

法律声明

□ 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：大数据分析挖掘

■ 新浪微博：ChinaHadoop



分布式爬虫

大纲

- 网页排重
- SimHash 算法
- SimHash + 海明距离 计算文本相似度
- Huffman 编码

重复网页类型

互联网存在大量的内容转载，也存在不同域名指向同一网站的情况，因此抓取回来的网页会有大量的重复。除了全文重复之外，还存在主要内容重复等情况，关于重复，我们定义以下4种类型：

- 如果2篇文章内容和格式上毫无差别，则这种重复叫做“完全重复页面”
- 如果2篇文章内容相同，但是格式不同，则叫做“内容重复页面”
- 如果2篇文章有部分重要的内容相同，并且格式相同，则称为“布局重复页面”
- 如果2篇文章有部分重要的内容相同，但是格式不同，则称为“部分重复页面”

查重的目的

- 就能节省一部分存储空间，提高检索的质量
- 对以往搜集信息的分析，预先发现重复网页，在今后的网页搜集过程中就可以避开这些网页
- 对重复次数多的网页，给予更高的 **page rank**
- 提高容错能力。比如一个网页的图片不存在了，我们可以到它类似的网页上尝试提取图片

查重算法 – 文本相似度

将一篇文章分词后，词汇加权行成一个向量，称之为这篇文章的特征向量，如果两篇文章的相似度很高，那么他们在高维空间里的夹角应该就比较小

计算向量的夹角：

$$v1 \times v2 = |v1||v2|\cos\theta$$

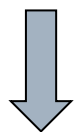
查重算法 – 文本相似度

从附加赛第三轮以3-0力擒素可泰，到小组赛首轮以1-0绝杀首尔FC，再到此役5-1大比分战胜西悉尼流浪者，上海上港在新赛季的亚冠赛场上一举拿到三连胜，为球队在赛季初始阶段开了个好头。此役战胜西悉尼流浪者后，上港不但继续保持着主场不败的金身，也成为在亚冠改制后，继泰达、国安和恒大之后第三支前2轮全胜的球队。

从附加赛以3-0力擒素可泰，到小组赛以1-0绝杀首尔FC，再到此役5-1大比分战胜西悉尼流浪者，上海上港在新赛季的亚冠赛场上拿到三连胜，为球队在赛季开了个好头。战胜西悉尼流浪者后，上港不但继续保持着主场不败的金身，也成为在亚冠改制后，继泰达、国安和恒大之后第三支前2轮全胜的球队。

文本相似度计算

附加赛、素可泰、小组赛、首轮、绝杀、首尔FC、此役、大比分、战胜、西悉尼流浪者、上海上港、赛季、亚冠、赛场、连胜、球队、赛季、好头、战胜、西悉尼流浪者、上港、主场、不败、金身、亚冠、改制、泰达、国安、恒大、全胜、球队



提取特征向量

[30234 23425 234235 234232 ...]

