1 Text Pre-Processing

通过处理文本 (process text), 可以从 unstructured data 中得到信息

- 文本预处理做得好不好对接下来的下游 application 有至关重要的影响
- 具体步骤可根据不同的 corpus 和 application 改动和删增
- Simple rule-based systems 可行,但是很少会有完美表现.

Term Definition

Word

A word is a sequence of characters with a meaning and/or function

Sentence

Sentence is a sequence of words, like “The student is enrolled at the University of Melbourne.”

Token (Word Token)

All separate words in a sentence, like each instance of “the” in the above sentence, like “student”, “is”, “enrolled”. 注意 “the” 在这个句子出现了很多次, 每次都算一个 token. 也就是说这句话有9 个 token, 其中有一些 token 是重复算的.

Type (Word Type)

The district word “the” in the above sentence. 注意所有 type 都必须是 unique 的, 即使 “the” 出现了很多次,在 type 里只能算一次. 也就是说上面这句话包含了 8 个 type.

Lexicon 词汇表

A group of word types.

Document

One or more sentences.

Corpus

A collection of documents.

Text Normalization 文本标准化

Remove Unwanted Formatting

比如说爬虫爬下来的网页包含了很多用来表达 HTML 格式的代码, 当我们不需要这些格式信息的时候,我们应该想办法去掉他们,只留下文本. 当然有些情况下这些格式信息还是有用的,比如他可以帮我们指出哪些文本是标题(give higher weight),哪些文本是普通正文(give lower weight).

Sentence Segment 分句

幼稚方法

用句号, 问号, 感叹号这三个标点符号来识别句子.

存在问题:

这三个标点符号除了用于断句,还有别的作用! 比如 “U.S. dollar”, “…”, “Yahoo!”. 如果只看标点来分句,会出很多问题.

正则方法

由于上面幼稚方法所存在的问题, 我们可以use regex to math 下一句开头的 capital ([.?!][A-Z])

存在问题:

但是人名也是大写字母开头, 而且缩写结束会有一个句号, 这种正则方法会错误地 match 到 Mr. Brown.

预存词汇表方法

由于正则方法还是存在问题 (比如 “Mr. Brown” 问题), 我们可以预存名字.

存在问题:

但是我们无法列举所有人名和缩写呀!

State-of-the-Art 方法

State-of-the-art uses machine learning, not just rules.

Tokenise Words 分词 (English)

幼稚方法

用正则separate out alphabetic strings (\w+)

存在问题:

缩写 (U.S.A), 连词符 (merry-go-round), 数字 (1,000,000.01), 日期 (3/1/2016), 附着词 (can't, Anna's), 网址 (http://www.google.com), hashtag (#metoo), 文本表情 (:-) , >3<), Multiword unit (New Zealand)

Tokenise Words 分词 (中文)

英文天然地带有空格来帮助分词, 但是中文并没有空格. 所以中文分词需要预先准备好一个词语表 (vocabulary).

词语表

这个词语表 (理想中) 应该列举所有中文词语

MaxMatch Algorithm

Greedily match 词汇表中最长的词语.

V = {墨,大,的,学,生,与,众,不,同, 墨大,学生,不同,与众不同}

墨**大的学生**与众不同

存在问题:

- 无法建立一个完美的词汇表, 新词太多
- 一味遵循 MaxMatch 有时候会出问题, 比如:

去	买	**新西兰**	**花**
go	buy	New Zealand	flowers

去	买	**新**	**西兰花**
go	buy	new	broccoli

Normalise Words 单词标准化

Lower Case

(Australia ➡️ australia)

Remove Morphology 去除词态

Lemmatisation

Inflectional Morphology (屈折词态)

可以产生 grammatical variants (加复数,加过去式,加比较级之类的, 但它们仍旧是同一个词,只是不同形态), 比如:

- Nouns: -s
- Verbs: -s, -ing, -ed
- Adjectives: -er, -est

Lemmatization 要做的就是把这些 variant 还原成 original form (the form in dictionary). 这个 original form 叫做 lemma, 比如:

- stopping → stop (not stopp)
- was → be (not wa)
- poked → poke (not pok)

注意:

- A lexicon of lemmas (lemma表) needed for accurate lemmatisation

Stemming

Derivational Morphology

creates distinct words (加derivational suffix/prefix, 变成衍生词,不再是同一个词)

- derivational suffix 往往改变词形, 比如:
  - personal → personally)
  - write → writer)
- derivational prefix 一半不改变词形,但改变 meaning, 比如:
  - write → rewrite
  - healthy → unhealthy

Stemming 要做的是去除所有 derivational 后缀, 剩下的部分叫做 stem. stem很可能不是一个 actual lexical item, 比如:

- automat**e**, automat**ic**, automat**ion** → automat (剩下的 stem 已经不是一个正常单词了)

注意:

- Even less lexical sparsity than lemmatisation. 更小的 lexicon size, 每个 lexicon 出现频率更高, lexicon 比较 dense.
- Popular in information retrieval.

The Porter Stemmer

Most popular stemmer for English.
第一步去除 inflectional suffixes (like -ies→ -i).
第二步去除 derivational suffixes (-isation → -ise).

