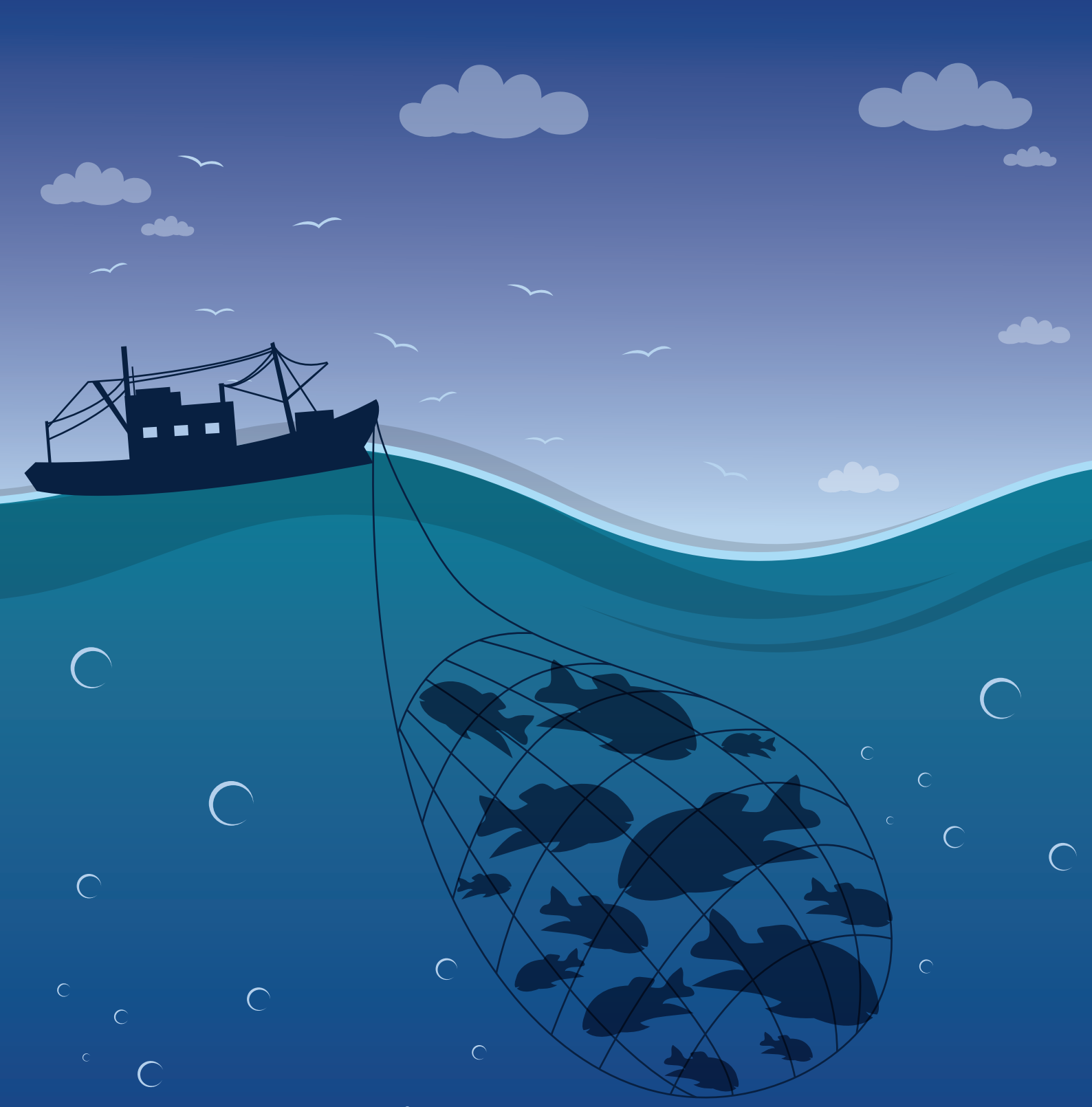


Web Information Retrieval

Edoardo Pugliesi

Web Information Retrieval = retrieve data/information from web resources

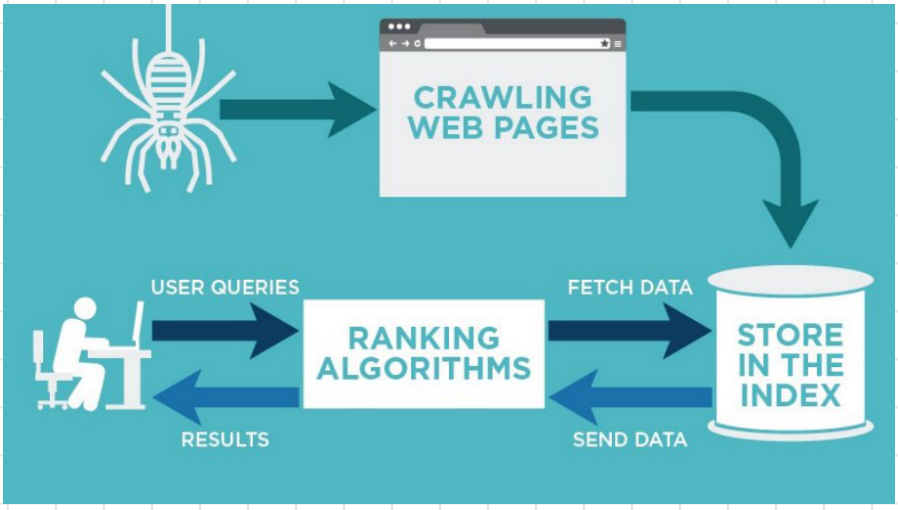
I.R. = retrieve information from a huge collection of unstructured docs (text) and analyze them
An example is what Google does during a web search: within milions of sites/articles it shows the ones related to user's query
What we always have:
1) huge collection of docs
2) Information need: what the user is looking for

On Web there are some factor that make things harder
- links
- Huge (Very) and distributed sources
- Different kinds of files
- web keep growing
- not homogeneous sources

COURSE OUTLINE
- collect a Web corpus (multiple documents)
- Pre-processing and organizing it
- Search for a document using a query
- Use web as platform to demand services

COLLECT A WEB CORPUS
Recursively explore all web pages iterating on links and collect informations: think of web as a graph with its necessities
This takes problems such as huge space requirement, computing power and competitiveness (if we use more machines for the search). Need to be sure to visit "all" pages to avoid information missing (Bias in data)

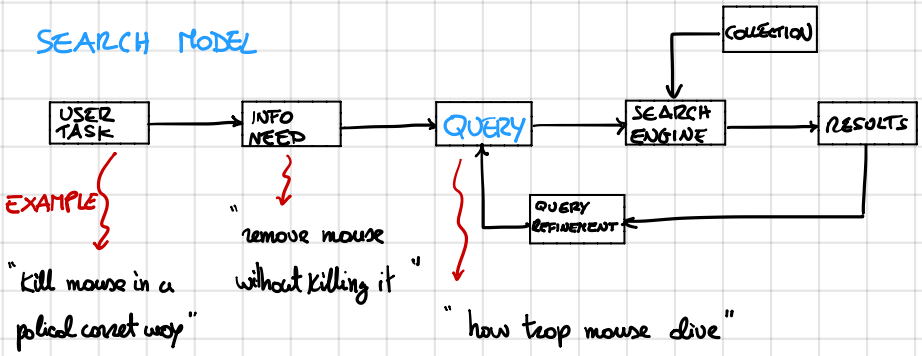
ORGANIZE A WEB CORPUS
Main goals: efficiency and relevance at same time
- Crowling: visit web corpus graph's nodes
- Indexing: organize nodes → efficiency
- Search Algorithms: how to solve user's information need using query → relevance (only on runtime ONLINE)
Relevance must be compared with authority of the page: pages must give user Real information



META - SEARCH : research using combinated results from multiple search - engines

COLLECTION : set of documents, not always static
↳ goal : collect documents that are relevant to user's information need

SEARCH MODEL



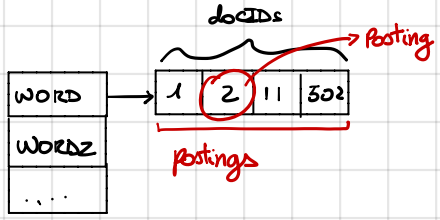
How to evaluate results:
- Precision: how many documents over the total are considered relevant to the user's information need
- Recall : how many documents, over the relevant ones, are shown (ex. First 10 links on Google)

More precision means less recall

Query based research use presence (or not) of keywords in documents : to do so we use Inverted Index : for every word we create a linked list with docIDs of documents that contain it.

Inverted index save space and time:
- every document has a docID that identify it
- avoid using fixed length array
- docIDs are sorted

hasmap / dictionary



A "normal" array, actually a matrix, with docIDs as keyword and words as postings is way bigger and sparse

How to build an inverted index:

documents to be indexed : Friends, Romans, countrymen

TOKENIZER

Token stream : Friends | Romans | Countrymen ...

linguistic module

Modified Token : friend | roman | countryman

Indexer

Inverte Index :

friend	2 → 4
roman	1 → 2 → 10 → 13
countryman	13

Note : often tokenization, we sort by word first then according to docIDs save doc frequency of every words and then sort postings.

TERM	docID	
WORD1	1	freq. = 2
WORD2	1	
WORD2	2	
WORD3	2	

... → WORD2 → 1 → 2

Ex.
WORD1 AND WORD2 : locate WORD1 and WORD2 posting and intersect them . Sorted postings guarantee more efficiency.
Complexity: $O(x+y)$ with x and y as postings lengths

This is an example of Boolean Query which follow the boolean query model : AND , OR , NOT (with other tools)
It's very precise : docs. match condition or not.
AND operand is very precise but recall is very low while OR has high recall and low precision making it hard to find a middle ground.

QUERY OPTIMIZATION

Consider a query that is an AND of n-terms.
Brutus AND Calpurnia AND Caesar → (Brutus AND Calpurnia) AND Caesar
Intersect the two smallest postings list grants next intersection to be more efficient (less postings to check)
In cases like (madding OR crowd) AND (ignoble OR strife) AND... we use frequency to estimate which OR must be first : the one which terms's frequency sum is lower
If we want a query to match "Stanford university" as phrase <term: docs> are no longer suffices cause we need a bi-term combining the two words.
We could index pair of terms : Friends, Romans, Countrymen → • friends romans • romans countrymen ⟹ too many dictionaries and false positive
We could save the position of term in docs :
- to : 2: 1, 17, 74 ; 4: 8, 16, 190, 429, 433 ; 7: 13...
- be : 1: 17, 19 ; 4: 17, 191, 291, 430, 434 ; 5: 14, 19...

But this use more space.

So we can find docs in which we have "to be" with an extra iteration. Using position we can even do proximity searches.

These methods can be used anyway in some cases and also combined , for example "name surname" can use bi-word method (ex search "Michael Jackson")

QUESTIONS :

- which format ? pdf , word , excel ...
- which language ?
- which character set ?

} classification problems : to identify the class of docs.

An index can contain docs with different format and language

TOKENIZATION

INPUT : "Friends , Romans and Countrymen"
OUTPUT : • Friends • Romans • Countrymen
A Token is an instance of terms grouped together as a useful semantic unit for processing : depends on how we want to process the index (by words , bi-words ecc)
After further processing they become index entries
Most simple strategy is splitting on white spaces , but "Finland's capital"? "Finland" AND "s" ? "Finlands" ? "Finland's" ?
"San Francisco is one or two tokens?
Similar problem with numbers : 3/20/91 and 20/3/91 is the same date but with different format → two different token or add "format" as meta-data to index entry.
Language : French → "L'ensemble" one or two token ? → L ? L' ? le ? → how to associate L' ensemble with un ensemble ?

Suppose we created the index: we now remove **stop words** like "the", "a" .... and create a separate list for them.
We need to **Normalize** words: "U.S.A" is equal to "USA" ⟶ unique dictionary entry.
Most of the times users doesn't use accent ⟶ we have to know how they like to write queries.
Synonyms and homonyms? car = automobile: we can search for car too when query contains automobile or rewrite to form equivalence-class terms indexing automobile under "car-automobile".
Need to to take care of spelling mistakes and so on...
Problems are a lot!

Lemmatization: reduce to base form. am, are, is ⟶ be
Stemming: reduce term to root before indexing. automate, automatic, automatic ⟶ automat
↳ **Algorith**: Porter's algorithm

- sses ⟶ ss
- ies ⟶ i
- ational ⟶ ate
- tional ⟶ tion

- words weight sensitive
- (m > 1) ENEMY:
  - replacement ⟶ replac
  - cement ⟶ cement

} Typical rules (there are a lot of stemmers)

Back to "AND query" we can improve merge efficiency using **jumps** in postings list:

- 2 — 4 — 8 — 41 — 48 — 64 — 128
- 1 — 2 — 3 — 8 — 11 — 17 — 21 — 31

we have to take common docID so we check one-by-one and confront them

When we reach 8 we match it and advance: we now have 41 and 11, but 11's successor is 31 which is smaller than 41 so we can skip to 31

Where do we put skips? Find a tradeoff.

DATA STRUCTURES

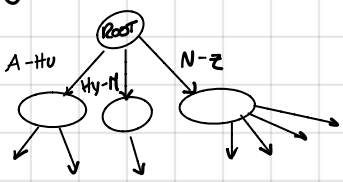
We use dictionary for inverted index. But which data structure we use?
Most common choices are hashtables and trees

Hashtable:

Pro: faster than tree finding postings list
Cons: - no prefix search Tolerant retrieval
- No easy way to find variants: judgement / judgment
- Need rehash if vocabulary keeps growing.

Tree

More usual type is B-Tree: self balancing tree which nodes have a number of child between [a, b] with a and b variable.
Pro: solve prefix problem
Cons: -Slower O(log M)
- Rebalance is expensive



WILD-CARD QUERIES

mon*: find all docs containing words beginning with "mon". Easy in B-Tree
*mon is way harder. → use an alternative tree for terms backwards
X*Y is an intersect of X* AND *Y → very expensive
Now we look up the posting lists.
More efficient way: permuterm index
Hello index under: hello$, ello$h, llohe, lohel, o$hell, $hello (permutations)
- X look up for X$ - X* → $X* - X*Y → Y$X*
- *X → X$* - *X* → X*
But 4 times bigger size (average)!
X*Y*Z → Z$X* → check every term if contains Y

BIGRAM (K-GRAM) INDEXES

We mantain an index for bigrams (or K-grams)
$m → all words that start with "m"
mo → all words that contain "mo"
→ mon* → $m AND mo AND on
we get "moon" too but it isnt a solution → use post filters

Spell Correction

- Correct indexed document
- Correct user query
} → check isolated word or context in which it is used

A language specific way is to use Soundex: phonetic equivalents.
There are many way to correct terms, for example we can search for terms with many K-grams in common with the query (using k-gram index), we can calculate the "distance" of the query with terms in dictionary (Levenshtein distone).