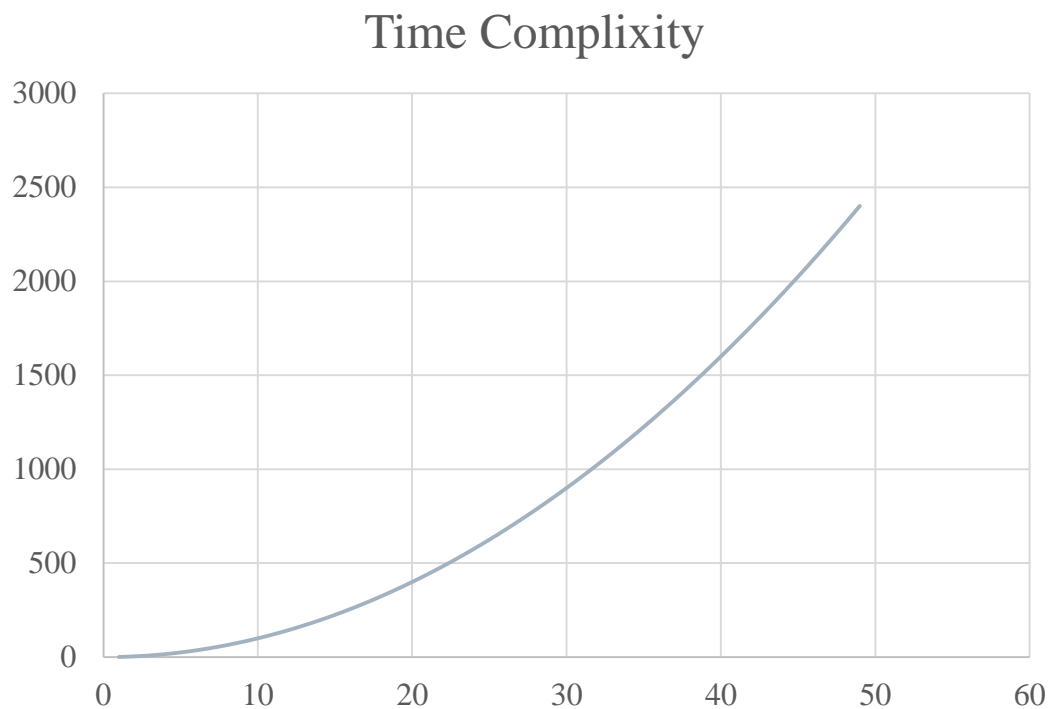


计算夹角

$$\theta = \arccos\left(\frac{v1 \times v2}{|v1| |v2|}\right)$$

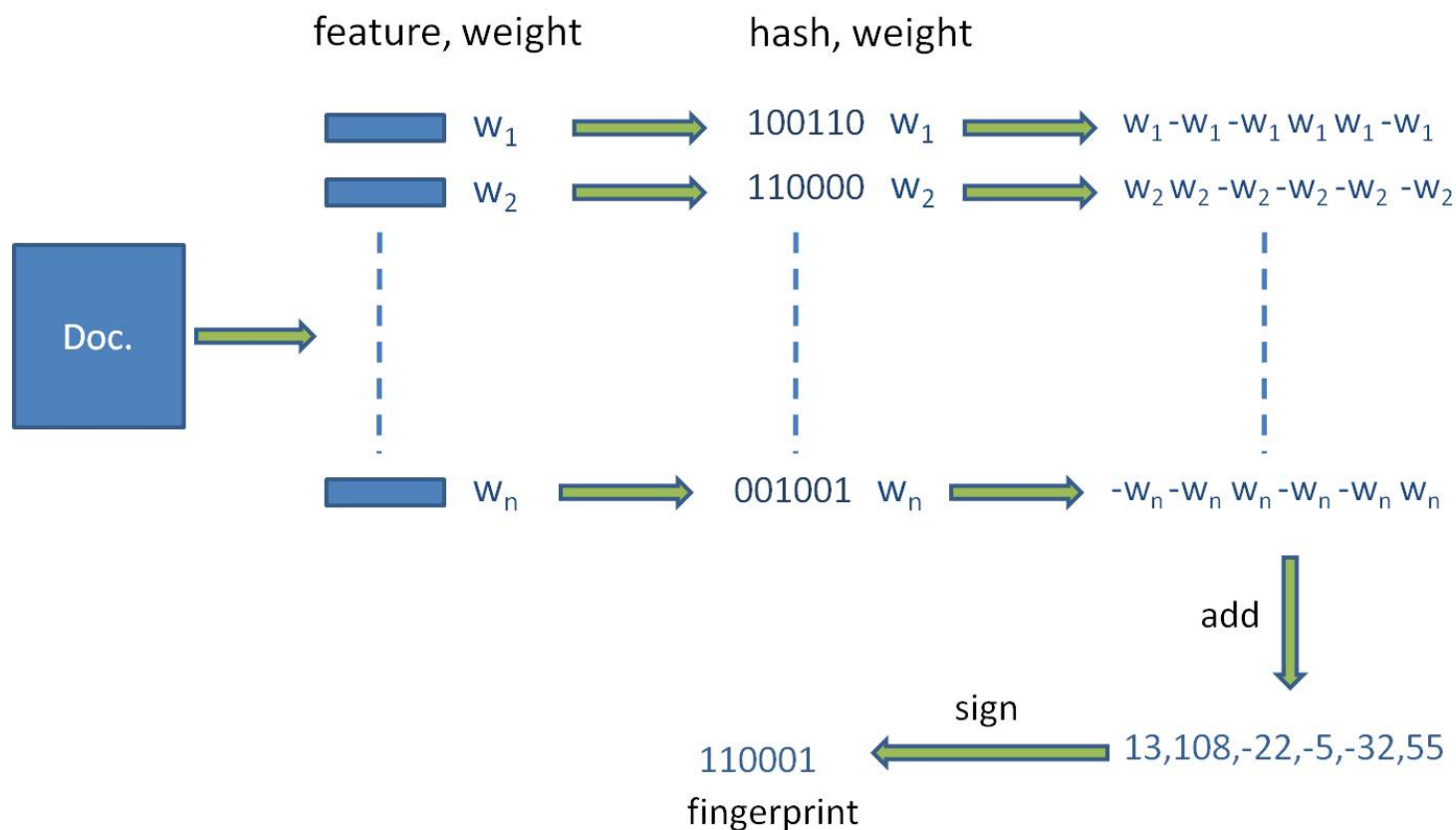
文本相似度计算

时间复杂度太高， n^2



查重算法 - SimHash

Simhash



SimHash – I 分词

分词，把需要判断文本分词形成这个文章的特征单词。最后形成去掉噪音词的单词序列并为每个词加上权重，我们假设权重分为5个级别（1~5）

附加赛	3
小组赛	4
首轮	1
绝杀	3
首尔FC	4
西悉尼流浪者	4
上海上港	5
赛季	5
亚冠	5

SimHash – II Hash

通过hash算法把每个词变成 Hash 值，比如“亚冠”通过 Hash 算法计算为 111000，“上海上港”通过 Hash 算法计算为 101011。这样我们的字符串就变成了一串串数字，要把文章变为数字计算才能提高相似度计算性能，现在是降维过程进行时

附加赛	3	101001
小组赛	4	101110
首轮	1	110001
绝杀	3	101000
上海上港	5	101011
赛季	5	101100
亚冠	5	111000

SimHash – III 加权