Correct Spelling

打错的 word 给它纠正

- word distance
- modelling error type (phonetic error, typo error…)
- n-gram language model

Expand Abbreviation 展开缩写

- US, U.S. → United States
- imho → in my humble opinion

Remove Unwanted Words (Stop Words)

Stop Words

A list of words to be removed from the document (words that not very important for our application)

- 一般用 bag-of-word representation (忽略顺序信息)
- 如果 word sequence is importance for our application, 那么不要 remove stop words

如何选出 Stop Words

- All closed-class word (封闭性词类，诸如介词、冠词、连词等，它们的共同特点就是每类词的成员个数基本都是固定的，不会随时有新成员添加进去。比如，冠词只有a ,an ,the，不会再有其他新单词加入到冠词类中. 相反open-class words是开放性词类，比如名词、动词、副词、形容词等，它们当中总是有新词语的加入)
- All function words (功能词, 如果没有实际、具体的意义则为“功能词”, 日不用来表转折的 but, 用来表因果的 so)
- Any high frequency words (like the, a, is)

2 Information Retrieval

信息检索就是, 给定大量 documents 和一个 query string, 找出最相关的几个 documents 给 user, 以满足 user 的 information need. 请注意: information need ≠query string, 不是每个 user 都能很好地用 query string 表达出他们真正的需求, 有时候我们还需要帮助 user 去 formulate 一个合适的 query string (比如 query 补全功能, 比如"people also ask" 功能)

Term-Document Matrix

动机: 如果用 bag-of-word 的方法表示一个 document (会 lose order information, lose 句子结构), 整个语料库会变成 a list of bags. 这样不能做到 efficient access (like find a specific word). 如果用 matrix 表达 document 和 word 的关系,则会快很多.

- 每行代表一个 document
- 每列代表一个 term (一般采用 word type, 剔除 stop word, 再 stem 后成为 term)
- 每个cell可以是 binary indicator (出现/没出现), 可以是 frequency count, 也可以是别的表达某 term 在某 document 里的重要的性的 score (比如各种 TF-IDF score和 BM25).

doc1	Two for tea and tea for two
doc2	Tea for me and tea for you
doc3	You for me and me for you

	two	tea	me	you
doc1	2	2	0	0
doc2	0	2	1	1
doc3	0	0	2	2

Vector 内的 Scores

每个cell可以是 binary indicator (出现/没出现), 可以是 frequency count, 也可以是别的表达某 term 在某 document 里的重要的性的 score (比如各种 TF-IDF score和 BM25).

Frequency Count

直接数一个 term 在一个 document 里的出现频率是一种简单方便的 scoring 方法.

TF-IDF Score (Weight)

TF-IDF score 用来描述某个 term 在某个 document/query 里的重要性(weight). TF-IDF score 可以有很多种公式变种,但是都围绕着一个思想: 我们希望给 在这个 document 里面高频出现, 同时又在只在少数几个 documents 里面存在的 term 非常高的 weight. 比如说有一个 term "volcano", 这个 term 在doc 5 里面出现了很多次, 同时它只在 整个corpus (比如1w docs) 中的 10 个 doc 里面出现, 那么 TF-IDF-score("volcano", doc 5) 就会非常大, 也就是说 "volcano" 这个词在 doc 5 中有非常重要的地位. 以下是一个例子:

$$TF-IDF_{d,t} = freq_{d,t} \times log_2\left(\frac{corpus\ size}{doc\ freq_t}\right)$$

BM25 Score (Weight)

BM25 是一种比较复杂, 效果很好, 而且业界常用的 TF-IDF scoring fucntion, 它符合 TF-IDF 思想. 它有k_1,k_3,b三个超参需要我们预设好, 一般默认值分别为$1.5, 0, 0.5$. 如有需要可用grid search调参. BM25 公式如下:

$$W_{d,t} = \left[log\frac{N - f_t + 0.5}{f_t + 0.5}\right] \times \frac{(k_1+1)\ f_{d,t}}{k_1\left(1 - b + b\frac{L_d}{L_{ave}}\right) + f_{d,t}} \times \frac{(k_3+1)\ f_{q,t}}{k_3 + f_{q,t}}$$

Document Vector Representation (Vector Space Model)

在 TD-matrix 中, 一个document 变成了一个 vector, vector 的长度是 term 的数量. 同样地, query 也可以用这样的 vector 表示. (这其实就是 vector space model)

存在问题:

由于很多 cell 的值都是 0 (term 没在这个 doc 里出现就会造成 0 值 cell), 这些 vector 其实是又长又 sparse. 而实际上在算 cosine similarity 的时候, 只有非 0 的 cell 才是提供信息的, 0 值 cell 是无意义的. 所以我们又开发了 inverted index 方法 (后面详述)

Similarity (Based on Vector Representation)

Cosine Similarity

当 document 和 query 都可以用 vector 表示后, 我们可以用两个 vector 之间的 cosine 值来衡量两个 doc/query 之间的相似度. 这个就叫做 cosine similarity, 也叫做 cosine distance. 注意 cosine 其实是表达了两个 vector 之间的角度.

$$cos(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}| \times |\mathbf{d}|}$$

Speed Up Cosine Calculation

把所有 vector 都 pre-normalized to unit length之后, 只要计算分子,因为分母固定为1. 并且由于 cosine 表达的是角度,而 normalize 之后 角度不会变, 所以cosine 值也不会变.

