Information Retrieval

Mids → 20%, Ends → 40%, Assignments : 20%, Attendance → 20%

Course material
First part : Manning's book

What is IR ?

→ Finding material (of documents / web documents) of an unstructured nature (text (for this course)) that satisfies "information need" from within large collections of data.

"information need" → query

large collections → corpus of documents.

What are potential documents

→ Legal documents / web pages / research papers

Challenges in IR:

→ Query & Data to retrieve is unstructured & hence understanding the NL query.
→ Erroneous queries & understanding the intent behind the query.

Components of query system

Document corpus

query string → IR algo

Relevant docs → usually ordered by extent of relevance

What does relevance mean

→ The response should be on the proper subject (topic).
→ Response should be timely (i.e not outdated response)
→ Responses should be authoritative (should be of good value).
→ Relevant to the information need of u / intent of query.

Most vanilla form of IR:

Keyword search (string matching) Ex: Match the query & document string (Moderately strict keyword matching)

Caveats of keyword matching

→ Word relations absent: Ex PRC & China of ~~are the~~ must be the same things
→ Polysemy: (Context based word meanings)
Ex: Apple — fruit
— company.

∴ IR → requires understanding of the intended meaning of word : requires context.

→ Not just syntax but semantics also important.

Boolean retrieval

Assumption: i) User interacting with an IR system.
ii) Document: unit of retrieval
iii) Doc ID: unique identifier of each document.
iv) Corpus: collection of documents.
v) User: has an info need
↓ expressed as
query ~~fed to~~ → IR system.
vi) Term: Units of information (word / phrase etc)
vii) IR system: Tasked with retrieving relevant documents.

- Running example for Boolean retrieval:
Corpus: All plays written by Shakespeare
Document: Individual play
Query: Boolean ~~expr~~ expression of terms connected by boolean operators ($\wedge$, $\vee$, $\sim$ etc)

Ex: Query: Which all plays of Shakespeare contain the words Brutus AND Caesar but not Calpurnia.
Possible solution: Grep all shakespearean plays containing Brutus AND Caesar.
Go line by line & omit all plays containing Calpurnia.

Cons of above solution

① Very slow

NOT Calpurnia is slowing things down.

ii) Queries like word A appearing near word B (Ex: Romans appears within 3 words of countrymen) is very hard to frame.

iii) Ranking not possible.

Term incidence matrix

A matrix of terms * Documents

Ex. Documents →

Terms ↓	Antonio & Cleopatra	Antonio & Cleopatra	Julius Caesar	The Tempest	Hamlet	Macbeth
Antonio	1	1	0	0	1	
Brutus	1	1	0	1	0	
Caesar	1	1	0	1	0	
Cleopatra	1	0	0	0	0	
Calpurnia	0	1	0	0	0	
Mercy	1	0	1	1	1	

$T_{ij} = 1$ indicates document j contains the term i.

Queries of form Brutus AND Caesar AND NOT Calpurnia can be processed using the AND of bit strings of appropriate terms.

Brutus AND Caesar AND NOT Calpurnia.

$\equiv [1\ 1\ 0\ 1\ 0] \wedge [1\ 1\ 0\ 1\ 0] \wedge [1\ 0\ 1\ 1\ 1]$

$\equiv [1\ 0\ 0\ 1\ 0]$

Problems with term incidence matrix

i) Size: Not scalable to something like the web.
Ex: 1M docs, each doc on avg: 1000 words. & we need 6 bytes per word, then total document corpus size: 6GB.
If there are 500K distinct terms across documents, then size to store the table = 500K × 1M × size/entry = $\frac{1}{2}$ trillion × size/entry

Possible solution: Observe that the matrix is sparse (Each word has on average only 1000 1's). ∴ Out of $\frac{1}{2}$ trillion entries, only 1B entries are informative