Hash 生成结果，需要按照单词的权重形成加权数字串，bit 为 1，权重值乘以1，bit 位为0，权重值乘以-1，比如“亚冠”的 Hash 值为 111000，通过加权计算为 5 5 5 -5 -5 -5

附加赛	3	101001	3 -3 3 -3 -3 3
小组赛	4	101110	4 -4 4 4 4 -4
首轮	1	110001	.
绝杀	3	101000	.
上海上港	5	101011	.
赛季	5	101100	
亚冠	5	111000	5 5 5 -5 -5 -5

SimHash – VI 合并

把上面各个单词算出来的序列值累加，变成只有一个序列串，也就是每一个数字位相加

附加赛	3	101001	3 -3 3 -3 -3 3
小组赛	4	101110	4 -4 4 4 4 -4
首轮	1	110001	1 1 -1 -1 -1 1
绝杀	3	101000	3 -3 3 -3 -3 -3
上海上港	5	101011	5 -5 5 -5 5 5
赛季	5	101100	5 -5 5 5 -5 -5
亚冠	5	111000	5 5 5 -5 -5 -5
			<hr/>
			26 -24 24 -8 -8 -8

SimHash – V 降维

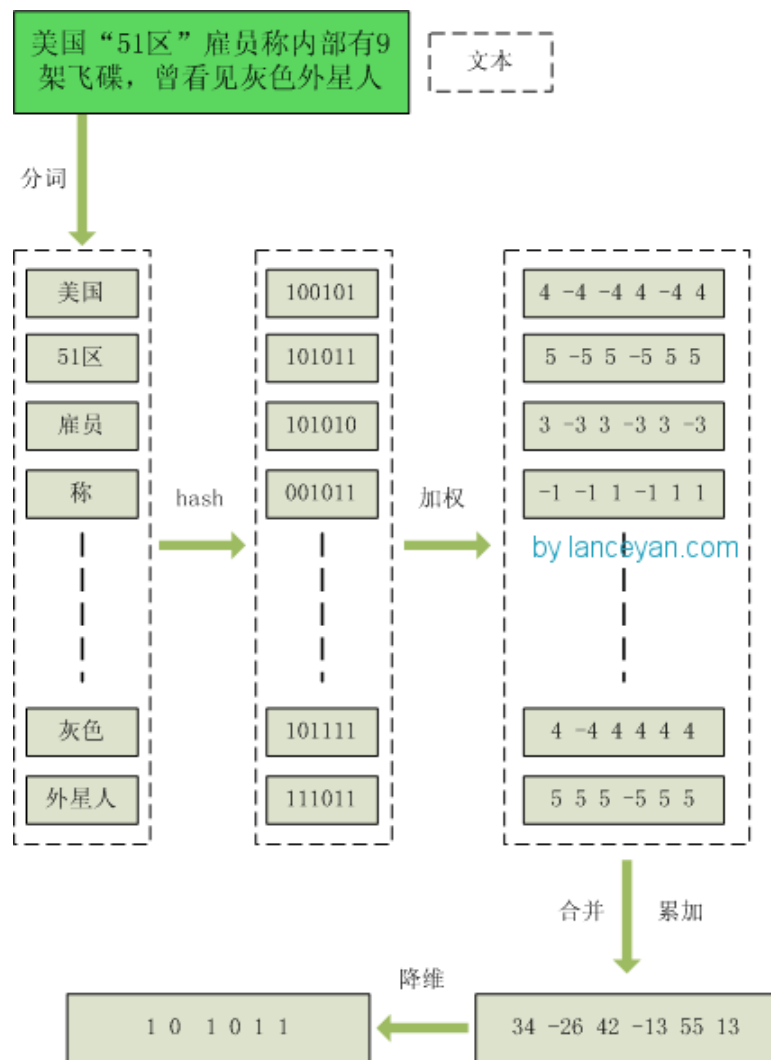
将结果转换为 0 1 这样的序列，规则是如果一个位上的和（例如第一位的和是26） > 0 ，则该位设置为1；否则为 0

