华中科技大学

大数据存储管理课程报告

姓　　名：　　张家豪

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　大数据 2101

学　　号：　　U202115434

指导教师：　　施展

分数	
教师签名	

2024 年 4 月 20 日

基于 LSH 算法的设计与实现

1. 实验背景

最近邻搜索是计算机科学中的一项重要任务，它在各种应用中都有着广泛的应用，如信息检索、推荐系统、模式识别等。随着数据规模的不断增大，传统的最近邻搜索算法在处理大规模数据时面临着严重的效率问题。

局部敏感哈希（Locality Sensitive Hashing，LSH）作为一种近似最近邻搜索技术，能够在大规模数据集上实现高效的最近邻搜索。LSH 通过将高维数据映射到低维空间，并利用哈希函数将相似的数据点映射到相同的哈希桶中，从而实现快速的近似最近邻搜索。

通过 hash function 映射变换操作，将原始数据集合分成了多个子集合，而每个子集合中的数据间是相邻的且该子集合中的元素个数较小，因此将一个在超大集合内查找相邻元素的问题转化为了在一个很小的集合内查找相邻元素的问题，显然计算量下降了很多。

本实验旨在研究和实现 LSH 算法，并对其性能进行评估，并提出空间优化的改进做法。通过设计和实现 LSH 算法，我们可以探索其在不同数据集和参数设置下的搜索质量、搜索时间以及空间开销等性能指标，进一步了解 LSH 在大规模数据集上的应用潜力。

2. 数据结构的设计

基于 go 语言设计了如下数据结构用于实现 lsh 算法

```
type LSHParameters struct {
    B int // 划分段数
    R int // 每段行数
    K int // 每段哈希表的桶数
}
type LSHIndex struct {
    hashFunctions []*func(int) int
    hashTables    []*map[int][]int
    Params    LSHParameters
}
```

3 操作流程分析

传统的 lsh 算法使用了如下三个步骤完成数据的索引建立过程：Shingling、MinHashing 和 LSH。

3.1 Shingling

首先，我们需要使用用 k-shingling（和 one-hot 编码）将文本转换为稀疏向量：

```
kShingling 将文本转换为 k-shingling 后的稀疏向量
func kShingling(text string, k int) map[string]bool {
    shingles := make(map[string]bool)
    words := strings.Fields(text)
    for i := 0; i <= len(words)-k; i++ {
        shingle := strings.Join(words[i:i+k], " ")
        shingles[shingle] = true
    }
    return shingles
}

OneHotEncoding 将 k-shingling 转换为稀疏向量（one-hot 编码）
func OneHotEncoding(shingles map[string]bool, vocabulary map[string]int) []int {
    vector := make([]int, len(vocabulary))
    for shingle := range shingles {
        index, ok := vocabulary[shingle]
        if ok {
            vector[index] = 1
        }
    }
    return vector
}
```

3.2 MiniHash

接着，我们需要用最小哈希将稀疏向量转为稠密向量并获取签名值

```
func (lsh *LSHIndex)MinHashing(vector []int) []int {
    signatures := make([]int, len(lsh.hashFunctions))
    for i, hf := range lsh.hashFunctions {
        minHash := -1
        for j, value := range vector {
            if value == 1 {
                hashValue := hf(fmt.Sprintf("%d-%d", j, 1))
                if minHash == -1 || hashValue < minHash {
                    minHash = hashValue
```

```
                        }
                    }
                }
                signatures[i] = minHash
        }
        return signatures
}
```

3.3 LSH

最后也是最重要的一步，使用 LSH，它将获取我们在上文得到的签名值并寻找哈希冲突。我们使用 band 方法。

1. 划分签名矩阵为 b 个段，每一个段有 r 行
2. 对于每一个段，hash 每一个列的一部分到一个有 k 个 bucket 的 hash 表
3. 候选对是能至少有一个 hash 到同一个 bucket 的对
4. 调节 b 和 r 来获取最多的相似对、最少的不相似对

```
// IndexData 将数据索引到 LSH 索引中
func (index *LSHIndex) IndexData(data [][]bool) {
        for i, row := range data {
                for j, val := range row {
                        for b := 0; b < index.Params.B; b++ {
                                hashValue := Hash(fmt.Sprintf("%d-%d-%d", i,
j, b))
                                tableIndex := hashValue %
uint32(len(index.Tables[b][j]))
                                index.Tables[b][j][tableIndex] =
append(index.Tables[b][j][tableIndex], i)
                        }
                }
        }
}
// GetCandidates 获取候选对
func (index *LSHIndex) GetCandidates() [][]int {
        candidates := make([][]int, 0)
        for _, table := range index.Tables {
                for _, bucket := range table {
                        for _, entries := range bucket {
                                if len(entries) > 1 {
                                        for i := range entries {
                                                for j := i + 1; j < len(entries);
j++ {
                                                        candidates =
append(candidates, []int{entries[i], entries[j]})
```

```
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                            return candidates
}
```

3.4 查找

查找的过程也不复杂

1. 将查询数据经过 LSH hash function 哈希得到相应的桶号；
2. 将桶号中对应的数据取出；（为了保证查找速度，通常只需要取出前 2L 个数据即可）；
3. 计算查询数据与这 2L 个数据之间的相似度或距离，返回最近邻的数据；

4 减小空间开销的方案

在构建 LSH 索引之前，可以对数据进行预处理和过滤，以去除冗余信息和噪声数据，从而减少 LSH 索引的存储空间。可以使用特征选择或者降维技术对数据进行预处理，以减少数据的维度和存储空间需求。

在实际应用中，可以考虑动态调整 LSH 索引的 k，b，r 的值，根据数据集的变化和查询需求来优化索引的存储空间和搜索性能。可以使用自适应算法或者在线学习方法来动态调整 LSH 索引的参数和结构，以适应不断变化的数据环境。