



華東師範大學

EAST CHINA NORMAL UNIVERSITY

数据科学与工程算法基础

Algorithm Foundations of Data Science and Engineering

第四章 哈希算法

$$(1+x)^n = 1 + \frac{nx}{1!} + \frac{n(n-1)x^2}{2!} + \dots$$

课程提纲

Content

1 算法引入

2 布隆过滤器

3 局部位置敏感哈希

课程提纲

Content

1 算法引入

2 布隆过滤器

3 局部位置敏感哈希

如何快速判断用户名是否被注册

- 判断用户名是否被注册相当于判断一个元素是否出现在一个集合中
- 如何能做到瞬间反馈结果？
 - 用链表表示集合，查找效率会很低
 - 对用户名建立 B+ 树索引，可以提高查询效率，复杂度为 $O(\log n)$ ，其中 n 为集合大小
 - 能否找到 $O(1)$ 复杂度的方法呢？

文本冗余检测

布隆过滤器(Bloom Filter)详解

直观的说, bloom算法类似一个hash set, 用来判断某个元素 (key) 是否在某个集合中。
和一般的hash set不同的是, 这个算法无需存储key的值, 对于每个key, 只需要k个比特位, 每个存储一个标志, 用来判断key是否在集合中。

算法:

1. 首先需要k个hash函数, 每个函数可以把key散列成为1个整数
2. 初始化时, 需要一个长度为n比特的数组, 每个比特位初始化为0
3. 某个key加入集合时, 用k个hash函数计算出k个散列值, 并把数组中对应的比特位置为1
4. 判断某个key是否在集合时, 用k个hash函数计算出k个散列值, 并查询数组中对应的比特位, 如果所有的比特位都是1, 认为在集合中。

优点: 不需要存储key, 节省空间

缺点:

1. 算法判断key在集合中时, 有一定的概率key其实不在集合中

论坛 问答 代码 直播 电子书 布隆过滤器 登录/注册 会员中心 收藏 消息 创作中心

布隆过滤器(Bloom Filter)详解

转载 shadow_zed 2019-04-21 19:32:30 229 收藏

分类专栏: java 文章标签: 布隆过滤器

原文:布隆过滤器(Bloom Filter)详解

直观的说, bloom算法类似一个hash set, 用来判断某个元素 (key) 是否在某个集合中。
和一般的hash set不同的是, 这个算法无需存储key的值, 对于每个key, 只需要k个比特位, 每个存储一个标志, 用来判断key是否在集合中。

算法:

1. 首先需要k个hash函数, 每个函数可以把key散列成为1个整数
2. 初始化时, 需要一个长度为n比特的数组, 每个比特位初始化为0
3. 某个key加入集合时, 用k个hash函数计算出k个散列值, 并把数组中对应的比特位置为1
4. 判断某个key是否在集合时, 用k个hash函数计算出k个散列值, 并查询数组中对应的比特位, 如果所有的比特位都是1, 认为在集合中。

优点: 不需要存储key, 节省空间

网络爬虫



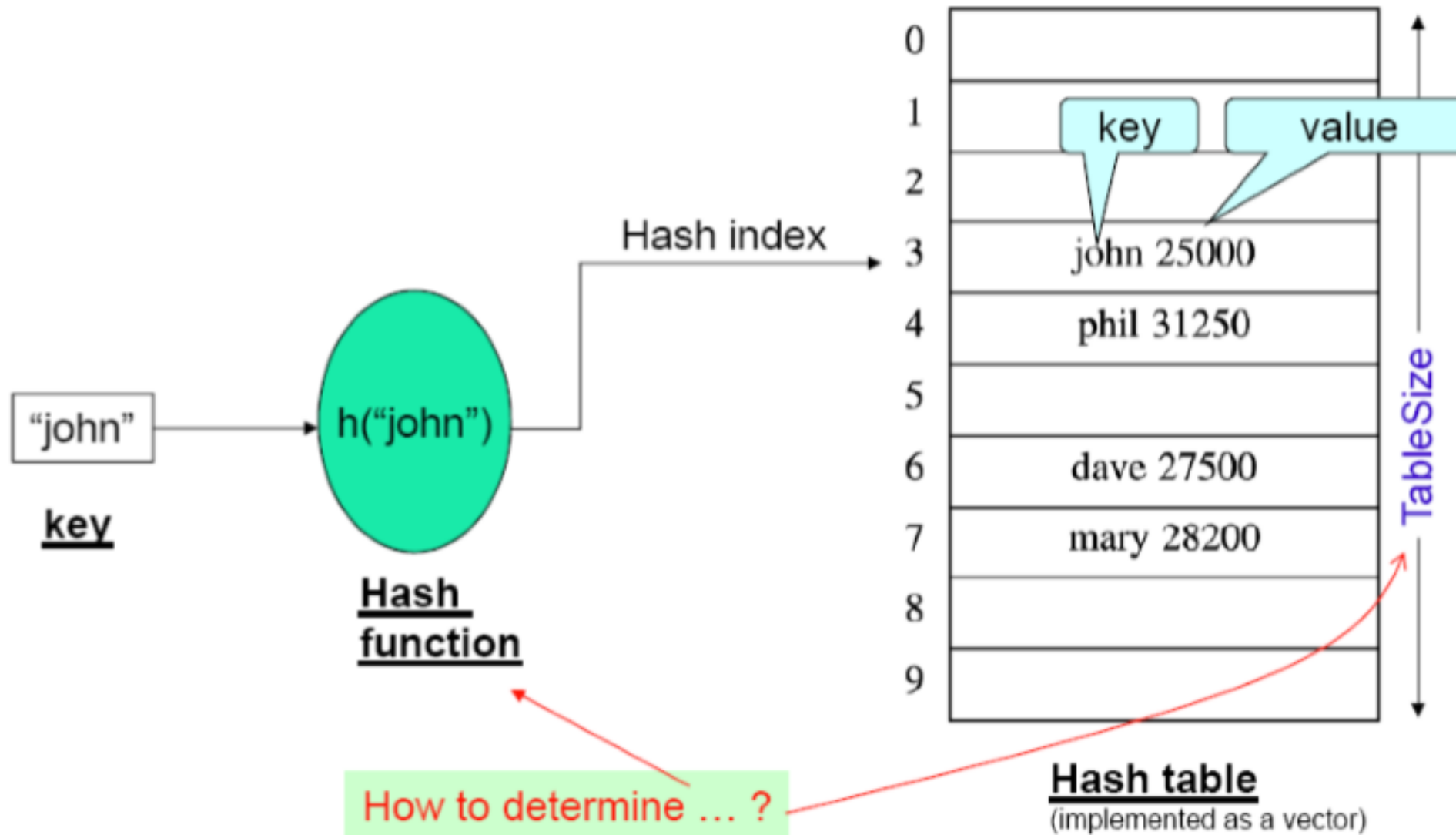
搜索引擎定期爬取网页内容

- 爬取太多冗余文本会浪费存储空间
- 但海量文本需要爬取, 逐篇对比过于低效
- 如何快速判断网页内容是否冗余?