26 -24 24 -8 -8 -8



1 0 1 0 0 0

SimHash – 图解



SimHash – 意义

将加权值合并的时候进行了第一步的降维，在最后一步根据正负换算为1和0（不考虑数值本身的大小，只考虑正负）进行了第二次降维，因此最终的Hash 值对字符串的改变变得不是很敏感。

传统 Hash 算法，任意字符的变化，都会导致整个 Hash 结果的剧烈变化

上港不但继续保持着主场不败的金身，也成为在亚冠改制后
上港不但继续保持主场不败的金身，成为在亚冠改制后

SimHash

1101100100010011101000101010110001010101101
110100010000001110000101011110001010101101

md5:

68254e797b2bdc7e022b0004649a8c96
a500532daefa7bdb50159fce4687208

SimHash – 海明距离

海明距离：两个二进制串中不同位的数量

可以通过异或，然后求bit为1的个数，即 $A \text{ xor } B$ 后二进制中1的个数

Python Library: SimHash

`__init__`: 构造函数，文本

`build_by_features()`: 输入一个 dictionary 或者 包含tuple的数组，以 key – weight 的方式输入，例如 `{'Apache':20, 'hadoop':25}` 或 `[('Apache', 20), ('hadoop', 25)]`

`distance(another)`: 计算与另一个Simhash 对象的海明距离

SimHash – 海明距离

```
pip install simhash
```

```
from simhash import Simhash
```

```
str0 = 'The Apache Hadoop software library is a framework that allows for  
the distributed processing large data'
```

```
str1 = 'The Apache Hadoop software library is a framework that allows for  
the distributed processing big data'
```

```
# 构造 SimHash 对象
```

```
sh0 = Simhash(str0)
```

```
sh1 = Simhash(str1)
```

```
# 构造特征值，关键字加权
```

```
features = [('Apache', 10), ('Hadoop', 15), ('framework', 3), ('distributed',  
10), ('data', 6)]
```

```
# 不加权计算
```

```
sh0.distance(sh1)
```

```
# 加权计算海明距离
```

```
sh0.build_by_features(features)
```

```
sh1.build_by_features(features)
```

```
sh0.distance(sh1)
```

处理数百万的网页？

假设我们需要计算海明距离在3以内的网页，可以将64位 Hash 所有变化在3以内组合列出来，一共是

$$C_{64}^3 = \frac{64 \times 63 \times 62}{3 \times 2} = 41664$$

当海明距离增加，所需要的空间也指数级增加，思考找一种更加高效的方法？

处理数百万的网页？

问题：一个80亿的64-bit指纹组成的集合Q，对于一个给定64-bit的指纹F，如何在毫秒级找到Q中和 f 至多只有 $k(k = 3)$ 位 差别的指纹？

如果将 f 的 64位 Hash 所有变化在3以内组合列出来，一共是

$$C_{64}^3 + C_{64}^2 + C_{64}^1 + C_{64}^0 = 43745$$

也就是说，所有与文档F，指纹距离在3以内的文档一定存在于这43745个指纹组成的列表里

两种简单粗暴的办法

- **online**，实时把 F 的 43745种距离为3个指纹结果计算出来，然后依次查找，因此需要计算出43745个结果并进行43745次查找
- **offline**，离线把 F 的 43745个距离为3个指纹结果缓存起来。意味着，如果一万个网页，每个网页需要存储 43745个SIMHASH拷贝，总量为 437,450,000 个 SIMHASH值，这样可以在 $O(1)$ 的时间查找出所有相似网页