BSBI ALGORITHM

The steps we saw previously to create and sort an inverted index are done in memory (RAM) but this causes many problems:

- postings lists are incomplete until the end
- low space

Use disk (hard disk)? TOO SLOW! We use it only to store intermediate results

We use then an external algorithm which uses very few disk seeks (operations that involve disk e.g take back intermediate results) (more precisely <term,docID> pairs)

E.g. To sort 100 M postings we define 10 blocks of 10 M postings. We can easily fit a block in memory.

IDEA: for each block: ① accumulate postings ② sort in memory ③ write to disk
then merge the blocks (sorted)

This is called Blocked Sort-Based Index (BSBI)

This algorithm uses one more dictionary to map terms with termID (like docID) to reduce size

Note: BSBI works with <termID-docID> pairs → collect → sort → merge

SPIMI ALGORITHM

BSBI use a dictionary to map terms to termID that grow dinamically and this is a problem

Single Pass in-memory index solves this problem avoiding the use of the extra dictionary.

- Separated dictionary for each block ⟶ we create an inverted index for each block and then merge them
- Sort sort postings (they are already sorted anyway cause docIDs are incremental when received)

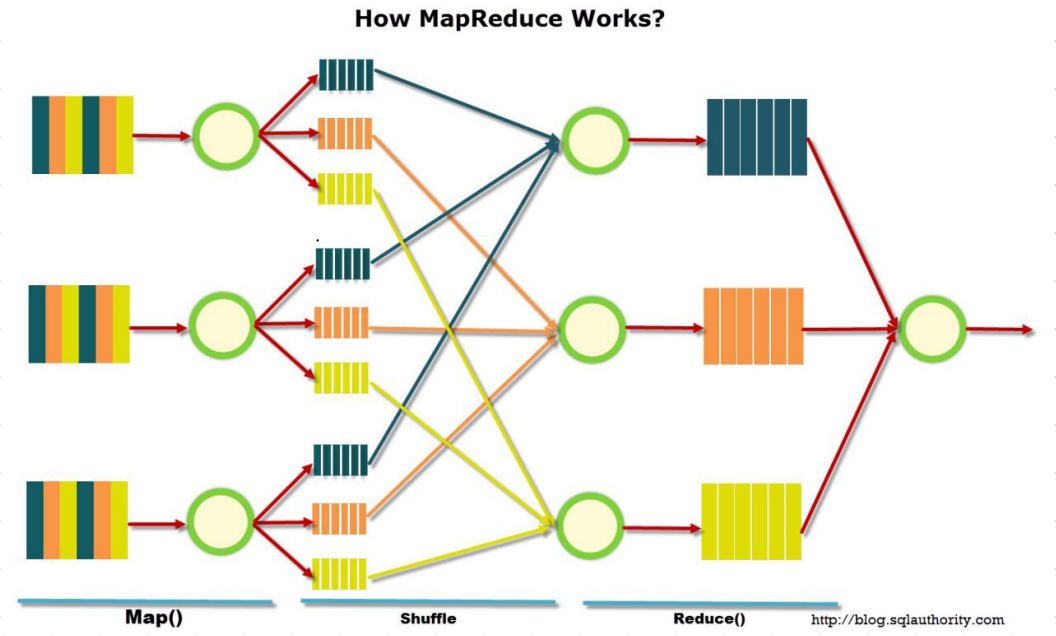
Very important

For web-scale indexing we (must) use distributed computer cluster. How?

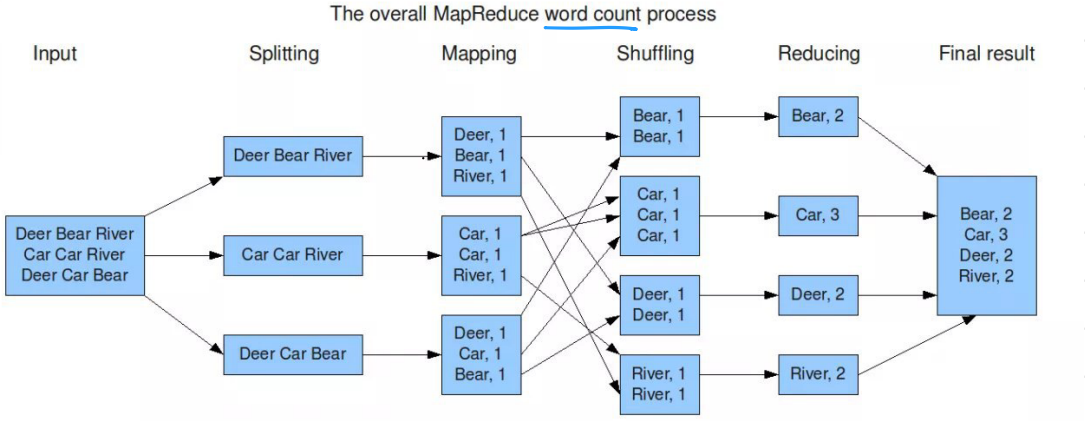
Montain a master machine considered "safe" which divide indexing into sets of parallel tasks assigned to other machines.

MapReduce is a framework that let task run in parallel

- Map: transforms an huge collection of documents (input) into a stream of <key,value> pairs, then it will group them by key (shuffle) (output)
- Reduce: takes in input Map's output and gives as result a set of distinct key and their corrisponding combined value (any mathematical function can be used)

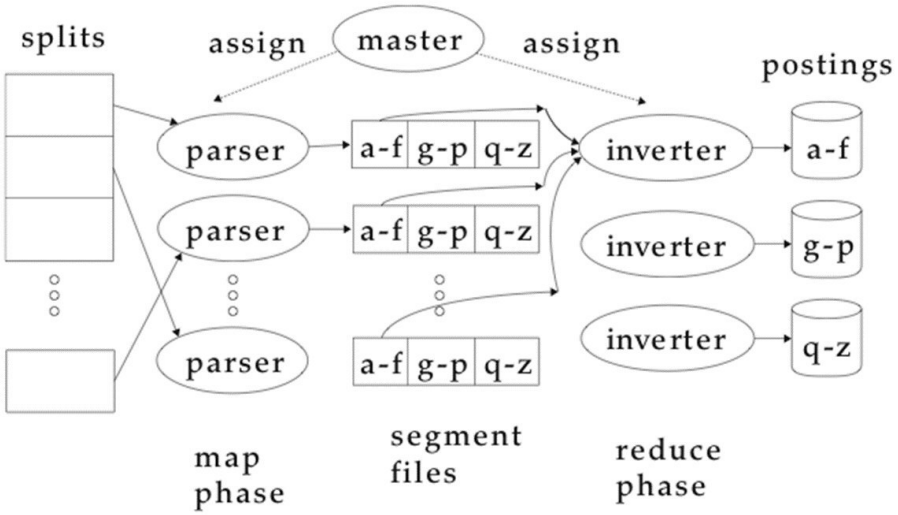


(particular example)



Nutch defines two type of machine: parsers and inverters and break document collection into splits (same as block we saw in BSBI / SPIM)

- Parsers: read file and emits <term, docID> pairs grouped in partition of j terms
- Inverter: collect these pair (only one group), sort and writes to posting list



We are assuming that collections are static which it's not true: posting must be dynamically modified.

The approach is simple:

- main big index on disk
- new docs go to small auxiliary index in memory
- search across both, merge results
- periodically merge auxiliary to main

Problems? Lot of merge and poor search performance during them

Logarithmic merge: amortizes the cost of merging index over time. It uses series of index each twice longer than the previous one. Smallest (Z_0) in memory and others ($I_0, I_1, I_2 \dots$) on disk: when the last one is full we create an other one. When Z_0 is full, write it on I_0 and empty Z_0. When Z_0 is full again and I_0 is too small, merge them in I_1 and empty both. Then again: we full every index before I_1 and then merge them together in I_1 ... And so on.

It use a binary number to save which index is full: e.g. 1011 (3 2 1 0) means I_2 has space, others are full.

$O(\log T)$ = # indexes with T = # posting read so for → query needs to merge $O(\log T)$ indexes → $O(T \log T)$ because each of T postings is merged $O(\log T)$ times, more efficient than $O(n^2)$ of auxiliary index method

COMPRESSION:

Prefacing decompression algorithms are fast, we want to compress index and posting lists to speed up the process e.g. keep index in memory and save space on disk.

- Lossy: discard some infos (e.g stop words) ⟶ what we "used" since now!
- Lossless: all infos are preserved

Heaps' law: relationship between collection size and vocabulary size (empirical law, but works in general)
⇒ $44 \cdot N^{0.49} \sim$ # terms (N = # tokens)

Zipf's law: we have vocabulary size now ⟶ we want to know which terms are more frequent or rare.
⇒ i^{TH} most frequent term has frequency cf_i proportional to $1/i$ (cf = collection frequency)
NOTE: more rare a term in a query is, more relevant it is for the document
$cf_2 = \frac{1}{2} cf_1$, $cf_3 = \frac{1}{3} cf_1$ ...

DICTIONARY COMPRESSION

Dictionary as array of fixed-width entries need 11.2 MB for pointers (1 year articles collection)
(20 bytes for term + 4 bytes for frequency + 4 bytes for pointer to posting list) · 400 000 = 11.2 MB
Problems: average length is 8 ⟶ space wasted
can't handle very long terms.

- Dictionary as string: bytes for frequency and pointers are still the same, but with only 3 byte we have a pointer to the position of term in dictionary string: e.g. 1 term1 6 term2 11 term3 ....
+ 8 bytes (on average) for term in string: (4+4+3+8) · 400.000 = 7.6 MB
- Dictionary as string with blocking: same as before but the dictionary-string is divided in blocks of k terms. We use an extra byte before every term to indicate the length: eg. ... 4 term 9 otherterm ...
k = 4: without blocks we use 3 bytes × 4 (terms) for pointers; now we use 3 byte for 1 pointer + 4 bytes for 4 length indicators ⟶ (4·3) − (3+4) = 5 bytes saved per block ⟶ (400 000/k) · 5 = 0.5 MB total
⟶ 7.6 MB becomes 7.1 MB
NOTE: look up of a term is slightly slower with blocking
Its possible to add front coding: eg. 8 automata 8 automata 9 automatic ⟶ 8 automat * a1 ◊ e2 ◊ ic
⟶ 5.9 MB

POSTINGS COMPRESSION

Posting lists total size is about 10 times larger than total dictionary.
docIDs are stored in increasing order.
Gaps: one idea is to store gaps instead of docIDs eg. COMPUTER → 283154, 283159, 283202 ⇒ COMPUTER → 283154, 5, 43 ..
(gap = 5, gap = 43)
gaps for frequent terms are much smaller than for rare terms so we have to make a division.

- Variable byte (VB) code:

docID	824	829
gap		5
VB	00000110 10111000	10000101

824 in binary is 1100111000 ⟶ we use 7 bits "blocks" + 1 bit
↳ 00000110 10111000 (0 and 1 are continuation bit)
1100111000 = 824
0 means that the number continues in next block

- Other VB: we can use other types of alignement: 32bit (words), 16 bits, 4bits (nibbles) etc.
- Gamma codes (bitwise code)
  - Unary code: n = n times 1 with one zero at the end ⟶ 3 = 1110

Gamma code uses length and offset of a gap G:
- offset = gap in binary without first bit eg 13 = 1101 ⟶ 101
- length = offset length eg 13 → 101 ⟶ 3

Then encode length in unary (1110)
Gamma code for 13 is concatenation of length in unary code and offset: 1110101

Length of offset is $\log_2 G$ bits, length of length is $\log_2 G + 1$ bits $\rightarrow$ total $= 2 \cdot \log_2 G + 1$ bits, 2 times the optimal encoding length.
This doesn't depend on distribution of gaps and its universal and parameter-free
Anyway machines have boundaries: 8, 16, 32 bit and this kind of compression and manipulation can be slow, making VB code a better choice sometimes

RANKED RETRIEVAL

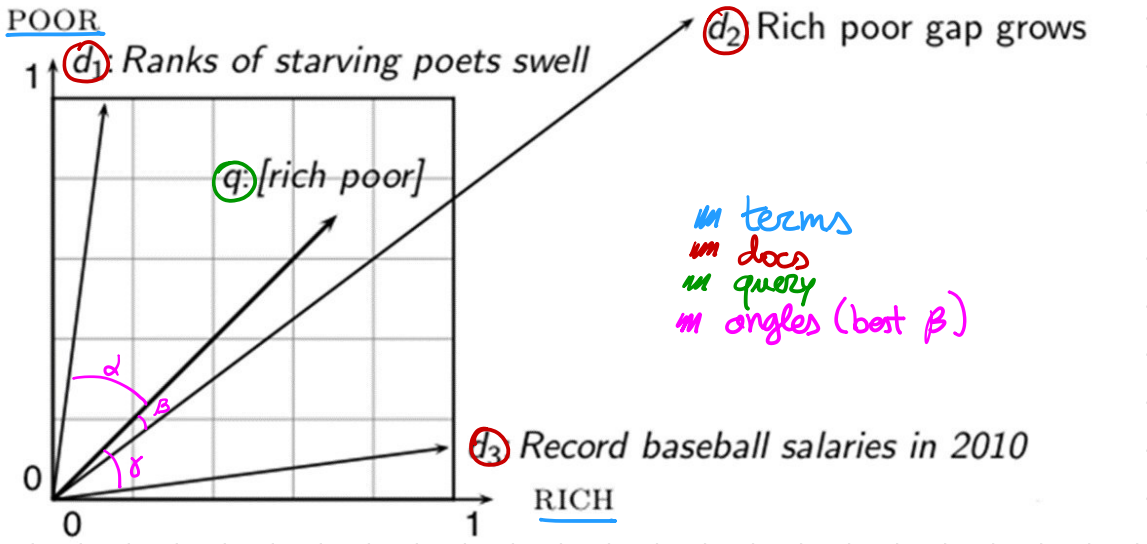
Boolean queries are too difficult to most of the users (not user-friendly)
They are not very optimized (too few or too many results: famine or feast).
With ranking we can give as result top documents, the most relevant ones
To do so we assign a score to each doc-query pair in $[0, 1]$ eg. one-term query if doc doesn't contains it $\rightarrow$ score $= 0$, more frequent the term is in a doc, the higher the score will be
There are a lot of alternatives.
Jaccard coefficient: A and B are two set $\neq \emptyset \rightarrow$ JACCARD$(A,B) = \frac{|A \cap B|}{|A \cup B|}$
eg. Q = "ides of March" D = "Caesar died in March" $\rightarrow J(Q,D) = 1/6$
Prob: doesn't take frequency in consideration.
Bags of words: each document is represented as a count vector with frequency for every term.

eg	Anthony and Cleopatra	The tempest
Anthony	157	0
Mercy	2	3

Prob: no order (John is quicker than Mary = Mary is quicker than John)
$tf_{t,d}$ = term freq. of term t in doc. d
How do we use it? A doc with tf = 10 is more relevant than a doc with tf = 1 BUT NOT 10 TIMES MORE RELEVANT (not proportional)
We add a weigth: $w_{t,d} = \begin{cases} 1 + \log tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$
Score for a doc-query pair: tf-matching-score$(q,d) = \sum_{t \in q \cap d} (1 + \log tf_{t,d})$ = sum of weigths of all terms which appear in both query and doc.