哈希函数

- 哈希函数 $h(\cdot)$ 是一个数学函数，满足 $h : \text{key} \rightarrow \text{value}$ ，即 $h(\text{key}) = \text{value} \in \mathbb{Z}^+$
- 哈希函数的作用
 - **压缩存储**：哈希值所需的存储空间远小于输入关键词占用的空间
 - ✓ 网页 URL 哈希到某个位置，可以表示为一个整数
 - ✓ 邮件地址哈希成一个整数
 - **无冲突**：理想状态下，输入不同的关键词会得到不同的哈希值
 - ✓ 即使是两个差异很小的关键词也会得到两个完全不同的哈希值
 - ✓ 相同关键词被相同哈希函数哈希后不可能得到两个不同的哈希值
 - **不可逆**：在不知道哈希函数的情形下，仅知道哈希值，不可能轻易地猜到此哈希值对应的关键词
 - ✓ 唯一找到关键词的方法是蛮力法（Brute-Force）
 - ✓ 正是因为这一点，哈希函数成为最重要的密码学工具之一

哈希表



数据结构的操作效率

操作	未排序数组	排序数组	链表	二叉排序树
插入	$O(1)$	$O(n)$	$O(1)$ 或 $O(n)$	$O(\log n)$
查找	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
删除	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$

- 通常，对哈希表的插入、查找和删除操作复杂度均为 $O(1)$ ，但并不总是这样
- 影响哈希表操作性能的因素
 - **哈希函数**：将冲突最小化，使 key 和 value 均匀地分布在表中
 - **冲突解决策略**：将 key/value 存储在不同的位置，或将多个 key/value 用链表串起来
 - **哈希表大小**：表过大会降低碰撞的可能，但会造成内存空间的浪费；过小的哈希表会增加碰撞的可能性

课程提纲

Content

1 算法引入

2 布隆过滤器

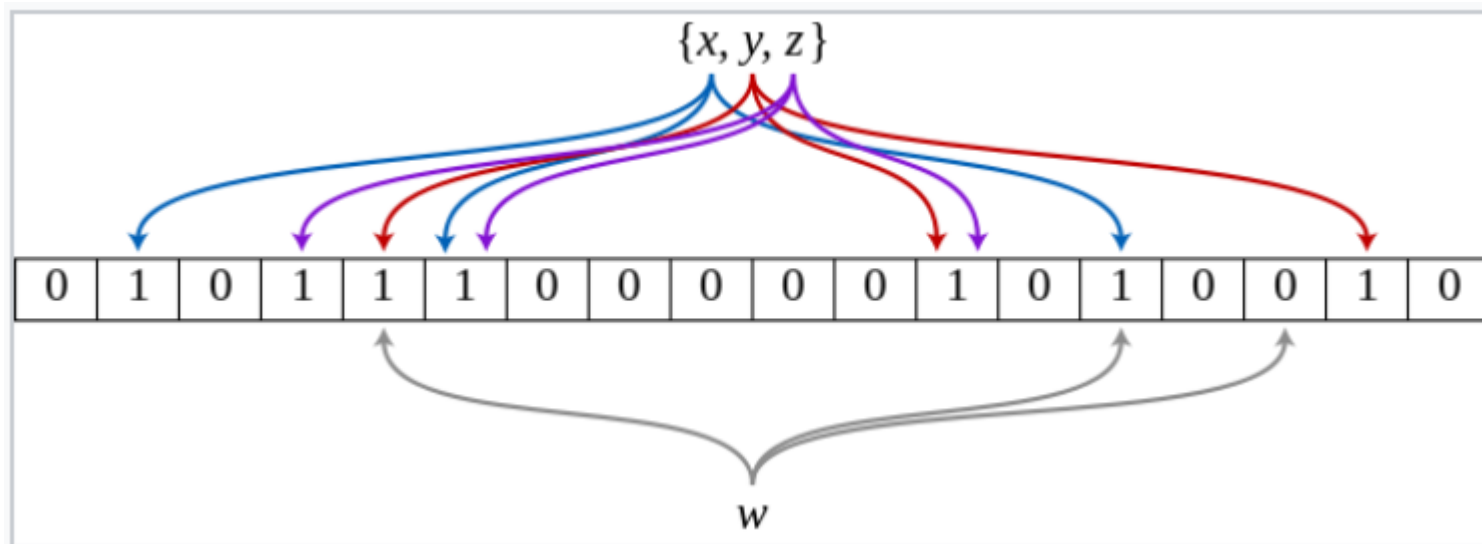
3 局部位置敏感哈希

布隆过滤器

- 回答一个元素是否出现在一个集合中时，仅仅使用一个哈希函数，碰撞可能是一个大的问题
- 布隆过滤器（Bloom Filter）是为了应对碰撞而提出的
 - 它是一种高空间效率的概率数据结构
 - 用于判断一个元素是否属于一个集合的成员
- 布隆过滤器具有广泛的应用
 - 垃圾邮件地址过滤器：可以有效过滤垃圾邮件地址
 - 拼写检查：能很快发现文字编辑软件中输入的错
 - 重复检查：用于网络爬虫判断某个 URL 是否已经被爬取过了，或用户名是否被注册过

布隆过滤器示例

假定集合中有 n 个元素，给定一组 k 个哈希函数 h_1, \dots, h_k ，其中 h_i 的范围为 $\{0, \dots, m-1\}$ ，初始状态长度为 m 的位数组每个位置被置为 0



- 如图为布隆过滤器示例
 - 示例中 $m = 18$, $k = 3$
 - 每个元素被哈希到位数组中的三个位置
 - 图示为插入元素 $\{x, y, z\}$ 后的情形