优化查找过程

把上面的粗暴方法结合一下，不要实时计算所有的 **SimHash** 可能性并逐一比较，提前进行一些离线计算，但同时也全部都算出来，以至于存储 43745 倍个拷贝，我们做一个折中

假设我们计算的是海明距离为 3 以内的网页，那么根据抽屉原则，如果把 **SimHash** 的 64 位分为 4 块，**ABCD**，如果 2 个海明距离为 3 以内的网页，它们至少有一个块是完全一样的

分为4等份

1100101100100010

A

1100100000000010

B

1100111100000010

C

0000101100101100

D

A

B

C

D

B

A

C

D

C

A

B

D

D

A

B

C

根据抽屉原则，如果海明距离为3，那么至少有1个块会完全重合

算法复杂度

空间会增加到 4 倍，搜索时间，由于排除了16个bit位，因此总的可能性

降低从 2^{64} 降低到了 4×2^{48} ，即当前概率是之前概率的 $4 \times \frac{2^{48}}{2^{64}} =$

$\frac{4}{2^{16}}$ ，假设网页总数为 $8B = 2^{34}$ ，排除掉16个bit后，还剩下 $2^{34-16} = 2^{18}$

也就是256K条数据，一一比较的话就是 $4 \times 256K = 1M$ 次

与之前的粗暴运算的方法比，这种方法不但增加了空间，比较的次数还增加了。注意到，排除16个bit位后，需要比较的次数是指数方式在减少，从总共80亿次比较减少到了100万次，我们是不是可以想办法继续增加排除的bit位？

算法复杂度

把64 bit分割为 16、12、12、12、12 这样5个块，根据抽屉原理，有至少2个块会完全一致，也就是我们会需要 $C_5^2 = 10$ 倍存储空间，这样可以把每个区块的数据减少到 $10 \times 2^{34-28} = 640$ 条