Inverted Index

由于在算两个 document/query vector 的 cosine similarity 的时候, 只有同时在两个 vector 中都出现的 term 才contribute to cosine value, 别的 term 要么自己是 0, 要么被乘以了 0, 它们都是无贡献的. 所以我们不必要浪费资源去储存这些 0 值, 只要储存非 0 的 score 就行了. 在这种动机下我们开发出了 Inverted Index 方法, 用来替代 TF-matrix , 去储存 scores (like. TF-IDF score, BM25 score)

- inverted index 的每行代表一个 term
- 每个 term 有一个 posting list
- 每个 posting list 长得像这样: $(d_1,\ W_{d_1,t}),\ (d_4,\ W_{d_4,t}),\ \ldots$

two	→	1: 0.88	
tea	→	1: 0.47	2: 0.9
me	→	2: 0.32	3: 0.71

Querying an Inverted Index

Assuming normalised TF*IDF weights:

Set accumulator $a_d \leftarrow 0$ for all documents d
for all terms t in query **do**
  Load postings list for t
  for all postings $\langle d, w_{t,d} \rangle$ in list **do**
    $a_d \leftarrow a_d + w_{t,d}$
  end for
end for
Sort documents by decreasing a_d
return sorted results to user

$w_{t,d}$ denotes normalised $tf_{d,t} \times idf_t$ vector

Example

our query: tea me

tea	→	1: 0.47	2: 0.9
me	→	2: 0.32	3: 0.71

- a_d = < 0, 0, 0 >
- term ***"tea"***
  - a_d[1] += 0.47
  - a_d[2] += 0.9
  - end of block, a_d = < 0.47, 0.9, 0 >
- term ***"me"***
  - a_d[2] += 0.32
  - a_d[3] += 0.71
  - end of block, a_d = < 0.47, 1.22, 0.71 >
- sort to produce doc ranking
  - 2: 1.22; 3: 0.71; 1: 0.47 → [2, 3, 1]

3 Index Compression & Efficient Query Processing

term frequency

term t	f_t	Postings list for t
		document list score list
and	6	$\langle 1,6,7,8,9,12\rangle$, $\langle 1,2,1,3,1,2\rangle$
big	3	$\langle 2,5,42\rangle$, $\langle 1,1,1\rangle$
old	1	$\langle 32\rangle$, $\langle 4\rangle$
in	7	$\langle 2,3,5,6,8,14,25\rangle$, $\langle 1,1,4,1,5,3,1\rangle$
the	52	$\langle 1,2,3,4,5,7,8,9,\ldots\rangle$, $\langle 10,21,10,42,12,14,12,4,\ldots\rangle$
night	4	$\langle 1,12,13,14\rangle$, $\langle 2,2,1,3\rangle$
house	5	$\langle 6,21,32,33,43\rangle$, $\langle 2,3,4,2,1\rangle$
sleep	3	$\langle 1,51,53\rangle$, $\langle 1,2,3\rangle$
where	4	$\langle 1,3,4,6\rangle$, $\langle 1,1,2,1\rangle$

注意: 以下所有提到的"index" 即为 inverted index.

Index Compression

由于我们的 term 数量太多, document 数量通常也很多, 所以 index 通常会占用非常大的储存空间. 并且为了计算速度, index 通常还会储存在 RAM 中, 使的size过大这个问题更加需要解决. 为了解决这个问题, 我们需要一些压缩 index 的方法. 本章所说的压缩 index 即为压缩 posting list, 并且注意: posting list 是升序整数数列.

压缩原则

Compressibility (可压缩性) 被 information content of a data set 所限制. 而information content 可以用 Entropy (常用 H 表示) 来描述, 单位为 bits:

$$H(T) = -\sum_{s\in\Sigma} \frac{f_s}{n}\log_2\frac{f_s}{n}$$

all symbols in the text, like all word in a sentence, or all char in a string

where f_s is the freq of symbol s in text T, n is the length of T

For example, H(abracadabra) = 2.040373 bits with $n = 11, f_a = 5, f_b = 2, f_c = 1, f_d = 1, f_r = 2$

- Intuition: Spend less bits on items that occur often. 越常见的 term 包含的信息量越低

Posting List Compression

普通 posting list 储存一串 increasing integer (也就是 document id). 每个 integer 可能是[1,N]之间的整数 (N 为 document 数量), 所以每个integer requires $log_2(N)$ bits (这么多个 bits 可以存下一个最大为 N 的 integer).

Idea: 某个 doc id 和前一个 doc id的差值会比 doc id 本身的值小很多，所以如果我们储存这个差值而不储存原本的 doc id，那么每个值需要的 bits 会少很多.

Variable Byte Compression

把 int 转换成二进制, 成为一串 binary bit. 这时候可以发现大数字会比较长, 小数字会比较短.

Idea: Use variable number of bytes to represent integers. 每个 byte 有 1 个 continuation bit + 7 bits payload.

Number	Encoding	某个 byte 的开头第一个 bit 为1, 说明结束了
824	00111000	10000110
5	10000101	要 decode 成二进制时,先读后面的，再读前面的, 824=0000110 0111000

OptPForDelta Compression

Idea: group k gaps, and try to encode each gap using b bits. If some gaps value is larger than 2^b (means that the gap cannot be encode using b bits), throw an exception and encode it separately (using more bits).

