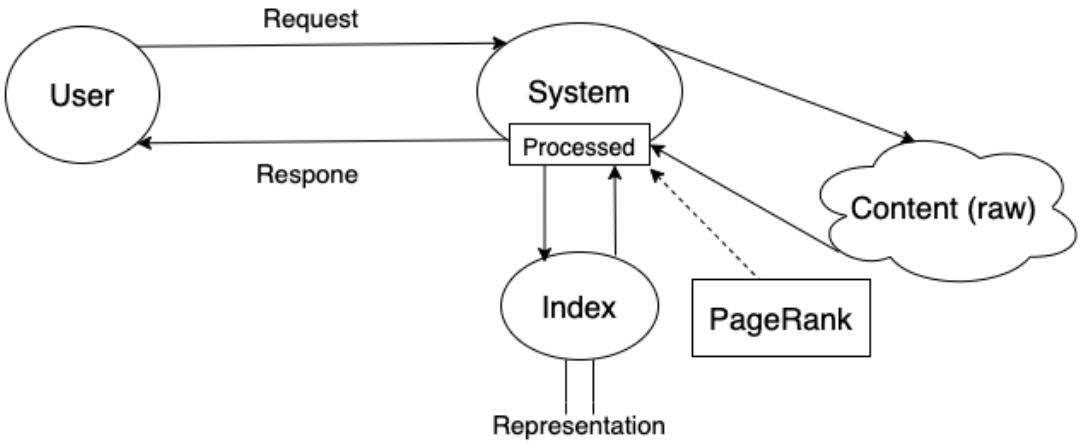
1. Introduction

1.1. Information Retrieval and Web Search Engine:

1.1.1. 定义：Examines key aspects of information retrieval as they apply to search engines; web crawling, indexing, querying and quality of results

1.2. Whole process overview



1.2.1. 文字描述：整个图包括两个流程：

1.2.1.1. user向system请求某些query, system根据index获得对应的raw content然后返回给user. query-> index -> raw content的过程称为processed; index又称为representation.

1.2.1.2. System定期进行爬虫取得raw content, 每一个raw content对应一个index存起来, 这个过程可能是insert, delete或者update.

1.2.1.3. 1.2.1.1和1.2.1.2是两个分别的流程, 互相之间独立运行.

1.2.1.4. Processed的目的: 加速用户搜索的速度

2. Search Engine Basics

2.1. Search Engine Elements

2.1.1. ***Spider*** (a.k.a. **crawler/robot**) - **builds corpus(维护语料库)**: Collects web pages recursively

2.1.2. **The *indexer* – creates inverted indexes**

2.1.3. ***Query processor* – serves query results**: Front end and Back end

2.2. Query Processing

2.2.1. Semantic analysis of the query includes

2.2.1.1. Determining the language of the query (确定语言)

2.2.1.2. Filtering of unnecessary words from the query (stop words) (把语气词去掉)

2.2.1.3. Looking for specific types of queries, e.g. (根据query中的词分类)

- Personalities (triggered on names)
- Cities (travel info, maps)
- Medical info (triggered on names and/or results)
- Stock quotes, news (triggered on stock symbol)
- Company info …

2.2.1.4. Determining the user's location or the target location of the query (用户地点)

2.2.1.5. Remembering previous queries (用户历史偏好)

2.2.1.6. Maintaining a user profile (将这次搜索纳入用户文档)

3. Crawlers and Crawling

3.1. 定义: A web crawler is a computer program that visits web pages in an organized way

3.2. Web Crawling Issues

3.2.1. **How to crawl?**

3.2.1.1. **Quality**: how to find the "Best" pages first (一般用BFS而不是DFS)

3.2.1.2. **Efficiency**: how to **avoid duplication**

3.2.1.3. **Etiquette**: behave politely by not disturbing a website's performance

3.2.2. **How much to crawl? How much to index? (crawl是要花钱的)**

3.2.2.1. **Coverage:** What percentage of the web should be covered?

3.2.2.2. **Relative Coverage:** How much do competitors have?

3.2.3. **How often to crawl? (crawl是要花钱的)**

3.2.3.1. **Freshness:** How much has changed?

3.2.3.2. **How much has really changed?**

3.3. Simplest Crawler Operation (crawling流程)

3.3.1. **Initialize** (begin with known "seed" pages) (**source code**)

3.3.2. **Loop**: Fetch and parse a page (用**queue**或者**stack**维护)

3.3.2.1. Place the page in a database

3.3.2.2. Extract the URLs within the page

3.3.2.3. Place the extracted URLs on a queue

3.3.2.4. Fetch a URL on the queue and repeat

3.4. Crawling的特点和面临的挑战

3.4.1. Crawling的过程可以distributed (可以处理大数据)

3.4.2. Challenges

3.4.2.1. Handling/Avoiding malicious pages

- Some pages contain spam
- Some pages contain spider traps – especially dynamically generated pages

3.4.2.2. Even non-malicious pages pose challenges

- Latency/bandwidth to remote servers can vary widely
- Robots.txt stipulations can prevent web pages from being visited
- How can one avoid mirrored sites and duplicate pages

3.4.2.3. Maintain politeness – don't hit a server too often

3.4.2.4. Robots.txt: The website announces its request on what can(not) be crawled by placing a robots.txt file in the root directory

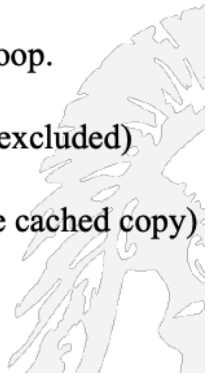
★ **Example: Disallow**中包含的路径都是爬虫不能访问的

```
# robots.txt for http://www.example.com/

User-agent: *
Disallow: /cyberworld/map/ # This is an infinite virtual URL space
Disallow: /tmp/ # these will soon disappear
Disallow: /foo.html
```

3.5. Crawling Algorithm

Initialize queue (Q) with initial set of known URL's.
Loop until Q empty or page or time limit exhausted:
    Pop a URL, call it L, from the front of Q.
    If L is not an HTML page (e.g. .gif, .jpeg, ….)
        continue the loop
    If L has already been visited, continue the loop.
    Download page, P, for L
    If cannot download P (e.g. 404 error, robot excluded)
        continue loop
    Index P (e.g. add to inverted index and store cached copy)
    Parse P to obtain list of new links N.
    Append N to the end of Q
End loop



3.5.1. BFS (FIFO先进先出): 上面的例子就是用的BFS, 用的数据结构是queue

3.5.2. DFS (LIFO后进先出): 将上面例子的数据结构改为stack就是DFS

3.5.3. **Heuristically ordering (**启发式排序**):** 根据一些特定的条件对下一个爬的网页进行排序(e.g. A document that changes frequently could be moved forward)

3.6. Avoiding Page Duplication

3.6.1. To determine if a **URL** has already been seen:

3.6.1.1. Must store URLs in a standard format (discussed ahead)

3.6.1.2. Must develop a fast way to check if a URL has already been seen

3.6.2. To determine if a **new page** has already been seen,

3.6.2.1. Must develop a fast way to determine if an *identical* page was already indexed

3.6.2.2. Must develop a fast way to determine if a *near-identical* page was already indexed

3.7. Representing URLs

3.7.1. 问题: URL太多太长了, 直接存太耗费空间.

3.7.2. 方法1: **To determine if a new URL has already been seen**

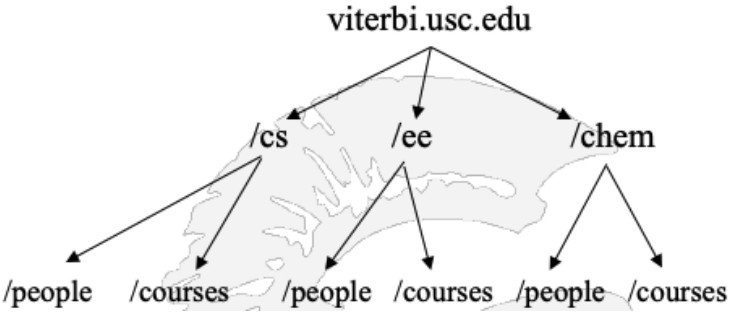
3.7.2.1. hash on host/domain name

3.7.2.2. Use a **trie data structure (**字典树**)** to determine if the path/resource is the same as one in the URL database

3.7.2.3. 最直接的比较两个URL是否相同的方式: 逐个字符比较, 复杂度**O(nk)**, n是URL个数, k是URL最大长度

3.7.2.4. 用了trie之后的复杂度: **O(k)**

★ **Example of tire:**



3.7.3. 方法2: **URLs are sorted lexicographically and then stored as a delta-encoded text file**

3.7.3.1. Each **entry is stored as the difference (delta) between the current and previous URL**; this substantially reduces storage

3.7.3.2. However, **restoring the actual URL is slower**, requiring all deltas to be applied to the initial URL

3.7.3.3. To improve speed, **checkpointing** (storing the full URL) is done periodically

3.8. Normalizing URLs

3.8.1. **Why Normalizing URLs is Important?** 很多相似的link指向同一个page但是一旦他们有一点点不同他们的hash值就会不同(e.g. http://www.google.com; http://www.google.com/; https://www.google.com)

3.8.2. **4 rules of Normalizing URLs**

3.8.2.1. **Convert the scheme and host to lower case.** The scheme and host components of the URL are case-insensitive.

★ **Example: HTTP**://www.**E**xample.com/ → **http**://www.**e**xample.com/

3.8.2.2. **Capitalize letters in escape sequences**. All letters within a percent-encoding triplet (e.g., "%3A") are case-insensitive, and should be capitalized.

★ **Example**: http://www.example.com/a**%c**2**%b**1b → http://www.example.com/a**%C**2**%B**1b

3.8.2.3. **Decode percent-encoded octets of unreserved characters.**

★ **Example:** http://www.example.com/**%7E**username/ → http://www.example.com/**~**username/

3.8.2.4. **Remove the default port.** The default port (port 80 for the "http" scheme) may be removed from (or added to) a URL.

★ **Example**: http://www.example.com**:80**/bar.html → http://www.example.com/bar.html

3.9. Avoiding Spider Traps

3.9.1. **Spider Traps定义: A spider trap is when a crawler re-visits the same page over and over again**

3.9.2. 最常见的**Spider Trap:** The most well-known spider trap is the one created by the use of Session ID's. A Session ID is often used to keep track of visitors, and some sites puts a unique ID in the URL (简单来说就是每个用户访问网页时会有一个ID, 有时候这个ID成为URL的一部分, 所以如果每次只改这一部分就会让爬虫一直重复在访问一个网页)

★ **Example:** www.webmasterworld.com/page.php?**id=264684413484654**

3.9.3. 解决方法**:**

3.9.3.1. For the crawler to be careful when the querystring "ID=" is present in the URL

3.9.3.2. Monitor the length of the URL and stop if the length gets "too long"

3.10. Handling Spam Web Pages

3.10.1. **Web Spam**的发展**:**

3.10.1.1. The first generation of spam web pages consisted of pages with a **high number of repeated terms**, so as to score high on search engines that ranked by word frequency

3.10.1.2. The second generation of spam used a technique called **cloaking**: When the web server **detects a request from a crawler, it returns a different page** than the page it returns from a user request. **The page is mistakenly indexed**.