Rarity is extremely informative $\rightarrow$ we want to use frequency of term in the collection too $\rightarrow$ we want high weigths for rare terms (higher than frequent terms)
df_t = # of docs in which term t occurs
$idf_t = \log N/df_t$ = measure of informativeness (N = # doc in collection), inverse docs frequency
$\Rightarrow$ Score$(q,d) = \sum_{t \in q} tf \ominus idf_{t,d}$ (this is not a minus)
idf affects ranking of docs for query with at least two terms eg. "arachnocentric line": idf increases weigth of "arach." and decreases weigth of "line"
The best known weighting scheme is tf-idf $\rightarrow w_{t,d} = (1 + \log tf_{t,d}) \cdot \log N/df_t$. It increases with number of occurences within a doc and within rarity in collection.
We now modify the previous matrix using tf-idf weigth instead $\rightarrow$ docs are now represented as real-valued vector in high-dimensional space
To make it more complex we do the same for query and rank docs according to their proximity to the query.
Proximity = Similarity.
To compare two (or more vectors) the best idea is to check the angle between them $\rightarrow 0°$ = same content

$\Rightarrow$



terms
docs
query
angles (best β)

Convert angles to cosine. But how to compute it?

$$\cos(\vec{q},\vec{d}) = \text{sim}(\vec{q},\vec{d}) = \frac{\vec{q}}{|\vec{q}|}\cdot\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

two normalization

q_i = tf-idf of term i in query
d_i = tf-idf of term i in doc
$|\vec{q}|$ and $|\vec{d}|$ are $\vec{q}$ and $\vec{d}$ lengths

Often we use different weightings for query and docs.
Most used combo is lnc.ltn (notation ddd.qqq) that stand for:
- docs: logarithmic tf (l), no dtf weighting (n), cosine normalization (c)
- query: logarithmic tf (l), idf (t), no normalization (n)

Computing cosine scores

```
CosineScore(q)
1  float Scores[N] = 0
2  float Length[N]
3  for each query term t
4  do calculate w_{t,q} and fetch postings list for t
5     for each pair(d, tf_{t,d}) in postings list
6     do Scores[d] += w_{t,d} × w_{t,q}
7  Read the array Length
8  for each d
9  do Scores[d] = Scores[d]/Length[d]
10 return Top K components of Scores[]
```

We will add "pages relevance" later: some pages are more important than others.
Line 10 could be very long process for huge doc collection: can we speed up?
Let J = # docs with cosine ≠ 0 → we seek the K best of J
We construct in 2J operation a binary tree (heap) in which node's value > node's children values
Each search of best node needs 2 log J steps

Since we are not able to read users mind we can accept some error: taking K docs SIMILAR to top K docs its ok!
How: find a set A of docs with $K < A \ll N$, A does not necessarily contain top K docs but has many docs from top K → return top K in A
Basic algorithm consider docs containing at least one query term → improve: consider only high-idf query terms and docs containing many query terms.

CHAMPION LISTS

For every term t, compute r docs of highest weight → champion list at index build time.
r can be > than K!
At query time compute scores only for docs in champion list → Pick the K top amongst these.

We want top-ranking docs to be both relevant and authoritative: relevance is modeled by cosine scores, authority is query-independent! (let's call it $g(d)$ function)

NET SCORE

Net-score $(q,d) = g(d) + \cos(q,d)$

It can use other linear combination → now we seek the top K docs by net-score.
[$f(d,q) = \alpha g(d) + \beta \cos(q,d)$ with $\alpha + \beta = 1$ in real world]
How i get top K by net-score? Order all postings by $g(d)$, in this way (usually) top K docs are in the first ones and so we can stop computation early

- Champion list + $g(d)$-ordering: each term has a champion list of r docs ordered by $g(d) + tf\text{-}idf_{t,d}$

- High and low list: each term has an high list (champion list) and a low one (others), when traversing postings only traverse high list. If we get more than K docs, select the best K, if not use the low list too.

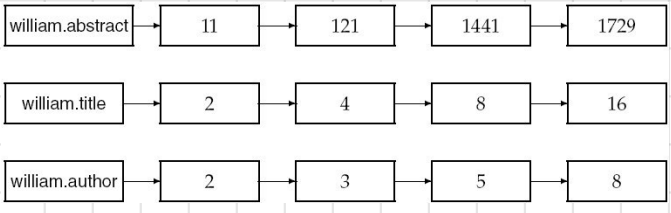
- Impact-ordered postings: we compute only docs with $wf_{t,d}$ high enough → order every posting list by $wf_{t,d}$ → all posting lists are now in different order. (a doc can be very relevant for a term and irrelevant for an other) → what we do?
  - ① Early termination
    When traversing postings stop after r docs or when $wf_{t,d}$ is too low.
    Take the union of each sets of r-docs, one set for each query term
    Compute the score for docs in union
  - ② idf-ordered terms
    When considering postings of a query, look at them by decreasing idf.
    As we update score contribution from each query term, stop if doc scores relatively unchanged (not very clear)

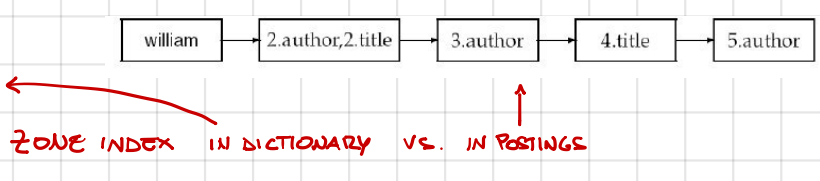
CLUSTER PRUNING

Preprocessing: pick $\sqrt{N}$ docs random called leaders, for all other (followers) compute nearest leader. Every leader has about $\sqrt{N}$ followers.
Query processing: given query Q, find its nearest leader L and seek k nearest docs in L's followers.

PARAMETRIC AND ZONE INDEXES

Some docs have specific term with special semantic called metadata (language, format, author ...)
- Field: e.g. YEAR = 1601 → postings for each field value → doc must be from year 1601
- Zone: e.g. title → build inverted index on zones as well to permit querying → e.g. find docs with "merchant" in title

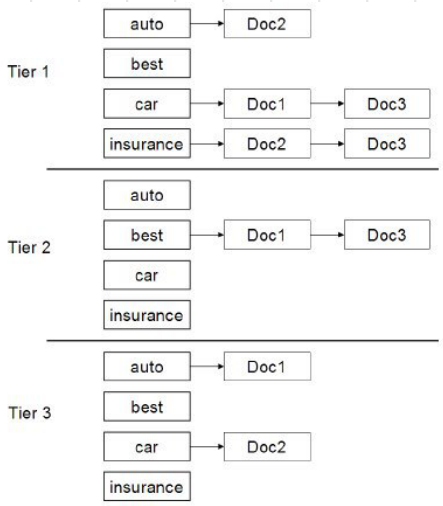




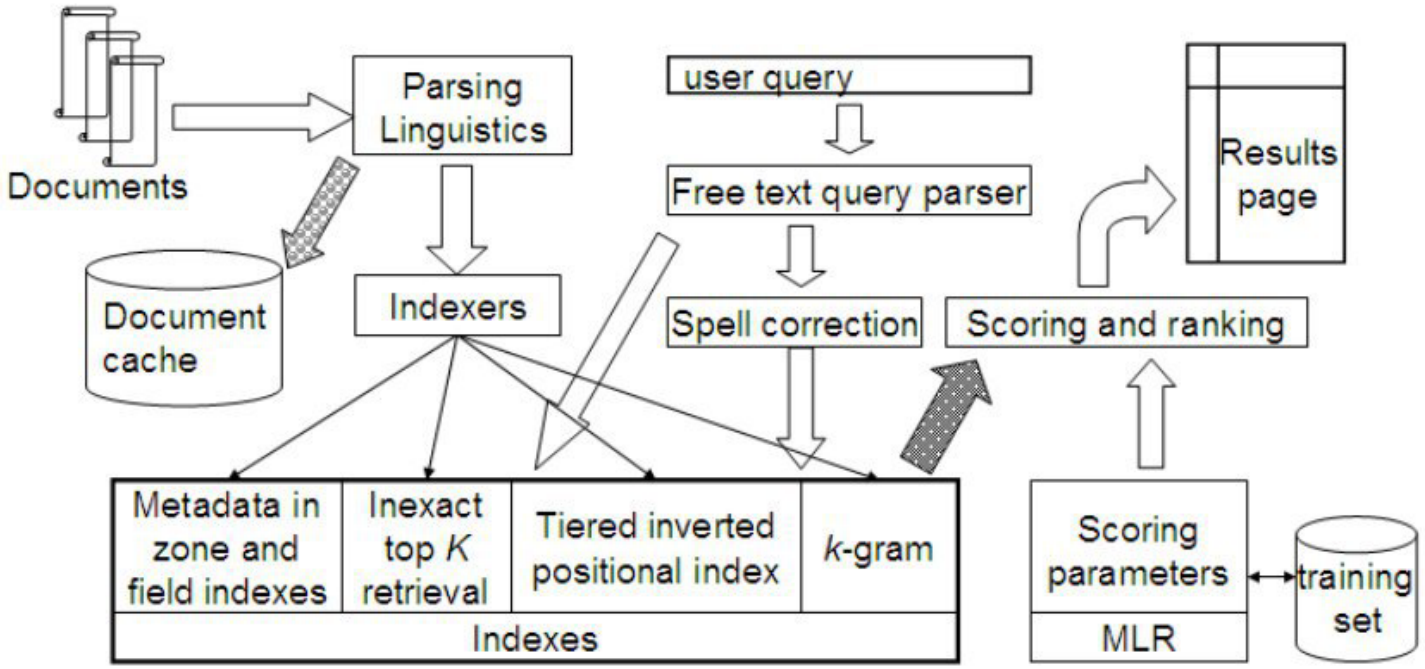
TIERED INDEXES

Break postings up into a hierarchy of lists, from the most important to the least one. Can be done with $g(d)$.

Start finding K docs in top tier, if not enough go lower.



PUTTING IT ALL TOGETHER



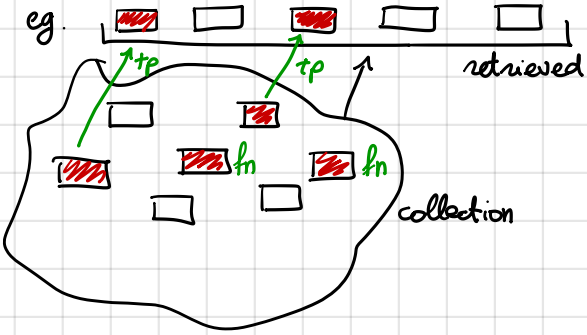
USER HAPPINESS

How good are query results? User happiness is not a "factor" as speed of response or size of indexes.
Need a way to quantify it! ⟹ RELEVANCE
Relevance measurement needs 3 element:

- A benchmark docs collection
- A benchmark suite of queries
- A (usually) binary assessment of either Relevant or Nonrelevant for each query and each doc.

Relevance is assessed relative to the information need (what the user wants for real in his mind) not the query.
A binary assessment is "compared" by Precision and Recall
Precision = fraction of retrieved docs that are relevant
Recall = fraction of relevant docs that are retrieved

eg.



	Relevant	Nonrelevant
Retrieved	tp true positive (2)	fp false positive (2)
Not Retrieved	fn false negative (2)	tn true negative (0)

Precision - P = $tp / (tp + fp) = 2/5$
Recall = R = $tp / \underbrace{(tp + fn)}_{\text{relevant items total}} = 2/(2+2) = \frac{1}{2}$

⟹ Precision@K = precision at first K docs
Same for Recall@K

- Measure for a search engine

How fast does it index? (e.g. bytes/hour) How fast does it search? (e.g latency as function of queries/second) How much does a query cost? (in dollar)
All those are measurable. But the key measure is user happines we saw before. It includes speed of response, user-friendly UI and ofcourse Relevance (being free maybe more important)
The user change depending on the type of search engine and with it also change the measure used to evaluate his happiness.
eg. Ecommerce: user = buyer, success = buyers buys something, measure = time to purchase after query, fraction of "conversions" from searchers to buyers
Back to relevance remember that given a query, a doc can match it very well but at some time can be not related to the user's information need!

About relevance, usually search engines make a tradeoff between recall and precision, lets call it (F).
$F = \frac{1}{\alpha/P + (1-\alpha)/R} = \frac{(\beta^2+1)PR}{\beta^2 P + R}$ with $\beta^2 = \frac{1-\alpha}{\alpha}$, $\alpha \in [0,1]$, $\beta^2 \in [0,\infty]$ → harmonic mean: $\frac{1}{F} = \frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)$
The most frequently used combination is $\alpha = 0.5$ and $\beta = 1$ called balanced F → $(F_1) = \frac{2PR}{P+R}$
Why we dont use accuracy = $(TP+TN)/(TP+FP+FN+TN)$? Cause TN is very huge and a system tuned to maximize accuracy will give a lot of false positive docs (even only FP)
Why harmonic mean? We want to punish bad performance either on precision or recall, and when we have numbers that differ greatly, the harmonic mean is closer to their minimum than aritmetic or geometric mean
Example on next page

To explain, consider for example, what the average of 30mph and 40mph is? if you drive for 1 hour at each speed, the average speed over the 2 hours is indeed the arithmetic average, 35mph.

However if you drive for the same distance at each speed -- say 10 miles -- then the average speed over 20 miles is the harmonic mean of 30 and 40, about 34.3mph.

The reason is that for the average to be valid, you really need the values to be in the same scaled units. Miles per hour need to be compared over the same number of hours; to compare over the same number of miles you need to average hours per mile instead, which is exactly what the harmonic mean does.

