# **文件指纹生成算法**

### **一、代码中需要生成文件指纹的四种数据类型**

在文件指纹生成代码种，我们需要对四种类型的数据进行指纹生成

（str、float、set()、dict())

1、domain\_pred\_res: 领域预测结果，str

2、domain\_pred\_match\_num: 领域预测结果的置信度，float

3、words: 文件名 / 标题关键词提取结果，set 其中不包括跨领域词

4、file\_key\_word 存储schema和对应schema的结果 dict()为string:list list中为提取出的信息

## **二、生成指纹的算法**

需求：对于相似的文本生成的Hash值也要相似

### **0、hashlib库**

hashlib是Python标准库中的一个模块，提供了多种最基本的哈希算法，包括MD5、SHA-1、SHA-256等。

1. import hashlib
2. ​
3. # algorithm='md5'/'sha1'/'sha256'
4. hash\_object = hashlib.new(algorithm)
5. hash\_object.update(data)

### **0、ssdeep库**

ssdeep 是一种用于计算文件和文本之间相似性的工具，基于 Nilsimsa 算法，可以生成文档的哈希值，用于比对文档的相似性。在 Python 中，可以使用 ssdeep 库进行文档指纹的计算和比对

对比两个文件相似度，python 中可通过 difflib.SequenceMatcher/ssdeep/python\_mmdt/tlsh 实现，在大量需要对比，且文件较大时，需要更高的效率，可以考虑模糊哈希（fuzzy hash）

PS：直接使用pip install XXX指令在本地环境配置有点问题

解决下载安装包，直接安装pip install .\ssdeep-windows-32\_64-master.zip

安装包在官网可以下下载

参考：Python 利用模糊哈希实现对比文件相似度详解

<https://my.oschina.net/u/3905674/blog/8400953>

1. import ssdeep
2. ​
3. ssdeep\_hash = ssdeep.hash(file\_content)

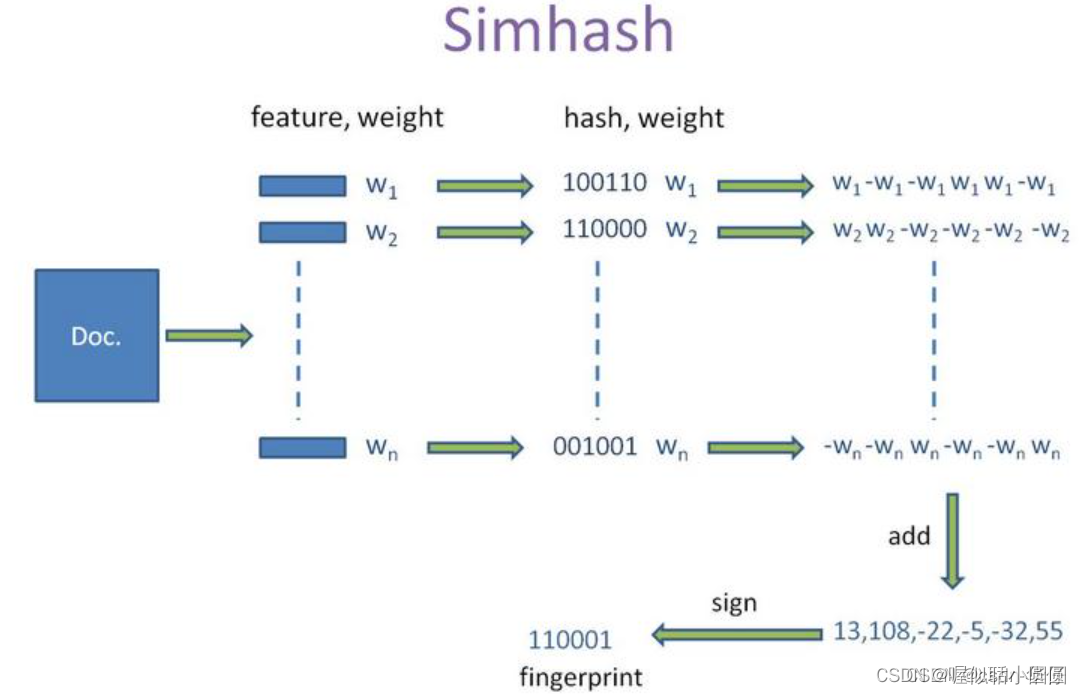
### **1、Simhash**

**Simhash**是一种用于文本相似性比较的哈希算法， google 用来处理海量文本去重的算法，可以将一个文档转换成一个 64 位的字节（特征字）。Simhash可以基于文本内容的特征向量，对相似的文本生成相似的哈希值（具有较好的局部变化容忍性，对于细微差异的文本可以生成相似的指纹）。

**算法步骤**：

1. **分词**：过滤标点等，移除停用词，提取n个特征关键词来表征文本
2. **哈希**：通过hash算法将分词转换为hash值
3. **加权**：分词权重与分词哈希值相乘
4. **累加**：将加权哈希值累加形成一个序列串
5. **二值化**：将序列串转化为0-1串
6. **比较**：（相似度计算-汉明距离）

**算法原理图：**



**例子：**



**应用场景：**一般是对**文本大于500+**的内容提前指纹做相似度比较，如果文本较短的话，相似度就会有较大的偏差。

**参考教程：**[海量数据去重之SimHash算法简介和应用 - 墨天轮 (modb.pro)](https://www.modb.pro/db/115729)

[simhash - 大辉\_FFf - 博客园 (cnblogs.com)](https://www.cnblogs.com/do-your-best/p/9846174.html)（包含相似度计算）

[simhash - 码农教程 (manongjc.com)](http://www.manongjc.com/detail/22-ogzpqhxcibhxigw.html)

[(107条消息) 文本相似度计算——Simhash算法（python实现）](https://blog.csdn.net/Trisyp/article/details/113623966?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=2)*[simhash文本相似度](https://blog.csdn.net/Trisyp/article/details/113623966?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=2)*[Trisyp的博客-CSDN博客](https://blog.csdn.net/Trisyp/article/details/113623966?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-1-113623966-blog-104106867.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=2)

1. from simhash import Simhash, SimhashIndex
2. def generate\_simhash\_fingerprint(text):
3. simhash = Simhash(text)
4. # 返回一个十六进制数
5. return simhash
6. # 返回Simhash值的哈希摘要（可以选择MD5、SHA-1等哈希算法）
7. # return hashlib.md5(str(simhash.value).encode()).hexdigest()
8. def calculate\_similarity(simhash1, simhash2):
9. distance = simhash1.distance(simhash2)
10. similarity = 1 - (distance / 64) # 假设Simhash的num\_perm为64
11. return similarity

#### **（1）文件指纹生成**

使用 generate\_simhash\_fingerprint() 函数生成每个text字符串的 Simhash 指纹

Simhash算法使用哈希函数将文本的特征转换为一个64位的二进制数，每个位上的值代表了该特征的重要性。最后，将这个64位的二进制数转换为十六进制表示形式，以方便查看和比较。如果要映射为哈希摘要，可以使用MD5等哈希算法（**但后续计算相似度时重新转换回二进制的Simhash值影响相似度计算**）

#### **（2）指纹相似度计算**

calculate\_similarity() 函数计算两两之间的相似度

1. 计算 simhash1 和 simhash2 之间的**汉明距离**，即两个 Simhash 值在二进制形式下不同位的数量。
2. 将汉明距离除以 Simhash 的长度，假设 Simhash 的长度为 64（num\_perm 的默认值）。
3. 计算相似度，通过用 1 减去步骤 2 中得到的比值，得到一个相似度值。
4. 返回相似度值。

**汉明距离**（Hamming distance）是衡量两个 Simhash 值之间差异程度的度量。在信息论中，两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数。

在这里，相似度是通过汉明距离计算得到的，值越接近 1，表示两个 Simhash 值越相似。因此，返回的相似度值越接近 1，表示两个 Simhash 值越相似。一般情况下，如果相似性大于某个阈值（例如0.8或0.9），我们可以认为两个Simhash值代表的文本或数据在内容上相似，可以视为相同或者具有相似性。

例如下： 10101 和 00110 从第一位开始依次有第一位、第四、第五位不同，则海明距离为 3.