3.10.1.3. A third generation, called a **doorway page**, contains **text and metadata chosen to rank highly on certain search keywords**, but when a browser requests the doorway page it instead gets a **more "commercially oriented" (more ads) page** (国内经典数字门户网站, 谁给的钱多推谁的网页)

3.11. Distributed Crawling

3.11.1. **Multi-Threaded Crawling: One bottleneck** is network delay in downloading individual pages. It is best to have **multiple threads** running in parallel each requesting a page from a different host. (在一台机器上多线程爬虫)

3.11.2. **Distributed Crawling Approaches:**

3.11.2.1. A **centralized crawler** controling a set of parallel crawlers all running on a LAN

3.11.2.2. A **distributed set of crawlers** running on widely distributed machines, with or without cross communication (MapReduce)

3.11.3. **If crawlers are running in diverse geographic locations, how do we organize them?**
→ Distributed crawlers must **periodically update** a master index (But incremental update is generally **"cheap"** because you need only send a **differential update**)

3.11.4. 优点**:**

3.11.4.1. **scalability:** for large-scale web-crawls

3.11.4.2. **costs:** use of cheaper machines

3.11.4.3. **network-load dispersion and reduction:** by dividing the web into regions and crawling only the nearest pages

3.11.5. 缺点**:**

3.11.5.1. **overlap:** minimization of multiple downloaded pages

3.11.5.2. **quality:** depends on the crawling strategy

3.11.5.3. **communication bandwidth:** minimization

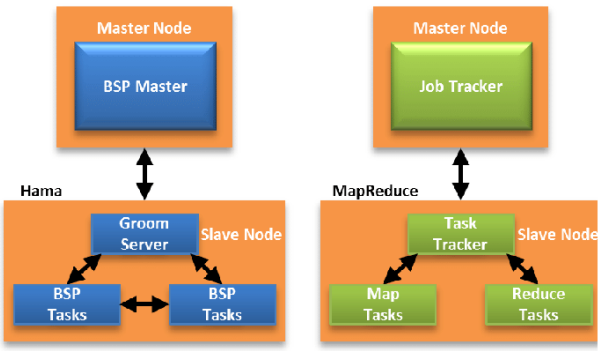
3.11.6. **Distributed Crawling**三种策略**:**

3.11.6.1. **Independent:** no coordination, every process follows its extracted links (所有分不开的服务器单独爬虫, 合并的时候需要**deduplication, Very Fast**)

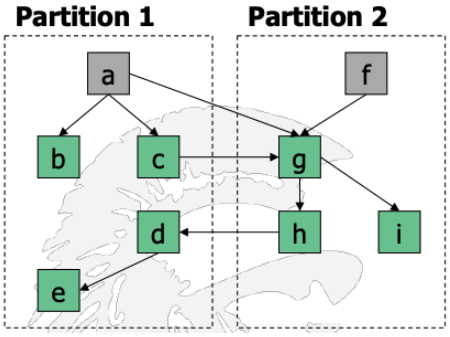
3.11.6.2. **Dynamic assignment:** a central coordinator dynamically divides the web into small partitions and assigns each partition to a process (一个主机来动态协调每个节点应该负责哪些link, 能避免**duplication**, 问题是主机可能**fail**)

3.11.6.3. **Static assignment:** Web is partitioned and assigned without a central coordinator before the crawl starts (在爬虫之前就好固定的访问link名单, 不需要动态协调所以不需要主机)

★ **Note:** 没有主机 **Static assignment** 如何交换不同 **partition**之间的 **links呢?** → BSP机制. 和MapReduce不同的是BSP中处理节点之间可以互相通信. (上课老师讲的例子就是罪犯逃到不同的地区时不同地区之间的警局会互相沟通交接)



★ **Example:** 右图中如果Partition2要把ghi交还给Partition1, 就把ghi都移到左边去, 然后点开fg的连接. (注意这里是转移而不是复制,不然会造成**duplication**)



3.12. Keeping Spidered Pages Up to Date

3.12.1. **Periodically check** crawled pages for updates and deletions: Just look at LastModified indicator(在metadata中) (🌰: 假设我爬到一个网页的LastModified在五年前, 而它的index在我的数据库里对应的上一次爬虫是两年前, 那我这次就不需要update; 假设我爬到一个网页LastModified在五分钟前, 而index对应的上一次爬虫是五天前, 那我就需要update)

3.12.2. **Track how often each page is updated and preferentially return to pages which are historically more dynamic.** (有点像机器学习, 根据不同网页更新的频率来设置相应的爬虫更新频率)

3.12.3. **When a crawler replaces an old version by a new page, does it do it "in-place" or "shadowing"?** ("in-place"的意思是立马更新之前index对应的内容, "shadowing"的

意思是某个时间段内的这个link的所有更新都先存在一个temp index里, 到达某个时间后再覆盖掉真正的index的内容) → 这是一个**availability**和**consistency**的**tradeoff**. **"in-place"**可以保证**consistency**, 也就是用户每次查询的内容都一定是最新的, 但是速度会变慢因为后台一直要更新内容; 而**"shadowing"**可以保证**availability**, 因为在某个时间段用户查询的内容都是**old version**, 不需要修改所以速度很快, 但是**consistency**就不能保证了.

3.13. Conclusion

3.13.1. **Running multiple types of crawlers is best**

3.13.2. Updating in-place keeps the index current

4. Search Engine Evaluation

4.1. Precision, Recall, F1-score

		Predicted: Negative	Predicted: Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

4.1.1. **relevant**指的是真实相关**(actual true), retrieved**是认为相关**(predicted true)**

4.1.2. **F1-score(F-measure)**是**Precision**和**Recall**的**harmonic mean**

$$\text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Harmonic Mean Formula} = \frac{n}{\left(\frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3} \cdots\cdots \frac{1}{x_n}\right)}$$

4.1.3. **You can get high recall (but low precision) by retrieving all docs for all queries!** (容易理解, 如果我每次都把所有**docs**推给用户里面一定有他想看的, 但是没有意义, **precision**很低)

4.1.4. **In a good system, precision decreases as the number of docs retrieved (or recall) increases** (通过事实总结, 没有理论依据)

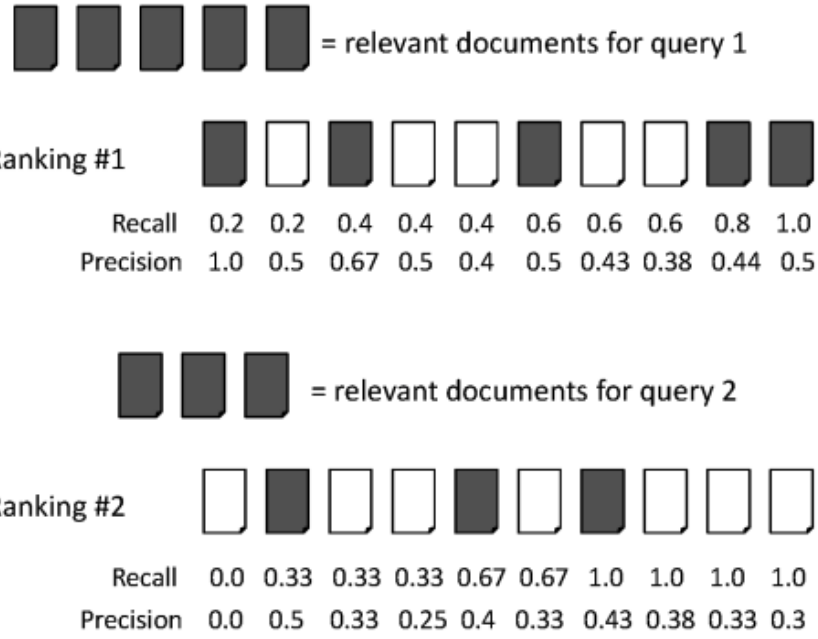
$$mAP = \frac{1}{n}\sum_{k=1}^{k=n} AP_k$$

$$AP_k = \text{the AP of class } k$$

$$n = \text{the number of classes}$$

4.2. Mean average precision

★ **Example:**



$$average\ precision\ query\ 1 = (1.0 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62$$
$$average\ precision\ query\ 2 = (0.5 + 0.4 + 0.43)/3 = 0.44$$

$$mean\ average\ precision = (0.62 + 0.44)/2 = 0.53$$

4.2.1. **mAP的缺点:**

4.2.1.1. **Each query counts equally**

4.2.1.2. If a relevant document never gets retrieved, we assume the precision corresponding to that relevant doc to be zero (this is actually reasonable)

4.2.1.3. mAP assumes user is interested in finding many relevant docs for each query

4.2.1.4. mAP requires many relevance judgments in the document collection

4.3. Discounted Cumulative Gain (DCG)

4.3.1. 定义: **Highly relevant documents appearing lower in a search result list should be penalized** as the graded relevance value is reduced logarithmically proportional to the position of the result

4.3.2. 公式: $DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i}-1}{log_2(i+1)}$, 其中p为一个结果列表的排序位置, rel_i 代表第i个位置上文档的相关度 (i通常情况下是rank)

★ **Example:** 假设搜索到6个结果, 其相关性分数分别是3,2,3,0,1,2. 求DCG

i	rel_i	log2(i+1)	rel_i / log2(i+1)
1	3	1	3
2	2	1.58	1.26
3	3	2	1.5
4	0	2.32	0
5	1	2.58	0.38
6	2	2.8	0.71

$DCG = 3 + 1.26 + 1.5 + 0 + 0.38 + 0.71 = 6.86$

4.4. Normalized Discounted Cumulative Gain (nDCG∈[0,1])

4.4.1. 首先对语料库中所有相关文档的相关性排序, 再通过位置生成最大可能的 DCG, 称为IDCG. 对于一个 Query, nDCG的公式$nDCG = \frac{DCG}{IDCG}$

★ **Example:** 沿用上面那个🌰, 假设我们实际召回了8个文档, 除了上面的6个, 还有两个结果. 假设第7个相关性为3, 第8个相关性为0. 那么在理想情况下的相关性分数排序应该是: 3, 3, 3, 2, 2, 1, 0, 0. 计算IDCG

i	rel_i	log2(i+1)	rel_i / log2(i+1)
1	3	1	3
2	3	1.58	1.89
3	3	2	1.5
4	2	2.32	0.86
5	2	2.58	0.77
6	1	2.8	0.35

$IDCG@6 = 3 + 1.89 + 1.5 + 0.86 + 0.77 + 0.35 = 8.37$

$nDCG = \frac{DCG}{IDCG@6} = \frac{6.86}{8.37} = 0.819$

4.5. Search engines also use non-relevance-based measures

4.5.1. **Click-through on first result** (看你推给用户的link有没有被用户点进去)

4.5.2. **A/B testing**: comparing two versions of a web page to see which one performs better. You compare two web pages by showing the two variants (let's call them **A** and **B**) to similar visitors at the same time. The one that gives a better conversion rate, wins!

5. Deduplication

5.1. 定义: De-duplication essentially refers to the identification of identical and nearly identical web pages and indexing only a single version to return as a search result

5.2. **Mirroring:** 镜像网站. Mirroring is the **single largest cause** of duplication on the web.

定义: Host1/a and Host2/b are mirrors if and only if for all paths p(所有子网页), http://Host1/a/p 存在, http://Host2/b/p 也存在, 并且内容几乎相同.

5.3. Duplication problems分类:

5.3.1. **Duplicate Problem: Exact match(**完全匹配**):**

5.3.1.1. 解决方案: compute **fingerprints** using **cryptographic hashing**(e.g. MD5)

5.3.1.2. Useful for URL matching and also works for **detecting identical web pages**

5.3.1.3. Hashes can be stored on sorted order for **logN** access(二分搜索)

5.3.2. **Near-Duplicate Problem: Approximate match(**近似相同**)**

5.3.2.1. 解决方案: compute the **syntactic similarity** with an **edit-distance measure**, and use a similarity threshold to detect near-duplicates (e.g. 相似度>80%就认为近似相同)

5.4. 通过Shingling检测两个网页是否相同:

5.4.1. **Shingling**概念: k-shingle (k-grams)是指文档中连续出现的 k 个字符构成的序列

★ **Example:** k=2, document D_1=abcab, 2-shingles $S(D_1)$ = {ab, bc, ca}, 这里的$S(D_1)$是一个集合

5.4.2. 怎么判断**k**多少合适？→ 选择一个总排列数比**buckets**数量大但又不会大很多的**k**

★ **Example:** 假设我们有20个不同的字符, shingles的总排列个数就是20^k, 那么现在我们对比4-shingles好还是9-shingles好

- 4-shingles: 可能的排列个数为$20^4=2^{17.3}$, 9-shingles: 可能的排列个数为$20^9=2^{39}$

- 假设我们用的hash tablen(bucket)用int来存, 一个int的范围在0~2^{32}-1

★ 我们更希望用9-shingles因为如果我们用**4-shingles,** 它的所有可能都要比我的空间小的多**($2^{17.3}$<<2^{32})**, 我们会有很多剩余空间没被利用. 而**9-shingles**的所有可能要比2^{32}大, 但又不至于大很多, 这样我们完全利用了**int**的范围, 并且在把**2^{39}hash**成**2^{32}**的时候也不至于有很多重复以至于降低准确性

5.4.3. **Jaccard similarity:** $Sim(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$