需要预设参数 k , 每个 group 根据里面 gap 数值大小不同,可以有不同 b. 对于每个 group ,我们希望找到一个合适的 b 值, 使得这个 group 里面大约有 10%的 gap 会触发 exception (也就是说大约有 10% 的 gap > 2^b)

Example $k = 8$

```
[1 4 7 2 4 5 123 6] [3 4 755 15 12 1 8 4]
 k gaps
      b=3                      b=4
```

Encode [1 4 7 2 4 5 123 6] as:

综合策略

把一大串 int 分成一个个 blocks, 每个 block 里面有 128 个 int. 对每个 block, 选择对它最佳的压缩方法.

- Often, posting lists of term “the” are represented more efficiently using bitvectors of size N. N is the number of documents in the collection
- State-of-the-art implementations use SIMD Instructions and bit-parallelism to increase decoding speed

Fast Searching

压缩之后的 int 必须先解压才能知道它究竟是个什么值, 如果我要 search for 某个 doc id, 我不可能吧所有 int 全部解压了在 search. Fast Searching 办法可以解决这个问题.

idea: 把 posting list 分成分成一个个 blocks, 每个 block 里面有 128 个 int. 每个 block 存出它里面的最大值 (最小值也可以), 然后再 compress 这个 block. 注意, posting list 是升序数列, 所以各个 block 的 uncompressed 最大值也是升序. 这时候如果我们要找某个 int, 只需要根据各个block 的最大值, 就可以定位到我的目标在哪个 block, 然后只解压这个一个 block 就行.

Efficient Query Processing

用户不想query 一次就等好几秒, 我们需要尽量使处理时间变快. 增加服务器, 提升硬件等等都可以是处理更快, 但是本章我们要谈论在软件(算法)层面上, 如何做到这一点.

一个重要前提: 我们不需要把所有 documents 按照 similarity 大小排序好, 然后全部 return 给用户, 因为用户只会点开 top-k 个 document. 我们只需要找出 top-k 个 document 就行了.

BM25 Similarity

一个用于计算 document/query 之间的相似度的公式 (和 cosine similarity 一样作用), 业界常用,非常 fancy! 但是由于这个公式很复杂, 计算耗时比较久, 需要配合使用一些聪明的算法 (like WAND) 来加快处理速度.

$$S_{Q,d}^{\text{BM25}} = \sum_{q \in Q} \underbrace{\frac{(k_1+1) f_{d,q}}{k_1\left(1-b+b\frac{n_d}{n_{\text{avg}}}\right)+f_{d,q}}}_{=w_{d,q}} \cdot \underbrace{\ln\left(\frac{N - F_{\mathcal{D},q} + 0.5}{F_{\mathcal{D},q} + 0.5}\right)}_{=w_{Q,q}}$$

$w_{d,q}$: term q 在 document d 中的 weight

$w_{Q,q}$: term q 本身的 weight

Note: 给定一个 query, 每个 document 和这个 query 的 similarity score 其实只是 query 里的各个 term 在这个 document 里的 BM25 weight 之和.

WAND Algorithm

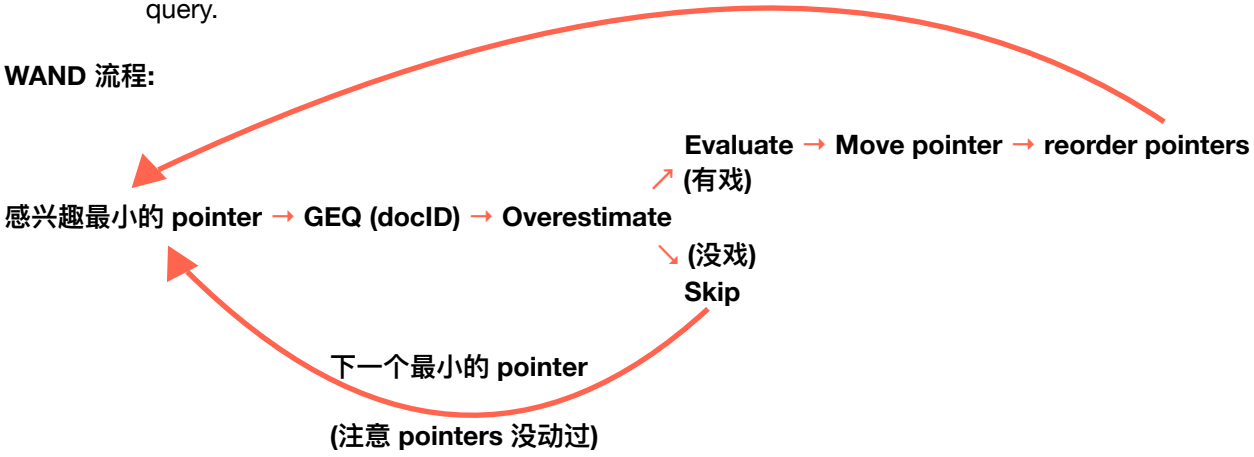
- Keep track the similarity score of top-k documents.
- For each term, calculate and store the maximum contribution it can have to the similarity score of any document in the collection
- Skip over the documents that are impossible to enter top-k.

Maximun Contribution

每个 term 都有一个 maximum contribution. 只用 term q 查询，看看哪个 document 会获得最高的 similarity score? 这个 largest score 就是 term q 的 Maximum contribution.

- 你用的 similarity 公式不同, 这个 largest score 也会不同.
- 每个 term 的 maximum contribution固定不变, 而且只是一个 float. 可以在建 index 的时候就算好, 然后存在 index 里面.
- maximum contribution 的作用是 overestimate similarity score of document in multi term query.