汉明距离是指两个**等长字符串**之间对应位置上不同字符的个数。这种距离计算方法可能并不适用于所有情况，特别是在文本长度差异较大时。可以尝试其他相似度计算方法，例如**余弦相似度**等，看看是否能得到更合理的结果。

谷歌经过工程验证认为当两个64bit的二值化simhash值的汉明距离超过3则认为不相似，所以判重问题就转换为求两个哈希值的汉明距离问题

#### **（3）测试效果：还不错(适合于长文本的str类型数据)**

1. 文件指纹1: <simhash.Simhash object at 0x0000016C01D086D0>
2. 文件指纹2: <simhash.Simhash object at 0x0000016C046C5940>
3. 文件指纹3: <simhash.Simhash object at 0x0000016C046C5820>
4. 相似度1和2: 0.875
5. 相似度1和3: 0.515625
6. 相似度2和3: 0.484375

### **2、Minhash（最小哈希）**

Minhash是一种用于**集合**相似性比较的哈希算法，可以用于处理文本的相似性。通过对文本内容进行集合表示，然后使用Minhash算法生成指纹。相似的文本内容会生成相似的Minhash指纹。主要用于计算两个集合的Jaccard相似度，其中Jaccard相似度表示两个集合的交集大小与并集大小的比值

**算法思想：**

1. **分词**： 首先对文件进行分词或处理，将文件拆分成一系列元素（如词语、字符等）。
2. **哈希**：对每个元素进行哈希，得到一个哈希值。
3. **矩阵化**：构建一个signature matrix（签名矩阵），其中每一列代表一个哈希函数对所有元素的哈希值进行映射。
4. **Minhash**: 对于每个哈希函数，找到其对应列中的最小哈希值，并组成一个指纹向量。
5. **相似度计算**：使用Jaccard相似度来比较两个文件的指纹相似度。Jaccard相似度是两个集合的交集大小除以并集大小，用来度量两个集合之间的相似性。

**例子：**

Min-hashing定义为：特征矩阵按行进行一个随机的排列后，第一个列值为1的行的行号。举例说明如下，假设之前的特征矩阵按行进行的一个随机排列如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **元素** | **S1** | **S2** | **S3** | **S4** |
| 他 | 0 | 0 | 1 | 0 |
| 成功 | 0 | 0 | 1 | 1 |
| 我 | 1 | 0 | 0 | 0 |
| 减肥 | 1 | 0 | 1 | 1 |
| 要 | 0 | 1 | 0 | 1 |

　　最小哈希值：h(S1)=3，h(S2)=5，h(S3)=1，h(S4)=2.

**应用场景：**适用于大规模数据集合的相似性问题，**适合set数据类型**。Minhash算法是一个近似算法，相似性的计算结果并不是完全精确的

**参考教程：**[文本内容相似度计算方法：minhash – 标点符 (biaodianfu.com)](https://www.biaodianfu.com/minhash.html)

[最小哈希签名（MinHash）简述-腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/2162118)

[文本去重之MinHash算法 - pathenon - 博客园 (cnblogs.com)](https://www.cnblogs.com/pathenon/archive/2012/07/17/2595778.html)

[minHash最小哈希原理 - stardsd - 博客园 (cnblogs.com)](https://www.cnblogs.com/sddai/p/6110704.html)

1. from datasketch import MinHash, MinHashLSH
2. # 定义一个函数，用于计算Jaccard相似度
3. def jaccard\_similarity(set1, set2):
4. intersection = len(set1.intersection(set2))
5. union = len(set1.union(set2))
6. return intersection / union
7. # 定义函数，用于生成MinHash
8. def generate\_minhash(data, num\_perm):
9. minhash = MinHash(num\_perm=num\_perm)
10. for item in data:
11. minhash.update(item.encode('utf8'))
12. return minhash
13. if \_\_name\_\_ == '\_\_main\_\_':
14. # 定义两个示例的set类型数据
15. set1 = {'apple', 'banana', 'orange', 'grape'}
16. set2 = {'apple', 'watermelon', 'banana', 'kiwi'}
17. # 使用MinHash生成文件指纹
18. num\_perm = 128 # MinHash的num\_perm参数，影响MinHash的精度
19. minhash1 = generate\_minhash(set1, num\_perm)
20. minhash2 = generate\_minhash(set2, num\_perm)
21. # 打印哈希值
22. print(f"文件指纹1: {minhash1}")
23. print(f"文件指纹2: {minhash2}")
24. # 查看Minhash的值（128维向量）
25. # print(minhash1.digest())
26. # print(minhash2.digest())
27. # 使用Jaccard相似度计算相似度
28. similarity = jaccard\_similarity(set1, set2)
29. print("Jaccard相似度：", similarity)
30. # 使用MinHashLSH查询相似项（这一步可以在大规模数据中进行快速相似项查询）
31. lsh = MinHashLSH(threshold=0.5, num\_perm=num\_perm)
32. lsh.insert("set1", minhash1)
33. lsh.insert("set2", minhash2)
34. result = lsh.query(minhash1)
35. print("MinHashLSH查询相似项：", result)

#### **（1）指纹生成**

generate\_minhash函数：用于生成MinHash。这个函数接受一个字符串类型的集合data和num\_perm参数，其中num\_perm会影响MinHash的精度，它表示生成MinHash时使用的哈希函数数量。

#### **（2）指纹相似度计算**

jaccard\_similarity(set1, set2)函数，用于计算Jaccard相似度。Jaccard相似度是两个集合的交集大小除以并集大小的比值，用于衡量两个集合的相似程度。

**Jaccard相似度**是一种度量集合相似性的指标，它衡量两个集合的交集大小与并集大小的比例。对于使用Minhash生成的两个文件指纹，它们可以看作是两个集合，其中包含了一系列的哈希值。

Jaccard相似度的计算公式如下：

Jaccard相似度 = (两个集合的交集大小) / (两个集合的并集大小)

具体步骤如下：

1. 使用Minhash算法生成文件的指纹，得到两个文件的Minhash集合。
2. 计算两个Minhash集合的交集大小，即两个文件指纹的相同哈希值的个数。
3. 计算两个Minhash集合的并集大小，即两个文件指纹的总哈希值的个数。
4. 使用Jaccard相似度公式计算相似度。

Jaccard相似度的取值范围在0到1之间，值越接近1表示两个文件的内容越相似，值越接近0表示两个文件的内容越不相似。

#### **（3）效果：还不错（适合于set类型数据）**

1. 文件指纹1: <datasketch.minhash.MinHash object at 0x00000130263A86D0>
2. 文件指纹2: <datasketch.minhash.MinHash object at 0x00000130263A85E0>
3. Jaccard相似度： 0.3333333333333333
4. MinHashLSH查询相似项： ['set1']
5. 进程已结束,退出代码0

PS：

minhash和simhash都属于局部敏感哈希（Local Sensitive Hash）。一般的哈希算法对于相似文本的哈希结果可能差别非常大，局部敏感哈希在普通哈希的基础上保留了一定程度的相似性，即相似文本的哈希结果距离较小。

### **3、Karp-Rabin卡普-拉宾算法**

Karp-Rabin是一种字符串匹配算法，用于在一个长文本中高效地查找一个短模式的出现位置。该算法是由Michael O. Rabin和Richard M. Karp于1987年提出的。