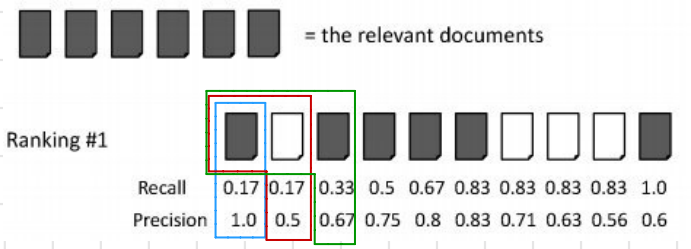
Precision and recall both have true positives in the numerator, and different denominators. To average them it really only makes sense to average their reciprocals, thus the harmonic mean.

share improve this answer

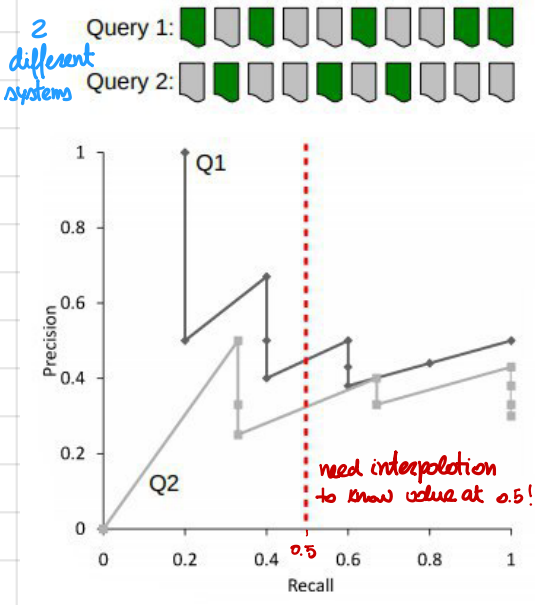
answered Oct 14 '14 at 14:54

Sean Owen
58.5k • 18 • 125 • 159

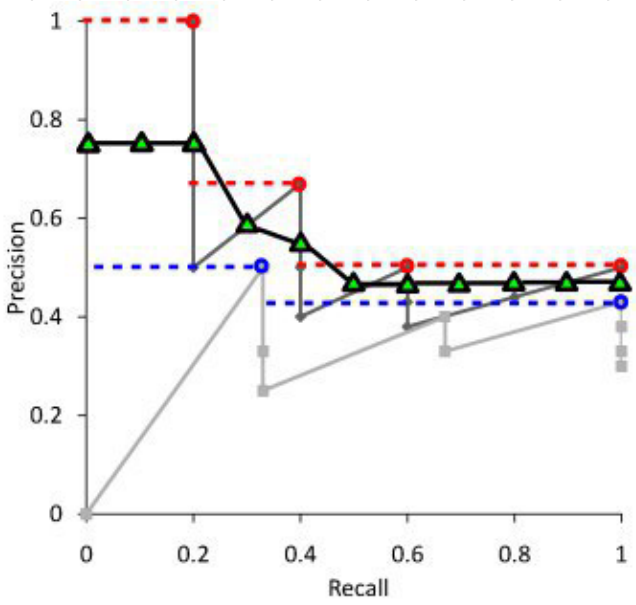
Precision/recall/F one measure for unranked sets. For ranked list just compute the set measure for each "prefix": top1, top2, top3 ... results giving a precision-recall curve



How to compare systems' curves? Interpolation!



⇒



On average precision drops as recall increases and so we define interpolation to preserve monotonicity

M.A.P. : mean average precision, its a single-number metric most frequently used in research papers

avg. precision query 1 = (1 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62 ┐
avg P.q.2 = 0.44 ┘→ M.A.P = (0.62 + 0.44)/2 = 0.53

these are precision@K corresponding to relevant doc

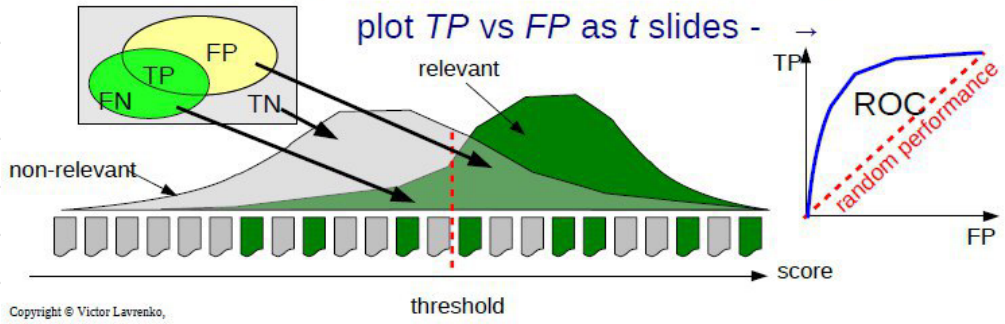
D.C.G. : discounted cumulative gain, its a measure of ranking quality.

Given n docs each has rating $r_1, r_2 \dots r_n$ (in ranked order) → $CG = r_1 + r_2 + \dots + r_n$

$DCG = r_1 + r_2/\log_2 + r_3/\log_3 + \dots + r_n/\log n$

ERRORS CLASSIFICATION

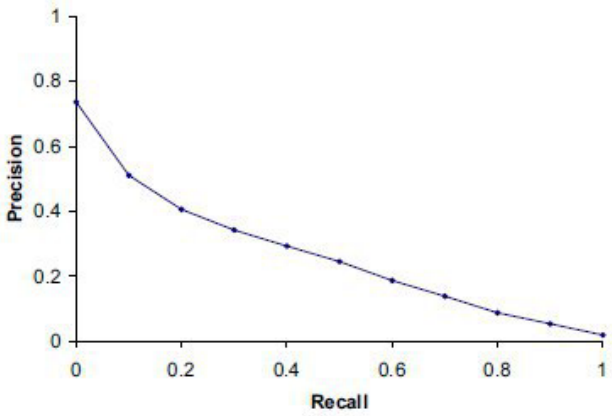
- False Pos. rate: Pr ($x > t \mid -$) = $FP / (FP+TN)$
- False Neg. rate = Pr ($x < t \mid +$) = $FN / (TP+FN)$
- True Pos. rate = Pr ($x > t \mid +$) = 1 - False Neg.
- Receiver Operating Characteristic (ROC):



Relevance assessment are usable only if consistent: how can we measure the consistency or agreement between who judge a doc relevant? → Kappa measure.

$K = \frac{P(A) - P(E)}{1 - P(E)}$ with $P(A)$ = percentage of agreement and $P(E)$ = probability to agree at random

$[\frac{2}{3}, 1]$ is considered acceptable.



one of the most common

- This curve is typical of performance levels for the TREC benchmark.
- 70% chance of getting the first document right (roughly)
- When we want to look at at least 50% of all relevant documents, then for each relevant document we find, we will have to look at about two nonrelevant documents.
- That's not very good.
- High-recall retrieval is an unsolved problem.! what to do? →

- **Interactive Relevance Feedback**: make a query, take the first results set and ask user to select relevant and nonrelevant docs. The user will see the result summaries (e.g Google) and will click docs that he thinks are relevant → improve overall performance
  Best method: Rocchio feedback
- **Query expansion**: improve retrieval result by adding synonyms / related term to the query. Used for example on one-term query.

The main idea is to do a "local" on-demand analysis for an user query with relevance feedback and a "global" one (e.g of collection) to produce thesaurus and use it for query expansion.

PART 1: the user issues a (short, simple) query, get a first set of result and mark the docs as relevant or non relevant (assumption). Then search engine computes a new representation of information need, hopefully better, and runs a new query with new (better) results

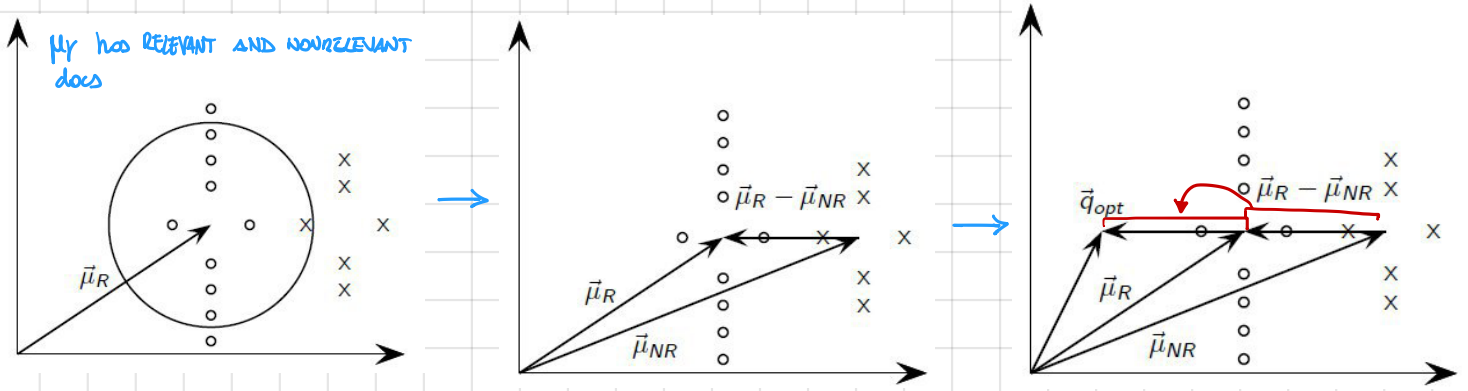
Centroid is the center of mass of a set of points that summarize their characteristics → $\vec{\mu}(D) = \frac{1}{|D|}\sum_{d\in D}\vec{v}(d)$

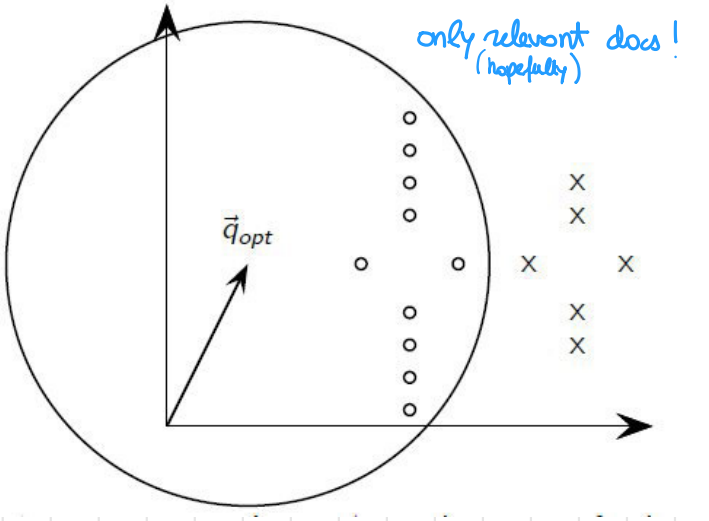
Remember we represent docs as vector/point in an highdimensional space.

Rocchio algorithm chooses the query $\vec{q}_{opt}$ defined as $\vec{q}_{opt} = \arg\max_{\vec{q}}[sim(\vec{q}, \mu(D_r)) - sim(\vec{q}, \mu(D_{nr}))]$ → difference btw q and D_r / D_{nr}

→ $\vec{q}_{opt}$ is the vector that separates relevant and nonrelevant docs maximally

with additional assumptions: $\vec{q}_{opt} = \mu(D_r) + [\mu(D_r) - \mu(D_{nr})]$





This is the dumb version of Rocchio algorithm
From now on we will use the SMART version used in practice:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\mu(D_r) - \gamma\mu(D_{nr})$$
$$= \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{d_j\in D_r} d_j - \gamma\frac{1}{|D_{nr}|}\sum_{d_j\in D_{nr}} d_j$$

α, β, γ are weights used for first raw query (α) and relevant and non relevant docs (β and γ)

Positive feedback (β) is more valuable than negative (γ) and many systems only allow positive feedback ($\gamma = 0$)

Relevance Feedback has 2 assumptions:
- A1: user knows the terms in collection for first query
  ↳ violation: mismatch of user's vocabulary and collection one (e.g. cosmonaut / astronaut)
- A2: relevant docs contain similar terms.
  ↳ violation: docs with different terms but some query concept are not helped by the feedback

Ofcourse "edited" query is better than raw one → a fair evaluation must be on docs not yet judged by user
Alternative: user revises and resubmits query (what a normal user does rigth now IRL)
Problems: expensive to process, users are reluctant to give feedback
⟹ Pseudo-Relevance Feedback: ① retrieve a ranked list of hints for the user's query ② assume top k docs are relevant ③ do relevance feedback.
Works very well on average but several iteration can cause query drift: no longer "close" to the query.

PART 2: in global query expansion, the query is modified based on some global query-indipendent resource.
A publication or DB that collect synonyms is called thesaurus: HOSPITAL → MEDICAL
- manual: maintained by editors
- automatic: generated automatically based on statistics (distribution of words)
Or can be used query-equivalence based on query log mining.
Manual th. is expensive and usually used in specialized search engines for science and engineering.
Automatic th. has a fondamental notion: similarity between two words
① two words are similar if they co-occur with similar words e.g. "car" ≈ "motorbyke" cause they both occur with "road", "gas", "plate".
② two words are similar if they occur in a given grammatical relation with the same words.
e.g. "apples" and "pears" are similar cause you can "harvest", "eat", "peel" both
① is more robust ② more accurate
Anyway query logs is the main source of query expansion: e.g. user searching "flower pics" and user searching for "flower clipart" both click on same link → "flower pics" and "flower clipart" are potential expansions of each other.
e.g. user issues "herbs" before than "herbal medicine" → "herbal medicine" is potential expansion of "herb"

TEXT CLASSIFICATION / CATEGORIZATION

Classification means distinguish relevant doc within collection ($\neq$ ranking)

- STANDING QUERY: a query, used for classification, that is run periodically to retrieve new relevant doc according to an information need.

e.g. Google Alert → Q = "bike" AND "yamaha" → notify new docs relevant for yamaha bikes.

Classification is used for spam filtering too: when you mark an email as spam, the system tries to create a query which describes it and use it to find new spam (+ training set for machine learning)

RECAP

- BAYES THEOREM → $P(B|A) = \frac{P(A \cap B)}{P(A)} = \frac{P(A|B)\,P(B)}{P(A)}$
- $P(A_1, A_2, A_3, A_4) = P(A_1|A_2,A_3,A_4) \cdot P(A_2|A_3,A_4) \cdot P(A_3|A_4) \cdot P(A_4)$

Given a rappresentation of a doc, we want to identify its class (category) → we need a classification function called classifier.

