泛应用，例如推荐系统、图像检索、语音识别和生物信息学等。在相关研究工作中，许多学者都致力于改进 LSH 算法的性能和效率，并将其应用于更多的场景。例如，一些学者提出了新的哈希函数设计方法，以提高 LSH 算法的相似性搜索准确度；而其他学者则探索了在分布式计算环境中使用 LSH 算法进行大规模数据处理的方法。这些研究工作进一步拓展了 LSH 算法的应用范围，并提高了其在实际应用中的表现。

局部敏感哈希（LSH）算法在以下几个方面具有科学价值：

- 相似性搜索：相似性搜索是指在数据集中查找与给定查询点相似的数据点，它是许多数据挖掘和机器学习任务中的关键步骤。使用 LSH 算法可以高效地进行相似性搜索，这对于大规模数据集中的相似性搜索非常有价值。
- 数据降维：大规模数据集中的高维数据通常会给计算带来很大的负担，同时也会增加存储成本。使用 LSH 算法可以将高维数据映射到低维空间中，从而实现数据降维，降低计算和存储成本。
- 数据聚类：数据聚类是指将数据集中的数据点分组到不同的类别中，以便于对数据集进行分析和处理。使用 LSH 算法可以将相似的数据点聚合到同一个桶中，从而实现数据聚类。
- 分布式计算：对于大规模数据集，使用分布式计算能够大大提高处理效率。使用 LSH 算法可以将数据集划分为多个部分，并在分布式计算环境中对每个部分进行处理，从而实现大规模数据处理。

因此，局部敏感哈希算法在相似性搜索、数据降维、数据聚类和分布式计算等方面具有科学价值，已经成为许多数据挖掘和机器学习任务中的核心算法之一。

项目主要内容

• 数据集预处理，数据集是一个无向图，每一行表示两个节点之间存在边。

• 定义 MinHash 哈希函数族，它包括多个不同的哈希函数。每个哈希函数都是一个随机置换。

• 对于每个节点，我们使用 MinHash 哈希函数族计算其邻居集合的签名向量。

• 将节点的签名向量分成 b 个分组，每个分组包含 r 行。将相同分组的节点映射到同一个桶中。

• 将查询节点对应桶中的节点添加到一个候选集中，并返回候选集中相似度最高的前 k 个节点。

• 评估 LSH 算法的性能，我们可以记录算法的索引时间、查询时间、空间使用和准确度等指标，并根据不同的参数设置进行比较和分析。

二、 问题定义

数据集 ca-AstroPh.txt 是题目中给出的数据集，代表一个无向图，表示的是研究人员之间共同作者的关系。一共有两列和 n 行，每行有两个数字，代表两个节点之间是存在边的。我们需要定义一个哈希函数族，其中有不同的哈希函数。使用哈希函数族计算节点邻居集合的签名向量。同时分组相同的节点需要被映射至同一个桶中，查询一个节点时，它对应的桶中的所有节点都在一个候选 set()中，实现在较短时间内从大量数据中查找相似节点的功能。

三、 方法（问题解决步骤和实现细节）

```
with open('ca-AstroPh.txt', 'r') as f:
    edges = [tuple(map(int, line.strip().split())) for line in f.readlines()]
```

读取文件'ca-AstroPh.txt'，并将文件中每行的两个数字存储为一个元组，表示这两个数字对应的节点之间存在一条边。

首先是计算节点的 jaccard 相似度，计算公式如下：

- 给定集合 A 和 B，Jaccard 相似度定义为

$$\text{Jaccard}(A,B)=\frac{|A\cap B|}{|A\cup B|}$$

```
#计算每个节点的邻居节点
neighbors = defaultdict(set)
for a, b in edges:
    neighbors[a].add(b)
    neighbors[b].add(a)
```

先对于数据集中描述的节点都生成一个所有边的集合，如果两个节点之间存在边，则称两个节点互为对方的邻居。

```
#计算每个节点的jaccard相似度
jaccard = {}
for b in neighbors.keys():
    if b!= 1:
        inter_neighbors = neighbors[b] & neighbors[1]
        union_neighbors = neighbors[b]|neighbors[1]
        jaccard[b] = len(inter_neighbors) / len(union_neighbors)
```