Karp-Rabin算法的核心思想是通过哈希函数对模式和文本中的子串进行哈希计算，然后比较哈希值来判断是否匹配。它使用了一种滑动窗口的方法，在文本中滑动一个与模式长度相同的窗口，通过哈希函数计算窗口内子串的哈希值，并将其与模式的哈希值进行比较。

具体步骤如下：

1. 计算模式的哈希值和第一个窗口子串的哈希值。
2. 在文本中滑动窗口，每次滑动一个字符，并通过哈希函数计算新窗口子串的哈希值。
3. 将新窗口子串的哈希值与模式的哈希值进行比较，如果相等，则说明找到了一个匹配。
4. 如果哈希值不相等，继续滑动窗口，重复步骤3。

**应用场景：**主要用于字符串匹配，因此适合处理字符串类型的数据，比如str

**参考教程：**[Rabin–Karp 算法 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/563551141?utm_id=0)

[Rabin-Karp算法 - 简书 (jianshu.com)](https://www.jianshu.com/p/e20994e5e33c)

[Rabin-Karp算法概述 - ChristianL - 博客园 (cnblogs.com)](https://www.cnblogs.com/christianl/p/13747580.html)

1. import hashlib
2. def generate\_karprabin\_hash(text, window\_size=5, base=256, prime=101):
3. """
4. 使用Karp-Rabin算法生成文本的指纹（哈希值）。
5. :param text: 要生成指纹的文本。
6. :param window\_size: 滑动窗口的大小。
7. :param base: 基数，用于计算哈希值。
8. :param prime: 大素数，用于计算哈希值。
9. :return: 文本的指纹（哈希值）。
10. """
11. # 初始化哈希值
12. hash\_value = 0
13. # 存储所有窗口的哈希值
14. window\_hashes = []
15. # 计算窗口内的初始哈希值
16. for i in range(window\_size):
17. hash\_value = (hash\_value \* base + ord(text[i])) % prime
18. window\_hashes.append(hash\_value)
19. # 滑动窗口生成哈希值
20. for i in range(1, len(text) - window\_size + 1):
21. # 移除滑动窗口前一个字符的贡献
22. hash\_value = (hash\_value - ord(text[i - 1]) \* pow(base, window\_size - 1, prime)) % prime
23. # 添加滑动窗口后一个字符的贡献
24. hash\_value = (hash\_value \* base + ord(text[i + window\_size - 1])) % prime
25. # 存储当前窗口的哈希值
26. window\_hashes.append(hash\_value)
27. # 将所有窗口的哈希值连接起来，作为整个字符串的指纹
28. fingerprint = ''.join(str(h) for h in window\_hashes)
29. return fingerprint
30. def hamming\_distance(hash1, hash2):
31. """
32. 计算两个字符串之间的汉明距离。
33. :param hash1: 第一个字符串的哈希值。
34. :param hash2: 第二个字符串的哈希值。
35. :return: 汉明距离。
36. """
37. distance = sum(bit1 != bit2 for bit1, bit2 in zip(hash1, hash2))
38. return distance
39. def calculate\_similarity(hash1, hash2):
40. """
41. 计算两个字符串之间的相似度。
42. :param hash1: 第一个字符串的哈希值。
43. :param hash2: 第二个字符串的哈希值。
44. :return: 相似度。
45. """
46. num\_bits = len(hash1)
47. distance = hamming\_distance(hash1, hash2)
48. similarity = 1 - (distance / num\_bits)
49. return similarity
50. if \_\_name\_\_ == '\_\_main\_\_':
51. str1 = 'This is a test text for similarity calculation using Karp-Rabin algorithm'
52. str2 = 'This is a test text for similarity calculation using Karp-Rabin algorithm and a little long'
53. # 生成示例文本的Karp-Rabin文件指纹
54. fingerprint1 = generate\_karprabin\_hash(str1)
55. fingerprint2 = generate\_karprabin\_hash(str2)
56. print("Karp-Rabin文件指纹示例:")
57. print("str1:", fingerprint1)
58. print("str2:", fingerprint2)
59. # 计算相似度
60. similarity = calculate\_similarity(fingerprint1, fingerprint2)
61. print(f"相似度: {similarity}")

#### **（1）指纹生成**

generate\_karprabin\_hash函数用于生成文本的指纹。它接受文本字符串作为输入，并使用Karp-Rabin算法计算滑动窗口的哈希值，然后将所有窗口的哈希值连接起来，作为整个字符串的指纹。

#### **（2）指纹相似度计算**

1. hamming\_distance函数用于计算两个字符串之间的汉明距离。它接受两个字符串的哈希值作为输入，并通过逐位比较两个哈希值的不同位数，计算汉明距离。
2. calculate\_similarity函数用于计算两个字符串之间的相似度。它接受两个字符串的哈希值作为输入，并使用汉明距离计算相似度。相似度计算的方法是通过将汉明距离除以哈希值的总位数，得到相似度。

#### **（3）效果：**常见的处理方法都具有局限性

存在哈希冲突的潜在问题，即不同的子串可能有相同的哈希值，导致误判（如str1和2直接完全相等)

由于滑动窗口的大小和基数、大素数等参数都是固定的，因此如果两个文本具有相同的滑动窗口，它们的指纹将会相同，而且str1和2文本长度较小

Karp-Rabin算法生成的文本指纹在相同位置具有相同的滑动窗口内容时，其指纹将会相同，导致相似度为1.0。为增加指纹的唯一性，可以尝试调整滑动窗口的大小或选择不同的哈希函数。

1. Karp-Rabin文件指纹示例:
2. str1: f457c545a9ded88f18ecee47145a72c0
3. str2: f457c545a9ded88f18ecee47145a72c0
4. str3: 7647966b7343c29048673252e490f736
5. 相似度1和2: 1.0
6. 相似度1和3: 0.484375
7. 相似度2和3: 0.484375

**局限性**：Karp-Rabin算法生成的指纹可能无法准确反映字符串的相似性，尤其是当两个字符串的相似部分位于不同位置时。Karp-Rabin算法是一种局部敏感哈希方法，它主要关注于字符串的局部特征。如果两个字符串在不同的位置有相似的子串，那么它们的指纹可能会有很大的不同，导致计算的相似度不太准确。

### **4、Winnowing**

Winnowing算法是一种用于文本数据的局部散列算法，主要用于文本去重和查找近似重复的文本片段。该算法基于散列函数，可以快速地生成文本的指纹，从而方便地进行文本相似性的计算。Winnowing可以有效地识别重复的文本段落或者检测抄袭文本。

基本思想是将文本分成固定大小的滑动窗口，在每个窗口内使用散列函数计算散列值，然后在所有窗口中选择散列值最小的一个作为该位置的指纹。通过这种方式，Winnowing算法可以过滤掉文本中的噪声和不重要的信息，保留重要的特征，从而实现文本去重和相似性计算。

Winnowing算法主要有以下几个步骤：

1. 预处理文本
2. 将文本分成固定大小的滑动窗口，窗口大小为k。
3. 在每个窗口内对文本片段使用哈希函数计算哈希值。
4. 在所有窗口中选择散列值最小的一个作为该位置的指纹。
5. 最后，得到的指纹序列即为文本的指纹，可以用于文本去重和相似性计算。

**应用场景：具有较好的时间和空间复杂度，适用于大规模文本数据的处理。**

**参考教程：**[(107条消息) 【文本相似性计算】winnowing算法\_夜谷子的博客-CSDN博客](https://blog.csdn.net/weixin_43098787/article/details/82837867)

[(107条消息) winnowing 算法 -- 提取文档指纹特征](https://blog.csdn.net/qq_32023541/article/details/82382808)*[winnowing算法](https://blog.csdn.net/qq_32023541/article/details/82382808)*[ouprince的博客-CSDN博客](https://blog.csdn.net/qq_32023541/article/details/82382808)