Inverted index

For each term T, store a list of all documents that contain T

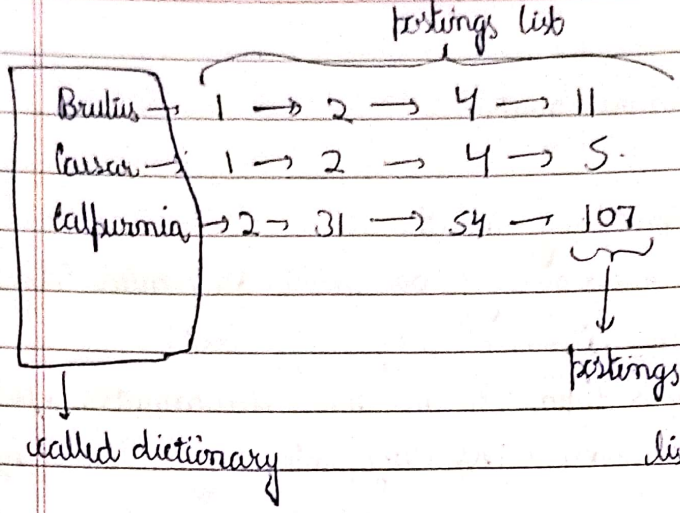
called inverted as terms point to documents.

Inverted index on four example:

Brutus → 1, 2, 4, 11, 31, 45, 173, 174.
Caesar → 1, 2, 4, 5, 6, 16, 57, 132.
Calpurnia → 2, 31, 54, 101

Some questions:

i) Keep the doc IDs contiguous or not:
In order to facilitate addition of new docs, linked list used.



ii) Where to store

Dictionary is usually stored in the main memory, postings list in disk.

iii) How to go from Raw text doc to Inverted index

a) Tokeniser : Splits running text into individual tokens (more abt this later) (Normalisation)
b) Language models : Does some basic processing on words (decapitalisation, prefix - suffix removal & so on)
c) Indexer : builds index

Ex: Friends, Romans, countrymen

↓ Tokeniser

Friends | Romans | country-men

↓ LM (process called stemming)

friend | roman | countrymen

called stems

Ex:

Tokenization

state-of-the-art → [state] [of] [the] [art]

Normalization

U.S.A ≡ USA

Stemming

authorize authorization

authorize

~~Stopwords~~ /

Stopwords / Function words

→ Words that do not have a meaning of their own (Ex: a, an, the etc). Occur very frequently.

→ Usually not indexed

Indexer

We have the tokens & the doc ID.

Token	Doc ID
I	1
did	1
:	
Caesar	2
was	2
cautious	2

Two times sorting is done

i) lexicographically on terms

ii) By doc ID

First time sort

ambitious → 2
⋮
caesar → 1
caesar → 2
⋮
I → 1

second time

ambitious → 2
be → 1
be → 2
⋮
caesar → 1
caesar → 2
caesar → 2.

After 2 time sorting, we sort document frequency of term (no of unique doc the term appears in)

Now this sorted list is converted to postings list.

Term	Doc freq	Postings list
ambitious	1	→ 2
be	2	→ 1 → 2
caesar	2	→ 1 → 2

Processing query Brutus AND Caesar from inverted index

i) Locate Brutus in dictionary
ii) Retrieve postings list of Brutus from ~~list~~ disk
iii) Locate Caesar in dictionary
iv) Retrieve postings list of Caesar from disk.
v) Find the AND of the 2 lists using the merge algorithm -
time required = $O(|X| + |Y|)$ / works as the lists are stored in sorted fashion.

```
Merge (P1, P2)
    ans = []
    while P1 ≠ NULL & P2 ≠ NULL:
        if *P1 < *P2:
            P1 = P1 -> next
        else if *P1 == *P2
            ans += *P1
            P1 = P1 -> next
            P2 = P2 -> next
        else
            P2 = P2 -> next
```

Boolean retrieval system views any document as a set of terms
→ Still no ranking system, but still very popular.

Ex: What is the statute of limitations in cases involving Federal tort law act?

Can be written as Boolean query:

LIMIT!/3 STATUTE ACTION/S FEDERAL /2 TORT/3 CLAIM

- LIMIT! — wildcard that mentions all variations of limit
- /3 — max 3 words
- STATUTE ACTION/S — STATUTE & ACTION must be in same sentence

Why doc frequency required

Ex: Merge Brutus AND Calpurnia AND Caesar

Brutus → 1 → 4 → 5 → 16 → 32 → 44
Caesar → 1 → 2 → 3 → 6 → 9 → 45.
Calpurnia → 13 → 40.

Optimal ~~sorting order~~ merge order (Calpurnia ∧ Brutus) ∧ Caesar.
This is known ~~as~~ by the doc freq. (Merge the smallest 2 doc freq lists every time)

What should be order of OR queries?

Ex
(madding or (covid) AND (ignoble or strife).