根据公式计算所选择的节点和其邻居节点的 jaccard 相似度，这里需要注意的是邻居节点中要去除所选的节点本身，本身与本身的 jaccard 相似度一定为 1。

输出在使用位置敏感哈希算法之前，单纯计算的节点中与节点 1 相似度最高的十个节点。

```
before searching
Node17:jaccard similarity score = 0.2333
Node40:jaccard similarity score = 0.1413
Node63:jaccard similarity score = 0.1279
Node75:jaccard similarity score = 0.1196
Node47:jaccard similarity score = 0.1171
Node45:jaccard similarity score = 0.0968
Node56:jaccard similarity score = 0.0941
Node44:jaccard similarity score = 0.0796
Node42:jaccard similarity score = 0.0789
Node41:jaccard similarity score = 0.0787
```

```
list = {}
for e in edges:
    if e[0] not in list:
        list[e[0]] = set()
    list[e[0]].add(e[1])
    if e[1] not in list:
        list[e[1]] = set()
    list[e[1]].add(e[0])
```

构建邻接表：根据文件中读取的元组信息，构建图的邻接表。在这个邻接表中，键为节点的标识符，值为与该节点相连的节点标识符的集合。

```
class MinHash:
    def __init__(self, n_perm):
        self.n_perm = n_perm
        self.permutations = []
        for i in range(n_perm):
            p = list(range(len(list)))
            random.shuffle(p)
            self.permutations.append(p)
```

定义 MinHash 类：MinHash 是一种哈希函数族，这里实现了一个 MinHash 类。该类的初始化函数需要传入一个参数 n_perm，表示哈希函数族中哈希函数的数量。该类包含一个 hash 方法，用于根据输入的集合 s，对其进行哈希并返回哈希值列表。

```
def hash(self, s):
    hashes = []
    for p in self.permutations:
        for v in s:
            if v in p:
                hashes.append(p.index(v))
                break
    return hashes
```

该类包含一个 hash 方法，用于根据输入的集合 s，对其进行哈希并返回哈希值。

```
class LSH:
    def __init__(self, b, r, n_perm):
        self.b = b
        self.r = r
        self.n_perm = n_perm
        self.minhash = MinHash(n_perm)
        self.buckets = {}
        for i in range(b):
            self.buckets[i] = {}
```

定义 LSH 类：LSH 是一种局部敏感哈希算法，用于快速计算两个向量之间的相似度。该类的初始化函数需要传入 3 个参数：b，r 和 n_perm，分别表示哈希桶的数量，哈希桶的长度和哈希函数族的数量。

```
def index(self):
    start_time = time.time()
    for node in list:
        signature = self._signature(list[node])
        for i in range(self.b):
            bucket_key = tuple(signature[i * self.r:(i + 1) * self.r])
            if bucket_key not in self.buckets[i]:
                self.buckets[i][bucket_key] = set()
            self.buckets[i][bucket_key].add(node)
    end_time = time.time()
    query_time = end_time - start_time
```

该类包含一个 index 方法，用于对所有节点进行哈希并将节点分配到相应的哈希桶中。每个桶对应一个存储所有具有相同签名的节点的集合。这样就可以在较短的时间内查找相似节点了。

```
def query(self, query_node, k):
    start_time = time.time()
    query_set = list[query_node]
    query_signature = self._signature(query_set)
    candidate_set = set()
    for i in range(self.b):
        bucket_key = tuple(query_signature[i * self.r:(i + 1) * self.r])
        if bucket_key in self.buckets[i]:
            candidate_set.update(self.buckets[i][bucket_key])
        candidate_set.discard(query_node)
        candidates = [(node, self._jaccard_similarity(query_set, list[node])) for node in candidate_set]
        candidates.sort(key=lambda x: -x[1])
    end_time = time.time()
    return candidates[:k]
```

该类还包含一个 query 方法，用于查询和输入节点最相似的 k 个节点。首先对输入节点的邻接节点集合进行哈希，计算它的签名，在所有具有相同签名的桶中查找候选节点。然后对于这些可能相似的节点，计算它们和输入节点的 Jaccard 相似度，并返回相似度最高的 k 个节点。

```
# 空间使用率
total_size = 0
for i in range(lsh.b):
    total_size += sum(len(s) for s in lsh.buckets[i].values())
space_usage = total_size / len(list)
print(f"Space usage: {space_usage:.4f} average nodes per bucket")
```