[(107条消息) 基于K-gram的winnowing特征提取剽窃查重检测技术（概念篇）](https://blog.csdn.net/chichoxian/article/details/53115067?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=3)*[kgramhash](https://blog.csdn.net/chichoxian/article/details/53115067?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=3)*[君的名字的博客-CSDN博客](https://blog.csdn.net/chichoxian/article/details/53115067?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~BlogCommendFromBaidu~Rate-2-53115067-blog-82382808.235%5Ev38%5Epc_relevant_anti_t3_base&utm_relevant_index=3)

**k-grams+Winnowing算法**步骤：

1. **生成k-grams**：将原始文本切分成连续的k个字符的子串，这些子串被称为k-grams。k-grams是文本的局部特征表示，能够捕捉文本的局部信息。
2. **计算k-grams的哈希值**：对于生成的每个k-gram子串，使用哈希函数将其转换为哈希值。这一步将k-grams映射为哈希值，缩小了指纹的维度。
3. **选取局部最小哈希值**：在生成的哈希值列表中，选择局部最小的哈希值作为该位置的指纹。具体做法是定义一个滑动窗口，在窗口内找到最小的哈希值，并记录下来。随着窗口滑动，不断更新最小值，得到文本的指纹。
4. **连接指纹**：将不同位置的指纹连接起来，形成整个文本的指纹表示。这样，每个文本都会有一串指纹值，代表了文本的局部特征。
5. **存储指纹**：将生成的文本指纹存储起来，用于后续的文本相似度计算和比较。

PS：k-grams+Winnowing算法主要用于文本相似度计算，特别是对于较长文本的相似度比较。通过局部特征的表示，可以捕捉文本的相似部分，而Winnowing算法的局部最小哈希选择策略能够进一步减少指纹的维度，提高计算效率。这种算法适用于大规模文本相似度比较和查重等应用场景。

1. from simhash import Simhash
2. def generate\_kgrams(text, k):
3. """
4. 生成k-grams列表。
5. :param text: 要生成k-grams的文本。
6. :param k: k-grams的长度。
7. :return: k-grams列表。
8. """
9. kgrams = []
10. for i in range(len(text) - k + 1):
11. kgram = text[i:i + k]
12. kgrams.append(kgram)
13. return kgrams
14. def calculate\_hashes(kgrams):
15. """
16. 计算k-grams的哈希值列表。
17. :param kgrams: k-grams列表。
18. :return: 哈希值列表。
19. """
20. hashes = []
21. for kgram in kgrams:
22. # 使用Simhash计算k-gram的哈希值(可以替换为其他哈希函数）
23. simhash = Simhash(kgram)
24. # 将Simhash值转换为十六进制数，并添加到哈希列表中
25. hash\_value = hex(simhash.value)
26. hashes.append(hash\_value)
27. return hashes
28. def winnowing(hashes, window\_size):
29. """
30. 使用Winnowing算法从哈希列表中生成文本的指纹。
31. :param hashes: 哈希值列表。
32. :param window\_size: 滑动窗口大小。
33. :return: 文本的指纹。
34. """
35. min\_hashes = []
36. # 找到初始窗口中的最小哈希值
37. min\_hash = min(hashes[:window\_size])
38. min\_hashes.append(min\_hash)
39. # 滑动窗口，选择局部最小哈希值
40. for i in range(1, len(hashes) - window\_size + 1):
41. if hashes[i + window\_size - 1] < min\_hash:
42. min\_hash = hashes[i + window\_size - 1]
43. min\_hashes.append(min\_hash)
44. return min\_hashes
45. def hamming\_distance(hash1, hash2):
46. """
47. 计算两个字符串的汉明距离。
48. :param hash1: 第一个字符串的哈希值。
49. :param hash2: 第二个字符串的哈希值。
50. :return: 汉明距离。
51. """
52. distance = sum(bit1 != bit2 for bit1, bit2 in zip(hash1, hash2))
53. return distance
54. def calculate\_similarity(fingerprints1, fingerprints2):
55. """
56. 计算两个文本指纹之间的相似度。
57. :param fingerprints1: 第一个文本的指纹。
58. :param fingerprints2: 第二个文本的指纹。
59. :return: 相似度。
60. """
61. num\_bits = len(fingerprints1[0]) \* 4 # 由于Simhash是64位的，每个十六进制数对应4位，所以要乘以4
62. distance = hamming\_distance(fingerprints1, fingerprints2)
63. similarity = 1 - (distance / num\_bits)
64. return similarity
65. def kgrams\_winnowing(text, k, window\_size):
66. """
67. 使用k-grams+Winnowing算法生成文本的指纹。
68. :param text: 要生成指纹的文本。
69. :param k: k-grams的长度。
70. :param window\_size: 滑动窗口大小。
71. :return: 文本的指纹。
72. """
73. kgrams = generate\_kgrams(text, k)
74. hashes = calculate\_hashes(kgrams)
75. fingerprints = winnowing(hashes, window\_size)
76. return fingerprints
77. if \_\_name\_\_ == '\_\_main\_\_':
78. str1 = 'This is a test text for similarity calculation using k-grams+Winnowing algorithm'
79. str2 = 'This is a calculation using k-grams+Winnowing algorithm and a little long'
80. k = 5 # k值，根据实际情况设定
81. window\_size = 5 # 滑动窗口大小，根据实际情况设定
82. fingerprints1 = kgrams\_winnowing(str1, k, window\_size)
83. fingerprints2 = kgrams\_winnowing(str2, k, window\_size)
84. print("Winnowing文件指纹示例:")
85. print("str1:", fingerprints1)
86. print("str2:", fingerprints2)
87. # 计算相似度
88. similarity = calculate\_similarity(fingerprints1, fingerprints2)
89. print(f"相似度: {similarity}")

#### **（1）指纹生成**

1. kgrams\_winnowing(text, k, window\_size)函数是整个算法的入口函数。它依次调用generate\_kgrams、calculate\_hashes和winnowing函数，生成文本的指纹。
2. generate\_kgrams(text, k)函数用于生成文本的k-grams列表。K-grams是文本中长度为k的连续子串。通过滑动窗口，从文本中提取出所有的k-grams，并存储在一个列表中。
3. calculate\_hashes(kgrams)函数用于计算k-grams的哈希值列表。这里使用了Simhash算法来计算k-grams的哈希值，将Simhash值转换成十六进制数，并将哈希值存储在一个列表中。
4. winnowing(hashes, window\_size)函数用于使用Winnowing算法从哈希值列表中生成文本的指纹。Winnowing算法通过滑动窗口，在哈希值列表中选择局部最小的哈希值，然后将这些最小哈希值作为文本的指纹。

#### **（2）指纹相似度计算**

1. hamming\_distance(hash1, hash2)函数用于计算两个字符串之间的汉明距离。汉明距离是指两个等长字符串之间相对应位置上不同字符的个数。
2. calculate\_similarity(fingerprints1, fingerprints2)函数用于计算两个文本指纹之间的相似度。首先计算每个文本指纹的位数，并计算两个指纹之间的汉明距离。然后，根据指纹位数和汉明距离计算相似度。

#### **（3）效果：还可以**

PS：需要根据后续测试来选择合适的k和window\_size以及哈希函数

1. Winnowing文件指纹示例:
2. str1: ['0xa83765b2ca5cc091', '0x42b5a6b3d156056c', '0x1f13be268715e04e', '0x1e62b32c2726cc99', '0x1660830820220c01', '0x10000e21108a9001']
3. str2: ['0xa83765b2ca5cc091', '0x7e1e9e4aedb8242d', '0x13ee4dd2201c7a73', '0x10000e21108a9001']
4. 相似度: 0.9583333333333334