5.4.4. **Jaccard distance:** $Jaccard\ Distance = 1 - Jaccard\ Similarity$

5.4.5. 判断duplication流程:

5.4.5.1. 对每个网页内容生成k-shingles, 每个网页维护成一个集合

5.4.5.2. 将每个集合中的每个shingle分别hash成hash values

5.4.5.3. 通过相同的某些条件筛选出每个集合中的一些hash values(aka. **fingerprints**). 这一步主要是为了牺牲部分准确度来换取计算时间和空间

5.4.5.4. 通过这些fingerprints计算jaccard similarity, 一个很高的jaccard similarity就暗示了这些网页就是duplicated ($J(fingerprint(A), fingerprint(B)) > k$, the pages are similar)

★ **Example:**

- **Original text**
  - "Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species"
- **All 3-shingles (there are 16 of them)**
  - (Tropical fish include), (fish include fish), (include fish found), (fish found in), (found in tropical), (in tropical environments), (tropical environments around), (environments around the), (around the world), (the world including), (world including both), (including both freshwater), (both freshwater and), (freshwater and salt), (and salt water), (salt water species)
- **Hash values for the 3-shingles (sets of shingles are large, so we hash them to make them more manageable, and we select a subset)**
  - 938, 664, 463, 822, 492, 798, 78, 969, 143, 236, 913, 908, 694, 553, 870, 779
- **Select only those hash values that are divisible by some number, e.g. here are selected hash values using *0 mod 4***
  - 664, 492, 236, 908; ***these are considered the fingerprints***
- **Near duplicates are found by comparing fingerprints and finding pairs with a high overlap**

5.5. 通过**SimHash**检测两个网页是否相同**:**

5.5.1. **SimHash**和普通**hash**的区别**: documents that are nearly identical have nearly similar fingerprints that differ only in a small # of bits.** In other words, similar inputs lead to similar outputs (hash values), hence 'Sim'Hash; other hashing techniques,

eg. MD5, do not have this property (in other words, even a tiny change in the input leads to a huge change in the output). **(**简单来说就是**SimHash**中差别越小的文本得到的**hash value**越相近**)**

★ **Example:**

- **A hash function usually hashes different values to totally different hash values; here is an example**

  p1 = 'the cat sat on the mat'
  p2 = 'the cat sat on a mat'
  p3 = 'we all scream for ice cream'
  p1.hash => 415542861
  p2.hash => 668720516
  p3.hash => 767429688

- **Simhash is one where similar items are hashed to similar hash values (by similar we mean the bitwise Hamming distance between hash values is small)**

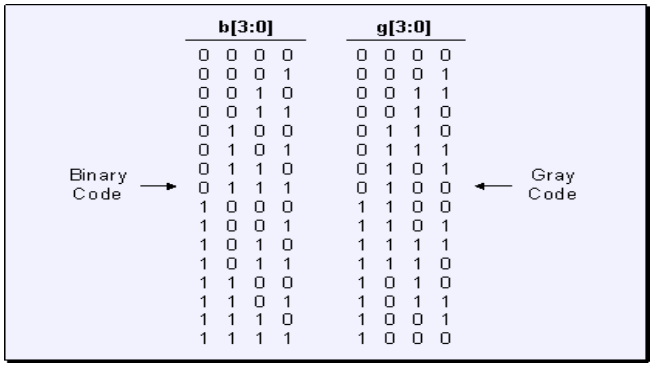
  p1.simhash => 851459198
  00110010110000000011110001111110
  p2.simhash => 847263864
  00110010100000000011100001111000
  p3.simhash => 984968088
  00111010101101010110101110011000

- **in this case we can see the hamming distance of the similar items (p1,p2)=4 is small whereas (p1,p3)=16 and (p2,p3)=12 are considerably larger**

5.5.2. 判断duplication流程:

5.5.2.1. comparing SimHash values is a great way to identify near-duplicates for 'n' documents, **comparing them all pairwise would take a long time [$O(n^2)$]**

5.5.2.2. as a shortcut, we can **sort their decimal representations and only compare adjacents** - this will identify similarities based on low-end bits; **but this will miss similarities based on the higher-end bits**; as an aside, we can look for one more possible low-bits near-duplicate by comparing the top-most and bottom-most values too, like in **Gray Code (Gray Code**的特点是前后两个二进制只会有一位是不同的**)**



5.5.2.3. To fix the problem of missing finding high order bit similarities, we can **rotate** all the docs' bits identically to the right (so that **the high order bits (left) become a 'bit' (lol) lower**) to produce 'new' hashes, sort *those*, compare for near-duplicates

5.5.2.4. we can progressively spin right by 1 bit, 2 bits, 3 bits... to discover more and more similarities [we will rediscover existing similarities but ignore those]

5.5.2.5. note that **we can rotate left as well**

5.5.2.6. doing the above is **$O(n) + O(n\log n) = O(n\log n) < O(n^2)$**

★ **Example:**

- **consider the eight numbers and their bit representations**
- **1 37586 1001001011010010**
- **2 50086 1100001110100110 7 <--(this column lists hamming**
- **3 2648 0000101001011000 11 distance to previous entry)**
- **4 934 0000001110100110 9**
- **5 40957 1001111111111101 9**
- **6 2650 0000101001011010 9**
- **7 64475 1111101111011011 7**
- **8 40955 1001111111111011 4**

if we sort them

4 934 0000001110100110
3 2648 0000101001011000 9
6 2650 0000101001011010 1
1 37586 1001001011010010 5
8 40955 1001111111111011 6
5 40957 1001111111111101 2
2 50086 1100001110100110 9
7 64475 1111101111011011 9

notice that two pairs with very smallest hamming distance hdist(3,6)=1 and hdist(8,5)=2 have ended up adjacent to each other.

- **A problem:**
  - **there is another pair with a low Hamming distance, *hdist(4,2)=2* that have ended up totally apart at other ends of the list...**
  - **sorting only picked up the pairs that differed in their lower order bits.**

'rotate' bits left twice

4 3736 0000111010011000
3 10592 0010100101100000 9
6 10600 0010100101101000 1
1 19274 0100101101001010 5
8 32750 0111111111101110 6
5 32758 0111111111110110 2
2 3739 0000111010011011 9
7 61295 1110111101101111 9

if we sort again by fingerprint

4 3736 0000111010011000
2 3739 0000111010011011 2
3 10592 0010100101100000 11
6 10600 0010100101101000 1
1 19274 0100101101001010 5
8 32750 0111111111101110 6
5 32758 0111111111110110 2
7 61295 1110111101101111 6

this time the (2,4) pair ended up adjacent
we also identified the (3,6) and (5,8) pairs as candidates again

6. Spearman Correlation (SRC)

6.1. 公式: $\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$, 其中**$d_i$**表示**difference between the two ranks of each observation; n**表示**number of observations (**在统计学领域正常情况下**Spearman Correlation**的值在**[-1,1])**

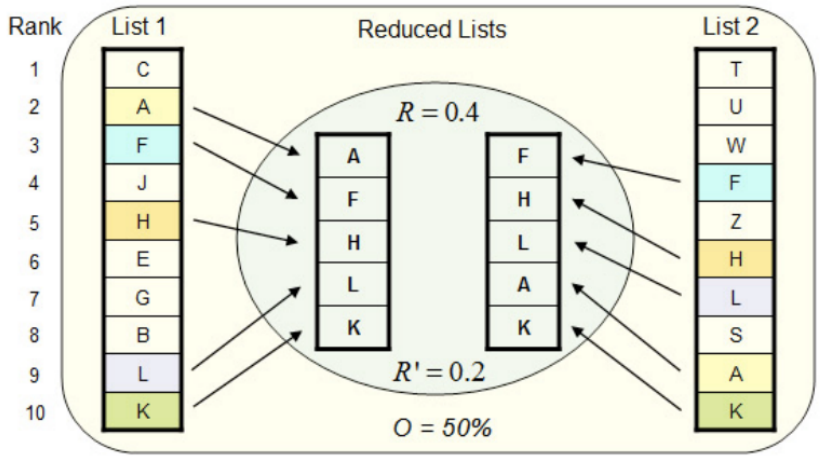
6.2. Modified Spearman Correlation:

6.2.1. 问题: 普通的spearman correlation假定了比较的内容都是相容的只能存在rank排序的不同(overlap=100%), 但是实际情况下overlap可能不是100% (e.g. A = [o,m,a,b,c,f,g], B = [s,t,b,c,a,f,u]), 这时算出来的**spearman correlation**就不一定在**[-1,1]**了

6.2.2. 解决方法: 改公式为$\rho' = \rho * overlap\%$, 这样就可以保证ρ'在[-1,1]

★ 这里**saty**课上好像讲错了**?** 用了这个**modified SRC**在**search engine**的结果上计算也不能保证一定在**[-1,1].** 而且下面这个例子计算**SRC**的时候**rank**重新从**12345**排了**,** 而**hw1**中是没有**rank**重新排序的**.**

★ **Example:**



6.3. Spearman Correlation in search engine: 公式和普通的**spearman correlation**相同但由于有**overlap**所以值不一定在**[-1,1]**

★ **Note:** 这里在计算**spearman correlation**时用到的**rank**都是在原始**list**中的**rank,** 而不是上面那个🌰中的取出**overlap list**之后的新**rank**

★ **Example:** 假设我们从两个不同的搜索引擎用同一个query爬到两列结果

Google	Bing
a	**a**
b	**f**
c	z
d	y
e	x
f	**g**
g	w
h	v
i	**e**
j	u

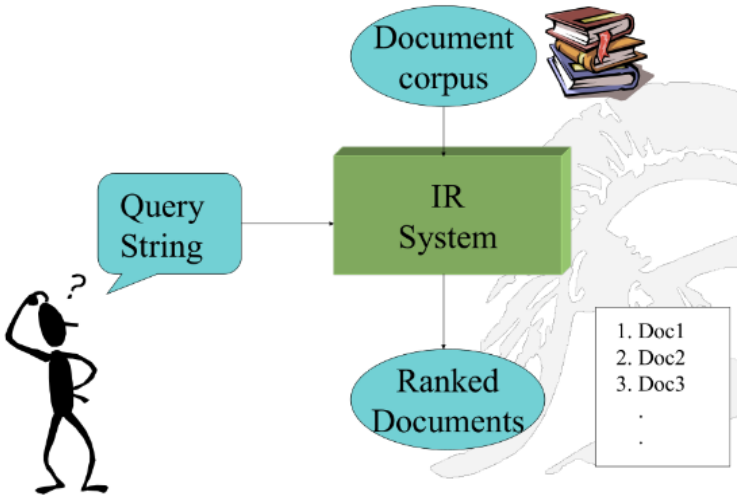
很明显这里overlap是4/10, 对应的rank是{1-1, 5-9, 6-2, 7-6},因此得到的d_i分别为{0,-4,4,1}, 得到

$\rho = 1 - \frac{6*(0+16+16+1)}{4*15} = -2.30$ (这里用到的**rank**都是原始结果中的**rank**)

7. Information Retrieval

7.1. 目的: 读入用户的**queries,** 根据已有的数据库推荐给用户他们想要看的内容

(要求在数据库里的存储格式是{word: [doc1,doc2 … dock]} (inverted index))



7.2. IR system的特点

7.2.1. A retrieval model specifies the details of:

7.2.1.1. Document representation (**vector of terms**)

7.2.1.2. Query representation (**vector of terms**)

7.2.1.3. Retrieval function (**TF-IDF** + **Similarity**)

7.2.2. Determines a notion of relevance (6.2.1.3)

7.2.2.1. Notion can be binary or continuous (i.e. ranked retrieval)

7.2.3. Three major Information Retrieval Models are:

7.2.3.1. Boolean models (set)

7.2.3.2. **Vector space models**

7.2.3.3. Probabilistic models

7.3. Document & Query representation

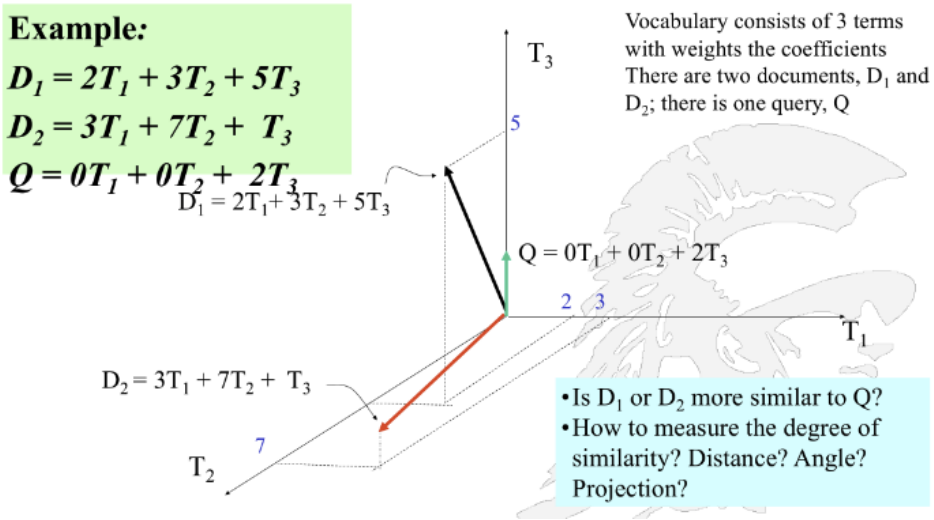
7.3.1. Documents和Queries都被表示成一堆keywords(aka terms)的集合

7.3.2. 语料库(vocabulary)是一个所有已知term的集合, 称为V

7.3.3. 假设我们用vector表示每个document和query

7.3.3.1. Size of V = Dimension

7.3.3.2. Document $D_i = (d_{i1}, d_{i2}, \ldots, d_{it})$, 其中$d_{ij}$表示第j个term在文档i中的权重(weight)是多少



★ 如何计算这个**weight**呢**?** → **TF-IDF**

7.4. TF-IDF

7.4.1. TF-IDF定义: Measure of Word Importance. **Item profile for a document** = set of words with **highest** TF-IDF scores, together with their scores.