WAND 流程:



4 Query Completion/Expansion

在搜索系统中, information need ≠query string, 不是每个 user 都能很好地用 query string 表达出他们真正的需求. 这时候, 我们需要帮助 user 写出一个合适的 query, 或者直接在后台改写 user 输入的 query, 然后再拿去搜索.

Query Completion 查询补全

查询补全功能的一个例子就是百度的 “广东人吃……..(福建人)”😂.

目标:

- 帮助用户 formulate 一个恰当的 query
- 减少用户按键盘的次数
- 防止/纠正拼写错误
- Cache results! Reduce server load.

策略:

- Generate list of completion suggestions based on partial query.
- Refine suggestions as more keys are pressed.
- Stop once users selects candidate or completion fails.
- Why not a Language Model? Might not return results!

High Level Algorithm

Given a query pattern P:

1. Relieve set of candidate queries which are matching P from set S which is a set of possible target queries.
2. Rank candidate queries by frequency.
3. Possibly re-rank set of high-ranked candidates queries using more complex method (like personalized ranking method for different users)
4. Return the top-k highest ranking candidate queries to user as suggestions.

注意:

以上描述的只是一个查询补全算法的粗略模板, 具体算法需要落实模板中的各种细节, 比如说怎么收集 possible target query set, 怎么定义 match, 怎么 rank (用哪个 similarity…), 怎么 re-rank. 下面写的 prefix completion algorithm 就是一种具象化的算法.

特性:

1. static (eg. 补全 “twi” 的建议永远是 “twitter”)
2. dynamic (eg. time-sensitive, 世界杯的时候, 应该在 “world” 后面建议 “cup”)
3. massive/small (google 需要处理超大量的可能 query, 但是查找联系人的时候, 所有可能 query 都在联系人列表里了)

Possible Target Query Set

这个set 从哪里来呢?

- Most popular queries (比如说鹿晗分手?)
- Items listed on website (一家卖衣服的淘宝店很可能会被查询很多与衣服有关的词, 比如外套, 裙子)
- Past queries by the user (一个用户的常用联系人邮箱会被他高频搜索)

Matching 方法

Prefix Match

Partial query 必须精确地为 一个 query 的前缀, 才可以 match

Substring Match

Partial query 必须精确地为一个 query 的 substring, 才可以 match

Multi-Term Prefix Match

当 partial query 有多个 term 时, 各个 term 必须为一个 query 中相对应的 term 的前缀, 才可以 match. 比如 “FI wor” 可以match “FIFA world cup 2018”

Relaxed Match

对于拼写错误放松. 比如 “FIFO warld cu” 也可以 match 到 “FIFA world cup 2018”

Prefix Completion Algorithm

给定一个 query prefix P, 找出 the top-k most popular completions.

Trie+RMQ Based Index

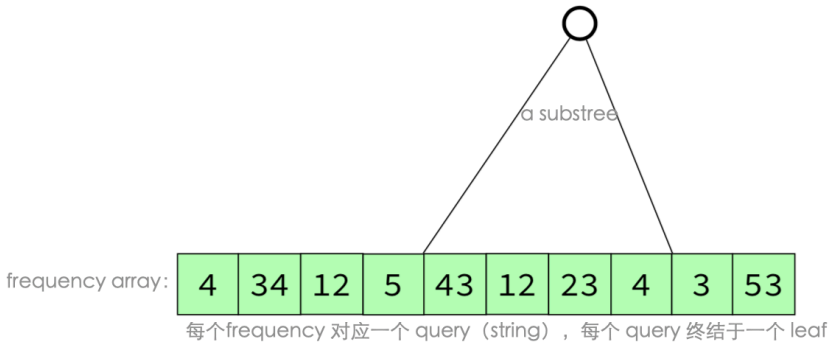
1. 我们有搜索系统的 query log, 储存了本系统的搜索历史记录. 把里面的历史 query 按字母顺序排序 (sorted unique query list), 并且 count 每个 query 的 frequency 存在 query 后面.
2. Insert all unique queries and their frequencies into a trie (also called a prefix tree).

Trie

A prefix tree, 每个 node 是一个字母, 每个 node 的 children 必须按字母顺序排序.

作用:

给定一个 prefix P, 跳 $|P|$ 次就可以找到这个 prefix 的最后一个 node, 这个 node 下面的 subtree 都可能是 P 的补全. 并且如果用 depth-first 的方式储存 tree, 我们可以做到一个 subtree 的所有 node 都存在 array 的某一段连续区域上. (每个 node 代表 一个 string/query 的终点, 所以每个 node 都带有一个其对应的 query 的 freq, 以下例子中显示的数字是 node 对应的 freq)



Range Maximun Query

Task: 给定一个 unsorted array A with length n 和一个 range [l , r] with length m. Find the positions (index) of the k largest numbers in A[l , r]　(很显然幼稚办法可以很简单地做到这件事, 但是我们需要 time and space efficient!!)

steps:

1. 预存 array A 的所有可能 sub range 的 maximum, so that we can find the max of any sub range in O(1). 注意: A 的长度为n, 所以 A 有n^2个可能的 sub range. 所以step1 需要 O(n^2)
2. For our sub range [l , r], find its max and the max position p. Note that it is the top-1 maximum
3. For sub range [l , p-1], find its max and the max position….
4. For sub range [p+1 , r], find its max and the max position….
5. Until we have k 个 maximum.