Classification methods:

- Manual: accurate when done by expert, difficult and expensive to scale
- Hand-coded rule-based: need to build very good rule (code) to categorize docs
- Supervised learning: what we said for spam, data set can be built by amateurs (no real experts)

Keeping the last method in mind we define the probabilistic relevance feedback: if user has told us some relevant and some nonrelevant docs we can build a probabilistic identifier

e.g. $P(\text{"dog"} \mid \text{Relevant}) \approx \frac{N_{r,dog}}{N_r} = \frac{\text{relevant docs with dog}}{\text{total relevant docs}} \rightarrow P(\text{"dog"} \mid NR) \approx \frac{N_{nr,dog}}{N_{nr}}$

Bayes' rule for text classification: $P(c,d) = \frac{P(d|c)\cdot P(c)}{P(d)}$

$P(c)$ = prob. that the doc i pick is class c

Assuming $d = \langle x_1, x_2, x_3 \ldots \rangle$ in position order then $P(d|c) = P(x_1, x_2, x_3 \ldots | c)$ → very complicated!

To "semplify" this, we assume that terms probability are indipendent: $P(d|c) = \prod_i^n P(x_i|c)$ but this is still on huge number because we still keep track of position.

Then: $\forall t\ P(x_i = t|c) = P(x_j = t|c)\ \ \forall i,j \neq i$

In this way "I go home" and "go I home" are computed in the same way because we dont keep track of position: maybe too much semplification but its ok!

$$\Rightarrow \frac{P(d|c)\,P(c)}{P(d)} = \frac{\prod_i^n P(x_i|c)\,P(c)}{\prod_i^n P(x_i)}$$

= → indipendent variables and position

What we want is something to maximize this probability:

$$\gamma(d) = \arg\max_{c \in C} \frac{\prod_i^n P(x_i|c)\,P(c)}{\prod_i^n P(x_i)} = \arg\max_{c \in C} \prod_i^n P(x_i|c)\,P(c)$$

$\prod_i^n P(x_i)$ → doesn't depend on classes

$\forall c: \hat{P}(c) = \frac{N_c}{N}$

$\forall t, \forall c: \hat{P}(t|c)$ = occurrences of term t over total number of terms in all docs of class c, easily done during dictionary creation

Problem: if $P(x_i|c) = 0$!!! (e.g. a misspell)

$\hat{P}(t|c) = \frac{N_{t,c}}{\sum_n N_{t_i,c}} \Rightarrow \hat{P}(t|c) = \frac{N_{t,c}+1}{\sum_n (N_{t_i,c}+1)}$ adding 1 doesn't make any problem even with a zero probability

MULTIVARIATE BERNOULLI MODEL

$P(d|c) = P(x_1, \dots, x_n|c)$ but in this case every doc is a binary vector with length $n = |V|$ (dictionary)

We can make some assumption as before:

$P(d|c) = \prod_n P(x_i|c)$ how many docs over the total have x_i in them (different from before!)

Naive Bayes (the edited version of Bayes' rule) is widely used for spam filtering and it's not so weak as someone could think: in fact it's very fast and robust and has low storage requirements

Note: evaluation must be done on test data that are indipendent of training data, sometimes it's possible to do a cross-validation (averaging results between multiple training and test sets)

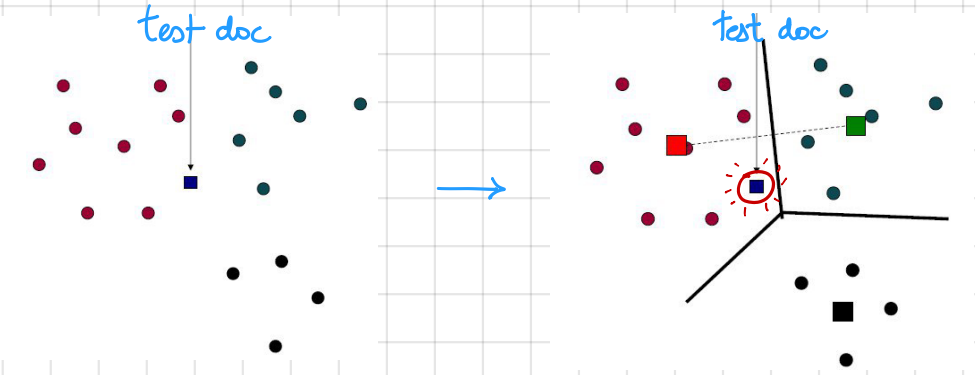
Evaluation is done according precision, recall, F_1 and classification accuracy ($r/n = \frac{\text{docs correctly classified}}{\text{total docs}}$)

Remember we are working with docs represented as vector in high dimensional space. How can we do classification?

Premise 1: docs in same class form a contiguous region in space.

Premise 2: docs from different classes don't overlap (much)

Learning a classifier means build surface to delineate classes in the space



How to find a good separator?
Use centroid and classify docs according to nearest centroid (Rocchio)

Rocchio defines line or hyperplan as $\sum_i^n w_i d_i = \theta$

$\vec{w} = \vec{\mu}(c_1) - \vec{\mu}(c_2)$

$\theta = 0.5 \cdot (|\vec{\mu}(c_1)|^2 - |\vec{\mu}(c_2)|^2)$

how? click here → https://youtu.be/eEEPAeweKqs?t=2516

Rocchio is little used outside text classification and in general is worse than Naive Bayes but cheap to train and test docs.

$\Rightarrow \boxed{x^T \vec{w} = \frac{1}{2}\theta} \Rightarrow$ Rocchio is a linear classifier

Why Naive Bayes is a linear classifier too?

(n_d = # terms in doc d)

We want to estimate $\hat{P}(c|d) \propto \hat{P}(c) \cdot \prod_{1}^{n_d} \hat{P}(t_k|c)$ where the probability to find t_k in class c is the same (for every same t_k) regardless of position (only in the model, not IRL)

Assume also $\hat{P}(\bar{c}|d) \propto \hat{P}(\bar{c}) \prod_{1}^{n_d} \hat{P}(t_k|\bar{c})$

When we try to classify a doc we are doing the following: odd ratio

$$\frac{\hat{P}(c|d)}{\hat{P}(\bar{c}|d)} = \frac{\hat{P}(c)}{\hat{P}(\bar{c})} \prod_{1}^{n_d} \frac{\hat{P}(t_k|c)}{\hat{P}(t_k|\bar{c})}$$

We could decide which class is doc d checking odd ratio: c if $\gg 1$, $\bar{c}$ otherwise, but for numerical reason we use logarithm

$\log(A \cdot B) = \log(A) + \log(B)$ applied 2 times

$$\log \frac{\hat{P}(c|d)}{\hat{P}(\bar{c}|d)} = \log \frac{\hat{P}(c)}{\hat{P}(\bar{c})} + \sum_{1}^{n_d} \log \frac{\hat{P}(t_k|c)}{\hat{P}(t_k|\bar{c})}$$

e.g. "a rose is a rose" $n_d = 5$ $|V| = 3$

$$= \log \frac{\hat{P}(c)}{\hat{P}(\bar{c})} + \sum_{i \in V} x_i \log \frac{\hat{P}(t_i|c)}{\hat{P}(t_i|\bar{c})}$$

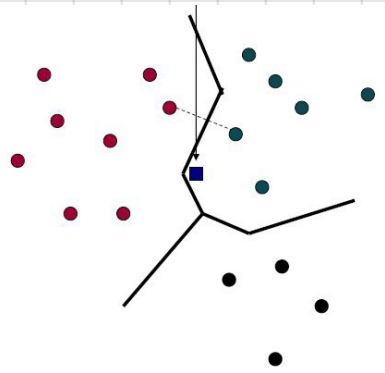
x_i = # occurrences of term i in d (e.g. $x_{rose} = 2$)

cause we said probability is position indipendent

$$0 \leq \underbrace{\log \frac{\hat{P}(c)}{\hat{P}(\bar{c})}}_{\theta} + \sum_{i \in V} x_i \underbrace{\log \frac{\hat{P}(t_i|c)}{\hat{P}(t_i|\bar{c})}}_{w_i} \Longrightarrow \sum x_i w_i \geq -\theta$$

KNN - K Nearest Neighbors Classification

To classify doc d we define its k-neighborhood as the k nearest neighbors and pick the majority class label in this neighborhood: for larger k $P(c|d) \approx \#(c)/k$



Voronoi diagram

As we can see this doesn't depend on a linear classifier!

Usually k is odd to avoid ties; 3 and 5 are most common

Naively finding nearest neighbors requires a linear search through $|D|$ docs in collection that is the same as finding the k best retrievals using the test doc as a query to a database of training docs $\longrightarrow$ use inverted index.

Pro: no training necessary, scales well, in most cases more accurate than NB and Rocchio (Bayes optimal)

cons: expensive at test time, small changes can have ripple effect.

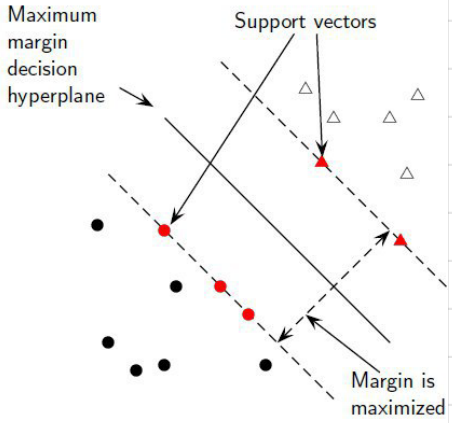
What we do with multiple-class docs? we have two types of answers:

- 1-of classifier: get only the most accurate class
- any-of classifier: get a vector composed by scores given to each class (for a given doc) based on "accuracy"

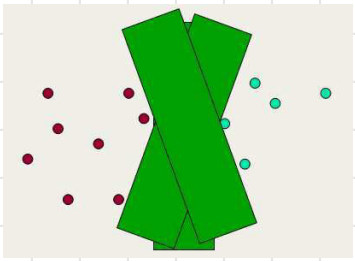
learning algorithms for vector space classification can be devided in two types: simple (N.B. Rocchio, KNN) and iterative (Support Vector Machines, Perceptron) that usually are the best choice. The simple ones often do only one linear pass.

SVM - Support Vector Machines

We still are in vector space (as previous algorithms) but we have now large margin classifier: we want to increase the division region btw classes and aim to find a separating hyperplane that is maximally far from any point in the training set.
In case of non-linear-separability we accept some points as noise (errors).



Why maximize the margin? Because points near the decision surface are less confident, uncertain classification decision and at the same time gives a classification safety margin respect to errors and random variation



All points $\vec{x}$ on the hyperplane satisfy $\vec{w}^T\vec{x} + b = 0$ (b is what we called θ)

- **Find boundary vector w that satisfies conditions:**

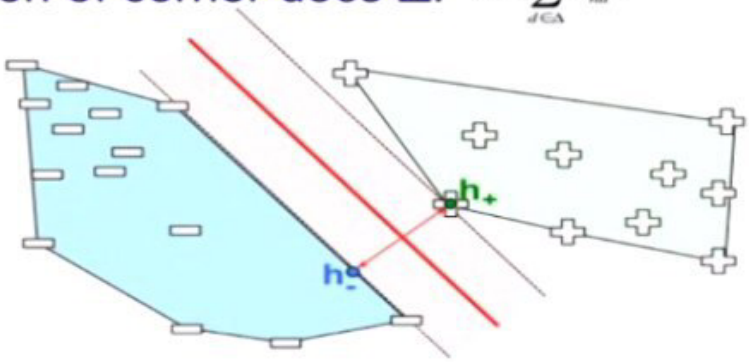
$\sum_i w_i d_{j,i} \geq +1$ all positives get scores ≥ +1
$\sum_i w_i d_{j,i} \leq -1$ all negatives get scores ≤ -1
$\min\|\vec{w}\|$ maximum margin around w

$$\vec{w} = \overbrace{\sum_{j\in+} \alpha_j \vec{d}_j}^{\text{"centroid" of positives}} - \overbrace{\sum_{j\in-} \alpha_j \vec{d}_j}^{\text{"centroid" of negatives}}$$

+/- docs have "weights"

- similar to centroid / Rocchio
- some docs more important
- most don't matter: $\alpha_j = 0$

- **Convex hull: polytope around all positives / negatives**
  - any point **h** is a linear combination of corner docs Δ: $\vec{h} = \sum_{d\in\Delta} \alpha_{hd} \vec{d}$
  - find two nearest points $\mathbf{h_+}$, $\mathbf{h_-}$
  - margin must be $\|\mathbf{h_+} - \mathbf{h_-}\|$
  - boundary must be half-way
  - perp. to: $\vec{w} = \vec{h}_+ - \vec{h}_- = \sum_{d\in+} \alpha_d \vec{d} - \sum_{d\in-} \alpha_d \vec{d}$



Note: polytopes are the lines around positive/negative docs.
e.g. 2 classes $y_i = +1$ and $y_i = -1$ $\longrightarrow$ $f(\vec{x}) = \text{sign}(\vec{w}^T\vec{x} + b)$
If a new doc is classified within the margin is possible to return "don't know" or use probability function to assign a class.

What if data is not linearly separable?

Allow to make errors and increase margin and reduce overfitting (accept the cost for every errors)

We introduce a Slack variable ξ_i, a non-zero value that takes in consideration these errors: the sum of ξ_i gives an upper bound on the number of errors.