7.4.1.1. Term Frequency: $TF_{i,j} = \frac{f_{i,j}}{max_k f_{k,j}} \leq 1$, 进行了一次**normalized**

$f_{i,j}$ = frequency of term (feature) i in document (item) j

$max_k f_{k,j}$ = maximum occurrences of any term in document j

7.4.1.2. Inverse Document Frequency: $IDF_i = log_2(\frac{N}{n_i})$

n_i = number of docs that mention term i

N = total number of docs

7.4.2. TF-IDF score: $w_{i,j} = TF_{i,j} \times IDF_i$

★ **Example:** 我们有2^{20}个document, term w出现在2^{10}个ducument里。

1. 假设w在document j里出现的次数是所有term里最多的。

   A: $TF_{w,j} = 1, IDF_j = log_2 \frac{2^{20}}{2^{10}} = 10,\ w_{w,j} = 1 \times 10 = 10$

2. 假设w在document i里出现的次数为1, 且文档里出现最多的term出现了20次。

   A: $TF_{w,i} = \frac{1}{20}, IDF_j = log_2 \frac{2^{20}}{2^{10}} = 10,\ w_{w,j} = \frac{1}{20} \times 10 = \frac{1}{2}$

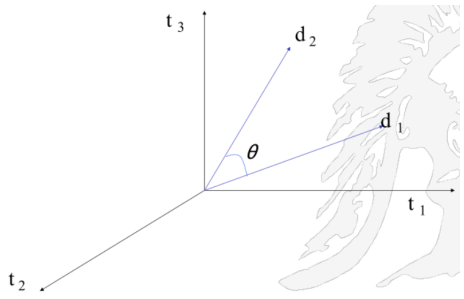
7.4.3. 给定一个query, 它和一个document的分数$Score(q,d) = \sum(TF_{t,d} \times IDF_t)$, 其中 $t = Set(q) \cap Set(d)$

★ **Note:** in some cases we **normalize** each tf•idf value using the **L2** (sqrt of sum of squares), as opposed to **L1** ((absolute)sum) norm

★ 有了**TF-IDF,** 如何知道哪些**document**更应该推荐给用户呢**?** → **Cosine Similarity**

7.5. Cosine Similarity

7.5.1. 用Cosine Similarity的原因: 我们用vector表示query或者document, 它们之间的相似度就应该是两个向量之间的距离, 和向量的长短无关只和它们的方向有关 (**degree of similarity**)



7.5.2. 朴素版本similarity: q和d_j之间的相似度就是他们的内积.

7.5.2.1. 公式$sim(d_j, q) = d_j \bullet q = \sum_{i=1}^{t} w_{ij} \bullet w_{iq}$, 其中$w_{ij}$表示term i和document j的权重, 而w_{iq}为term i和query之间的权重

7.5.2.2. 对于boolean vectors, 内积就表示哪些query中的terms在document中也出现了(aka size of intersection/Hamming distance)

7.5.2.3. 对于weighted term vectors, 内积表示所有共同出现的terms的加权乘积

★ **Example:**

Binary:

	retrieval	database	architecture	computer	text	management	information
D	1	1	1	0	1	1	0
Q	1	0	1	0	0	1	1

Size of vector = size of vocabulary = 7
0 means corresponding term not found in document or query

similarity(D, Q) = 3 **(the inner product)**

Weighted:

$D_1 = 2T_1 + 3T_2 + 5T_3$ $D_2 = 3T_1 + 7T_2 + 1T_3$
$Q = 0T_1 + 0T_2 + 2T_3$

$sim(D_1, Q) = 2*0 + 3*0 + 5*2 = 10$
$sim(D_2, Q) = 3*0 + 7*0 + 1*2 = 2$

7.5.2.4. 存在的问题: 文档的长度**(vector**的长度**)**会影响**similarity**(和7.5.1的要求不符)

7.5.3. 使用Cosine Similarity(相当于做了一个normalization):

7.5.3.1. **公式**: $similarity = cos(\theta) = \frac{A \cdot B}{||A|| ||B||} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$

★ **Example1:**

$D_1 = 2T_1 + 3T_2 + 5T_3$ $CosSim(D_1, Q) = 10 / \sqrt{(4+9+25)(0+0+4)} = 0.81$
$D_2 = 3T_1 + 7T_2 + 1T_3$ $CosSim(D_2, Q) = 2 / \sqrt{(9+49+1)(0+0+4)} = 0.13$
$Q = 0T_1 + 0T_2 + 2T_3$

D_1 is 6 times better than D_2 using cosine similarity but only 5 times better using inner product.

★ **Example2:** A=[1,2,-1], B=[2,1,1], $similarity = cos(A, B) = \frac{1\times2+2\times1-1\times1}{\sqrt{1+4+1}\times\sqrt{4+1+1}} = \frac{1}{2}$

7.6. 完整TF-IDF + Cosine Similarity流程:

7.6.1. 将文档集合D中的所有文档都通过TF-IDF表示成vector, d_j表示第j个文档, 维护一个语料库V

7.6.2. 将每个query q都通过TF-IDF表示成vector

7.6.3. 对每个d_j, 计算q和d_j之间的score, $s_j = cosSim(d_j, q)$

7.6.4. 根据score从大到小对文档进行排序

7.6.5. 将top k个文档推荐给用户

★ **Note:** 时间复杂度**O(|V|·|D|)**, 当|V|和|D|很大的时候很慢! 如何加速**?** → **preprocessing** (对于每一个term, 提前算出它们相似度最高的若干个文档(把每个**term**当做一个长度为**1**的**query**), 存成一个preferred list → 对于一个t-term query, 取这t个term的preferred list的交集或者并集, 得到一个新的集合S → 将**S**当作新的文档集合去跑**7.6**)

7.7. TF-IDF + Cosine Similarity的缺点:

7.7.1. Missing semantic information (无法理解语义)

7.7.2. Missing syntactic information (无法理解语法 e.g. phrase structure, word order, proximity information)

7.7.3. **Assumption of term independence** (总结上面两点, 就是对待每个单词是独立的)

7.7.4. Lacks the control of a Boolean model (e.g. 搜索一个2-term query "A B", 用户可能要求A一定要频繁出现但是B没那么重要, 可以不出现在结果文档里; 但是在我们的算法中会平等地对待A, B所以会推荐A, B都出现但是都不频繁的文档)

8. Text Processing (Classification)

8.1. 模型:

- **Given:**
  - **A representation of a document *d***
    - **Issue: how to represent text documents.**
    - **Usually some type of high-dimensional space – bag of words**
  - **A fixed set of classes:**
    $C = \{c_1, c_2, \ldots, c_J\}$
- **Determine:**
  - **The category of *d* by generating a classification function, say *γ(d)***
  - **We want to build classification functions ("classifiers").**

8.1.1. 意义: 简单来说就是给定一个文档的向量, 将这个文档进行分类, 目的是方便推荐给有特定方向需求的用户

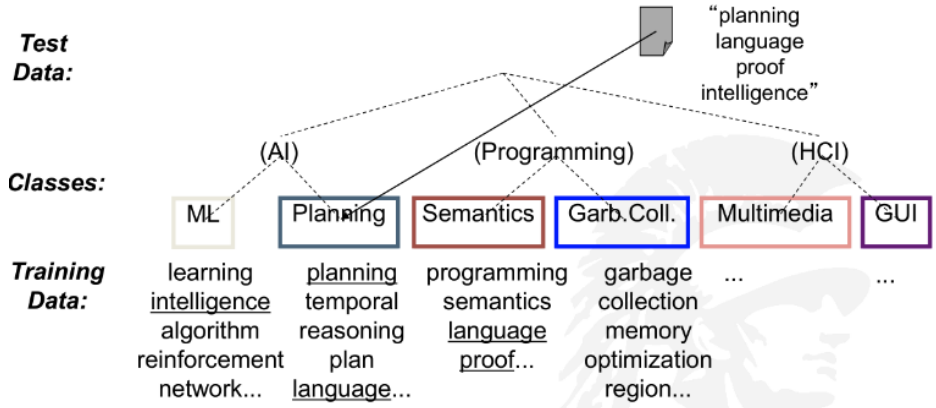
8.1.2. 🌰:

8.1.2.1. Standing Queries: 一直监视某些query的结果, 一旦有新的文档或者文档更新

就推送给用户 (本质上是监视某一个或多个类别)

8.1.2.2. Spam Filtering: 根据某些文本特性筛选出被归类为spam的email (事实上某一个email都可以根据用户需求被分类为某一类email自动进入某个folder)

★ **Example:**



8.2. Classification Methods

8.2.1. Manual classification: 人为的给每个URL分类, 像是在维护一个图书馆. 需要experts才能很好地完成分类, 在数据规模很大的时候无法实现

★ **Note:** 这个方法最早由**Yahoo!**提出, saty在上课时对比了Yahoo!和Google的方法. Google的方法就是index法, 不需要分类, 在数据规模大的时候更有优势

8.2.2. Hand-coded rule-based classifiers:

8.2.2.1. 定义: 人为的制定某些规则进行分类 (e.g.如果你发的邮件符合某些特定的pattern, 你的邮件就会被NSA捕捉), 多用于news, government和enterprises.