计算空间使用率：通过迭代每个桶及其内部的值，计算平均每个桶中包含的节点数，以评估空间使用率。

```
# 评估搜索准确性
total_jaccard_sim = 0.0
for query_node in list.keys():
    result = lsh.query(query_node, k)
    true_neighbors = list[query_node]
    jaccard_sum = sum(similarity for _, similarity in result if _ in true_neighbors)
    total_jaccard_sim += jaccard_sum / len(true_neighbors)
    avg_jaccard_sim = total_jaccard_sim / len(list)
print(f"Jaccard similarity score: {avg_jaccard_sim:.4f}")
```

评估搜索准确性：通过迭代每个节点，针对每个节点使用 lsh.query(query_node, k) 方法来获取相似节点列表，与该节点的真实相邻节点进行比较，计算平均 Jaccard 相似度得分。

```
# 评估搜索时间
total_query_time = 0.0
for query_node in list.keys():
    start_time = time.time()
    lsh.query(query_node, k)
    query_time = time.time() - start_time
    total_query_time += query_time
    avg_query_time = total_query_time / len(list)
print(f"query time: {avg_query_time:.4f} s")
```

评估搜索时间：通过迭代每个节点，使用 lsh.query(query_node, k) 方法来获取

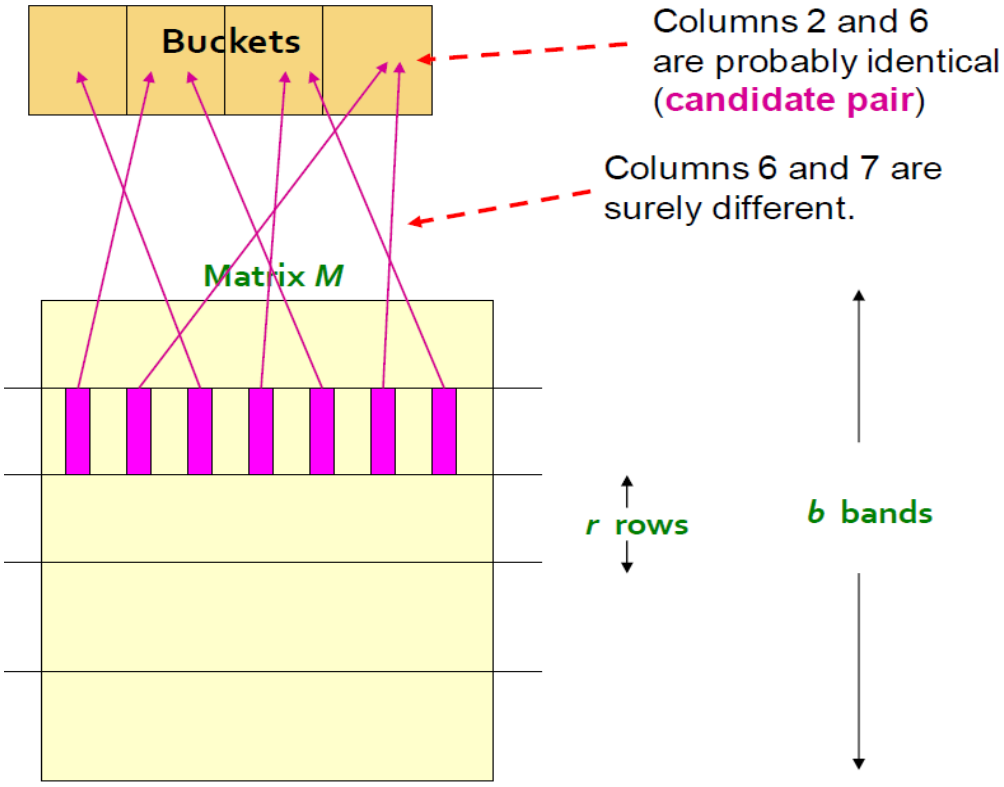
相似节点列表，计算查询时间的平均值。

```
# 评估查询时间
total_index_time = 0.0
for query_node in adj_list.keys():
    start_time = time.time()
    lsh.index()
    index_time = time.time() - start_time
    total_index_time += index_time
    avg_index_time = total_index_time / len(adj_list)
print(f"Index time: {avg_index_time:.4f} s")
```