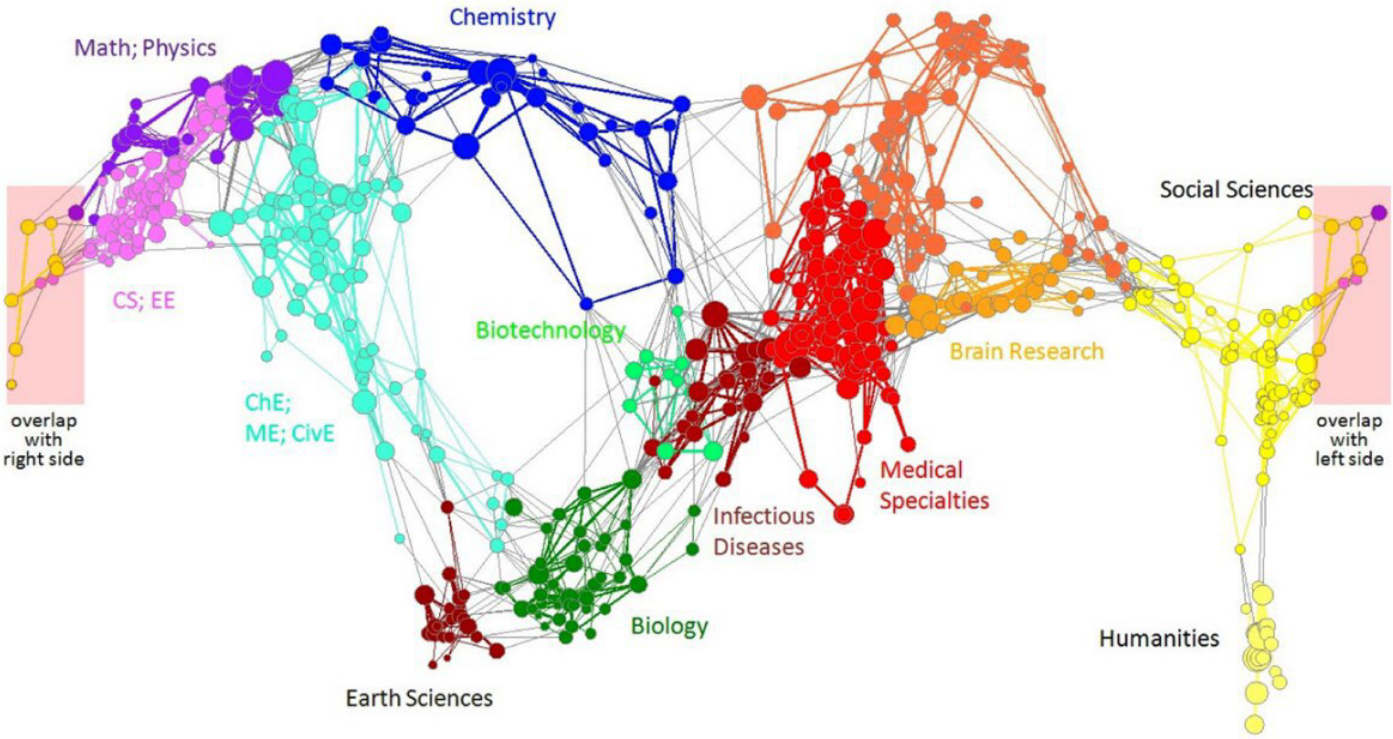
Ways to use SVM (for one-of classification)

1. Train and run K classifier and then select the class with the highest confidence
2. Build $K(K-1)/2$ 1vs1 classifier and choose the class selected by the most classifier: more classifier but less time of training due to data set is much smaller
3. Structured prediction

CHOOSING CLASSIFIER

- How much training data is there currently avaible?
- No labeled training data? → Use hand written rules (even more complicated than boolean)

LINK ANALYSIS



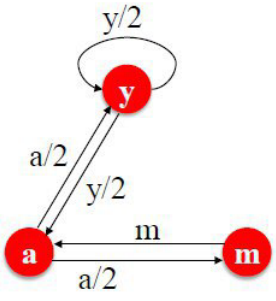
Challenges:
- Web contains many sources, who trust? → Trustworthy page may point to each other
- What is the "best" answer to query e.g. "newspaper"? → Pages that actually know about newspapers might all be pointing to many "newspapers.

All webpages are not equally "important" and for this reason we will rank the graph's nodes by the link structure!

Importances of a node:
- Page Rank
- Hubs and Authorities (HITS)
- Topic-specified (Personalized) Page Rank
- Web Spam detection Algorithms

Idea: links as vote. More links more points. Link to/from important pages count more. Recursive!

e.g Page j with importance r_j has n out-links, each get r_j/n votes. Page j importance is the sum of in-links. Rank $j = r_j = \sum r_i/d_i$ $(d_i = n_i)$



"Flow" equations:

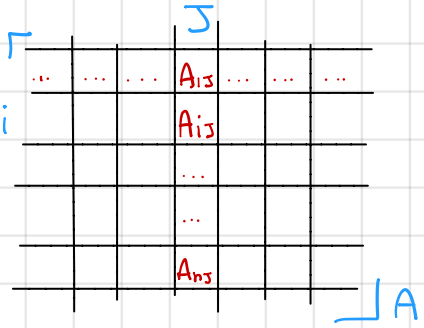
$r_y = r_y/2 + r_a/2$

$r_a = r_y/2 + r_m$

$r_m = r_a/2$

No unique solution! Gaussian elimination is not suitable for web-size graphs.

New solution: Stochastic adjacency matrix. ⟹



if $i \to j$ $A_{ij} = 1$ else $A_{ij} = 0$ $\longrightarrow r_j = \sum \frac{A_{ij} \cdot r_i}{d_i} = \sum M_{ij} \cdot r_i$

r^T (row-vector) $= r^T M$

$\forall i \sum_{j=1}^{n} M_{ij} = 1 \longrightarrow$ stochastic property (row stochastic)

Note: $r^T = r^T M$ means r^T is an eigenvector with eigenvalue 1

How to solve $M \cdot r^T = r^T$? Power Iteration Method

To remplify formula we will start from $r = M^T r = A r$ (not previous A, A is column stochastic now)

Initialize $r^{(0)} = x$ and iterate $r^{(t+1)} = A \cdot r^{(t)} \longrightarrow r^{(1)} = Ax$, $r^{(2)} = A r^{(1)} = A^2 x$ ... $r^{(t)} = A^t x$

x is a vector $[1/N, \dots, 1/N]$ (usually)

$(w_1, \lambda_1) \dots (w_n, \lambda_n)$ are eigenvector (w_i) and corrisponding eigenvalues (λ_i) of A

λ_1 in stochastic matrix is equal to 1 and all eigenvalue are real $\leq \lambda_1$

Since this is a system of eigenvectors, its a base of $\mathbb{R}^n$ so every other vector can be expressed as linear combination of these ones.

$\Rightarrow x = \sum \alpha_i w_i$

$\hookrightarrow A^t x = A^t (\sum \alpha_i w_i) = \sum \alpha_i A^t w_i$

$A w_i = \lambda_i w_i$, $A^2 w_i = A \cdot A w_i = A \lambda_i w_i = \lambda_i^2 w_i$ ...

$\Rightarrow A^t x = \sum \alpha_i \lambda_i^t w_i$ Note: $|\lambda_1| = \lambda_1 = 1$ $|\lambda_i| < 1$ $i > 1$ ALWAYS (if no error occurs)

$= \alpha_1 w_1 + \sum_{2}^{n} \alpha_i |\lambda_i^t| w_i \quad (|\lambda_i| < 1) = \alpha_1 w_1$ $(n \to \infty, \sum_{2}^{n} \alpha_i \lambda_i^t w_i \to 0)$

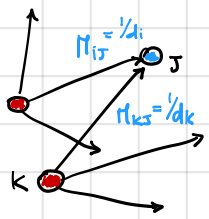
The main eigenvalue is 1! $\longrightarrow r$ = main eigenvector $\to \alpha_1 w_1 = \alpha_1 r$ ALWAYS NOT NEGATIVE

So: to solve $r^T = r^T M$ we do $(r^{(t+1)})^T = (r^{(t)})^T M$

Term $\sum_{2}^{n} \alpha_i \lambda_i^t w_i$ converges and how fast depends on how small $|\lambda_i|$ are $\longrightarrow$ depends on $\max_{i \geq 2} |\lambda_i|$ exponentially

Lets assume $r^{(0)} = x = \begin{pmatrix} 1/n \\ \vdots \\ 1/n \end{pmatrix}$. This give all probability to be at time $t=0$ in any given point.

$(r^{(1)})^T = (r^{(0)})^T M \iff r_j^{(1)} = \sum_{i \to j} M_{ij} r_i^{(0)}$



Explain: The probability to be in node j at time $t=1$ $(r_j^{(1)})$ is equal to the probability to be in any incoming neighbours $(r_i^{(0)})$ times the probability to go to j (M_{ij}), at time $t=0$.

So the probability distribution is given by $(r^{(t+1)})^T = (r^{(t)})^T M$

This is called random walk and r is its stationary distribution.

MARKOV CHAINS

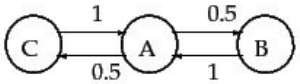
A Markov chain is a discrete-time stochastic process: a process that occurs in a series of time-steps in each of which a random choice is made. A Markov chain consists of N states. Each web page will correspond to a state in the Markov chain we will formulate.

It is characterized by an $N \times N$ transition probability matrix P each of whose entries is in the interval $[0,1]$; the entries in each row of P add up to 1. The chain can be in one of the N states at any given time-step; then, the entry P_{ij} tell us the probability that the state at next time-step is j, conditioned on the current state being i. Each entry P_{ij} is know as a transition probability and depends only on the current state i; this is known as the Markov property: $\forall i,j, P_{ij} \in [0,1]$ and $\forall i, \sum_{j=1}^{n} P_{ij} = 1$ (Second property not always true)

The second property is the sum of the stochastic matrix.

e.g

	A	B	C
A	0	0.5	0.5
B	1	0	0
C	1	0	0



In a Markov chain, the probability distribution of next state depends only on the current state and NOT on how we reached that state.

Starting from A we have 0.5 probability to proceed either to B or C.

From them we can only reach A, so probability is 1

We can view a random surfer on the web graph as a Markov chain, with one state for each page, and each transition probability representing the probability of moving from one page to another.

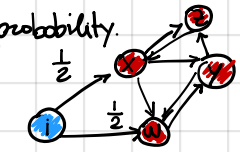
Back to page rank! We have $r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$ or $r^{(t+1)} = M r^{(t)}$ but:

1) does this converge?
2) does it converge to what we want?
3) are results reasonable?

We saw previously that $r_j^{(t+1)}$ can be either rank or probability.

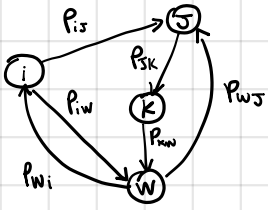
$(r^{(t+1)})_j^T = (r^{(t)})^T M$ is equal to $r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$

$M_{i*} = i^{th}$ row $= \left(\frac{1}{d_i}, 0, 0, \frac{1}{d_i} \dots\right)$ e.g. $\left(0, \frac{1}{2}, 0, 0, \frac{1}{2}\right)$ (i, x, y, z, w)



The difference with a markov chain is that all non-zero entries (in the same row) are equal!

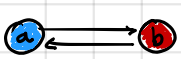
(if not stochastic)



$\forall i \sum_{1}^{n} P_{ij} = 1$

$X^{(t)}$ = node (state) in which M.c. is at step t

$\hookrightarrow r_j^{(t)} = P(X^{(t)} = j)$ $\longrightarrow$ $M = \begin{pmatrix} & j & \\ i & P_{ij} & \end{pmatrix}$ $\quad [(r^{(t+1)})^T = (r^{(t)})^T M]$



r_a =	1	0	1	...
r_b =	0	1	0	...
t =	0	1	2	3 ...

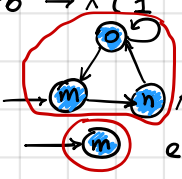
$\longrightarrow$ $\begin{matrix} & a & b \\ a & 0 & 1 \\ b & 1 & 0 \end{matrix} = A$ $\longrightarrow \det(A - \lambda I) = 0 = P(\lambda)$

$A - \lambda I = \begin{pmatrix} -\lambda & 1 \\ 1 & -\lambda \end{pmatrix}$ $\longrightarrow P(\lambda) = \lambda^2 - 1 = 0$ $\longrightarrow \lambda \begin{cases} -1 \\ 1 \end{cases}$

This is an example of periodic markov chain that doesn't converge!

Other problem: Spider traps

Similar problem is: Dead End



All pagerank score get "trapped" in this circle

e.g. a link to a file without outgoing links

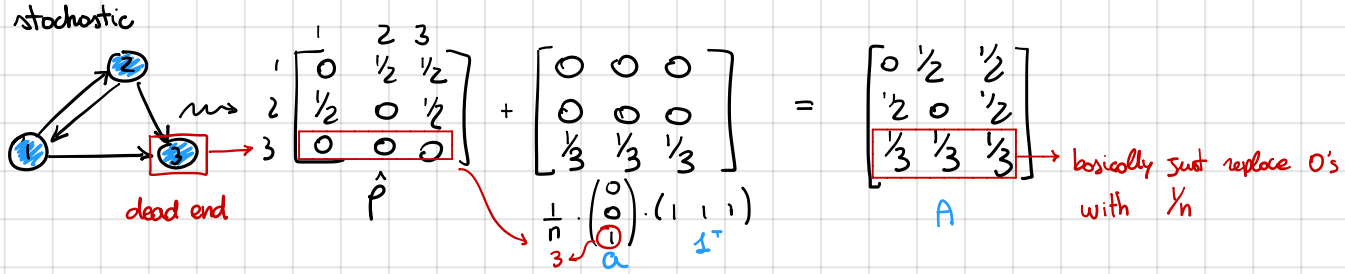
To fix dead ends we adjust matrix with a link to ALL others nodes with $1/n$ probability
To fix spiderchaps we use teleports: at each time step, the random walker has two options
- with prob. β follow a link at random
- with prob $1-\beta$ jump to some random node

$\beta = [0.8, 0.9]$ usually.

When any nodes is reachable from any other nodes, the Markov chain is called irreducible
irreducible + aperiodic = ergodic $\longrightarrow \begin{cases} \lim_{t\to\infty} r^{(t)} = r \\ r \text{ is unique} \end{cases}$

A M.C. has the following property: $P(X^{(t)}=j \mid X^{(t-1)}=a_{t-1}, \dots X^{(0)}=a_0) = P(X^{(t)}=i \mid X^{(t-1)}=a_{t-1})$
with a_t = state at end of round t
Note: $P_{ij} = P(X^{(t)}=j \mid X^{(t-1)}=i)$
We assume that each state i has at least one outgoing link and the matrix is stochastic (we enforce this property adding n links to dead ends)
After t step, M.C. is in each state with different probabilities. $p^{(t)}$ corresponds to a probability distribution, an n-dimensional vector such that its j^{th} entry is $p_j^{(t)} = P(X^{(t)}=j)$: since at the end of each round M.C. has to be in some state we have $\sum p_j^{(t)} = 1 \quad \forall t$
$p_j^{(t)} = \sum_{i\to j} p_i^{(t-1)} P_{ij} \iff (p^{(t)})^T = (p^{(t-1)})^T \cdot P \implies (p^{(t)})^T = (p^{(0)})^T P^{(t)}$ iterating over different rounds.

Steps for random walk:
① Remove dead ends — a dead end corresponds to a row of zeros. We replace it with a row of $1/n$ which corresponds to adding links to all nodes (even itself)
From now on we consider $A = \hat{P} + \left(\frac{1}{n}a\right)[1]^T$ where a is a column vector such that $a_i = 1$ if i is a dead end, $a_i = 0$ otherwise. ($[1]^T$ is a row of 1's)
Now it is stochastic
e.g.