8.2.2.2. 特点: 如果规则制定的好准确度就会很高, 但是随着时间规则可能要一直改变, 制定和维护好的规则是很expensive的 (1980年的规则用在现在可能很差)

8.2.3. **Supervised learning**:

8.2.3.1. 逻辑模型:

- **Given:**
  - **A document *d***
  - **A fixed set of classes:**
    $C = \{c_1, c_2, \ldots, c_J\}$
  - **A training set *D* of documents each with a label in *C***
- **Determine:**
  - **A learning method or algorithm which will enable us to learn a classifier *γ***
  - **For a test document *d*, we assign it the class**
    $\gamma(d) \in C$

8.2.3.2. Naive Bayes: 假定每个类别之间没有关联性 (比如一个水果具有weight, shape, smell等属性, 但它们之间我们假定是互相没有联系的, 也就是水果的shape不会影响它的smell)

8.2.3.3. K-Nearest Neighbors (KNN): 把所有objects作为向量, 通过最近的**K**个邻居的类别投票出自己的类别 (**K**绝大多数情况下是奇数以保证不会出现平局)

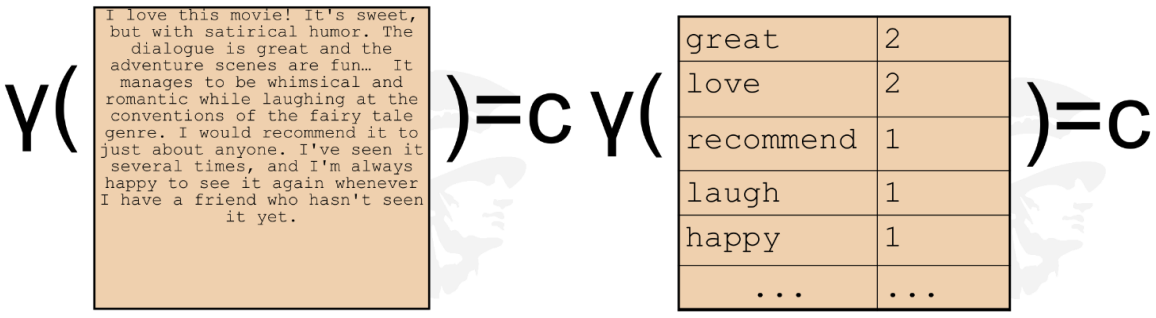
8.2.3.4. Support-vector machines(SVM): 把所有objects作为坐标点, 根据已知点的类别构造出一个**barrier**来区分所有点, 目标是尽可能把属于同一类别的点都分在同一边 (barrier可以是直线, 曲线, 超平面等, 取决于类别数量和feature数量)

★ **Note:** 上课的时候saty说KNN和SVM都是解决二分类问题的, 但其实他们都可以解决多分类问题, 可能只是他简化了 (都以spam email分类为例)

8.2.4. Many commercial systems use a **mixture** of methods **(ensemble learning)**: 先用多种模型跑同一个object看得到的分类, 然后根据结果结合experts给出的分类最后决定这个object属于哪一类

8.2.5. Feature特点: Supervised learning classifiers **<u>can use any sort of feature</u>** (并不局限于words. URL, email address, network features都可以作为feature)

8.3. Naive Classification representation



8.3.1. 过程: 得到一段话以后统计每个word出现的次数, 每个word作为一个feature给模型得到一个class

8.3.2. 特点: 只用word作为feature并且每个word都统计 (不管stop words)

8.3.3. 问题:

8.3.3.1. 存在noise: 很多stop words的出现其实在段落中是没有意义的, 但是会影响模型的学习

8.3.3.2. 可能导致overfitting: 过多feature会导致模型学习到的东西不够robust

8.3.4. 改进方法: **Feature Selection** (删去**stop words**之后只用**most common terms**来训练)

8.4. Evaluating Categorization

8.4.1. 常用Meatures: precision, F1 score, **classification accuracy**

8.4.1.1. **classification accuracy**: $\frac{r}{n}$, 其中n为测试文档总数, r为分类正确的文档数

8.5. Naive Bayes

8.5.1. 一个应用: **SpamAssassin**

8.5.1.1. 过程: 得到一个邮件以后通过某些feature判断它是一个spam email的可能性

8.5.1.2. 特点: feature不仅限于words, 还用blacklist (黑名单), hand-crafted text pattern (符合某些结构的邮件很可能是spam)

8.5.2. 优点:

8.5.2.1. **Very fast learning and testing** (不需要神经网络)

8.5.2.2. **Low storage requirements** (基本上只需要存一张概率表)

8.5.2.3. **Very good in domains with many equally important features** (因为Naive Bayes不能对feature添加重要性这个概念)

8.5.2.4. **More robust to irrelevant features than many learning methods (independent**

8.6. Classification using Vector Space

8.6.1. Vector Space:

8.6.1.1. 每个文档视为一个vector, 每个component是一个term

8.6.1.2. 通常情况下都被normalize成单位长度 (回顾7.5)

8.6.2. 可以分类的前提:

8.6.2.1. 训练集中documents in the same class form a **contiguous region of space** (同一类的**documents**不会分散在空间距离很大的位置**,** 并且有很好的形状)

8.6.2.2. 训练集中documents from different classes don't overlap **much** (极少情况下会

有**overlap,** 即某个**document**同时属于多个类别)

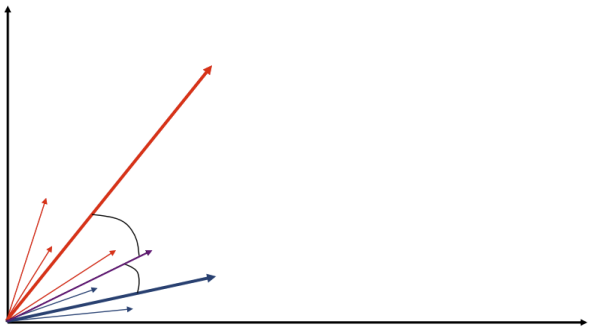
8.6.3. 目标: build surfaces to delineate classes in space

8.6.4. **Rocchio Classification (linear classifier)**

8.6.4.1. Centroid公式:

$\bar{\mu}(c) = \frac{1}{|D_c|}\sum_{d\in D_c}\bar{v}(d)$, 其中

D_c是c这个类别所有的documents的集合, v(d)是某个document的空间向量 (其实就是找出所有向量的平均方向)



8.6.4.2. 过程: 每个类别形成自己的一个**centroid,** 当一个新的文档进来分类的时候**,** 比较它和所有**centroids**之间的距离**,** 选择**nearest**的那个一个**centroid**对应的**class**作为这个文档的**class (**如果恰好距离相等就同时属于多类**)**

8.6.4.3. 特点:

- 在**text classification**里效果很好, 但是之外的应用很少, 总体比Naive Bayes差
- **Cheap to train and test documents**
- It does not guarantee that classifications are consistent with the given training data

8.6.5. **KNN (non-linear classifier)**

8.6.5.1. 过程: 每个文档作为一个向量**,** 一个新的文档进来时找出离它最近的**k**个邻居进行投票**,** 得票最多的**class**作为新文档的**class (**尽可能设**k**为奇数防止平局情况**,** 万一真平了就同时属于多类**)**

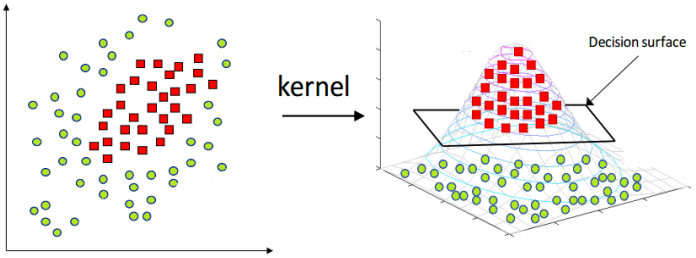
8.6.5.2. 特点:

- **Case-based, Memory-based learning**
- **Lazy learning (**每次只需要计算**distance; no need for training)**
- ★ 💥💥**Exam**题目**(saty**上课说的**):** 已知我们算距离的公式$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$, 每次得到一个新的文档就要重新计算k次这个公式, 有什么办法可以加速吗**?**
  - ➢ 答案: 不计算根号. 因为我们最后是比较距离而不是真正需要这些距离**,** 有没有根号没区别 (有点无厘头, 一个根号能省多少时间?)
- **No feature selection necessary**

- **No training necessary**
- 对于**large number of classes**能很好地**handle**
- **Small changes to one class can affect other classes**
- **Very expensive at test time** (要算距离)
- **In most cases accuracy > Naive Bayes and Rocchio**

8.6.5.3. 必要前提条件: **contiguity hypothesis** (Documents in the same class form a **contiguous region** and regions of **different classes do not overlap**. 详见8.6.2)

★ 如果出现了不满足**contiguity hypothesis**的情况怎么办**?** → 用**kernel trick**把它在转换维度变成**contiguity hypothesis**



★ 如果找不到合适的**kernel function**怎么办**?** → **neural network** (相信玄学😅)

8.6.6. 对比KNN和Rocchio在**polymorphic categories**的效果:

8.6.6.1. **Polymorphic Categories:** 多态性分类 (🌰: 假设我有一个blog是关于旅行的, 但是在这个blog中我大部分时间在介绍旅行过程中的美食, 那么这个blog应该被归类为旅行、美食, 而不单单是一个类别)

★ 为什么**Rocchio**在**polymorphic categories**上效果不好**?** → **Rocchio**本质上把每个类别的所有个体都糅合成了某一个特质, 从而缺失了个体的特征**.** 我们回顾**8.6.4.2**中的内容**,** 只有在一个**doc**到多个**centroid**的距离完全相等时我们才认为这个**doc**属于多个类别**,** 这个概率是很低的

★ 为什么**KNN**在**polymorphic categories**上效果比**Rocchio, Naive Bayes**都好**?** → **KNN**本质上是个体与个体之间的比较**,** 找到和自己最像的**k**个个体**.** 因此并不是用一个**polygon**来描述每个个体

8.7. Bias-Variance Tradeoff (💥💥Exam题目 Why have to tradeoff???)

8.7.1. KNN has **high variance** and **low bias**

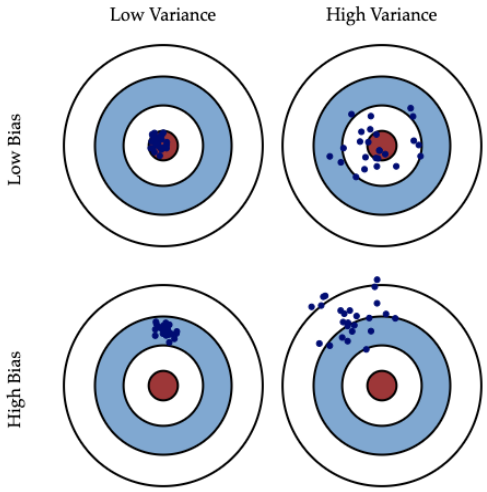
8.7.2. Rocchio/Naive Bayes has **low variance** and **high bias**

8.7.3. 尝试理解Bais和Variance:

8.7.3.1. **Bias**偏差是预测值与实际平均值的差值. 想象一下你可以多次重复构建整个

模型的过程: 每次收集一波新的数据并进行新的分析, 创建一个新的模型. 由于潜在数据的随机性, 所得到的这些模型将具有一系列预测值. 偏差就是衡量总体上这些模型的预测和真正正确值的偏差.

8.7.3.2. **Variance**方差是给定数据点的模型预测的波动性. 再次想像一下你可以多次重复整个建模过程. 方差是不同模型的预测结果对于一个固定点的变化多少.



8.7.3.3. 以上面这个靶子图为例: 中间的红心是正确值. 一个点就代表了一次测试.

- 如果一个模型有低偏差和低方差(左上), 那么我无论给定数据集的哪个部分去测试, 得到的预测值都差不多且接近正确值
- 如果一个模型有低偏差和高方差(右上), 那么我给不同的数据集部分就会得到比较分散但是都比较接近正确值的预测值
- 如果一个模型有高偏差和低方差(左下), 那么我给不同的数据集部分就会得到比较集中但是都离正确值比较远的预测值
- 如果一个模型有高偏差和高方差(右下), 那么我给不同的数据集部分就会得到比较分散且都离正确值比较远的预测值

8.7.3.4. 根据上面的分析, 理论上最好的模型是符合具有低偏差和低方差特点的. 但是我们做得到吗? → 不行.

★ 为什么不能同时获得低偏差和低方差?