元素插入

给定 3 个哈希函数 h_1, h_2, h_3 , 其中 h_i 的范围为 $\{0, 1, \dots, 9\}$, 初始状态长度为 10 的位数组每个位置被置为 0

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

令 $H(x) = \{h_1(x), h_2(x), h_3(x)\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

插入 x_1

$H(x_1) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

插入 x_2

$H(x_2) = \{4, 5, 8\}$

元素查询

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$$H(x_1) = \{1, 4, 9\}$$

$$H(x_2) = \{4, 5, 8\}$$

- 元素查询
 - 查询 y_1 , $H(y_1) = \{1, 4, 9\} \rightarrow \text{Yes}$
 - 查询 y_2 , $H(y_2) = \{0, 4, 8\} \rightarrow \text{No}$
 - 查询 y_3 , $H(y_3) = \{1, 5, 8\} \rightarrow \text{Yes (假阳性)}$
- 布隆过滤器可能会误判
 - 不会出现拒真的情形（假阴性不会出现）
 - 但是可能出现纳伪的情形（假阳性）
- 如何降低误判率？

误判率分析

- 当插入一个元素到布隆过滤器，一个哈希函数未将某个特定位置置为 1 的概率为 $1 - \frac{1}{m}$
- 当一个元素插入布隆过滤器后， k 个哈希函数未将特定位置置为 1 的概率为 $\left(1 - \frac{1}{m}\right)^k$
- 将 n 个元素插入布隆过滤器后，特定位置未被置为 1 的概率为 $\left(1 - \frac{1}{m}\right)^{kn}$
- 因此，某个特定位置为 1 的概率为 $1 - \left(1 - \frac{1}{m}\right)^{kn}$

误判率分析（续）

- 查询某个元素时，当 k 个哈希函数对应的位置均为 1，则过滤器声称该元素属于该集合

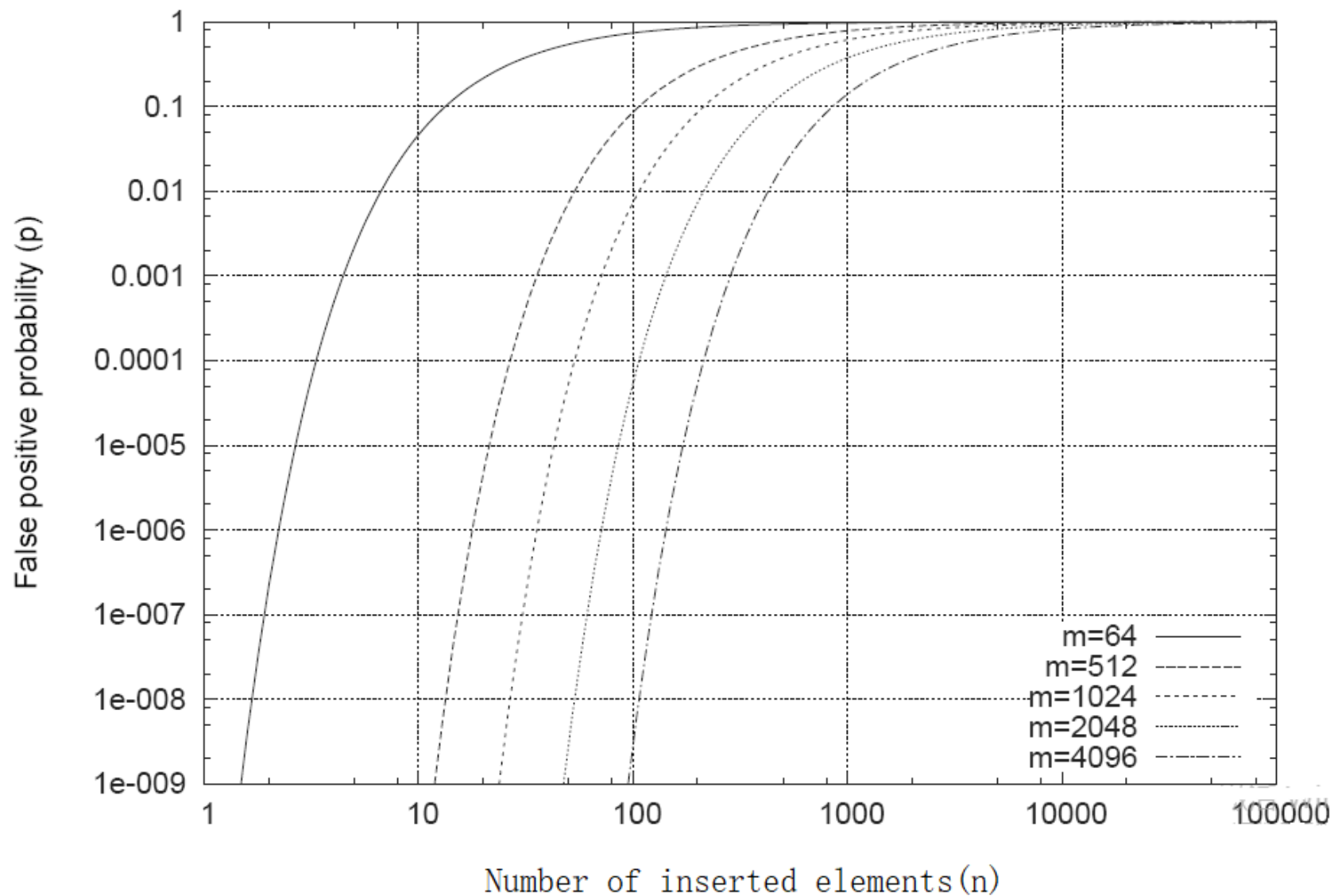
- 当该元素不属于该集合时，会发生误判，其概率为

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

x	$\left(1 - \frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

- 误判概率随布隆过滤器 m 的增大而减小
- 随着更多元素的加入，误判概率随 n 的增加而增加
- 如何降低误判概率
 - 集合中元素个数相对固定，空间大小可能受限
 - 可以通过适当选择哈希函数的个数即 k 值，最小化概率 $\left(1 - e^{-\frac{kn}{m}}\right)^k$