② Teleportation — the random walk described by P is modified as follow: when at a generic node i
- with probability β → follow one of i's outgoing links
- with probability $1-\beta$ → jump to any node at random (even i)

$r_j = \sum_{i\to j} \beta\, r_i/d_i + (1-\beta)/n$
Longer β is (<1) then more similar the graph is to the original one but without periodicity and irreducible
$\longrightarrow [r^{(t)}]^T = \beta \cdot [r^{(t-1)}]^T A + \frac{1-\beta}{n} \cdot 1^T$
$\sum r_j^{(t-1)} = 1 \longrightarrow = \beta [r^{(t-1)}]^T A + \frac{1-\beta}{n} \underbrace{[r^{(t-1)}]^T [1]}_{1} [1]^T = [r^{(t-1)}]^T \underbrace{\left(\beta A + \frac{1-\beta}{n}[1][1]^T\right)}_{P \text{ stochastic}}$ $(\to r^{(t)} = r^{(t-1)} M)$

Smaller β means faster convergence but lose original topology → find a compromise $\alpha \sim [0.8, 0.9]$ usually

SOME PROBLEM WITH PAGERANK (this kind of pagerank)

- Measures generic popularity of a page and biased against topic-specific authorities
  ↳ Topic Specific Page Rank
- Uses a single measure of importance when there can be other ones
  ↳ Hubs-and-Authorities
- Susceptible to Link Spam: artificial in-link created in order to boost page rank
  ↳ TrustRank

TOPIC-SPECIFIC PAGERANK

Goal: evaluate web pages not just according to popularity, but by how close they are to a particular topic.
Search engines use browser history, cache and cookies to profile the user and select the best topic for him.
$r_j(t) = \beta \sum_{i \to j} \frac{r_i(t-1)}{d_i} + \frac{1-\beta}{n}$, let assume we want to rank pages according to topic "sport".
We update teleportation in the following way:

$$\begin{cases} \beta M_{ij} + \frac{1-\beta}{|S|} & \text{if } i \in S = \text{sport pages} \\ \beta M_{ij} + 0 & \text{otherwise} \end{cases}$$

→ this means that when (if) teleporting, jump only to "sport" pages

$$\to r(t)^T = r(t-1)^T \left(\beta A + \frac{1-\beta}{|S|}[1]P^T\right) \quad P_i = \begin{cases} 1 & i \in S \\ 0 & \text{otherwise} \end{cases}$$ (same as a in dead ends elimination)

P^T is called personalization vector. How to compute it? Usually you don't cause there are open source repository with million pages handmade categorized. (DMOZ)
P can be composed by multiple topic with different percentage of "usage" by the user
e.g. $P_{slice} = (0.3, \dots 0 \dots, 0.2, \dots 0 \dots, 0.5, \dots 0 \dots)$ (not a personalization vector)
Books, Movies, Music

$$\to \begin{bmatrix} P_{music} \\ P_{movies} \\ P_{books} \end{bmatrix} \to P = 0.3 P_{books} + 0.2 P_{movies} + 0.5 P_{music}$$

Let assume we have P_1 and P_2, two personalization vector. Then we can compute two stationary distribution:
$r_1^T = r_1^T \left(\beta A + \frac{1-\beta}{|S_1|}[1]P_1^T\right) = r_1^T \underbrace{\left(\beta A + (1-\beta)[1]P_1^T\right)}_{R_1}$ with $\sum P_{1i} = 1$
$r_2^T = r_2^T \left(\beta A + \frac{1-\beta}{|S_2|}[1]P_2^T\right) = r_2^T \underbrace{\left(\beta A + (1-\beta)[1]P_2^T\right)}_{R_2}$ with $\sum P_{2i} = 1$
Note: Personalization vectors are stored somewhere
Assume, by user profile, $P = \alpha_1 P_1 + \alpha_2 P_2$ and we have to compute $r^T = r^T(\beta A + (1-\beta)[1]P^T)$.
Can we compute it without applying power method again?
P_1 and P_2 of course are two different M.C.

$$\begin{cases} r_1^T = r_1^T P_1 \\ r_2^T = r_2^T P_2 \end{cases} \to \alpha_1 r_1^T + \alpha_2 r_2^T = \alpha_1 r_1^T P_1 + \alpha_2 r_2^T P_2 = \alpha_1 r_1^T \beta A + \alpha_2 r_2^T \beta A + (1-\beta)\alpha_1 P_1^T + (1-\beta)\alpha_2 P_2^T =$$
$$= \underbrace{(\alpha_1 r_1^T + \alpha_2 r_2^T)}_{r^T} \beta A + (1-\beta)\underbrace{(\alpha_1 P_1^T + \alpha_2 P_2^T)}_{P^T} \quad (r^T \cdot [1] = 1)$$

↳ $r^T = r^T(\beta A + (1-\beta)[1]P^T)$? → yes

So we can use the linear combination on the fly. Its also true that this operation is not done for every user: users are classified in groups.

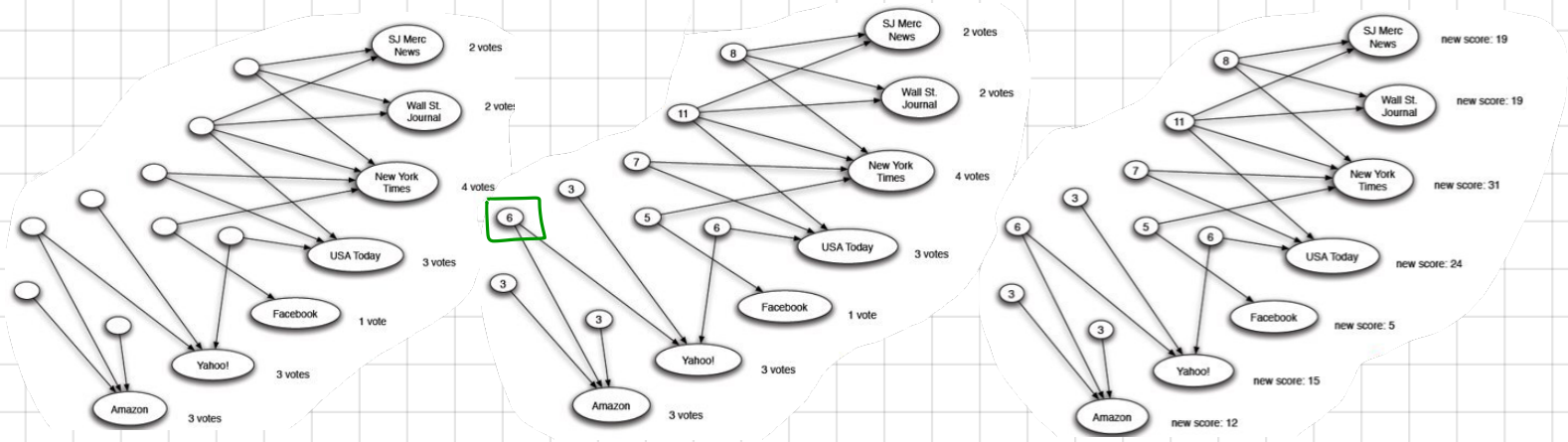
Note: $\sum \pi_i = 1$ with π_i = pagerank of i
→ if there are dead ends NO! → $\sum \pi_i < 1$

HUBS AND AUTHORITIES

HITS: hyperlink-induced Topic Selection, a measure of importance similar to PageRank. How it works: let's say we want to find good newspapers → don't just find newspapers but also "experts" who link in a coordinated way to good newspapers. Votes are always links, but this time also outgoing links count! "I'm a good page if trusted pages link to me, but also if I link to trusted pages"

Each page has 2 score:

- as hub = total sum of votes of authorities pointed to
- as authority = total sum of votes coming from experts.



① Each page start with hub score 1 → authorities collect their votes e.g. 3 hubs point to Amazon → vote 3 as authority

② Hubs collect authorities score e.g. green hub score now is 3 (Amazon) + 3 (Yahoo!) = 6

③ Authorities again collect hub scores.

Note: this is an example. IRL graph is not bipartite and each page has both the scores

- good hub links to many good authorities
- good authority is linked from many good hubs

} two vectors h and a

$$h(x) = \sum_{y: x \to y} a(y) \qquad A_{xy} = \begin{cases} 1, & x \to y \\ 0, & \text{otherwise} \end{cases}$$

$$= \sum_y A_{xy}\, a(y) = A_{x*} \cdot \underline{a}$$

$$A = \begin{pmatrix} & y \\ x & \end{pmatrix}$$

$$\underline{a}(x) = \sum_{y: y \to x} h(y) = \sum_y A_{yx} \cdot h(y) = A^T_{*x} \cdot \underline{h}$$

$$\begin{cases} h^{new} = A\, a^{old} \\ a^{new} = A^T h^{old} \end{cases}$$ and so on ....

$\underline{h}(0) = [1] = \mathbb{1}$

for $t \geq 1$:

$$\underline{a}(t) = A^T \underline{h}(t-1) = A^T A\, \underline{a}(t-1)$$

$$\underline{h}(t) = A\, \underline{a}(t) \longrightarrow \underline{h}(t-1) = A\, \underline{a}(t-1) \to A A^T \underline{h}(t-1)$$

$$\Rightarrow \begin{cases} \underline{a}(t) = A^T A\, \underline{a}(t-1) \\ \underline{h}(t) = A A^T \underline{h}(t-1) \end{cases}$$

$[\vec{c}(t) = M \cdot \vec{c}(t-1)]$ power method

Entries in h and a tend to grow, so:

$$\underline{a}(t) = \frac{\underline{a}(t)}{\|\underline{a}(t)\|_2}$$

$$\underline{h}(t) = \frac{\underline{h}(t)}{\|\underline{h}(t)\|_2}$$

RECOMMENDATIONS

When an user search for something on the web usually he get recommendations from different sites. We come from scarcity (low amount of content in the past) to abundance and so we need to filter results

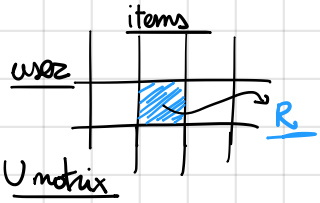
Recommendations are very important also for "items" that are rare and without recommendations would never be found.

Different types:

- Editorial and hand curated e.g. list of favorites
- Simple aggregates e.g. top10, most popular, recent..
- TAILORED to individual user e.g. Amazon, Netflix – thats what we will talk about

X: set of users, S: set of items → Utility function: $u = X \times S \to R$

R: set of ratings, totally ordered, 0-5 stars, real numbers $[0, 1]$



Problems:

1) How to collect ratings
2) How to extrapolate unknown ratings from the known ones
3) Evaluate success of recommendations

1 Explicit way → ask users to rate items (not so good)
Implicit way → learn from past: purchase = high rate → What about low ratings?

2 U matrix is sparse → new items have no ratings, new users have no history

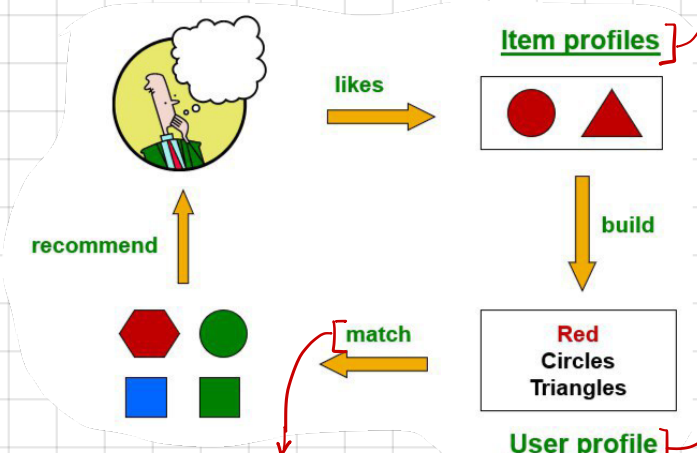
3 approaches:

- content based
- collaborative
- latent factor based

→ used today!

CONTENT-BASED

Recommend items to costumer x similar to previous items rated highly by x



vector of features e.g. Movie → author, title, actor

How? → TF-IDF → doc profile = set of words with highest TF-IDF scores.

user has rated items with profiles $i_1, \dots i_n$ → i_k vector

User Profile 1: average of rated items

User Profile 2: Weighted average using rates as weights.

Given user profile x and item profile i, estimate $u(x,i) = \cos(x,i) = \frac{x \cdot i}{\|x\| \cdot \|i\|}$ → cosine similarity

PRO: No cold start, able to recommend to users with unique tastes, able to recommend new and unpopular items, able to provide explanations (e.g. list of common features).

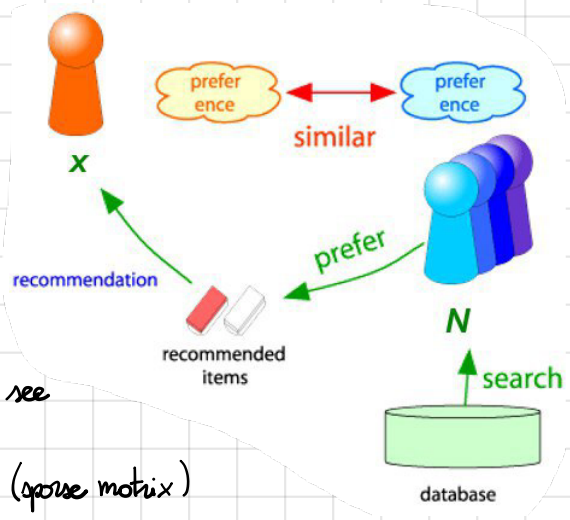
CONS: hard to find appropriate features, no recommendations for new users, too specific (overspecialization), unable to exploit quality judgements of other users.

COLLABORATIVE FILTERING

Consider customer x

Find set N of other users whose ratings are similar to x's ratings.

Estimate x's ratings based on ratings of users in N



how to find "similar" users?

r_x = vector of user x's ratings

- Jaccard similarity → no, cause it ignores the value of ratings (it only see if both users rated the same item
- Cosine similarity → no, cause treats missing ratings as "bad ratings" (sparse matrix)
- Pearson correlation coefficient: S_{xy} = items rated by both users x and y

$\bar{r}_i$ = avg ratings of r_i

$$sim(x,y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}}$$

e.g.

ITEMS

USERS	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

OK — not similar — already close

As we can see $sim(A,B) > sim(A,C)$: we are interested in high rated items and similar so ✔ are not similar and ✘ is not important cause A rated it 1.