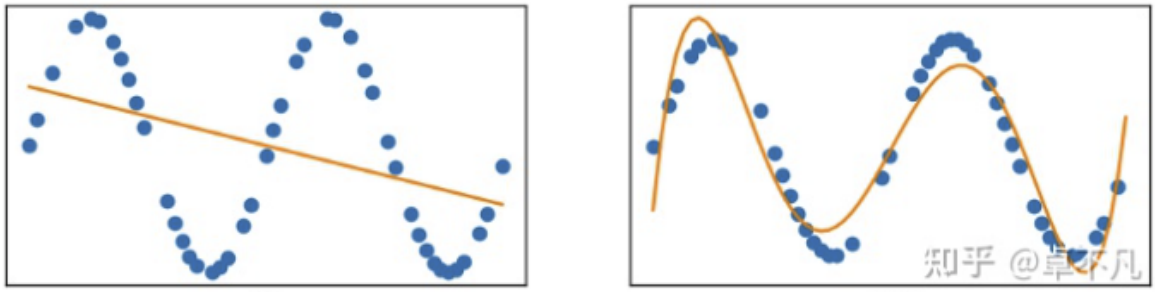
- 在机器学习中, **high bias**意味着模型没有学到数据的真实分布特点, 这是一种欠拟合**underfitting (high bias → underfitting)**
- 而**high variance**意味着模型只专注于一部分数据的分布特点, 这是一种过拟合**overfitting (high variance → overfitting)**
- 理论上如果我们训练的数据完全符合完整数据的分布特点, 那么只要模型的复杂度够高就能达到低偏差和低方差, 但现实的数据基本上都有**noise**,

并且是很多noise. 因此高复杂度的模型就意味着它还会拟合noise, 当然就造成了high variance

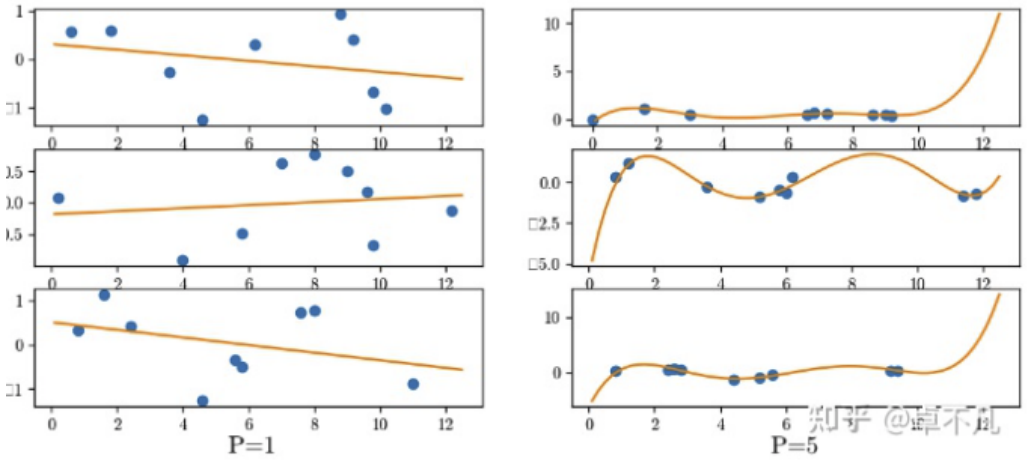
- 由上可得, 现实世界里增加深度学习系统的复杂度, 系统的Bias会减少, 而方差(Variance)会增加. 它们此消彼涨. 你不能同时减少它们, 这一点是Bias-Variance Tradeoff的基础

8.7.3.5. 因此, 一个算法越强调平均, 整体, 那它就越可能有**low variance**和**high bias**; 反之, 一个算法越强调个体, 那它就越可能有**high variance**和**low bias**

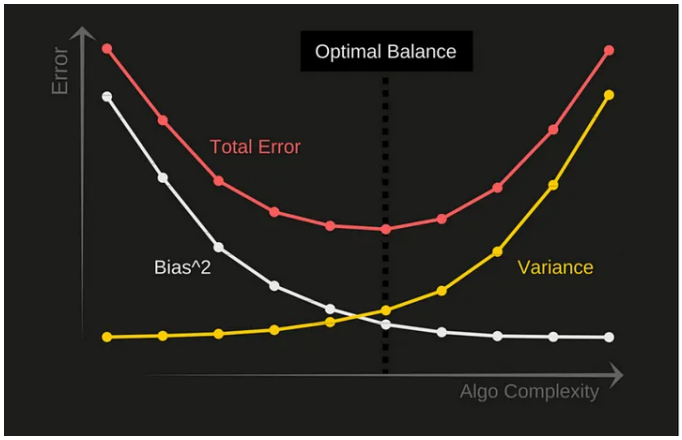
★ **Example1: Bias.** 分别用线性回归(左)和非线性回归(右)拟合到非线性模式的数据集. 结果说明了线性回归学不到非线性的特点, 因此**Bias**很大.



★ **Example2: Variance.** 分别用线性回归(左)和非线性回归(右)拟合到非线性模式的不同部分数据集. 结果说明了右边的模型过于复杂, 导致在不同的数据集的部分中的拟合曲线差别很大, 因此**Variance**很大. 想象假设以右边第一行数据作为训练集, 右边第二行的点作为测试集, 那么第一行的曲线就拟合不到测试集的点了, 这时候就是overfitting



8.7.3.6. Bias-Variance Tradeoff曲线. 我们的目标就是找到optimal balance使total error最小, 过小的bias和过小的variance都不是我们想要的



★ **Is there a learning method that is optimal for all text classification problems? → No!**
因为不同的问题的特点不同, 需要在**Bias**和**Variance**中做的**tradeoff**也不同. **The BEST way is to combine different methods!**

👉参考文献**1**(建议完整看完)**:** [WTF is the Bias-Variance Tradeoff? (Infographic)]
👉参考文献**2**(建议完整看完)**:** [Understanding the Bias-Variance Tradeoff]

9. Inverted indexing

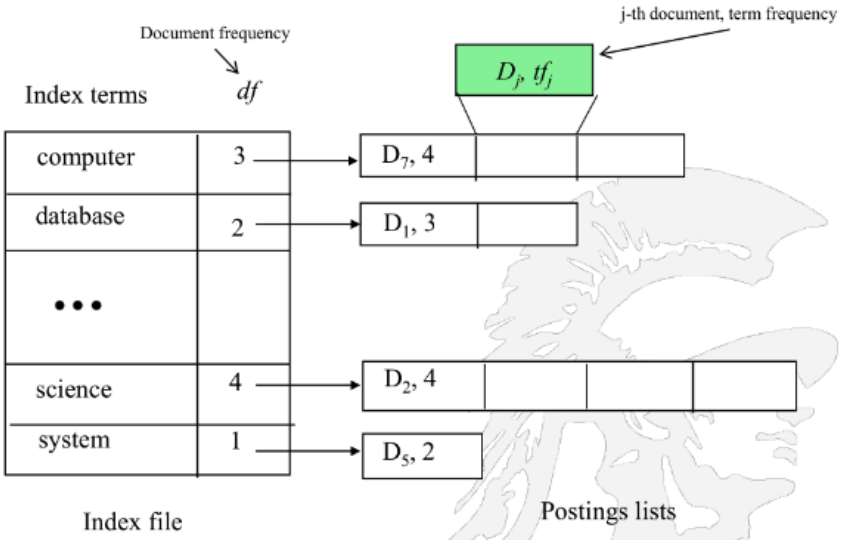
9.1. 定义:

9.1.1. 维护一个dictionary, 其中key是term, value就是包含有这个term的所有docs.
9.1.2. 之所以叫Inverted index是用于区别forward index(即正常情况下普遍以docs作为key来统计不同单词在这个docs里出现的次数)

9.2. 特点: Terms in the inverted file index are refined

9.2.1. **Case folding**: converting all uppercase letters to lowercase (所有大写转小写)
9.2.2. **Stemming**: reducing words to their morphological roots (所有词态还原为词根)
9.2.3. **Stop words**: removing words that are so common they provide no information (把所有无意义词(I, you, me…以及语气词)都去掉)

9.3. 经典数据结构(用一个🌰展现)



9.3.1. **Index file:** 把整个inverted index看作一个dictionary, 那么index file就是dictionary中的key, 但同时index file本身也是一个(key, value)的形式.

9.3.1.1. 每个**term**对应一个**IDF**(Inverse Document Frequency, 表示包含这个词的文档一共有多少个, 也就是当前这个**term**对应的**postings list**的长度)

9.3.1.2. Index file被保存在**memory**中, 用**pointers**指向**postings lists**

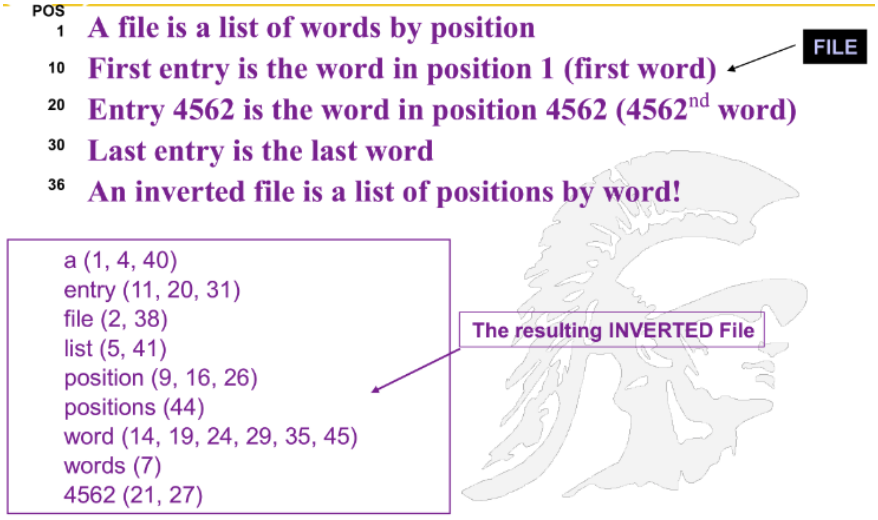
9.3.1.3. Index file的terms是按照**alphabetically**从小到大排序

9.3.2. **Postings lists:** 每个postings list是一个链表**(link-list)**, 每个元素存两个值, 一个存对应的**term**的**document ID,** 另一个是**TF(Term Frequency, 7.4.1.1)**

9.3.2.1. Posting lists被保存在**disk**中, 按照**document ID**从小到大排序

★ 一个不太一样的**Example:** 在这个🌰里只处理一个document, POS代表了term在文档中的位置, 可以看到这时的key就是term, 而value变成了对应的term在document中出现的位置的集合. 虽然看上去和上面讲的结构不太一样, 但本质上都是一个**dictionary,** 并且**postings lists**都可以用链表存储.

★ **Note:** 这里有个小问题就其实position和positions应该被认为是一个term, 同样的word和words也应该被认为是一个term



9.4. Inverted indexing处理一个query的流程

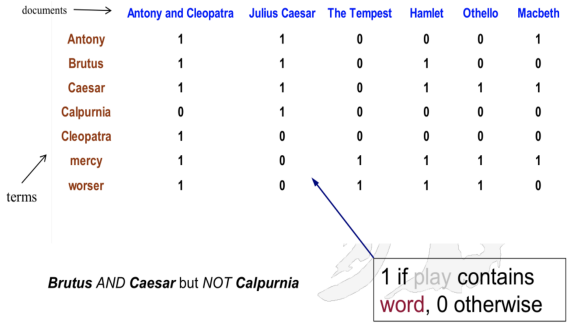
★ **Example:** 假设我的**query**是**which plays of Shakespeare contain the words Brutus AND Caesar but NOT Calpurnia (**注意这个**query**并不是真正我查询的输入**,** 只是一个逻辑表达而已**)**

9.4.1. Naive Solution:

9.4.1.1. 方法: 找出所有包含**Brutus**和**Caesar**的**Shakespeare plays**然后筛选掉那些有**Calpurnia**的

9.4.1.2. 问题: **too slow**; **needs large space**; **doesn't allow for other operations**即只能针对这一个query

9.4.2. Term-Document Incidence Matrix:

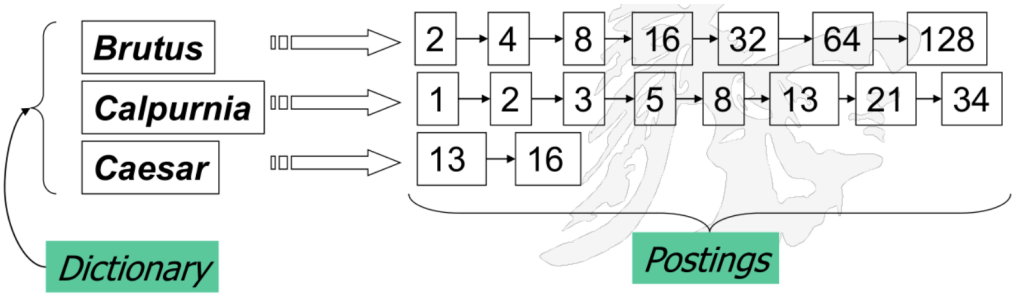


9.4.2.1. 方法: 维护一个01矩阵来表示哪些term在哪些document里出现过, 只需要根据逻辑关系进行相应的位运算即可得到最后结果(以上面的query为例, 位运算就是**110100 AND 110111 AND (NOT 010000) = 100100**, 即"Antony and Cleopatra"和"Hamlet"是符合要求的结果)