Doc freq of OR is atmost sum of doc freqs of individual terms.
∴ Sort by sum of doc freqs & process in increasing order.

Google also uses boolean retrieval.

$[W_1 W_2 \cdots W_n] \equiv W_1$ AND $W_2 \cdots$ AND W_n.

Problems
→ Very few docs with all words
→ Words might be in anchor texts & not doc
→ Synonyms.

Term vocabulary / Tokenisation & normalisation in IR.

Inorder to deal with documents, we need to process documents (parse documents).
refers to converting byte sequences into a linear sequence of characters.

Issues:

i) Different format & language
ii) Different encoding
iii) Character set in use

} Each of these is a classification problem.

But generally, we use heuristics to deal with these problems rather than full fledged classification.

The term document itself is ill defined:

i) Is document a single file or collection of files.
ii) Is document a single email, chain of emails, email with attachment

} Design choices that depend upon downstream appl

Some definitions we'll use

i) Word:
Delimited string of characters as it occurs in running text (corpus)

ii) Term:
Normalised word.

iii) Token
An instance of a word.

Style

An instance of distinct occurrence of a word (i.e token)

In the inverted index construction

Input :

Doc1 → [friends, Romans, countrymen ...]

Doc2 → [let us be with caesar ...] → the created terms to build posting list.

Output : tokens

[friends] [Romans] [us] [be]

Steps to go from input to output :

Tokenization : Splitting of string of characters into tokens / words.

Usually done with splitting using whitespace delimiter

Issues :

i) One word or many :

Hewlett-Packard

State-of-the-art.

San Francisco.

Things like : San Francisco - Los Angeles Apres.

** Design issue : Consider these as single token (will have to build a dictionary with these multi-word tokens) or break them into multiple tokens.

ii) Numbers
a) Differentiating b/w various formats of dates.
b) Differentiating b/w phone numbers & IPs.

iii) Languages without delimiters (like chinese)
iv) Language specific nuances.
v) Morphology of words.
vi) New words
vii) Accents while indexing
Do we ~~Dots~~ Universität & Universitaet differently? (like résumé)
→ Handled usually using careful observation of users: Users usually drop accents while querying. ∴ Drop accents while indexing.

viii) Case folding: Usually indexing is done after blindly converting everything to lowercase.
Caveat: MIT & mit.
Fed v/s fed.

Normalisation: we want to identify query terms that have the same form.
ex: U.S.A & USA.
~~cars~~ cars & automobiles } ideally we would want these in the same index

Requires generation of equivalence classes among tokens. Can be done using handwritten rules or asymmetric expansion.

Asymmetric expansion:
window → window, windows
windows → window, Windows
Unix Windows → ✗ restricted to expand (∴ asymmetric)
Asymmetric expansion used to replace query words with equivalents

Stopwords & their removal

→ Stopwords: words that do not have a meaning of their own but rather render meaning to other words.
Ex: Articles, prepositions, conjunctions, auxiliary verbs.
→ Occur very frequently in a document. Some of the most frequently occuring.
→ Typical design policy is to not index these words.
Caveats: Phrase queries like "Joan of Arc"
↳ can't be removed cause provides a greater purpose than just stopword

Stemming & ~~lemmatization~~ lemmatization (~~Morph~~ Morphological processing)

Converting the word to its root form
↳ is called stem / lemma.

lemmatisation: proper reduction of the word to its lemma. Usually by matching the headword from a dictionary

Stemming: Just chops the end of the word & assumes the stem has been obtained. Usually done using heuristics.

is, am, are —lemmatize→ be
car, cars, car's, cars' —lemmatize→ car.

automation, automatic... —stemming→ ~~aut~~ automat

↓

Stemming usually good enough for practical purposes.

Inflectional morphology

Word gets an inflection. POS remains same.

Ex- car → cars

car: noun, cars: noun.

Derivational morphology

POS changes.

Ex- automate → automation

verb → noun

Stemmer: Stemming algorithms are called stemmers.

Porter stemmer

Only 1 rule applies at each stage. Rule decided by longest suffix match.

Stage 1

sses → ss (caresses → caress)

ies → i (ponies → poni)

ss → ss (caress → caress)

s → ϕ (cats → cat)

caress matches both s & ss. ss is chosen.