## **LSH—局部敏感哈希**

**Locality Sensitive Hashing**

主要用于高效处理海量高维数据的最近邻问题 ，使得 2 个相似度很高的数据以较高的概率映射成同一个hash 值，而令 2 个相似度很低的数据以极低的概率映射成同一个 hash 值。

#### **K-gram/N-gram + LSH算法步骤：**

1. 选择合适的K-gram或者N-gram大小：根据应用的需要，选择合适的K-gram或者N-gram大小，将文本划分为连续的K个字符或者N个连续的词组。
2. 生成文本的指纹：对于每个文本，使用K-gram或者N-gram来生成文本的指纹。可以使用哈希函数将K-gram或者N-gram序列映射为一个哈希值，得到文本的指纹。
3. 建立LSH索引：使用LSH算法来建立索引，将所有生成的文本指纹划分到不同的桶中。在datasketch库中，可以使用MinHashLSH/SimhashLSH类来实现LSH索引。
4. 添加指纹到LSH索引：将生成的文本指纹添加到LSH索引中的对应桶中。
5. 查询相似指纹：对于要查询相似度的文本，先生成其指纹，然后使用LSH索引进行查询，找到与查询指纹相似的其他指纹。可以使用MinHashLSH/SimhashLSH类的query方法来实现查询。
6. 计算相似度：对于从LSH索引中检索到的相似指纹，可以使用某种相似度度量方法（如汉明距离、Jaccard相似度等）来计算它们与查询指纹之间的相似度。

#### **K-gram和N-gram比较**

1. K-gram适用场景：
   * K-gram适用于较小的文本或字符级别的文本分析，特别是当文本长度较短时，K-gram可以提供更细粒度的划分。
   * 对于字符级别的文本，K-gram可以生成长度为K的子串，可以捕捉到字符级别的相似性，例如拼写错误、字母重排等。
   * 在某些文本处理任务中，需要对文本进行较小窗口的滑动分析，这时使用K-gram更合适。
2. N-gram适用场景：
   * N-gram适用于较大的文本或词语级别的文本分析，特别是当文本较长且具有较复杂的语义结构时，N-gram可以提供更有意义的划分。
   * 对于词语级别的文本，N-gram可以生成长度为N的词语序列，可以更好地捕捉语义信息和词序关系。
   * 在自然语言处理等任务中，常常使用N-gram来提取特征、训练模型等。

### **5、N-gram + Minhash+Locality Sensitive Hashing (LSH)**

N-gram是一种文本特征表示方法，可以将文本切分为连续的N个字符或词语。通过将字符串内容转换为N-gram特征，并使用LSH将相似的N-gram特征分到同一个桶中，可以生成类似的文本指纹。

1. **N-gram：** N-gram是一种文本特征提取方法，它将文本划分为连续的N个字符或词语。N-gram模型可以捕捉文本中的局部特征和上下文信息。常见的N-gram模型包括unigram（单个字符或词语）、bigram（两个连续字符或词语）、trigram（三个连续字符或词语）等。通过提取文本的N-gram特征，我们可以将文本转化为向量表示，便于后续的相似度计算。
2. **Locality Sensitive Hashing (LSH)：** LSH是一种近似搜索算法，用于快速查找近似相似的数据项。在文本相似度计算中，LSH可用于快速搜索相似的N-gram向量，以便在大规模文本数据集中找到相似的文本。LSH通过哈希函数将向量映射到桶（bucket）中，相似的向量在同一个桶中的概率较高。通过只计算同一桶内的向量之间的相似度，大大减少了相似度计算的时间复杂度。

**N-gram + LSH的流程**如下：

1. 对文本进行N-gram特征提取，将文本转化为N-gram向量表示。
2. 使用LSH算法将N-gram向量映射到不同的桶中。
3. 在每个桶内计算向量之间的相似度，找到近似相似的文本。

**应用场景：**处理大规模文本数据的相似度搜索和近似匹配

**参考教程：[可搜索加密：基础知识-腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/2205117?areaSource=102001.7&traceId=eEiYFBEodUbaVIwPmqCP1)**

1. import hashlib
2. from nltk importngrams
3. from datasketch import MinHash, MinHashLSH
4. ​
5. ​
6. defgenerate\_ngram\_lsh\_fingerprint(text, n=3, num\_perm=128, threshold=0.5):
7. """
8. 使用N-gram + LSH算法生成文本的指纹（哈希值）。
9. :param text: 要生成指纹的文本。
10. :param n: N-gram的大小。
11. :param num\_perm: MinHash的排列数。
12. :param threshold: LSH的阈值。
13. :return: 文本的指纹（哈希值）。
14. """
15. # 创建MinHash对象
16. minhash = MinHash(num\_perm=num\_perm)
17. ​
18. # 创建哈希函数（可以选择MD5、SHA-1等哈希算法）
19. hash\_function = hashlib.md5
20. ​
21. # 生成N-gram序列
22. ngram\_sequence = ngrams(text, n)
23. ​
24. # 添加N-gram序列到MinHash中
25. forngraminngram\_sequence:
26. foriteminngram:
27. minhash.update(hash\_function(item.encode()).digest())
28. ​
29. # 返回指纹的哈希摘要
30. fingerprint\_digest = minhash.digest()
31. ​
32. returnfingerprint\_digest
33. ​
34. ​
35. defhash\_fingerprint(fingerprint):
36. """
37. 使用哈希函数对指纹进行进一步的处理。
38. :param fingerprint: 要处理的指纹。
39. :return: 处理后的指纹（哈希值）。
40. """
41. # 创建哈希函数（可以选择MD5、SHA-1、SHA-256等哈希算法）
42. hash\_function = hashlib.md5
43. ​
44. # 计算指纹的哈希值
45. hashed\_fingerprint = hash\_function(fingerprint).hexdigest()
46. ​
47. returnhashed\_fingerprint
48. ​
49. ​
50. defhamming\_distance(fingerprint1, fingerprint2):
51. """
52. 计算两个指纹之间的汉明距离。
53. :param fingerprint1: 第一个指纹。
54. :param fingerprint2: 第二个指纹。
55. :return: 汉明距离。
56. """
57. assertlen(fingerprint1) == len(fingerprint2), "指纹长度不一致"
58. ​
59. distance = sum(c1!= c2forc1, c2inzip(fingerprint1, fingerprint2))
60. returndistance
61. ​
62. ​
63. defcalculate\_similarity(fingerprint1, fingerprint2):
64. """
65. 计算两个指纹之间的相似度。
66. :param fingerprint1: 第一个指纹。
67. :param fingerprint2: 第二个指纹。
68. :return: 相似度。
69. """
70. hamming\_dist = hamming\_distance(fingerprint1, fingerprint2)
71. similarity = 1- (hamming\_dist/len(fingerprint1))
72. returnsimilarity
73. ​
74. ​
75. if\_\_name\_\_ == '\_\_main\_\_':
76. str1 = '北京增值税电子普通发票.pdf'
77. str2 = '福建增值税电子普通发票.pdf'
78. str3 = '福建工程学院计算机学院培养方案.pdf'
79. # str3 = '0.813762178'
80. # 生成示例文本的N-gram + LSH文件指纹
81. fingerprint1 = generate\_ngram\_lsh\_fingerprint(str1)
82. fingerprint2 = generate\_ngram\_lsh\_fingerprint(str2)
83. fingerprint3 = generate\_ngram\_lsh\_fingerprint(str3)
84. ​
85. print("N-gram + LSH文件指纹示例:")
86. print("str1:", fingerprint1)
87. print("str2:", fingerprint2)
88. print("str3:", fingerprint3)
89. ​
90. # 使用MD5哈希函数对指纹进行进一步处理
91. hashed\_fingerprint1 = hash\_fingerprint(fingerprint1)
92. hashed\_fingerprint2 = hash\_fingerprint(fingerprint2)
93. hashed\_fingerprint3 = hash\_fingerprint(fingerprint3)
94. ​
95. print("处理后的指纹（哈希值）:")
96. print("str1:", hashed\_fingerprint1)
97. print("str2:", hashed\_fingerprint2)
98. print("str3:", hashed\_fingerprint3)
99. ​
100. # 计算相似度
101. similarity1 = calculate\_similarity(fingerprint1, fingerprint2)
102. similarity2 = calculate\_similarity(fingerprint1, fingerprint3)
103. similarity3 = calculate\_similarity(fingerprint2, fingerprint3)
104. print("1和2两个文本的相似度为:", similarity1)
105. print("1和3两个文本的相似度为:", similarity2)
106. print("2和3两个文本的相似度为:", similarity3)