Time cost: $O(n^2) + O(klogk)$

Space cost: $O(nlogn)$

Query Expansion

主要是同义词问题. 明明是同一个 concept, 用户和 document 可能用了不同的单词去表达它. 比如说, 都是在说电影, 用户查 movie 但是 document 里面用的是 film. 这样会导致这个 document 无法被 return. 遇到这种问题, 有时候用户会人工地 reformulate query, 比如说先查查 movie 看看结果, 再查查 film 看看结果.

Query Expansion 通用方法

- Retrieve synonyms from thesaurus or WordNet (medical domain)
- Word2Vec (what words are close to the query words?)
- Spell correction (importamt → important)

User Relevance Feedback

用户用原始 query 查询之后, 手动点击按钮告诉 system, 你给我看的这个 document 非常好非常 relevant. 然后这些被表扬的 document 可以被利用去 formulate 更好的 query.

Pseudo-Relevance Feedback

Also called blind relevance feedback.

先用用户输入的原始 query 查到 top-k 个 documents. 然后看看这些 documents 有什么共有的 importance term/ informative term/ topic. 把这些 doc 共有的关键词融入到新的 query 里去再重新搜索. 注意这个过程中不需要用户的任何反应, 所以也叫做 blind relevance feedback.

Indirect Relevance Feedback

用户用原始 query 查询之后, 会点开一下他们感兴趣的 documents. “点开”这个行为间接地告诉 system, 你给我看的这个 document is relevant. 然后这些被表扬的 document 可以被利用去 formulate 更好的 query.

- Learning-2-Rank uses neural models to rerank result pages (later this semester)

5 Index Construction & Phrase Query

Inverted Index Construction

Static情况:

- Static collection of documents
- Offline construction
- Fixed at query time (no delete or append)

现实中 (dynamic 情况):

- Documents collection keeps growing (at a given rate like 1000 new docs/second in web search case)
- New arrived document should be searchable immediately
- Documents may be delated or modified.

Batch Indexing (for Static 情况)

注意这个算法用于static情况下. 这个算法是一种 map and reduce.

InvertBatch: 先把 document collection 分成多个 batch, 为每一个 batch 里的 document 分别建立 inverted index. (share global vocabulary)

MergeBatches: 根据 term, 把多个 inverted index 一条一条合并起来. 比如 term1 在这个 index 里面一个 posting list, 在那个 index 里面也有一个 posting list, 我们要把这两个 posting list 合并起来. Note: 如果上一步的 posting list 已经用 various bytes 方法压缩了, 那么这一步 merge 只要 appending bytes.

Incremental Indexing (for Dynamic 情况)

Incremental indexing allows you to add, modify, or delete documents without reindexing the entire document collection.

在这个算法里, 我们需要两个 index. 第一个是主要 index (static large index) on disk, 第二个是辅助 index (auxiliary index) on memory. 当有新 document过来时, 更新auxiliary index以包含这个新 doc 的信息. Query 时我们需要同时 query 两个 index 然后把结果合并,再返回给用户. 当auxiliary index 长太大了的时候, 把它跟 large static index merge.

Construction cost: 设定小 index 的上限为 n 个 document, 最后我们要得到一个包含 N 个 document 的大 index 需要 $O\left(N^2/n\right)$ 次 I/O

存在问题:

需要频繁 merge, 每次 merge 需要读整个 static index, 读写次数太多太多 (甚至多到 infeasible) , 而且 static index 只会越来越大.

Logarithmic Indexing (for Dynamic 情况)

分多层设立小 index. 底层的小 index 最大 size 设为 n 个 documents, 当底层的小 index 长到了它的上限, 开设另一个底层小 index, 最大 size 同样为 n. 当两个底层小 index 都满了, 我们把它们 merge 成一个2n大的上一层的 index. 然后继续开设新的底层 index, 当又攒满两个底层 index, 把它们 merge 成一个两倍大的上一层的 index. 这时候可以发现, 我们已经攒满了两个两倍大的上一层index, 这两个两倍大的上一层 index 又可以 merge 成一个四倍大的上上层 index.....

注意: Query 时我们还是需要同时 query 所有 index 然后把结果合并,再返回给用户.

Construction cost: 设定最底层小 index 的上限为 n 个 document, 最后我们要得到一个包含 N 个 document 的大 index 需要 $O\left(Nlog(N/n)\right)$ 次 I/O

Distributed Index Construction

把 document collection 分成多个 shards, 尽量均匀的分布在各个 processing nodes 上, 最好使得每个 processing node 有着均匀的 indexing 和 search workload.

注意: 每次 query 时我们还是需要同时 query 所有 index 然后把结果合并,再返回给用户.

注意: 这个方法不再有merge步骤!

Phrase Query

有时候用户想要搜索一个multi-term 的固定短语, 比如 “the University of Melbourne” 和 “New Zealand”, 这些固定短语不应该被分词, 如果分词了它们的意思就改变了, 应该把它们当做一个 term 来看.

Phrase as Term

找出搜索系统收到的历史搜索记录里 top-k frequency phrases, 把他们看着做一个 term, 像 term 里一样建立在 inverted index里面, 这样做可能解决 90% 的 incoming phrase query!