误判率



最小化误判率

- 给定 m 和 n , 选择一个合适的 k 使得误判率最小
- 令 $\rho = e^{-\frac{kn}{m}}$, 则 $f \approx (1 - \rho)^k = e^{k \ln(1 - \rho)}$
- 注意到 $\ln \rho = -\frac{nk}{m}$, 即 $k = -\frac{m \ln \rho}{n}$, 令
$$g = k \ln(1 - \rho) = -\frac{m}{n} \ln \rho \ln(1 - \rho)$$
- 因此, 对固定的 m 和 n , 当 $\rho = 1/2$ 时, 误判概率最小

k 值的选择

- 当 $\rho = 1/2$ 时, $k = \ln 2 \cdot \frac{m}{n}$
- 将其值代入, 得 $f = (\frac{1}{2})^k \approx 0.6185^{\frac{m}{n}}$
- 因此, 最优布隆过滤器结构为 $\rho = 1/2$ 时, 即布隆过滤器有一半位置为空
- 此时, $k = \ln 2 \cdot \frac{m}{n}$

误判概率表

- 给定 m/n , 不同 k 值对应的误判率
 - 当 k 值太小时, 两个元素被哈希到相同位置的概率增加, 误判概率也会增加
 - 当 k 值太大时, 更多位置被置为 1, 因此误判概率增加

m/n	k	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
2	1.39	0.393	0.400			
3	2.08	0.283	0.237	0.253		
4	2.77	0.221	0.155	0.147	0.160	
5	3.46	0.181	0.109	0.092	0.092	0.101
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347
8	5.55	0.118	0.0489	0.0306	0.024	0.0217

课程提纲

Content

1 算法引入

2 布隆过滤器

3 局部位置敏感哈希

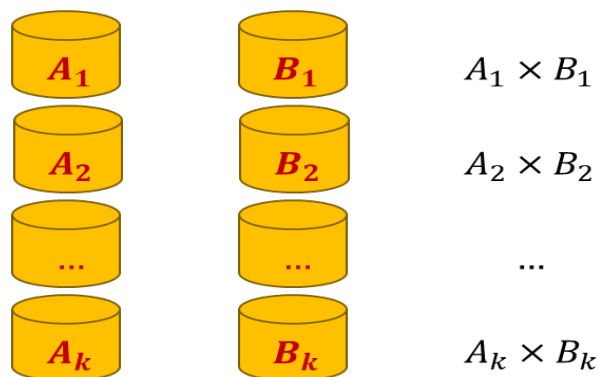
冗余文本检测

- 给定海量文档（百万或数十亿计）查找其中的冗余文本

- 文档太多，逐一比较费时费力
- 文档太大或太多，以致无法放入主存
- 分块或者索引是常用的方法



$A \times B$



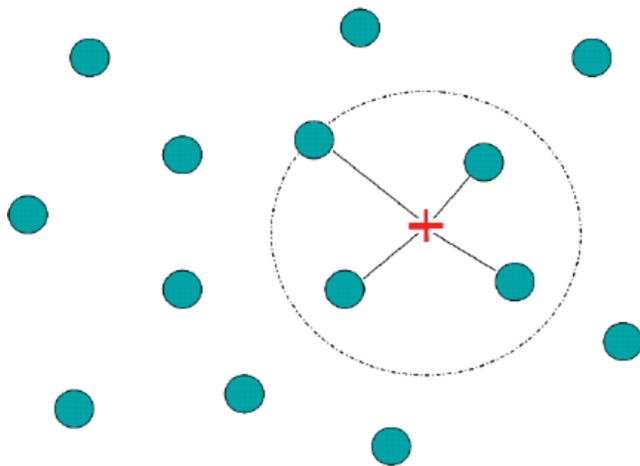
$$\bigcup_{i=1}^k A_i \times B_i$$

- 应用

- 镜像网站发现
- 抄袭检测
- 对象分块

(近似) 最近邻查找

- 给定海量高维数据，找到与查询对象最相似的对象
 - 应用：分类，聚类，文本检索，图像和音频检索
 - 最近邻查找 (NN)：树型索引结构，如二叉搜索树，B+ 树等
 - “维度诅咒”：当数据维度很高时，树型索引结构的查找效率会接近线性查找
- 近似最近邻查找 (ANN)：局部敏感哈希 (LSH)