A B C D E

ABCDE
ACDEB
ADEBC
AEB CD
BCDEA
BDEAC
BEACD
CDEAB
CEABD
DEABC

把一个SimHash的指纹，分割为ABCDE 5块，对前两个块完全匹配，后面的通过异或来比较海明距离

准确率与召回率

准确率（**Precision**）：判断为**True**的数据中，判断正确的比例

召回率（**Recall**）：所有**True**的数据，被判断正确的比例

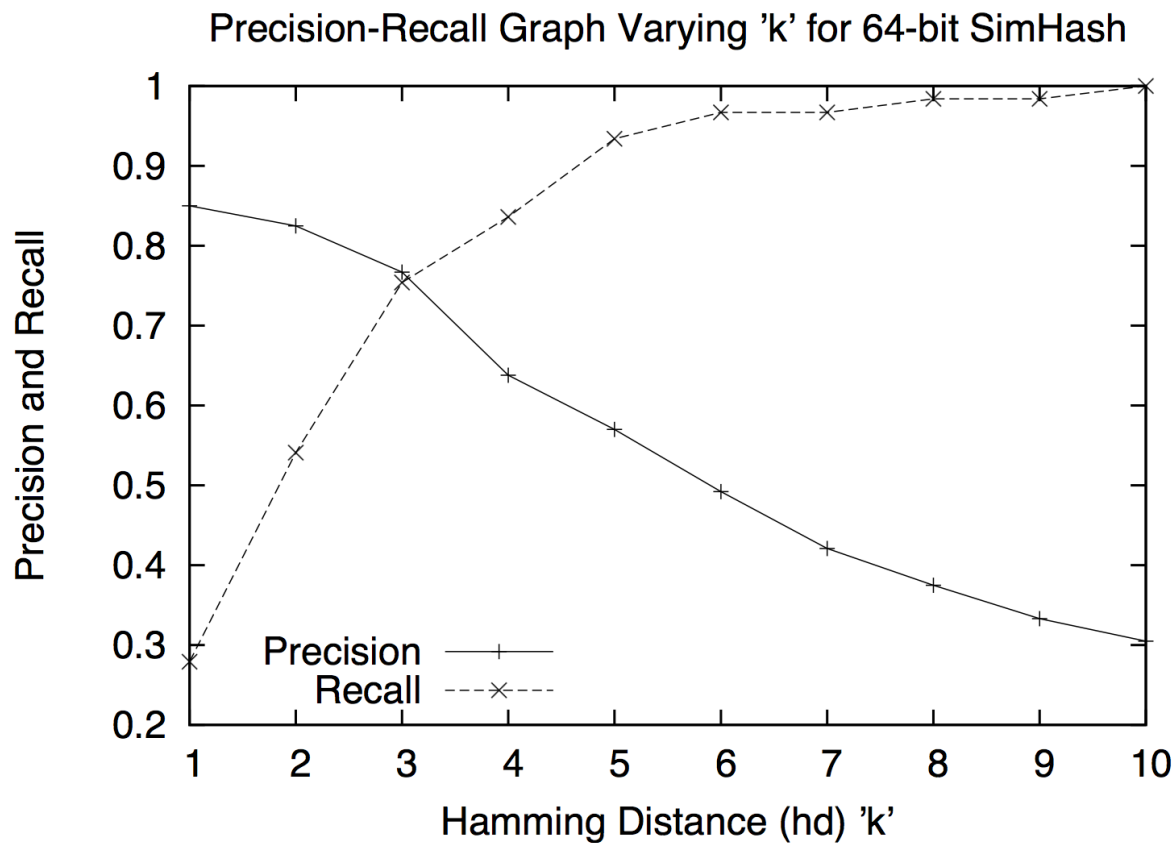
E.g: 有120个嫌疑人，其中100个人是罪犯，20个人是无辜的。一套识别系统，找出了90个罪犯，但是其中有5个人是无辜的，因此 准确率 = $\frac{85}{90}$

100个罪犯，最终找到90个，因此召回率 = $\frac{90}{100}$

假设极端一点，把120个人都认定为罪犯，那么召回率就是100%，而准确率就只有100/120；对应的如果把指标设定非常严格，系统只能找出1个罪犯，此时准确率就是100%，但是召回率就极低

所以准确率与召回率很难两个都做得很高，但是如果两个都很低，系统就设计得有问题了

k 的影响



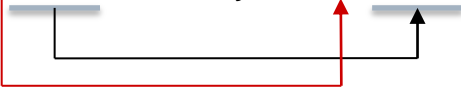
海明距离为 k 的时候，召回率与准确率的影响

k 的影响

- k 的值设置得小，海明距离更近，因此两篇文章的相似度更高，因此准确率会很高，但是召回率会降低
- k 的值设置得大，海明距离更远也认为是类似，因此召回率会提高，但是准确率会降低
- k 的值设置得大，意味着变化组合更多，因此计算量会指数级地增加，例如 $k = 3$ 的时候，大约是 $C_{64}^3 = 41664$ 种海明距离为 3 的组合，而 $k = 4$ 的时候，就会有 $C_{64}^4 = 635376$ 种组合

存储空间的压缩

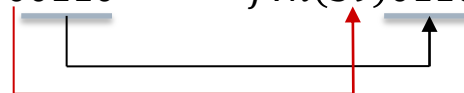
<i>a</i> : 0100000101011100101110101100001010100010100010010	<i>a</i> :
<i>b</i> : 0100000101011100101110101100001010100010100010110	<i>b</i> : <i>h</i> (61)10
<i>c</i> : 0100000101011100101110101100001010100010101011110	<i>c</i> : <i>h</i> (60)110
<i>d</i> : 0100000101011100101110101100001010100010101111110	<i>d</i> : <i>h</i> (58)11110
<i>e</i> : 0100000101011100101110101100001010100010101110110	<i>e</i> : <i>h</i> (60)110
<i>f</i> : 0100000101011100101110101100001010100010101000110	<i>f</i> : <i>h</i> (59)0110



如果我们把 **8B** 个数进行排序，会发现相邻两个数前面的**bit**位是高度重合的，因此我们我们需要每个数都用**64bit**来存储，可以将前面相似的部分记下来。比如 *b* 是第**61**位与 *a* 不同，因此 *b* 可以记为 *h*(61)10，*c* 记为 *h*(60)110，*d* 记为 *h*(58)11110 ...