Positional Inverted Index

In addition to document ids and frequencies, store the **positions** of each term in the document.

term t	f_t	Postings list for t (docids,freqs,positions)
		document frequency position of each occurrence
and	6	$\langle 1,6,7,\ldots\rangle, \langle 1,2,3,\ldots\rangle, \langle\langle 5\rangle, \langle 12,43\rangle, \langle 6,45,212\rangle, \ldots\rangle$
big	3	$\langle 2,5,42\rangle, \langle 1,1,1\rangle, \langle\langle 8\rangle, \langle 43\rangle, \langle 65\rangle\rangle$
old	1	$\langle 32\rangle, \langle 4\rangle, \langle\langle 6,34,56,59\rangle\rangle$
⋮	⋮	⋮

搜索的时候, 先分开搜索phrase里的每个 term. 对于每个 term, 找出包含这个 term 的 document 组成一个 set. 把这些 set 取交集, 交集里的 document 包含了 phrase 里的所有term. 再查看这些 document 里面, 固定短语中的 term 的 position 是不是连在一起. 如果连在一起, 说明这些 term 是以固定短语的形式存在在这个 document 里面. 如果没连在一起, 说明这些 term 是分散地存在在这个 document 里面, 那么这个 document 就不是我们的搜索目标. 把最后留下的目标 document ranking, 然后 return 前几名给用户.

存在问题:

- inverted index 的储存 size 会变得很大, 会占用大量储存空间.
- 如果一个 phrase (比如 “the who”) 存在于非常多的 document 中, 那么这个算法会很慢
- 如果一个 phrase (比如 “the University of Melbourne”) 的里各个 term (“the”, “university”, “Melbourne”) 常常分开出现在很多 document 里面，并不是以固定短语形式出现, 那么这个算法也会很慢.

Exact String Search

在一些 bioinformatics 应用场景中, text常常是无法分词的, 比如说基因序列. 但用户常常想要精确查找一些其中的 pattern, 比如说 “AGCTAGCAGAA” in genome databases.

Suffix Arrays

由于普通 string matching 方法需要$O(nm)$ where n is the length of long text and m is the length of short pattern string. 这个算法太慢了,我们需要一个高效一点的算法来做到这件事.
suffix arrays可用来替代 inverted index, 并且不需要进行分词. 在需要精确搜索 bioinformatics 的应用场景中广泛应用.

1. 把一个 long text string 循环播放, 每个位置都有一次出现在开头的机会. 得到n个 suffix where n is the length of long text, 并把每个 suffix 的开头字符在原本的long text string中的位置和suffix储存在一起. 储存一个字母需要$log\sigma$ bits where σ是字母表大小, 比如说英语是 26 (因为有σ种字符,每个字符需要$log\sigma$个 bits去表示,才能让每个字符区分开来). 储存一个 suffix 需要 space cost $nlog\sigma$ bits. 储存所有 suffix 需要 space cost $n^2log\sigma$ bits. 储存一个 freq 需要 $logn$ bits (因为freq最大可能为n, 所以每个 freq 需要$logn$个 bits去表示,才能让每个数字区分开来), 储存所有 freq 需要$nlogn$ bits.
2. 把suffix 们按字母顺序排序. 使用 quick sort 的话, time cost 是 $O(n^2logn)$, 使用 best algorithm (??) 的话是$O(n)$.
3. Each occurrence of the pattern string is the prefix of a suffix. 由于 suffix 们是 sorted, 我们可以用 binary search. Time cost $O(mlogn)$, space cost $O(nlogn+n)$

优点:

More runtime guarantees than inverted index

延伸:

Compressed Suffix Arrays (**CSA**) use space equivalent to the compressed size of the text. Example: 1GB text, 8GB Suffix Array, Compressed text (gzip) 200MB, CSA 150MB

More Advanced Query

- Wildcard/mis-spelling queries (Sydney vs. Sidney? User may want to query “S?dney”)
- Proximity 近义词 queries (“president” close to “obama”)
- Regular expression queries (“[Jj]ohn.*”@smith.com???”)

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra [sp3..ep3]
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: $P_{[0,2]}$ *bar* interval [9..10]

Often require specialized indexing structures to solve those kind of queries efficiently. Generally much slower than regular queries.

6 IR System Evaluation & Re-Ranking

IR System Evaluation

评价一个搜索系统返回的 document 是否质量高, 是否满足了用户的 information need 是个非常主观的问题. 我们需要 human judgement. 但是人工判定太贵太慢, 而且每次对 system 做出细微调整(比如调参)都要重新判定. 所以我们需要寻求一些 automatic evaluation 方法.

TREC Collection:
用来 evaluate IR system 的数据集. 包括很 documents, 很多 query (带有相应的 information need 的详细叙述), 对每个 query-doc pair 都有人工做的相关性判定 (人工认为这个 doc 和这个 query 背的需求 相关,则判定为 1, 否则判定为 0)

一些前提:

- Retrieval is ad-hoc (query performed only once, and with no prior knowledge of the user or their behavior)
- Effectiveness based on relevance (相关性描述一个 document 是否和 information need 相关, 可以是 0/1, 也可以是几个相关性等级. 并且每个 document 的相关性是独立的, 与其他 document 无关. Effectiveness is a function of the relevance of documents returned by the system)

Indirect Relevance Feedback

除了直接的人工判定, 我们还有一种方法可以间接地了解到用户的态度. 用户用原始 query 查询之后, 会点开一下他们感兴趣的 documents. “点开”这个行为间接地告诉 system, 你给我看的这个 document is relevant. 然后这些被表扬的 document 可以被利用去 formulate 更好的 query.