文本相似度

- 集合相似度和距离

- 给定集合 A 和 B , Jaccard 相似度定义为

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

✓ 用于衡量文本、图中顶点间的相似度

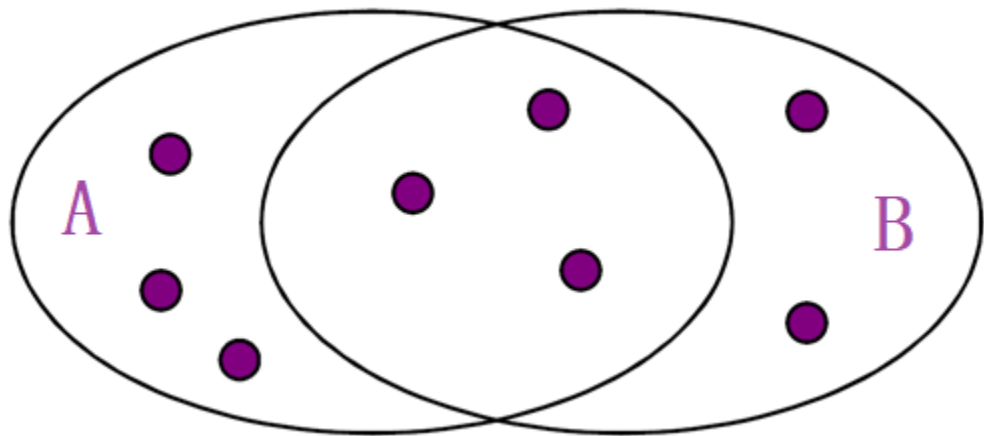
- 给定集合 A 和 B , Jaccard 距离定义为

$$\checkmark d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

- 示例

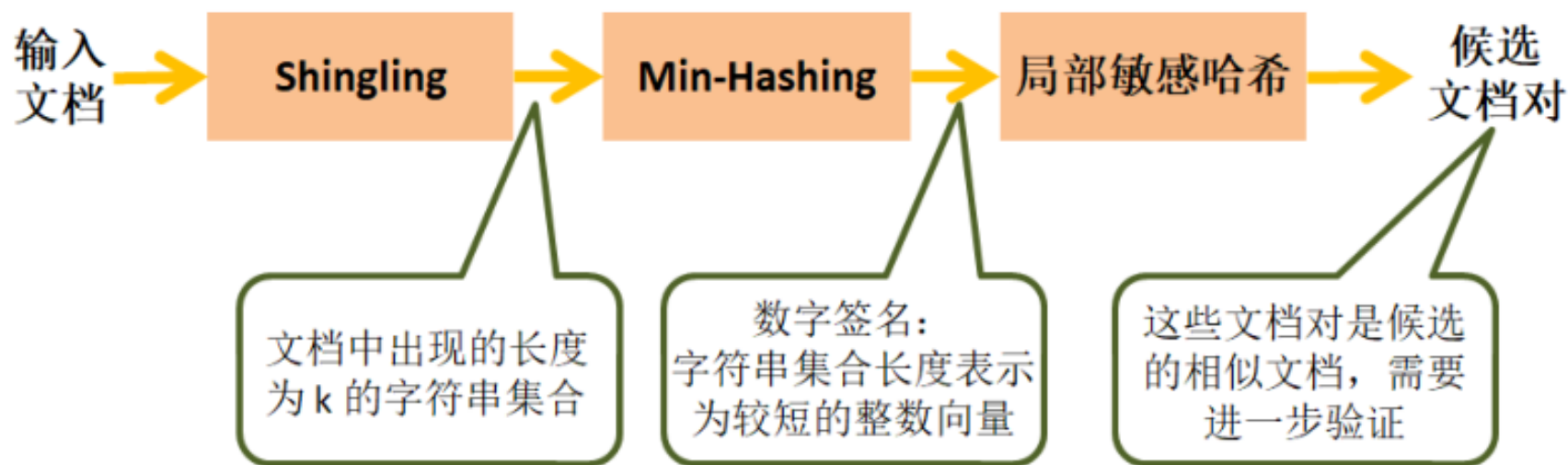
$$\checkmark \text{Jaccard}(A, B) = \frac{3}{8}$$

$$\checkmark d(A, B) = 1 - \frac{3}{8} = \frac{5}{8}$$



冗余文本检测的步骤

- **Shingling**: 将文档转换为集合
- **Min-Hashing**: 将大的集合转换为短的签名, 同时保留相似性
- **局部敏感哈希**: 筛选或寻找相似文档候选对



Shingling: 文档的集合表示

- 文本建模
 - Document = 文档中出现的单词集合
 - Document = 一组“重要”单词的集合
 - 简单分词会损失上下文信息
 - 为关注单词的顺序, 定义 Shingles
- 文本中的 k -Shingles (k -gram) 是文本中连续 k 个 token 组成的序列
 - Token 可能是字符、单词等
 - 给定字符串 $D = abcab$, 令 $k = 2$, 字符串的 2-Shingles 为
$$S(D) = \{ab, bc, ca\}$$
 - 如果 k -Shingles 看作是 multi-set, 表示为 $S(D) = \{ab, bc, ca, ab\}$

位向量编码

					Documents			
Shingles	1	1	1	0				
	1	1	0	1				
	0	1	0	1				
	0	0	0	1				
	1	0	0	1				
	1	1	1	0				
	1	0	1	0				

- 所有 k -Shingles 构成全集
- 文本的位向量
 - 行 = k -Shingles, 列 = 单个文本
 - 第 i 行第 j 列的数值表明第 i 个 k -Shingles 出现在第 j 个文本中
 - 文本的 Jaccard 相似度也可以运用位向量进行计算
- 对文本 C_1 和 C_2
 - 交集大小为 3, 并集大小为 6
 - $\text{Jaccard}(C_1, C_2) = \frac{1}{2}$
 - 距离 $d(C_1, C_2) = \frac{1}{2}$

Min-Hashing

- 目标：找到哈希函数 $h(\cdot)$ ，使得集合 C_1 和 C_2
 - 当 $\text{Jaccard}(C_1, C_2)$ 较高时，则 $P(h(C_1) = h(C_2))$ 较高
 - 当 $\text{Jaccard}(C_1, C_2)$ 较低时，则 $P(h(C_1) \neq h(C_2))$ 较高
- 一旦目标达成
 - 相似文档大概率会被哈希到同一个桶中
 - 不相似文档大概率会被哈希到不同的桶中
- 如何选择哈希函数？
 - 显然，哈希函数的选择依赖于相似度量
 - 并非所有相似度量都能找到合适的哈希函数
 - 幸运的是，对 Jaccard 相似度，**最小哈希**（Min-Hashing）可以实现这一目标