#### **（1）指纹生成**

generate\_ngram\_lsh\_fingerprint

1. 对于每个文本，首先进行N-gram处理，获取输入文本的N-gram序列。N-gram是指将文本划分为连续的N个字符或词组的序列。N可以根据需要进行调整。
2. 对于每个N-gram序列，将其转换为字符串形式，并计算其哈希值（哈希函数可以进行调整）。
3. 将计算得到的哈希值添加到 MinHash 对象中。
4. 返回 MinHash 对象的摘要值：

PS：可以调整N-gram的大小（n），MinHash的排列数（num\_perm）和LSH的阈值（threshold）来获得更好的指纹效果

打印二进制摘要、哈希摘要

#### **（2）指纹相似度计算**

hamming\_distance函数用于计算两个指纹之间的汉明距离。然后，calculate\_similarity函数使用**汉明距离**计算两个指纹的相似度。相似度的计算公式是 1 - (汉明距离 / 指纹长度)。

PS：可以考虑使用Jaccard相似度计算

效果：效果不错

1. 处理后的指纹（哈希值）:
2. str1: a33d4c4e78166ea389699c2db12b7973
3. str2: 882ed74f91d19ee27582bd43b244751a
4. str3: 20a43e7e1509294247f91bc87891fa83
5. 1和2两个文本的相似度为: 0.78125
6. 1和3两个文本的相似度为: 0.15625
7. 2和3两个文本的相似度为: 0.2734375

### **6、（K/N**-gram） + **Simhash + Locality Sensitive Hashing (LSH)：**

**局部敏感哈希**(Locality Sensitive Hashing，LSH)的基本思想类似于一种空间域转换思想，LSH算法基于一个假设，如果两个文本在原有的数据空间是相似的，那么分别经过哈希函数转换以后的它们也具有很高的相似度；相反，如果它们本身是不相似的，那么经过转换后它们应仍不具有相似性。

LSH是一种用于快速近似近邻搜索的技术，可以与Simhash结合使用。通过Simhash计算文本指纹，然后使用LSH将相似的指纹分到同一个桶中。这样，在进行文本相似性搜索时，只需比较同一桶中的指纹，从而提高搜索效率。

1. Simhash算法：首先，对每个文本进行Simhash处理，将文本转换为一个固定长度的二进制数，称为Simhash值。Simhash值的生成过程中，使用了hash函数和加权求和技术，保证了相似文本的Simhash值在汉明空间中的距离很小。
2. LSH技术：然后，使用LSH技术对Simhash值进行分组，将相似的Simhash值归为一组。LSH技术是一种用于在大规模数据中查找相似项的方法，通过将数据集划分成多个小的子集（称为哈希桶），将相似的项映射到同一个桶中，从而加速相似性查询。

参考1：python\_mmdt:一种基于敏感哈希生成特征向量的python库(一)

<https://bbs.kanxue.com/thread-265211.htm>

参考2：python\_mmdt:一种基于敏感哈希生成特征向量的python库(一)

<https://zhuanlan.zhihu.com/p/346260945>

1. import hashlib
2. from collections importdefaultdict
3. from nltk importngrams
4. from sklearn.feature\_extraction.text import TfidfVectorizer
5. from sklearn.metrics.pairwise import cosine\_similarity
6. import Levenshtein
7. ​
8. ​
9. defgenerate\_simhash\_lsh\_fingerprint(texts, k=3, num\_perm=64):
10. """
11. 使用Simhash + LSH算法生成文本的指纹（哈希值）。
12. :param texts: 要生成指纹的文本列表。
13. :param k: Simhash的k-gram参数。
14. :param num\_perm: Simhash的排列数。
15. :return: 文本的指纹（哈希值）。
16. """
17. # 创建LSH索引
18. lsh\_index = defaultdict(list)
19. ​
20. # 生成Simhash值并添加到LSH索引中
21. fortextintexts:
22. simhash = generate\_simhash(text, k=k, num\_perm=num\_perm)
23. lsh\_index[simhash].append(text)
24. ​
25. returnlsh\_index
26. ​
27. ​
28. defgenerate\_simhash(text, k=3, num\_perm=64):
29. """
30. 生成文本的Simhash值。
31. :param text: 要生成Simhash的文本。
32. :param k: Simhash的k-gram参数。
33. :param num\_perm: Simhash的排列数。
34. :return: 文本的Simhash值。
35. """
36. # 初始化特征向量
37. feature\_vector = [0] \*num\_perm
38. ​
39. # 生成N-gram序列
40. ngram\_sequence = ngrams(text, k)
41. ​
42. # 计算特征向量
43. forngraminngram\_sequence:
44. ngram\_hash = hash(' '.join(ngram))
45. foriinrange(num\_perm):
46. ifngram\_hash& (1<<i):
47. feature\_vector[i] += 1
48. else:
49. feature\_vector[i] -= 1
50. ​
51. # 构建Simhash值
52. simhash = 0
53. foriinrange(num\_perm):
54. iffeature\_vector[i] >= 0:
55. simhash |= (1<<i)
56. ​
57. returnsimhash
58. ​
59. ​
60. defcalculate\_similarity(texts1, texts2, threshold=0.5):
61. """
62. 计算两个文本列表的相似度。
63. :param texts1: 第一个文本列表。
64. :param texts2: 第二个文本列表。
65. :param threshold: 相似度阈值。
66. :return: 两个文本列表的相似度。
67. """
68. # 生成指纹
69. fingerprint1 = generate\_simhash\_lsh\_fingerprint(texts1)
70. fingerprint2 = generate\_simhash\_lsh\_fingerprint(texts2)
71. ​
72. # 计算相似度
73. similarity = 0.0
74. ​
75. forsimhash1infingerprint1:
76. similar\_texts = find\_similar\_texts(fingerprint2, simhash1, threshold)
77. fortext1infingerprint1[simhash1]:
78. fortext2insimilar\_texts:
79. similarity += calculate\_cosine\_similarity\_text(text1, text2)
80. ​
81. similarity /= len(texts1) \*len(texts2)
82. ​
83. returnsimilarity
84. ​
85. ​
86. deffind\_similar\_texts(lsh\_index, simhash, threshold):
87. """
88. 根据相似度阈值在LSH索引中查询相似的文本。
89. :param lsh\_index: LSH索引。
90. :param simhash: 目标Simhash值。
91. :param threshold: 相似度阈值。
92. :return: 相似的文本列表。
93. """
94. similar\_texts = []
95. forstored\_simhash, textsinlsh\_index.items():
96. ifhamming\_distance(simhash, stored\_simhash) <= threshold:
97. similar\_texts.extend(texts)
98. returnsimilar\_texts
99. ​
100. ​
101. defhamming\_distance(simhash1, simhash2):
102. """
103. 计算两个Simhash值的汉明距离。
104. :param simhash1: 第一个Simhash值。
105. :param simhash2: 第二个Simhash值。
106. :return: 汉明距离。
107. """
108. xor\_result = simhash1^simhash2
109. hamming\_dist = bin(xor\_result).count('1')
110. returnhamming\_dist
111. ​
112. ​
113. defcalculate\_cosine\_similarity\_text(text1, text2):
114. """
115. 计算两个文本之间的余弦相似度。
116. :param text1: 第一个文本。
117. :param text2: 第二个文本。
118. :return: 余弦相似度。
119. """
120. # 使用正确的TF-IDF计算方法
121. vectorizer = TfidfVectorizer(use\_idf=True, norm='l2')
122. tfidf\_matrix = vectorizer.fit\_transform([text1, text2])
123. cosine\_sim = cosine\_similarity(tfidf\_matrix[0], tfidf\_matrix[1])[0][0]
124. returncosine\_sim
125. ​
126. ​
127. defcalculate\_similarity\_str(str1, str2):
128. """
129. 计算两个字符串的相似度。
130. :param str1: 第一个字符串。
131. :param str2: 第二个字符串。
132. :return: 相似度。
133. """
134. distance = Levenshtein.distance(str1, str2)
135. max\_length = max(len(str1), len(str2))
136. similarity = 1- (distance/max\_length)
137. returnsimilarity
138. ​
139. ​
140. if\_\_name\_\_ == '\_\_main\_\_':
141. # 示例文本列表
142. str1 = '北京增值税电子普通发票.pdf'
143. str2 = '福建增值税电子普通发票.pdf'
144. str3 = '福建工程学院计算机学院培养方案.pdf'
145. ​
146. # 生成文件指纹
147. texts = [str1, str2, str3]
148. fingerprints = generate\_simhash\_lsh\_fingerprint(texts)
149. print("文件指纹：", fingerprints)
150. # 计算相似度
151. similarity1 = calculate\_similarity\_str(str1, str2)
152. similarity2 = calculate\_similarity\_str(str1, str3)
153. similarity3 = calculate\_similarity\_str(str2, str3)
154. ​
155. print("相似度1和2:", similarity1)
156. print("相似度1和3:", similarity2)
157. print("相似度2和3:", similarity3)