Stage 2

(*v*) ing → ϕ (walking → walk)
(*v*) ed → ϕ (played → play)

Stage 3

~~ret~~ ational → ate
ization → ize
ator → ate
⋮

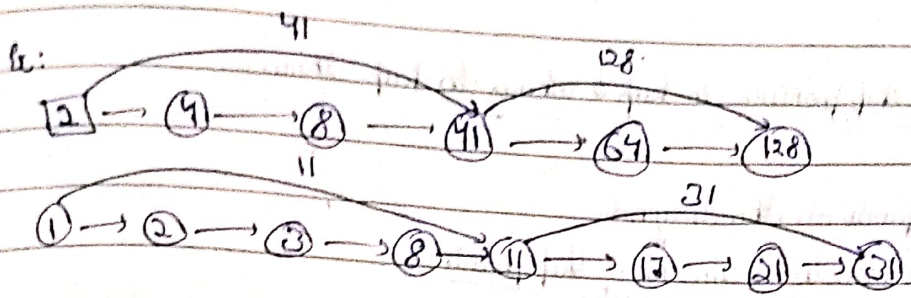
Stage 4

al → ϕ (revival → reviv)
ate → ϕ (activate → activ)
able → ϕ (adjustable → adjust)
⋮

These are not the only rules. (these are just examples).
There are many more rules that aren't mentioned here.

Skip pointers

Used to make posting list operations efficient
→ We saw the merge operation took O(X+Y) time.
→ We can make it faster by augmenting the list with skip pointers.
→ Skip pointers can be used to skip over all doc ids that do not feature in the search.



Modified merge algorithm using skip pointers:

```
Intersect (p1, p2)
  answer = [ ].
  while p1 ≠ NULL & p2 ≠ NULL
  if p1 → docID == p2 → docID
      answer.pb (p1 → docID)
          p1 = p1 → next
          p2 = p2 → next
  else:
      if p1 → docID < p2 → docID:
          if hasskip(p1) && p1 → skip → docID ≤ p2 → docID:
              p1 = p1 → skip.
          else
              p1 = p1 → next.
      else:
          if hasskip (p2) && p2 → skip → docID ≤ p1 → docID:
              p2 = p2 → skip
          else
              p2 = p2 → next.
```

How many skip pointers to keep & where to keep them?

Depends on the designer

Tradeoffs involved in no of skip pointers

More skips	Less skips
→ Shorter skip spans (so will usually choose skips)	→ Longer skip spans (so will usually not choose skips)
→ Increase comparisons	→ Fewer pointer comparisons.

Where to place the skip pointers?

If the list size is L then keeping $\sqrt{L}$ skip pointers evenly placed gives empirically the best performance (Moffat & Zobel 1996)

Easy if L is not growing frequently (would require pointer redistribution otherwise).

Note

Skip pointers are generally only used when postings list doesn't change frequently.

~~More memory means no~~ More skip pointers means more memory required.

Problematic when loading the list onto memory for merging. I/O cost might surpass the benefit of having skip pointers.

Phrase queries

Queries that enable us to do searches like ["Stanford University"] together as a unit. (instead of searching Stanford & University) AND

Boolean queries won't work here because Stanford AND University might bring up documents about a guy named Stanford that went to University & documents regarding other universities in Stanford.

Ways to resolve this problem

i) Biword indexing: Index bigrams (2 consecutive words)
Ex: "Friends, Romans, Countrymen" will have biword index entries "friend roman", "roman countrymen".
Each of these biwords are added to the dictionary.
Queries like "Stanford University Palo Alto" can be broken as "stanford university" AND "Palo Alto".

Issues: Phrases like "abolition of slavery" cannot still be processed. (as this cannot be broken to bigrams).

Extended biwords: Requires POS tagging of query & document.
Bucket all the nouns (N) and all articles & prepositions (X)
Extended biwords: NX^*N

Ex: catcher in the rye
N X X N

King of Denmark
N X N

Now the vocabulary contains all extended biwords, biwords & single words.

Caveats: → Index could easily blow up.

→ Queries with greater than 3 words can't be processed.

Note

If we're using extended biword indexing, we can't remove stopwords.

ii) Positional indexes (reconstruction of postings lists)

Postings list in a non positional index is just a list of DocID.

But in positional index, each postings is a docID + a list of positions.

Ex: We have a query "to be or not to be". → (to,1) is different than (to,5)
1 2 3 4 5 6.