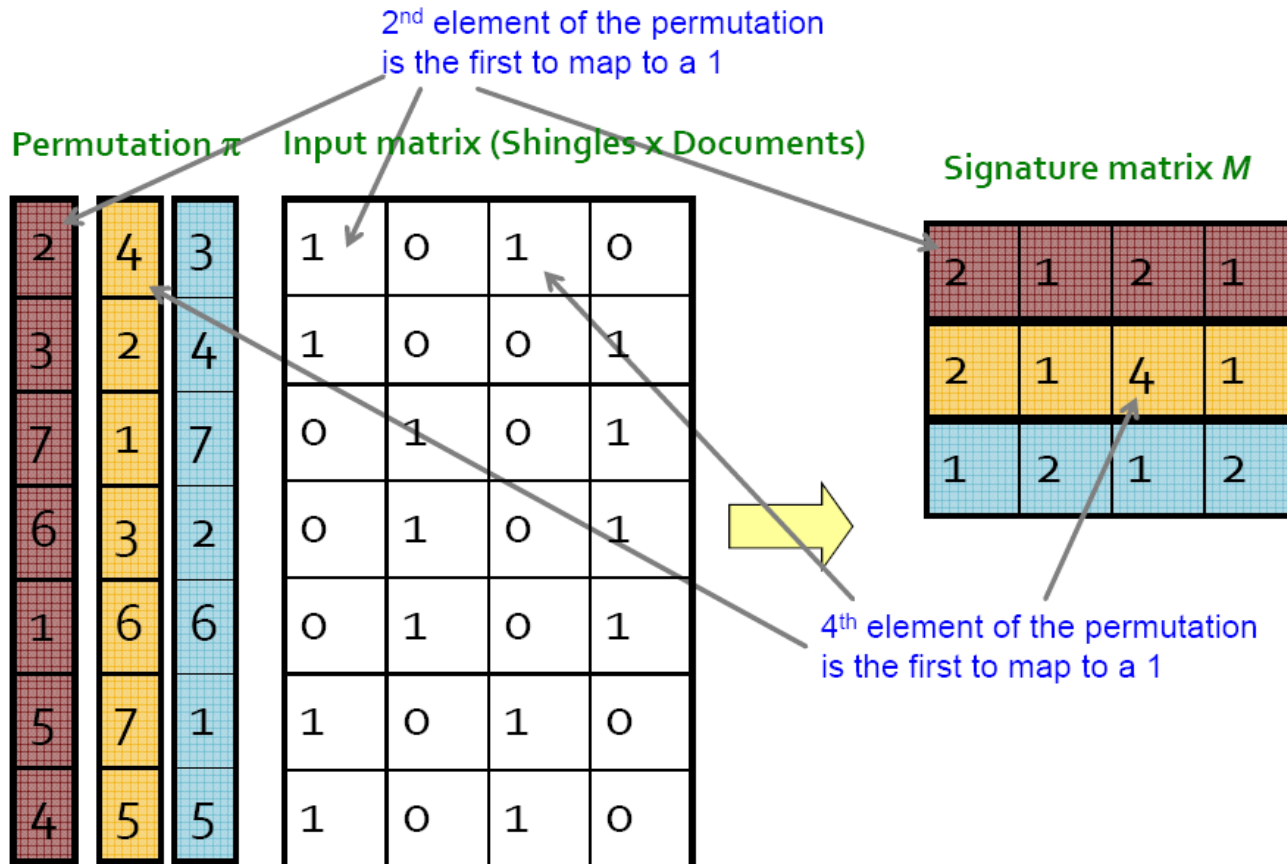
随机排列

- 随机排列是 n 个不同元素的随机排序
 - 洗一副扑克是随机排列最好的例子
 - 因此，随机排列的数量为 $n!$
 - 如图为数字 1—7 的三个随机排列
 - 以第一列为例，新的排列
 - ✓ 第一行是原来的第 5 个位置
 - ✓ 第二行是原来的第 1 个位置
 - ✓ 第三行是原来的第 2 个位置
 - ✓ 第四行是原来的第 7 个位置
 - ✓ 第五行是原来的第 6 个位置
 - ✓ 第六行是原来的第 4 个位置
 - ✓ 第七行是原来的第 3 个位置

2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

最小哈希

- 令 π 为一个随机排列和布尔向量 C ，定义一个“哈希”函数 $h_\pi(C) =$ 随机排列后第一个不为 0 的行号索引，即 $h_\pi(C) = \min_{\pi} \pi(C)$



最小哈希的性质

- 给定一个随机排列 π , 我们有 $P(h_\pi(C_1) = h_\pi(C_2)) = \text{Jaccard}(C_1, C_2)$

- 证明:

集合 C_1 和 C_2 的特征矩阵每行只有三类情况: (1) 均为 1; (2) 有且只有一个为 1; (3) 均为 0。

假设第一类有 a 行, 第二类有 b 行, 第三类有 c 行, 则 $\text{Jaccard}(C_1, C_2) = a/(a + b)$ 。

随机重排之后 $h_\pi(C_1) = h_\pi(C_2)$ 则意味着在遇到第一类情况之前没有遇到第二类情况。由于重排是随机的, 根据古典概型, 可以知道

$$P(h_\pi(C_1) = h_\pi(C_2)) = \frac{a}{a + b} = \text{Jaccard}(C_1, C_2)$$

基于最小哈希的相似度

- 对任意的 π , $h_\pi(C_1) = h_\pi(C_2)$ 是一个贝努里随机变量, 可以通过成功频数估计概率
- 因此, 可以基于最小哈希估计集合间的相似度

Permutation π

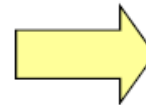
2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Input matrix (Shingles x Documents)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix M

2	1	2	1
2	1	4	1
1	2	1	2



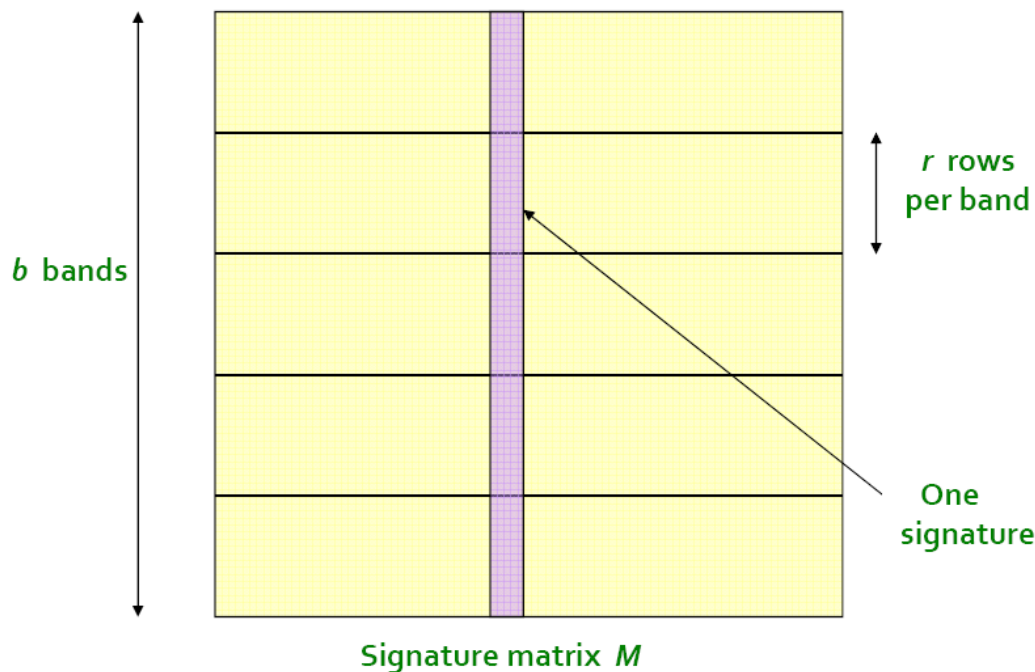
Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