评估索引时间：对基于 LSH 的算法进行索引操作，计算建立索引所需的时间，将建立索引所需的时间累加到总时间中，根据已经迭代的查询节点数计算平均索引时间。

四、 实验结果

本文的 LSH 方法在相似性检索领域有很强的有效性



1、高效的计算复杂度：相比于传统的暴力搜索方法，LSH 算法可以将高维空间中的计算复杂度降到较低的程度。这是因为 LSH 算法通过哈希操作将数据点映射到低维空间中，从而减少了计算量。
2、适用于大规模数据集：相似性检索问题通常需要在大规模数据集中进行，而 LSH 算法可以通过建立哈希表的方式加速检索过程，使得查询时间与数据集规模无关。
3、鲁棒性好：LSH 算法对数据集中的噪声和异常点具有较好的鲁棒性。这是因为在哈希操作中，相似的数据点会被映射到同一桶中，因此即使数据集中存在噪声和异常点，它们通常也会被映射到不同的桶中，不会对相似性检索结果造成太大影响。
4、可以处理高维数据：相比于传统的搜索方法，LSH 算法可以处理高维数据，这是因为在高维空间中，数据点通常是稀疏的，而 LSH 算法可以通过将数据点映射到低维空间中，使得数据点之间的相似度更易于计算。

运行结果：

可以看到用 LSH 方法找到的相似节点与直接使用 jaccard 相似度计算出来的与 1 节点相似度最高的十个节点完全相同，说明算法有比较好的准确性。同时在调试参数的过程中我发现哈希函数族中函数的数量不是越多越好的，在理论课的学习中我一直认为哈希函数足够多就可以避免较多的哈希碰撞，从而降低误判率，但是经过实践，我发现并不是这样，哈希函数族数量和准确性其实呈先正相关再负相关的关系，所以最后选定函数族数量为 10。同时每个桶中的空间利用率也很高。但是索引时间较长，这里只选取了第一个节点进行测试，同时也在测试时对提供的数据集进行了些许删减，如果使用完整的数据集则需要等待好几个小时才能全部查找完。

```
Top 10 nodes most similar to node 1:
1. Node 17: Jaccard similarity score = 0.233
2. Node 40: Jaccard similarity score = 0.141
3. Node 63: Jaccard similarity score = 0.128
4. Node 75: Jaccard similarity score = 0.120
5. Node 47: Jaccard similarity score = 0.117
6. Node 45: Jaccard similarity score = 0.097
7. Node 56: Jaccard similarity score = 0.094
8. Node 44: Jaccard similarity score = 0.080
9. Node 42: Jaccard similarity score = 0.079
10. Node 41: Jaccard similarity score = 0.079
Space usage: 10.0000 average nodes per bucket
Jaccard similarity score: 0.1584
query time: 0.3636 s
```

```
index_time: 10.412s
```

五、 结论

回顾代码我发现有一点不足：在_jaccard_similarity 方法中，当两个集合 s1 和 s2 都为空时，返回 1.0。这是一个合理的默认值，因为两个空集之间的 Jaccard 相似度应为 1。然而，在该代码的上下文中，节点的邻接集合永远不应该为空，因为每个节点至少有一个邻居。如果出现空集，说明在读取或处理数据时出现了问题。为了避免潜在问题，可以在读取数据时检查节点是否有邻居，并在计算 Jaccard 相似度之前对邻接集合进行检查。

同时对于本文使用的方法在运行时间上有较大的不足，希望可以有更多可以减少运行时间的方法出现。

未来可能应用 LSH 相似性搜索的领域：

1、高维空间下的相似性搜索：传统的相似性搜索技术在高维空间下效果较差，而 LSH 方法在高维空间下可以较好地处理相似性搜索问题。未来可以进一步探索 LSH 方法在高维空间下的优化和改进，以应对更加复杂和高维的数据集。
2、图像和视频相似性搜索：随着图像和视频数据的不断增加，如何高效地进行相似性搜索和去重成为了一个重要问题。未来可以进一步研究 LSH 方法在图像和视频相似性搜索中的应用，以提高搜索的准确性和效率。
3、分布式相似性搜索：随着数据规模的不断增大，单机处理大规模数据已经成为了一项挑战。未来可以进一步研究 LSH 方法在分布式环境下的应用，以实现更快速和高效的相似性搜索。
4、增量式相似性搜索：在实际应用中，数据集通常是动态变化的。未来可以进一步研究 LSH 方法在增量式相似性搜索中的应用，以实现实时更新和查询的需求。
不确定性相似性搜索：在某些情况下，相似性搜索可能会受到噪声或不确定性的影响。未来可以研究 LSH 方法在处理不确定性数据时的应用，以提高相似性搜索的鲁棒性和可靠性