Postings list changes as

to; 993427: → doc freq

<1: <7,18,33,57,86,231>,

2: <1,17,222>, → positions

4: <8,16,90,400,429,433>,

5: <363,367>,

7: <13,23>, ... >

docID

be, 178239

<1: <17, 25>

4: <17,191,291,430,434>

5: <14,19,101>, ... >

Next likely match at docID 4, word 429.

Merge method

i) Get postings list of to, be, or, not.

ii) Progressively intersect the postings list of words starting with to, be.

(Position list could be merged by a slight modification of merge algorithm (we check for difference instead of equality))

Bi word indexing used when bigrams are extremely frequent, otherwise we use positional index.

Multiword phrases with bigram indexing:

State of the art ≡ "State of" "AND" of the" "AND" "the art"

Document structure:

→ Some fields in the document need to be given special treatment

Ex: Dates, titles: Main titles/ section titles.

→ These fields are usually indexed separately.

One of the ways: "Extends lists"

Format: Skidus.

Precomputed scores

Postings list now contains Doc ID & also relevance score of that word for the document.

Ex: fish → [1, 3.6] → [32.2]

How does this help?

While merging, ignore postings with relevance less than a certain threshold. Reconstruct the postings list after removing these entries → smaller postings list ⇒ faster merge.

Suppose we query "tropical fish aquarium".

Some questions that come up are

i) what is the size of the result set for this query?

i.e for a query "a b c" how many docs contain all 3 of these terms.

Let f_a = frequency of "a" in the corpus. Similarly f_b & f_c.

Assuming a, b & c are independent & N is the number of tokens in the corpus,

Expected frequency of a, b & c together is:

Probability of a's occurrence = $\frac{f_a}{N}$, similarly for b & c it is $\frac{f_b}{N}$ & $\frac{f_c}{N}$.

probability of occurrence of a b c = $\frac{f_a f_b f_c}{N^3}$.

Expected no of documents = $N\left(\frac{f_a f_b f_c}{N^3}\right)$

$= \frac{f_a f_b f_c}{N^2}$

** Independence assumption leads to very poor results.

One solution is to use document frequency of bi word indices

$P(a \cap b \cap c) = P(a \cap b)\, P(c \mid a \cap b)$

↓

can be estimated using $\min(P(c|b)$ & $P(c|a))$

$P(a \cap b \cap c) = P(a \cap b) \min(P(c|b), P(c|a))$

$P(a \cap b) = \frac{\text{docfreq}(a \cap b)}{N}$ = ↳ no of documents.

$E(f_{a \cap b \cap c}) = N \times P(a \cap b) \times \min(P(c|b), P(c|a))$

$= f_{a \cap b} \times \min\left(\frac{f_{c \cap b}}{f_b}, \frac{f_{c \cap a}}{f_a}\right)$.

$\therefore f_{aq \cap tropical \cap fish} = f_{aq \cap tropical} \times \min\left(\frac{f_{aq \cap fish}}{f_{aq}}, \frac{f_{tropical \cap fish}}{f_{tropical}}\right)$.

Still tedious as requires bigram frequencies.

A simpler estimate

Let 26480 documents have the word aquatic. Select some random 3000 docs out of it. Let 258 of them have tropical, fish & aquarium all 3 phrases. Then we estimate about $\frac{258}{3000} \times 26480$ documents with have tropical, fish & aquarium.

Retrieved documents must not only be relevant to the query but the intents of the user (even if the query may not accurately represent the intent) ← i.e. information needs of the user.

∴ We require relevant retrieval & not exact retrieval.

↓

documents matching the query & documents matching perturbations in the query.

Hashtables not very useful because they don't help in finding minor variants / prefix searches which are necessary for relevant retrieval.

Two algorithms

Take log n time complexity for operations, which is not good enough.

~~Retrieval~~ Retrieving terms matching pro* wont:

Naive idea:

Query pro* & * wont & intersect.

Problem: Intersection is costly.

How bigram indices are more efficient compared to permuterm indices?

Consider a word like among. In permuterm index, it creates 5 entries but in bigram index, we just store it once (the unsorted index just stores pointer to the word). (Extra space for the bigrams is atmost 27^2)

Bigram indices faster permuterm indices because we can use hashtable instead of trees.

- Intersection of bigram index list faster than intersection of words ~~in~~ retrieved from trees (the case where we retrieve pro* & *ent & intersect) because of the sorted structure of lists (→ O(m+n) V/s O(m*n))