存储压缩数据

<i>a</i> : 0100000101011100101110101100001010100010100010010	<i>a</i> :
<i>b</i> : 0100000101011100101110101100001010100010100010110	<i>b</i> : <i>h</i> (61)10
<i>c</i> : 0100000101011100101110101100001010100010101011110	<i>c</i> : <i>h</i> (60)110
<i>d</i> : 0100000101011100101110101100001010100010101111110	<i>d</i> : <i>h</i> (58)11110
<i>e</i> : 0100000101011100101110101100001010100010101110110	<i>e</i> : <i>h</i> (60)110
<i>f</i> : 0100000101011100101110101100001010100010101000110	<i>f</i> : <i>h</i> (59)0110



$h(n)$ 中, n 最多为63, 意味着可以用6个bit位来表示 $h(n)$, 因此综合起来可以用 6个bit 位+后续的比特位来组成一个SimHash值, 例如 $h(61)10$ 就可以表示为 **111101**10, $h(58)11110$ 就可以表示为 **111010**11110

存储压缩数据

a: 0100000101011100101110101100001010100010100010010

a:

b: 0100000101011100101110101100001010100010100010110

b: $h(61)10$

c: 0100000101011100101110101100001010100010101011110

c: $h(60)110$

d: 0100000101011100101110101100001010100010101111110

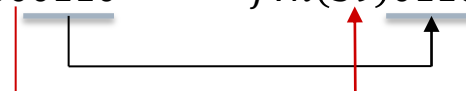
d: $h(58)11110$

e: 0100000101011100101110101100001010100010101110110

e: $h(60)110$

f: 0100000101011100101110101100001010100010101000110

f: $h(59)0110$



上面的 $abcdef$ 可以连续存储为:

01000001010111001011101011000010101000101000100101110110111100110
111010111101111001101110110110

压缩过程

1. 存储第一个SimHash

a : 0100000101011100101110101100001010100010100010010

2. 将第二个 SimHash b 与 a 做异或，得到 100 （前面的0都会忽略）
3. 记下这一位的位置 n ，并右移 $64 - n$ 位，得到后面需要存储的bit 位 10
4. 将 n 然后append到block后面
5. 将 不同的 bit 位 10 append 到block后面
6. 如此反复直到 block 的 size full，开始下一个block

解压过程

1. 读出一个block到内存，然后读出64位
2. 读出6个bit位，得到 n ，于是 $64 - n$ 就是后面bit位的长度
3. 将前一个SimHash值 hsa 赋值给当前要解码的 hsb
4. $hsb \oplus 1 \ll n$ ，将标记不同的起始位取反
5. 读出后面 $64 - n$ 个 bit 位，并依次通过 OR 操作设置到 hsb 对应的位置
6. 重复此过程直到block的数据全部读出并解码

进一步压缩

010000010101110010111010110000101010001010001001011110110111100110
111010111101111001101110110110

前面的例子中，每一个 $h(n)$ 我们用了6个bit位来表示，这里我们完全可以利用 Huffman 编码，根据 $h(n)$ 的出现概率对 $h(n)$ 做变长的编码，以进一步压缩 $h(n)$ 的存储空间

霍夫曼编码（Huffman Coding）

用于无损数据压缩的熵编码（权编码）算法。由大卫·霍夫曼在1952年发明。在计算机数据处理中，霍夫曼编码使用**变长编码表**对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种**评估来源符号出现概率**的方法得到的，**出现概率高的字母使用较短的编码**，反之出现概率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。