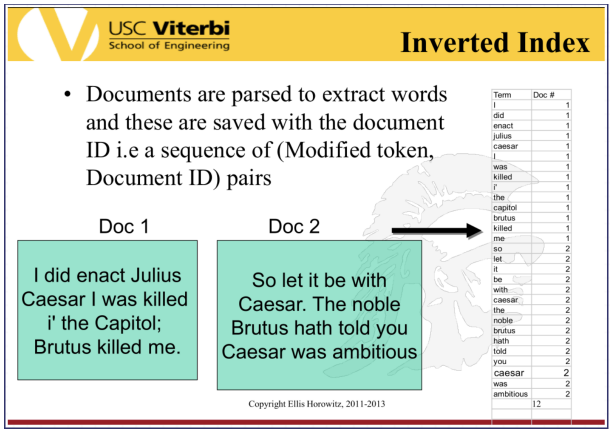
9.4.2.2. 问题: 虽然结果是正确的且可以覆盖其他query, 但是从上面的矩阵可以看出有很多为0的位, 这些**0**位占据了大量空间(即这个矩阵是**sparse matrix**)

9.4.3. Inverted index with Link-list(实际上就是9.3的模型):

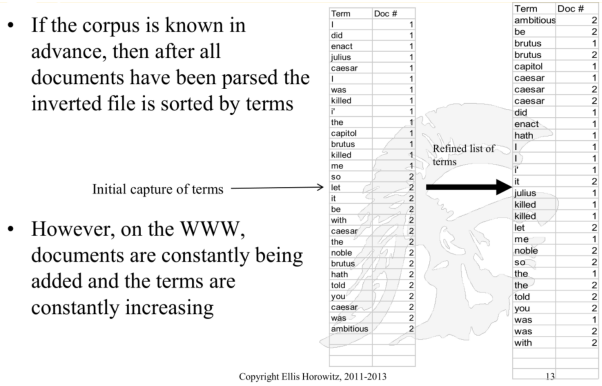
9.4.3.1. 基本结构:



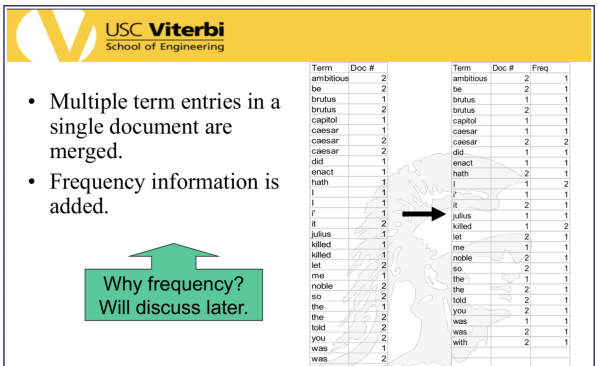
9.4.3.2. 第一步: **Extract words** (将每个doc的term都提取出来, 并且以**(term, doc ID)** 的方式存在一个表里. 这个时候不需要任何**merge,** 即哪怕一个**doc**里的**term** 出现了两次也分成两行来记录)



9.4.3.3. 第二步: **Sorted terms** (将所有terms按照 **alphabetically order**从小到大排序, 此时仍然不需要做任何的merge操作)

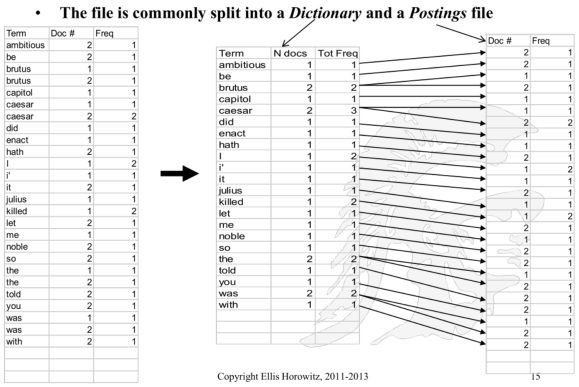


9.4.3.4. 第三步: **merge single document** (将同一个文档的同一个term进行merge, 在表格中增加一列来记录



frequency)

9.4.3.5. 第四步: **merge multiple documents** (将terms进一步merge, 形成了最终的dictionary形式. 右图里左边的部分是index file, 右边的部分是postings lists. 这一步完成后就得到了9.4.3.1中的图)



★ 为什么要用**Link-list**而不用**array**来维护呢? (即**array**和**link-list**的优缺点各是什么)

g

➢ **Array pros:**

➢.1. **randomly access to any element**

➢ **Array cons:**

➢.1. **fixed memory (<u>partially correct</u>, not fixed in JS);**

➢.2. **difficult to insert new or delete a middle element;**

➢.3. **Operating system has to find a block of memory all next to each other, if the memory is too fragmented, it's hard to find a continuous block of memory** (完整连续的空间)

➢ **Link-list pros:**

➢.1. **Easy insert or delete a middle element;**

➢.2. **Can distribute the data anywhere in the memory, just need pointers to point to each other (do not need a continuous block**完整连续)

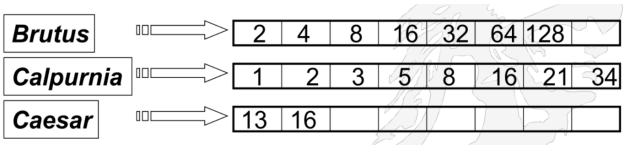
➢ **Link-list cons:**

➢.1. **Can only follow the order to access elements** (如果使用循环链表 **circular linked list,** 可以从头或者从尾开始访问, 否则只能从头访问)

★ 该怎么改进**link-list**的不足呢? → **Skip Pointers!** (<u>但是注意即使用了**skip pointers, link-list**仍然做不到**randomly access**</u>)

9.4.4. **Skip Pointers**

9.4.4.1. 正常link-list得到最终结果: 三个link-list根据长度从小到大的顺序进行merge (比如在这里就是Caesar先和Brutus合并, 合并出的list再和Calpurnia合并得到最后的结果). 目的是找到相同的 每次merge的复杂度是**O(m+n)** (m, n分别是两个list的长度

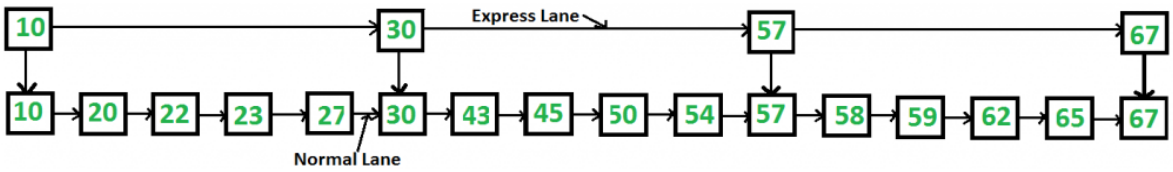


Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

★ 为什么要按长度从小到大排序**?** → 节省平均运算次数 (比如这里如果Brutus先和Calpurnia合并, 得到的list是[2,8,16], 然后再和Caesar合并, 总运算次数=7+8+3+2=15+5=**20**; 如果Brutus先和Caesar合并, 得到的list是[16], 然后再和Calpurnia合并, 总运算次数=7+2+1+8=9+9=**18**)

★ 为什么需要**Skip Pointers?** → 以上图为例, 假设我们正在合并Brutus和Calpurnia, 并且我们已经对比到16, 16了. 此时Brutus的下一个是32, 我们需要在Calpurnia里一个一个访问16后面的元素看有没有32. 但是如果我们能模拟出类似array的特点, 即可以跳着访问元素, 就可以把明显不可能存在32的区间跳过以节省时间了.

9.4.4.2. Skip Pointers结构: 将整个link-list均分成若干部分, 新建pointers连接这些部分 (相当于一条快速通道, 可以想象高速上的express)

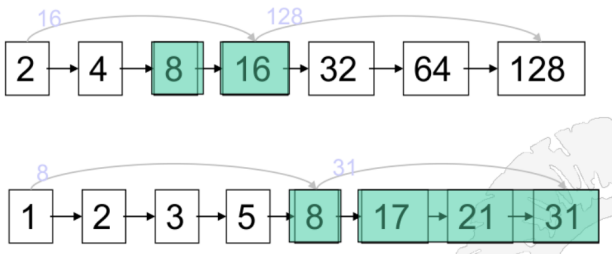


★ 怎么均分最高效**?** → 尽可能将整个**link-list**分成$\sqrt{n}$份**,** 额外用**O(**$\sqrt{n}$**)**空间

9.4.4.3. Skip Pointers查询复杂度: **O(**$\sqrt{n}$**)**, 其中n为link-list长度

~~扩展**(不会考):** 这是只添加一条express lane的复杂度, 事实上我们可以添加logn条express lane, 最下面一层是每两个elements连一个, 往上一层是每四个elements连一个, 再往上是8, 16, 32... 可以想像此时我们其实构建出了一个三叉树结构, 因此简单可以证明搜索某一个元素的复杂度为**O(logn)**~~

★ **Example:** 假设我们下一个处理的是上面的32, 下面我们正在8的位置. 我们可以从8处的skip pointer看到8对应的是31, 而31比32小, 且这里有序. 所以我们可以直接跳过17->21->31这个部分



9.5. Phrase Queries

9.5.1. 问题: 有时候我们会把**query**作为一个**phrase**去查询, 我们不希望它是按照term拆开来分别对应的, 因为有时候phrase有特别的意义 (例如stanford university)

9.5.2. **Solution1: N-grams** (其实也就是N-shingle). 这里介绍Biword (2-gram): 将两个连续的words作为一个dictionary term

9.5.2.1. Biwords will cause an explosion in the vocabulary database (很好理解)

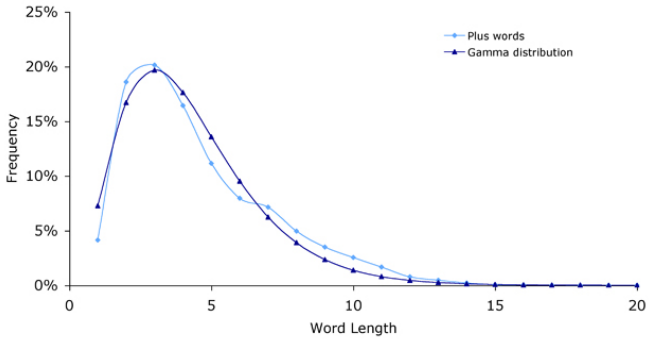
9.5.2.2. Queries longer than 2 words will have to be broken into biword segments

★ **Example:** Query: "stanford university palo alto" 对应的Biwords就是{stanford university, university palo, palo alto}

9.5.2.3. 会造成很多false positive (很多结果会包含biwords但是不包含完整的query)

9.5.2.4. **N-grams**遵循**Zipf distribution** (即随着n的增大渐渐地很多n-word-term的frequency都会特别低)

★ 💥课后思考**(**可能会考**):** 下图是真正的term长度和在网络中的出现频率的曲线, 可以发现它和我们上面描述的Zipf其实有所不同, 为什么**?** → 搜不到答案**,** 个人认为和语言学相关**,** 例如搜索**ceramics comes from,** 或者**serve as the inspiration.** 也可能是**term**太短不能准确描述用户的搜索需求**,** 而用户通常又不想完整打完一句话**,** 而是用一个短语或者几个**noun**形容自己的**query**



9.5.3. **Solution2: Positional Indexes(9.3的🌰里有所展示)**: 每个term在对应的doc ID之后还多存一个位置信息, 表示这个term在这个doc里的哪些位置出现了.