#### **（1）指纹生成**

generate\_simhash函数：根据输入的文本，生成Simhash值。Simhash是一种哈希算法，用于计算文本的指纹，以便快速进行相似度比较。在这个函数中，首先初始化一个长度为64的特征向量，用于存储Simhash值。然后，通过将文本切分成k-gram（默认为3-gram）序列，并使用hash函数将每个k-gram映射到一个哈希值。根据每个哈希值的位信息，更新特征向量。最后，根据特征向量的值构建Simhash值并返回。

generate\_simhash\_lsh\_fingerprint函数：使用Simhash + LSH算法生成文本的指纹。在这个函数中，首先创建一个LSH索引，用于存储Simhash值和对应的文本列表。然后，对输入的文本列表中的每个文本，使用generate\_simhash函数生成Simhash值，并将文本添加到LSH索引中。最后，返回LSH索引。

find\_similar\_texts函数：根据相似度阈值在LSH索引中查询相似的文本。遍历LSH索引中的每个Simhash值，计算它与目标Simhash值之间的汉明距离。如果汉明距离小于等于阈值，将对应的文本添加到相似文本列表中。

#### **（2）相似度计算**

hamming\_distance函数：计算两个Simhash值之间的汉明距离。汉明距离是指两个Simhash值之间对应位不同的数量。通过异或运算获得不同的位，并计算二进制中1的个数。

calculate\_cosine\_similarity\_text函数：计算两个文本之间的余弦相似度。使用TF-IDF计算文本的向量表示，并计算向量之间的余弦相似度。

calculate\_similarity\_str函数：计算两个字符串之间的相似度。使用Levenshtein距离计算字符串的编辑距离，并将其转化为相似度。

#### **（3）效果：还可以**

1. 文件指纹： defaultdict(<class 'list'>, {5095444621731347073: ['北京增值税电子普通发票.pdf'], 5518853269549265427: ['福建增值税电子普通发票.pdf'], 17439802823161103605: ['福建工程学院计算机学院培养方案.pdf']})
2. 相似度1和2: 0.8666666666666667
3. 相似度1和3: 0.21052631578947367
4. 相似度2和3: 0.3157894736842105

**另一种版本**

1. from simhash import Simhash, SimhashIndex
2. def generate\_simhash\_fingerprint(text):
3. """
4. 使用Simhash算法生成文本的指纹。
5. :param text: 要生成指纹的文本。
6. :return: 文本的指纹（Simhash对象）。
7. """
8. simhash = Simhash(text)
9. return simhash
10. def calculate\_similarity(simhash1, simhash2):
11. """
12. 计算两个Simhash指纹之间的相似度。
13. :param simhash1: 第一个Simhash指纹。
14. :param simhash2: 第二个Simhash指纹。
15. :return: 相似度。
16. """
17. similarity = simhash1.distance(simhash2)
18. return 1 - (similarity / 64) # 64为Simhash指纹的位数
19. if \_\_name\_\_ == '\_\_main\_\_':
20. str1 = 'This is a test text for similarity calculation using Simhash and LSH algorithm'
21. str2 = 'This is a test text for similarity calculation using Simhash and LSH algorithm and a little long'
22. str3 = 'This is a completely different text'
23. # 生成示例文本的Simhash指纹
24. simhash1 = generate\_simhash\_fingerprint(str1)
25. simhash2 = generate\_simhash\_fingerprint(str2)
26. simhash3 = generate\_simhash\_fingerprint(str3)
27. # 创建SimhashIndex对象，用于存储指纹和检索相似指纹
28. index = SimhashIndex([(str(i), simhash) for i, simhash in enumerate([simhash1, simhash2])], k=3)
29. # 查询与str3相似的指纹
30. similar\_simhashes = index.get\_near\_dups(simhash3)
31. # 计算相似度
32. similarity1 = calculate\_similarity(simhash1, simhash2)
33. similarity2 = calculate\_similarity(simhash1, simhash3)
34. similarity3 = calculate\_similarity(simhash2, simhash3)
35. print("相似度示例:")
36. print("str1和str2的相似度:", similarity1)
37. print("str1和str3的相似度:", similarity2)
38. print("str2和str3的相似度:", similarity3)
39. print("与str3相似的指纹:", similar\_simhashes)

算法思路步骤如下：

1. 使用Simhash算法生成文本的指纹：
   * generate\_simhash\_fingerprint函数接收文本作为输入，使用Simhash算法生成文本的指纹，返回一个Simhash对象，表示文本的指纹。
2. 计算两个Simhash指纹之间的相似度：
   * calculate\_similarity函数接收两个Simhash对象作为输入，通过计算汉明距离（Hamming distance）来计算两个指纹之间的相似度。汉明距离是指两个等长字符串之间对应位置上不同字符的个数，也可以看作是两个指纹在位级上的不同之处。
3. 创建SimhashIndex对象，用于存储指纹和检索相似指纹：
   * 创建SimhashIndex对象，并将需要检索的Simhash指纹添加到其中，k参数表示LSH中的参数k，用于控制索引中的桶的数量。
4. 查询与给定文本相似的指纹：
   * 使用index.get\_near\_dups(simhash)方法从SimhashIndex中检索与给定文本相似的指纹。该方法返回一个包含相似指纹的列表，这些指纹与给定的Simhash指纹在汉明距离上很近。
5. 计算相似度：
   * 使用计算汉明距离的方法来计算相似度。相似度的计算方法是通过计算汉明距离与指纹长度的比例，然后用1减去该比例得到的。