原理

考虑一颗带权重的二叉树，假设

- 叶子节点到根节点的长度为 l
- 每个节点的权重为 w

那么，一棵树的权重纸盒就是 $\sum_{i=0}^n w_i l_i$ ，对于下面的二叉树，图3权重最小

霍夫曼编码的基本思想就是把概率最低的字符，路径最长，而出现概率最高的字符路径最短（也就是编码最短）

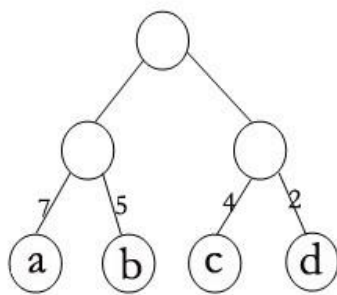


图 1

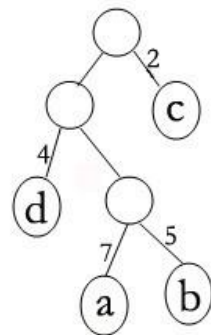


图 2

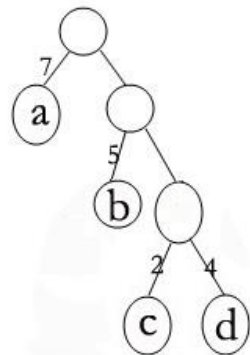
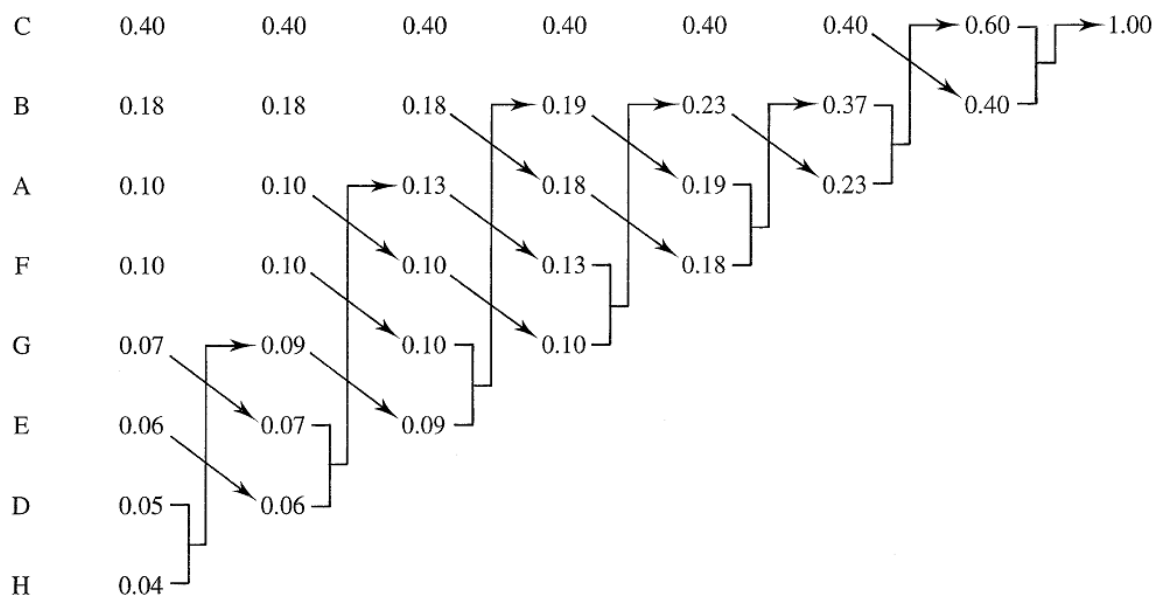


图 3

编码过程

1. 按照概率进行排序
2. 取出最小概率的2个取出，把它们概率相加后得到一个新的数，重新再加入队列
3. 重复过程2，直到所有符号都被取出



对 $h(n)$ 进行霍夫曼编码

扫描全库，记录各个 $h(n)$ 的出现次数，然后对 $h(n)$ 进行霍夫曼编码

Symbol	Weight	Huffman Code
62	5	00
58	4	011
59	4	100
63	4	110
64	4	111
60	2	1010
61	2	1011
69	1	0100
30	1	01010
32	1	01011

网页排重过程

1. 建立10个分块存储区
 2. 计算一篇文章的SimHash，对于主题抓取，可以对特定主题词增加权重
 3. 把SimHash值分为16、12、12、12、12 这样的5块，计算出10个组合
 4. 将这10个组合，分别插入到各自对应的分块中，通过二分法插入
 5. 重复2~4直到所有的文档都计算完SimHash并插入到索引表
 6. 扫描索引表，进行压缩，将每一个块的第一个SimHash保存到单独索引表
 7. 对于一个要检查的文档，计算SimHash，分块后计算出10个查找Key，通过SimHash 的 block 索引表，利用插补法进行查找，找到AB区域相同的block
 8. 解压block，再一一比较计算海明距离，找出距离小于k的文档
- * 1~6 蓝色部分是建立索引过程，7~8 绿色部分是排重的过程

疑问

□ 问题答疑：<http://www.xxwenda.com/>

■ 可邀请老师或者其他回答问题

联系我们

小象学院：互联网新技术在线教育领航者

- 微信公众号：大数据分析挖掘
- 新浪微博：ChinaHadoop