最小哈希签名

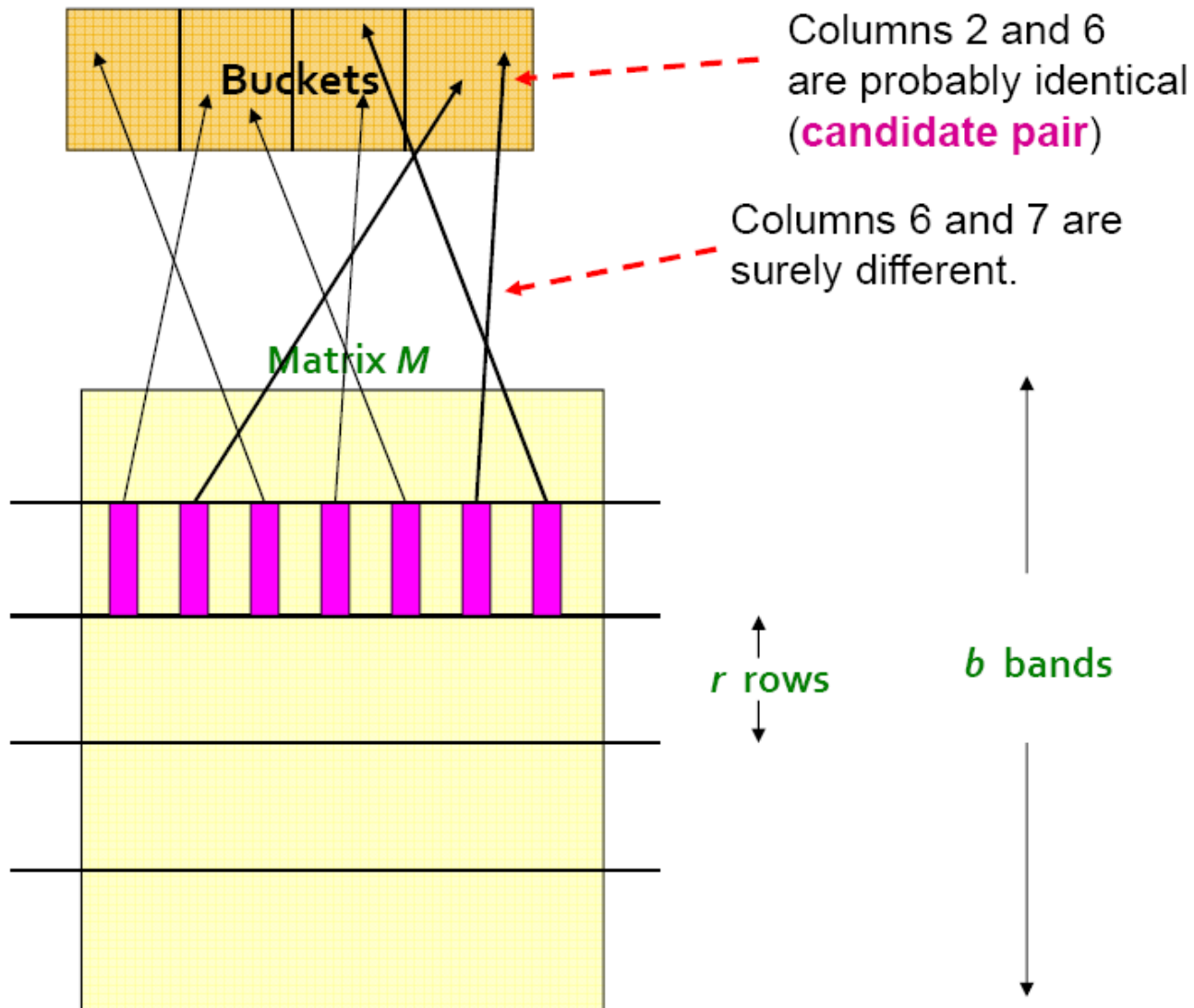
- 假定有 100 个随机排列，每一个文本 C 对应 100 个最小哈希值
- 把这 100 个哈希值看作是一个列向量，记为 $\text{sig}(C)$ ，其中
$$\text{sig}(C)[i] = \min(h_{\pi_i}(C))$$
 - 将高维布尔向量“压缩”为低维向
 - 低维向量记作一个文本的签名向量
- 重要观察
 - 100 维签名向量相同的两个集合大概率是相同的
 - 即使很相似，稍有不同的集合可能签名向量也不同
 - 为了能够找到相似文本，对签名向量进行分组

签名矩阵分组



- 将签名矩阵 M 划分为 r 行，每行由 b 个最小哈希值构成
 - 每 b 个最小哈希值构成一个签名，每个集合由 r 个签名构成
 - 在每行条中，如果两个集合的签名相同，它们将被哈希到同一个桶中
 - 在 r 个行条中，两个集合至少一次被哈希到同一个桶中被认为是相似的
- 重要观察
 - 固定相似度（小于 1），两个集合被哈希到同一桶的概率随着 b 值的增加而减少
 - 随着行条数 r 的增加，两个集合被哈希到同一桶的概率会增加

Locality Sensitive Hashing (LSH)