- Learning-2-Rank uses neural models to rerank result pages (later this semester)

Evaluation Metric

需要预先准备一份 human judgement, 选出 truly relevant documents for each query.

Precision @ K

Compute precision using only ranks 1 .. k

$P@k = \frac{TP}{k}$ where TP is 前k个 (top-k) returned documents 中有多少个是truly relevant 的?

注意:

- This metric ignores ranking inside the top-k set
- 而且只评价一个 query 的结果

Average Precision

Take average over precision@k for each k where rank k item is truly relevant.

注意:

- This metric becomes rank-sensitive
- 仍旧只评价一个 query 的结果

Mean Average Precision (MAP)

Precision averaged across multiple queries.

注意:

- 综合评价了多个 query 的结果

Reciprocal Rank (RR)

第一个正确(相关)答案在第几位

Reciprocal rank=1/rank of first correct answer

注意:

- 只评价了一个 query 的结果

Mean Reciprocal Rank (MRR)

Average RR over multiple queries

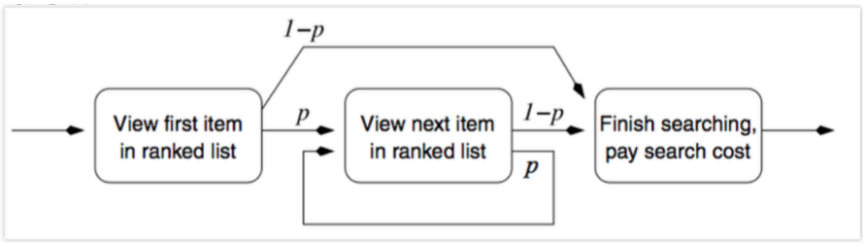
注意:

- 综合评价了多个 query 的结果
- 这个方法忽略了第一个正确答案之后的其他答案是否好

Rank-Biased Precision (RBP)

RBP是一种utility-based metrics.

假如用户每点开一个 relevant doc, 我们就可以收到$1, 并且用户一定花打开第一个 doc, 而接下来的 Prob(点开下一个 doc) 恒



为 p. 也就是说打开第二个的概率是 p, 打开第三个的概率是 p^2...用户打开了多少个相关 doc,开发者就能收到多少钱. 钱越多说明这个 query 的结果越好.

$RBP = \left(r_1 + r_2 p + r_3 p^2 + r_4 p^3 + \cdots + r_k p^{k-1}\right) \times (1-p)$ where r_i is the i-th element of relevance vector (of length k)

注意:

- 只评价了一个 query 的结果
- 对于每个用户, p值不同. 耐心的可能有 0.95, 不耐心的可能就 0.5

Multi-Stage Retrieval System (Re-Rank Documents)

1. Use a cheap, fast, simple similarity metric (like BM25) to retrieval top-k documents.
2. For these top-k documents, apply a Machine Learning algorithm which uses more features to re-rank the k documents.

注意:

为什么不直接用 ML rank all documents? Too Expensive!

Learning-To-Rank

Training data: $< q, d_i, u, r_i >$ which represent the query q, one document from top-k set $(d_1, \ldots, d_k)$, user u and document d_i's corresponding relevance indicator from relevance vector $(r_1, \ldots, r_k)$ for query q.

Input $< q, d_i, u >$ to the model to predict r_i (某个 user 用某个 query 在 simple system里查询,得到了 k 个 doc. 但是其中哪几个是真的 relevant 呢?)

User Features

- What kind of documents has the user been looking for?
- What kind of links is the user clicking on?
- What are the user's friends searching/clicking on?
- User's location
- User's native language
- User's age
- …

Document Features

- Various TF-IDF formula
- Number of slashes in URL
- Main topics of the document(see Topic Models!)
- Length of URL
- Page-rank / number of in-links or out-links
- How long do users stay on the URL before returning to search engine?
- Quality score (spam or no spam?)
- Navigational vs Informational
- For a given query Q, how often was document D first click, last click, only click?
- Users that come view the document come from the same location as the document?
- …

Query Features

- Number of terms in the query
- Popularity of the query
- Time-sensitive? Like "World Cup"
- Number of matching documents
- BM25 score distribution
- …

Point-Wise Objective

Objective is thing that we want to optimum

Input: features vector x_i for each $< q, d_i, u >$ tuple

Model: model $r_i = f(x_i)$ that outputs a real number that measure the relevance level of document d_i for query q.

也可以说是 train 一个 classifier that output $P(r_i = relevant \mid x_i)$

To "learn a model" we define an objective that we try to minimize. This is usually referred to as a loss function (minimize loss function = trained a model)

Pair-Wise Objective

Train 一个 classifier that predict if $r_u < r_v$ based on pairs of training documents for a same query.

RankNet Algorithm

RankNet outputs $P(y_{u,v} \mid x_u, x_v) = \frac{1}{1 + e^{[f(x_u) - f(x_v)]}}$ where f is a scoring function, x_u and $x_v$vectors representing the two documents, and $y_{u,v}$ is a binary value (1 → u better than v, 0 → v better than u)