在这个算法中，Simhash和LSH技术被结合使用，Simhash用于生成文本的指纹，而LSH用于快速检索与给定文本相似的指纹，从而提高相似度计算的效率。这种组合能够有效地处理大量文本数据，并找出相似的文本。

## **三、距离计算方法**

Jaccard相似度适用于集合数据的相似度计算，汉明距离适用于等长字符串的相似度计算，Simhash相似度适用于保持文本信息的压缩指纹计算，Minhash相似度适用于大规模数据集的近似相似度计算，Karp-Rabin相似度适用于文本数据的快速相似度估计，Winnowing相似度适用于滑动窗口处理的文本相似度计算。

#### **1、汉明距离**

汉明距离（Hamming distance）是一种衡量两个等长字符串之间差异的度量方法。它用于计算两个字符串在相同位置上不同字符的个数。换句话说，汉明距离衡量了将一个字符串转换为另一个字符串所需的最小替换次数。

汉明距离计算方法：

1. 首先，将两个字符串对齐，使它们的长度相同。如果两个字符串长度不同，需要在较短的字符串末尾补充空字符，直到两个字符串的长度相等。
2. 然后，逐个比较两个字符串在相同位置上的字符。如果两个字符不相同，则汉明距离加1；如果两个字符相同，则汉明距离不变。
3. 最后，将所有不相同字符的个数相加，得到最终的汉明距离。

PS：如果是**不等长字符串**：常见的做法是先将它们转换成等长的特征向量，然后再计算特征向量之间的相似度。常见的方法包括使用N-gram特征或TF-IDF特征等。

N-gram特征：N-gram是一种将文本切分为连续的N个字符或单词的方法。对于每个字符串，我们可以将其切分为N-gram，并统计每个N-gram在字符串中出现的次数，形成一个特征向量。然后，通过比较两个特征向量之间的相似度，可以得到两个不等长字符串的相似度。

**代码示例（针对二进制）**

1. 首先，将两个文件指纹转换成二进制串，确保它们长度相等。
2. 然后，计算两个二进制串之间的汉明距离，即不同位置上的位数不同的个数。
3. 最后，通过计算汉明距离与二进制串长度的比值来得到相似度。
4. def hamming\_distance(s1, s2):
5. # 计算两个二进制串的汉明距离
6. if len(s1) != len(s2):
7. raise ValueError("两个二进制串必须具有相同的长度")
8. return sum(bit1 != bit2 for bit1, bit2 in zip(s1, s2))
9. def calculate\_similarity(fp1, fp2):
10. # 计算两个文件指纹的相似度
11. hamming\_dist = hamming\_distance(fp1, fp2)
12. similarity = 1 - (hamming\_dist / len(fp1))
13. return similarity
14. if \_\_name\_\_ == '\_\_main\_\_':
15. # 两个等长的文件指纹，长度为8
16. fp1 = "10010110"
17. fp2 = "10100110"
18. similarity = calculate\_similarity(fp1, fp2)
19. print("汉明距离：", hamming\_distance(fp1, fp2))
20. print("相似度：", similarity)

​

#### **2、Jaccard相似度**

Jaccard相似度是一种用于计算两个集合之间相似性的指标，它衡量两个集合的交集元素与并集元素之间的比例。Jaccard相似度的取值范围在0到1之间，其中0表示两个集合没有共同的元素，1表示两个集合完全相同。

Jaccard相似度的计算方法如下：

1. 首先，计算两个集合的交集，即两个集合中共有的元素。
2. 然后，计算两个集合的并集，即两个集合中所有的元素，包括重复的元素。
3. 最后，用交集的大小除以并集的大小，得到Jaccard相似度。

数学表达式为：J(A, B) = |A ∩ B| / |A ∪ B|

其中，J(A, B)表示集合A和集合B的Jaccard相似度，|A ∩ B|表示集合A和集合B的交集大小，|A ∪ B|表示集合A和集合B的并集大小。

Jaccard相似度在数据处理和信息检索中广泛应用，特别适用于处理文本数据、集合数据、网络图等。在文件指纹相似度计算中，Jaccard相似度也常用于衡量两个文件指纹之间的相似性。

**代码示例：（常用于set集合类型数据计算相似度）**

​

1. def generate\_ngrams(text, n):
2. ngrams = set()
3. for i in range(len(text) - n + 1):
4. ngrams.add(text[i:i + n])
5. return ngrams
6. def jaccard\_similarity(set1, set2):
7. intersection = set1.intersection(set2)
8. union = set1.union(set2)
9. return len(intersection) / len(union)
10. # 示例文本
11. text1 = "This is a sample text for testing Jaccard similarity."
12. text2 = "Another text for testing the Jaccard similarity."
13. # 设置N-gram的大小
14. n = 3
15. # 生成两个文本的N-grams集合
16. ngrams1 = generate\_ngrams(text1, n)
17. ngrams2 = generate\_ngrams(text2, n)
18. # 计算Jaccard相似度
19. similarity = jaccard\_similarity(ngrams1, ngrams2)
20. print("Jaccard相似度：", similarity)

hamming\_distance函数用于计算两个二进制串之间的汉明距离，

calculate\_similarity函数用于计算两个文件指纹的相似度。

#### **3、余弦相似度**

余弦相似度（Cosine Similarity）是一种常用的相似度计算方法，它可以用于衡量两个向量之间的相似程度。在文本相似度计算中，可以将文本表示成向量，然后使用余弦相似度来比较两个文本的相似性。

余弦相似度的计算方法如下：

假设有两个向量A和B，它们分别是n维向量（可以是词频向量、TF-IDF向量等），那么余弦相似度可以通过以下公式计算：

其中，A · B表示向量A和向量B的点积（内积），||A||表示向量A的模长，||B||表示向量B的模长。

余弦相似度的取值范围在[-1, 1]之间，取值越接近1表示两个向量越相似，取值越接近-1表示两个向量越不相似，取值为0表示两个向量正交（无相似性）。

**代码示例：**当比较两个文件指纹的相似度时，可以使用余弦相似度来计算它们之间的相似性。首先，需要将文件指纹表示成向量形式，可以使用N-gram + LSH算法生成文件指纹，并将生成的文件指纹表示成向量。然后，使用余弦相似度计算这两个向量之间的相似度。

假设已经有了两个文件指纹的向量表示，分别是vec1和vec2，可以使用以下代码计算它们之间的余弦相似度：

1. import numpy as np
2. def cosine\_similarity(vec1, vec2):
3. # 计算余弦相似度
4. dot\_product = np.dot(vec1, vec2)
5. norm1 = np.linalg.norm(vec1)
6. norm2 = np.linalg.norm(vec2)
7. similarity = dot\_product / (norm1 \* norm2)
8. return similarity
9. if \_\_name\_\_ == '\_\_main\_\_':
10. # 假设vec1和vec2是两个文件指纹的向量表示
11. vec1 = np.array([0, 1, 1, 0, 1, 1]) # 向量长度为N，其中N是文件指纹维度
12. vec2 = np.array([1, 0, 1, 1, 1, 0])
13. similarity = cosine\_similarity(vec1, vec2)
14. print("余弦相似度：", similarity)

​

## **四、代码中应用**

——选择合适的哈希算法或文件指纹生成算法取决于对哈希的需求以及所处理数据的特点。

#### **对于 str 类型数据的哈希处理：**

#### **对于 float 类型数据的哈希处理：**

#### **对于 set() 类型数据的哈希处理：**

#### **对于 dict() 类型数据的哈希处理：**