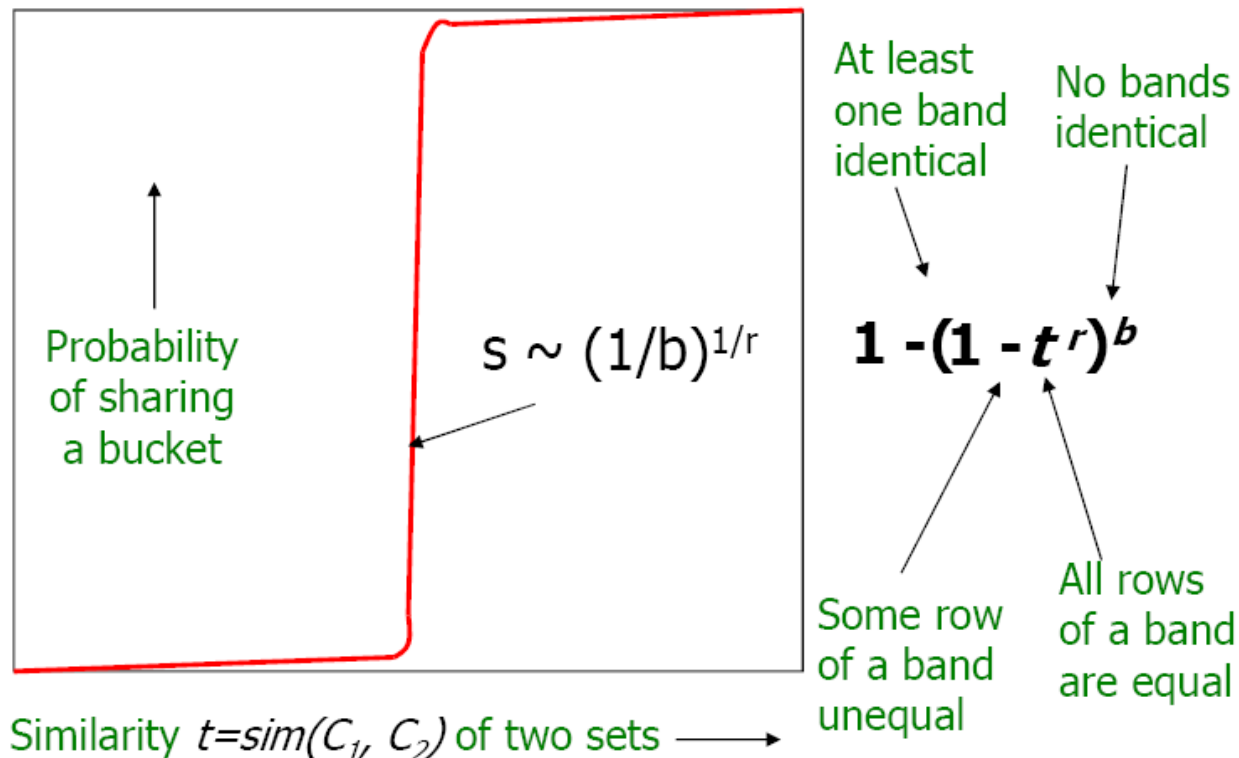
Jaccard 相似度为 0.8

- 令集合 C_1 和 C_2 的相似度为 0.8, $b = 20$, $r = 5$
 - C_1, C_2 的签名相同概率为 $0.8^5 = 0.328$
 - C_1, C_2 的 20 个行条都不相同的概率为 $(1 - 0.328)^{20} \approx 0.00035$
- C_1, C_2 为候选相似集合, 即两个集合被哈希到至少一个公共 bucket 中, 因此发生概率为 $1 - (1 - 0.328)^{20} \approx 99.965\%$
- 计算结果表明
 - 相似集合以大概率被哈希到同一个桶中
 - 相似度越高越可能被哈希到同一个桶中

Jaccard 相似度为 0.3

- 令集合 C_1 和 C_2 的相似度为 0.3, $b = 20$, $r = 5$
 - C_1, C_2 的签名相同, 其概率为 $0.3^5 = 0.00243$
 - C_1, C_2 的 20 个行条都不相同的概率为 $(1 - 0.00243)^{20} \approx 0.9526$
- C_1, C_2 为候选相似集合, 即两个集合被哈希到至少一个公共 bucket 中, 因此发生概率为 $1 - (1 - 0.00243)^{20} \approx 4.74\%$
- 计算结果表明
 - 不相似集合以小概率被哈希到同一个桶中
 - 相似度越低越不可能被哈希到同一个桶中

LSH 分析

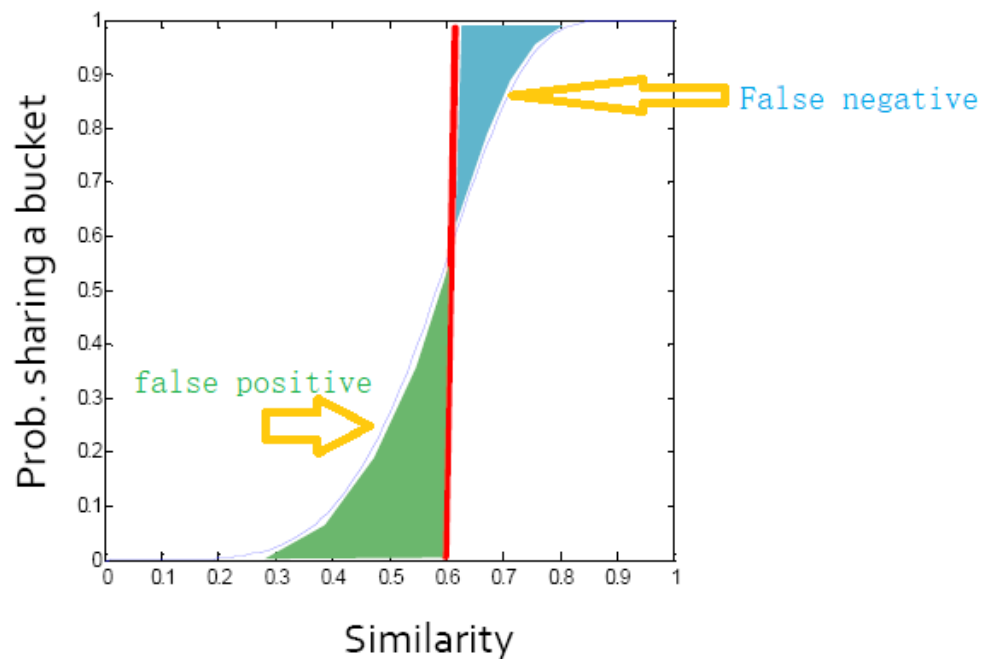


- 设两个集合相似度为 t
 - 某个签名相同的概率为 t^r
 - 没有一次被哈希到同一个桶中的概率为 $(1 - t^r)^b$
 - 至少一次被哈希到同一个桶中的概率为 $1 - (1 - t^r)^b$

LSH 分析 (续)

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

50 hash-functions ($r=5, b=10$)



- 通过适当选择 b 和 r 得到最好的 s 曲线
- 因此, LSH 能够实现文本分块
 - 落在同一桶中的文本大概率是相同的
 - 不在同一桶中的文本大概率是不相同的

本章小结

- 哈希算法
 - 哈希函数
 - 布隆过滤器
 - 局部敏感哈希 (LSH)
 - ✓ Shingling, 最小哈希, 局部敏感哈希
- 针对如下相似度或者距离, 可以利用局部敏感哈希解决冗余检测和最近邻搜索的问题
 - Jaccard 相似度
 - Hamming 距离
 - Cosine 距离
 - 欧几里得距离