★ **Example:** 假设我们搜了"to be". 这里比如to的"2:1,17…"表示的就是在2这个doc里第1个词, 第17个词... 是to. 所以如果我们看to和be都在一个文档出现并且有存在**positional index**相邻的情况就说明这个文档里出现了我们要的phrase

- ***to:***
  - ***2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...**
- ***be:***
  - ***1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...**

9.5.4. **Solution3: Biword +** part-of-speech-tagging (词性分析): 先用词性分析得到query中所有单词的词性, 只考虑nouns进行biwords的构建再回到solution1 (基于假说"nouns and noun phrase经常出现且最能概括用户的意思")

★ **Example:** 假设query是 "renegotiation of the constitution", 经过词性分析后的到的是 **N X X N** (X代表preposition, N代表noun). 因此我们就得到了**biword "renegotiation constitution"**

★ **Note**: part-of-speech-tagging的程序可以通过statistical或者rule-based的方法训练出来

~~9.6. Distributed Indexing: 用MapReduce~~ (之后会讲的)

9.7. Dynamic indexing

9.7.1. 问题: 之前讨论的inverted index都是在数据不变的情况下, 但是现实生活中应用的时候数据可能随时都在变化, 需要维护一个动态的index

9.7.2. 方法: 维护一个大的**main index**和一个小的**auxiliary index.**

9.7.2.1. 定义: main index相当于主数据库, auxiliary index相当于一个小型数据库, 存在memory中.

9.7.2.2. 查询过程: 在一段时间内, 新的docs会先存在auxiliary index里. 这时用户给一个query, 我们会在**main index**和**auxiliary index**中都去查询**,** 会得到两个**link-list**, 对这两个**link-list**再**merge**一次就得到了最终给用户的结果.

9.7.2.3. 每过一个时间段**,** 我们把**auxiliary index**里的内容都转移到**main index**

★ 💥课后思考**(**可能会考**):** 如何动态删除某个**doc?** → 我们把要删除的**doc ID**都和在一起维护成一个**bit-vector,** 在**9.7.2.2**里得到最终结果后和这个**vector**在进行一次**merge**来**filter**那些删除后的文档 (Deletions are stored in an invalidation bit vector. We can then filter out deleted documents before returning the search result. Documents are updated by deleting and re-inserting them.)

➢ 扩展**: KNN with Inverted Index**(利用inverted index给KNN加速)**:** 得到一个query后, 首先用**Inverted indexing**得到一些**candidate docs**, 然后把query当成一个test document, 而candidate docs就是train documents. 由此跑**KNN**就能快速得到接近**query**想要得到的**docs**顺序**.** Testing Time: $O(B|V_t|)$, 其中B是一个test-document words在training documents中出现的平均次数, V_t 是test document的vocabulary. 通常情况下B<<文档总数

10. Video Search Engines

10.1. How to index a video into database? → Metadata

10.1.1. Author, title, creation date, duration, coding quality, tags, description

10.1.2. Other aspects of video recognition are subtitles and transcription

10.2. How to rank a video (or sort videos)?

10.2.1. Relevance: using metadata and user preferences

10.2.2. Ordered by date of upload

10.2.3. Ordered by number of views

10.2.4. Ordered by duration

10.2.5. Ordered by user rating

★ **Note:** 用户可以选择sort by which, 如果没有选择则会对每个因素添加一个weight, 最后算出一个final ranking (不知道具体的公式)

10.3. YouTube Recommendation System (即YouTube如何推荐更多video给用户)

10.3.1. **Association Rule Mining**: 每一对视频 (v_i, v_j), relatedness公式: $r(v_i, v_j) = \frac{c_{i,j}}{f(v_i, v_j)}$, 其中$c_{i,j}$表示视频i和j被co-watched的次数, c_i表示视频i被观看的次数, c_j表示视频j被观看的次数, $f(v_i, v_j)$是一个normalization function (e.g. $f(v_i, v_j) = c_i * c_j$)

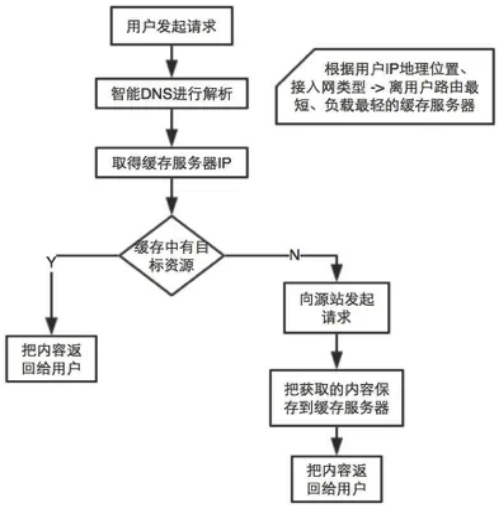
10.3.2. 当用户观看了一个视频**i, system**就会通过**10.3.1**中的公式计算出所有其他视频和**i**之间的**relatedness,** 并且从大到小排序**,** 最后选择**topk**推荐给用户

10.4. Two Technology Challenges for YouTube

10.4.1. **How to identify billions of videos?** → YouTube用固定长度11位的string来作为video ID (一共可以identify **(26+26+10)^{11}**个视频), 得到一个新的视频之后, 系统会随机给它分配一个**11**位的**string**作为**video ID**, 之后系统会搜索这个ID在database里存不

存在, 如果存在就重新随机给一个11位的string, 直到ID不存在重复. 把ID添加到video对应的URL尾部 (e.g. https://www.youtube.com/watch?v=**gocwRvLhDf8**)

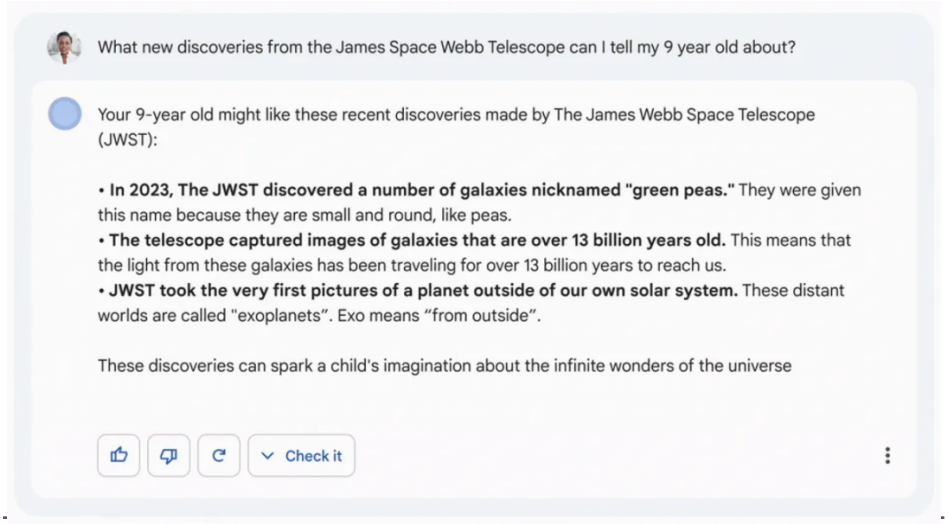
10.4.1.1. **How to efficiently deliver the video to the desktop/mobile device? → CDN (Content Delivery/Distribution Networks).** 简单来说就是网络会解析用户的IP并且根据用户的IP地址分配一个最近的服务器来读取用户需要的资源 (比如你在北京看一个youtube上的视频, youtube就会找到一个离北京最近的服务器, 通过这个服务器把视频加载到你的网页上)



10.5. YouTube如何判断video duplication (即如何维护copyright)? → ContentID

10.5.1. ContentID: a **fingerprint** database of copyrighted content. 每个视频都会被YouTube将内容(音频, 画面)取样(sampling)之后处理成一个**spectrogram**. 然后spectrogram会被hash. 之后上传的视频也会得到一个hash value, 如果这个hash value和database中的某个hash value很接近, 那就是侵犯了著作权

★ **Note:** 这里的**hash function**会使用**SimHash(5.5),** 也就是内容越接近的视频得到的**hash**值最接近

- [x] **02/14/2023**课堂扩展内容**: Amplifies Scam**
- [x] 大致可以理解为网上存在很多错误的信息或者不实的信息但是很难分辨真伪. Saty用以表达对ChatGPT深深的担忧
- [x] 🌰: Google版ChatGPT: Bard, 在demo演示的时候被发现回答的内容出现了fact wrong(下图), 在回答问题「关于詹姆斯韦伯太空望远镜(JWST)有哪些新发现, 我可以告诉我 9 岁孩子哪些内容？」Bard 提供了三个要点, 其中一个指出「该望远镜拍摄了太阳系外行星的第一张照片。」
- [x] 然而, 推特上的一些天文学家指出这是不正确的, 第一张系外行星图像是在2004年不是用韦伯望远镜拍摄的.
- [x] 

- [x] 个人理解: 如果了解ChatGPT本质其实很好理解这种错误. 因为它们本质上是自动生成系统.当前的AI不是查询已证实事实的数据库来回答问题, 而是接受大量文本语料库的训练并分析模式, 以推定任何给定句子中的下一个单词出现的概率. 换句话说ChatGPT在guessing问题的答案而不是searching

11. Query formulation

11.1. 问题**:** 传统的搜索无法处理需要筛选条件的**query (**🌰**:** 搜索**Apple AND orchard NOT computer**是想要得到不包含**computer**的结果**,** 但是搜索引擎无法判断**and**和**not**是**query**的一部分还是判断条件**)**

11.2. **Google**用符号表达式表示筛选信息**:**

11.2.1. Default (什么都不加): 表示**AND**表达式 (e.g. Apple orchard = Apple **AND** orchard)

11.2.2. **Quotes (“”**引号): 表达**exact phrase** (e.g. “Apple orchard”只会显示完整包含Apple orchard这个短语的结果)

11.2.3. **NOT (-**减号): 表达不要 (e.g. Apple -computer就不会返回computer相关的apple产品)

11.2.4. **Square bracket ([]**中括号):

11.2.4.1. 表达自动匹配相似单词 (e.g. [child bicycle]会得到包含“child”, “children”, “children’s”, “bicycles”, “bicycling”等结果)

11.2.4.2. 表达不要匹配**stop word** (e.g. [the who]就不会把the和who当作stop word二是搜索the who的结果)

11.2.5. **OR** (或): A or B表达返回匹配A的结果或者匹配B的结果 (e.g. Apple OR Pen会得到和Apple有关的结果, 也会得到和Pen有关的结果, 但不会得到Apple Pen的结果)

11.2.6. **Connect (+**加号**) and Anyword (***星号**)**: (e.g. it’s +a * world中会将+后的a当作正常单词而不是stop word 被删掉, 而*会返回任何符合a “somewords” world的结果)

11.2.7. **filetype:** 表示只搜索后缀为某一类文件的结果 (e.g. filetype:**pdf** apple就只会返回apple结果中的pdf文件)

11.2.8. **inanchor:** 返回的是包含anchor text指向的网页 (e.g. restaurants inanchor:gourmet返回的是网页中含有restaurants并且被一些anchor text中含有gourmet的网页指向的网页 (anchor text就是链接上的文字, 比如GOOGLE))

11.2.9. **intext:** 返回必须要网页里的文字包含query的网页 (e.g. 搜索youtube.com不会返回link里包含youtube.com的网页, 而是返回网页里body text有youtube.com的网页)

11.2.10. **intitle:** 只返回网页标题里包含搜索query的网页

11.2.11. **inurl:** 只返回url里包含query的网页

11.2.12. **site:** 只返回某些域名里的网站 (e.g. site:usc.com就只会返回usc.com里的网页)

11.2.13. **info:** 只返回google关于搜索query的信息 (e.g. 搜索info:applepie就会先返回recipe)

11.2.14. 其他特殊符号(e.g. @表示social network; $表示price)

11.3. Google其他的query rules

11.3.1. Query的上限长度只有32个words

11.3.2. 会优先返回query中words位置最接近的结果 (e.g. snake grass返回的是plants; snake in the grass返回的是sneaky people) (不会中间插词)

11.3.3. 会优先返回query中words顺序不变的结果(e.g. Apple watch返回的是苹果手表而不是watch apple)

11.3.4. 搜索不管大小写 (e.g. NEWS和news是一个结果)

11.3.5. 会自动忽略一些符号 (e.g. ! ? .)

11.4. Relevance Feedback & Query Expansion

11.4.1. Relevance Feedback: 用户搜索了query之后Google还会推荐给他们相似的query点击 (相关搜索)

11.4.2. Auto-Completion: 用户一边输入Google会一边帮用户补全他们可能想输入的query

★ **Note:** 这里的用的数据结构是字典树, 自动补全的query排序的顺序是根据之前大数据的搜索次数决定的 (搜索越多的越先被推荐)

11.4.3. Spelling Correction: 会在用户输入query后识别拼写错误并推断出用户最可能想搜索的query