Jaccard similarity: $sim(A,B) = \frac{1}{5}$, $sim(A,C) = \frac{2}{4}$ → $sim(A,B) < sim(A,C)$ → NOT TRUE

Cosine similarity: $sim(A,B) = 0.386$, $sim(A,C) = 0.322$ → $sim(A,B) > sim(A,C)$ → TRUE but considers missing ratings "bad ratings" (too similar)

Solution:

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	2/3			5/3	−7/3		
B	1/3	1/3	−2/3				
C				−5/3	1/3	4/3	
D		0					0

The main idea of Pearson correlation coefficient is to calculate cosine similarity subtracting average ratings

e.g. user A's avg $= \left(\frac{4+5+1}{3}\right) = 10/3$ → $HP1_A = 4 - 10/3 = \frac{2}{3}$

→ $sim(A,B) = 0.092$, $sim(A,C) = -0.559$ → $sim(A,B) > sim(A,C)$ and they are very different now!

Note: 0's in second matrix dont bother anyone

Now we have r_x and N = k users most similar to x which rated item i

Prediction for item i of user x:

$r_{xi} = 1/k \sum_{y \in N} r_{yi}$ or $r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}$ $(s_{xy} = sim(x,y))$ or many other tricks...

Some concept (user-user collaborative filtering) can be applied to item-item scenario:

for item i find similar items and estimate rating for item i based on ratings for similar items

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

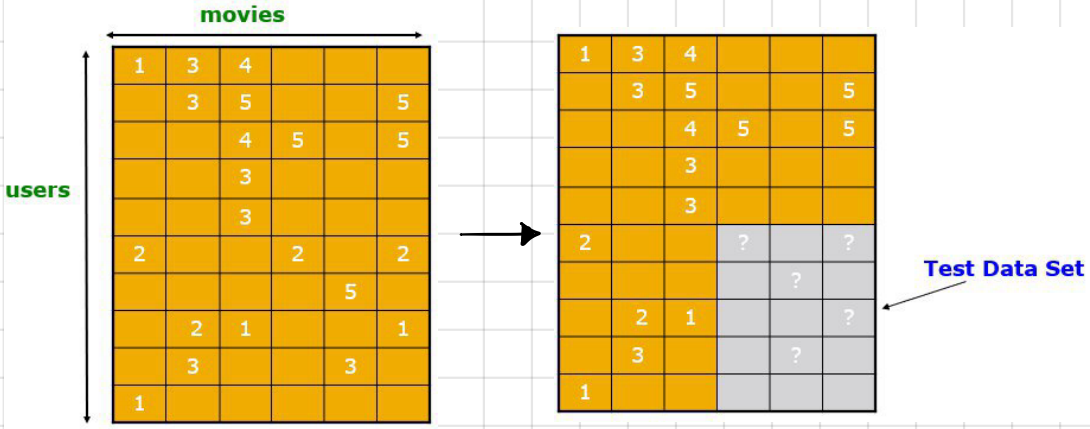
- s_{ij} = similarity of items i and j
- r_{xi} = rating of user x on item i
- $N(i,x)$ = set of items rated by x similar to i

It has been observed that item-item is better than user-user cause items are simpler while users have multiple tastes.

PRO: works for any kind of item

CONS: cold start (need data to find match), sparsity, cannot recommend items not already rated, tend to recommend popular items

In many case we can combine multiple methods.
But how do we evaluate performance? 3



Basically we compare predictions for test data set with known ratings: root-mean-square error (RMSE)
$\sqrt{\sum_{xi}(r_{xi}-r^{*}_{xi})^2}$ where r_{xi} is predicted and r^{*}_{xi} is the true rating of x on i
Then we consider precision at top "10" cause errors on high ratings are more relevant than ones on low ratings.

LOCAL & GLOBAL EFFECTS

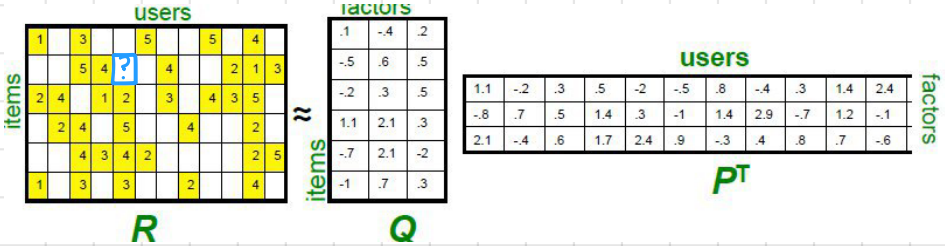
In practice we get better estimates if we model deviations: $\hat{r}_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} s_{ij}(r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} s_{ij}}$

↳ we are combining local (user x) and global (all users) effects

$b_{xi} = \mu + b_x + b_i$
μ = overall mean rating
b_x = rating deviation of user x = (avg rating of user x) $- \mu$
b_j = (avg rating of movie i) $- \mu$
e.g.
Mean movie rating = 3.7 , "The sixth sense" is 4.2 stars
Joe usually rates 0.2 stars below avg. → 4.2 − 0.2 = 4 = baseline estimation
Joe didn't like related movie "Signs" → 4 − 0.2 = 3.8 stars = final estimation

To improve recommendations we want lower RMSE and try to build a system that works well on known ratings and HOPE it predict well the unknown ratings.

LATENT FACTOR MODELS



Suppose we can do this: $R = Q \cdot P^T$

The results is a 3D space in which users and items are 3D points.
$\hat{r}_{xi} = q_i \cdot p_x = \sum_f q_{if} \cdot p_{xf}$ q_i = row i of Q p_x = column x of P^T → even missing ratings
? $= [-0.5, 0.6, 0.5] \cdot \begin{bmatrix} -2 \\ 0.3 \\ 2.4 \end{bmatrix} = 1 + 0.18 + 1.2 \approx 2.4$

How the fuck we find P and Q?

we want to minimize error $\rightarrow \min_{P,Q} \sum_{(i,x)\in R} (r_{xi} - q_i \cdot p_x)^2$

To avoid overfitting we introduce regularization: allow rich model where there are sufficient data and shrink aggressively where data are scarce

$$\min_{P,Q} \sum_{\text{training}} \underbrace{(r_{xi} - q_i p_x)^2}_{\text{error}} + \underbrace{\left[\lambda_1 \sum_x \|p_x\|^2 + \lambda_2 \sum_i \|q_i\|^2\right]}_{\text{length}}$$ minimize error and complexity

λ_1 and λ_2 are user set regularization parameters

$\Rightarrow$ Stochastic Gradient Descent:

initialize P and Q then iterate over the ratings and update factors: Not Very Clear!

for each $r_{xi} \rightarrow \varepsilon_{xi} = 2(r_{xi} - q_i \cdot p_x)$ $\rightarrow$ derivative of the error

$q_i = q_i + \mu_1(\varepsilon_{xi} p_x - \lambda_2 q_i)$ $\rightarrow$ update

$p_x = p_x + \mu_2(\varepsilon_{xi} q_i - \lambda_1 p_x)$ $\rightarrow$ update

for until convergence:

for each r_{xi}

compute gradient

$\rightarrow$ all together: $r_{xi} = \mu + b_x + b_i + q_i p_x$

to complicate this a bit more u can make b_x and b_i time dependent: $r_{xi} = \mu + b_x(t) + b_i(t) + q_i p_x$

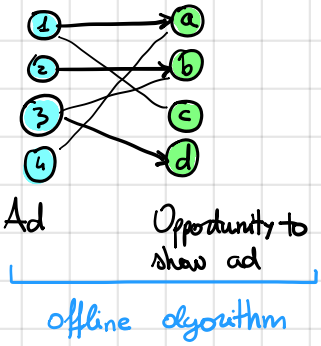
WEB ADVERTISING

First of all we have to make a difference between offline and online algorithms. In the second ones the input isn't known at prior but one piece at time (like a continous data stream) and "decisions" depends on its history.

The goal: show right ad depending on query (and its history)

Note: we dont know anything about future

e.g.



We talk about perfect matching when all vertices are matched
Maximum matching when we have the max number of connections

Online version consist of graph not know upfront. For example we want to pair boys and girls according to their preferences (tinder?). Initially we only have boys set and in each round one girl make choices: she reveals her edges ⟶ we have to decide to pair her with someone or not at that time

Greedy algorith: pair the new girl with any eligible boy, if there is none do not pair

How good is it? Compare it with an offline matching

⟶ competitive ratio = $\min_{\text{all possible inputs } I} (|M_{greedy}| / |M_{optimal}|)$ where M_{greedy} is the matching of greedy algorith and $M_{optimal}$ is the one of offline optimal algorithm

Competitive ratio gives the greedy's worst performance over all possible inputs I

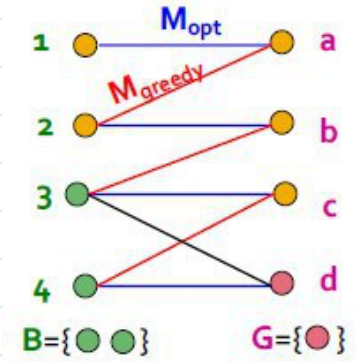
Consider $M_g \neq M_{opt}$ and set G of girls matched in M_{opt} but not in M_g

- $|M_{opt}| \leq |M_g| + |G|$
- B = boys linked to girls in G ⟶ $|M_g| \geq |B|$
  (boys in B are already matched in M_g because a non matched boy linked to a non matched girl are matched by M_g.)

Opt matches all girls in G to some boy in B ⟶ $|G| \leq |B|$

$\Rightarrow |G| \leq |B| \leq |M_g|$ worst is when $|G| = |B| = |M_g|$

$|M_{opt}| \leq |M_g| + |M_g| \longrightarrow |M_g| / |M_{opt}| \geq \frac{1}{2}$



Performance-based Advertising: advertisers bid on search keywords called adwords

A stream of query arrives at search engine $q_1 \dots q_n$; several advertiser bid on each query; when q_i arrives search engine select a subset of ads to show

Goal: maximize profit ⟶ simple solution BID · CTR (click through rate = probability ad is clicked)

Limitation: CTR unknown and advertisers have limited budgets and bid on multiple query.

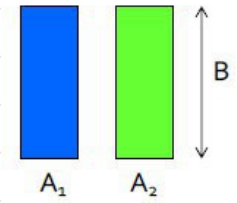
CTR is predicted or measured historically

Search engine has to be sure that the size of ads set is not longer than the number of ads per query, each advertiser has bid on the search query and each advertiser has enough budget to pay if ad is clicked.

In the semplified scenario all budget and ctr are the same , we accept 1 ad for query and ad price = 1
C.ratio = ½ (greedy algorithm)
Bad scenario: A bids on X , B bids on X and Y . Budgets = 4$, query stream xxxx yyyy
→ worst case greedy choice = BBBB____ cause of Y B finished budget and noone can bid for it → c ratio = ½
→ optimal = AAAA BBBB
⟹ Balance algorithm by Mehta, Saberi, Vazirani and Vazirani (MSVV): for each query, pick the advertiser with largest unspent budget
→ balance choice ABABBB__ c. ratio = $\frac{Mg}{Mopt} = \frac{6}{8} = \frac{3}{4}$

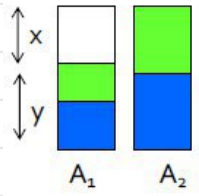
In a simple case with 2 advertiser A_1 and A_2 with budgets $B \geq 1$ optimal solution exhaust both budgets.
Balance must exhaust at least one budget (if not, allocate more queries) → Balance profit = opt. profit − x
= 2B − x . (x = queries not allocated)

e.g.

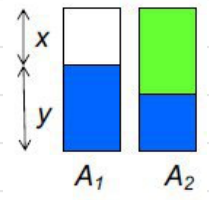


Blue = queries allocated to A_1 in optimal solution
Green = " to A_2 "

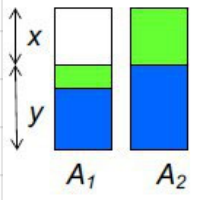
Suppose we exhaust A_2 budget: balance profit = 2B − x = B + y
B is A_2 exhausted budget and y is the allocated profit on A_1
Goal: show $y \geq B/2$



• Case 1: Balance assigns ≥ B/2 blue queries to A_1
→ $y \geq B/2$



• Case 2: Balance assigns ≥ B/2 blue queries to A_2
Consider the last blue query assigned to A_2: at this time A_2's unspent budget ≥ A_1's because balance assigns queries to the one with greatest budget
This means A_1 and A_2 have the same number of query.
→ $x \leq B/2$ and $x + y = B$ → $y \geq B/2$



→ Profit = B + y but $y \geq B/2$ → Profit = 3B/2 competitive ratio = $\frac{3B/2}{2B} = \frac{3}{4}$
In general case, worst competitive ratio is $1 - 1/e \approx 0.63$ THE BEST RESULT POSSIBLE

THE END

EXERCISES

- Prove that for any graph the pagerank of each node is at least α/N.

$\forall v: \pi_v \geq \alpha/N$

Pagerank → Ergodic MC. ⇒

$\pi^T = \pi^T P$

$\pi_v = \alpha/N + (1-\alpha) \sum_{u: u \to v} \frac{\pi_u}{d_u^{out}}$ ← def of pagerank

α/N → Teleport; $(1-\alpha)$ → following links > 0; π_v → probability

$\Rightarrow \pi_v \geq \alpha/N \quad \forall v$

- An algorith gives 30 docs as result in the following order

RRNRN NRRNN NRNNN NRNNR NNNNN NRNRN

① What is the precision on top 20?

P@20 = (# relevant docs retrieved) / (# total retrieved) = $8/20 = 2/5$

② What is the recall on top 20?

R@20 = (# relevant docs retrieved) / (# total relevant docs) = $8/10 = 4/5$

③ What is the F1 measure on top 20?

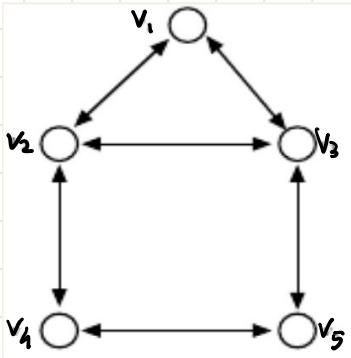
F1@20 = $2 \cdot [(P@20 \cdot R@20)/(P@20 + R@20)] = \frac{8}{15}$

④ What is MAP (mean average precision)?

$\forall k$: P@k $\forall q$ - query

avg. P(q) = $\frac{1}{m} \sum_i^m P@k$ $\Rightarrow q \in Q \Rightarrow MAP(Q) = \frac{1}{|Q|} \sum_{q \in Q}$ avg P(q)

- Compute PR (pagerank scores for $\alpha = 0$ and $\alpha = 1/2$ (α = prob to teleport)



- $\pi_1 = \frac{\alpha}{5} + \frac{1-\alpha}{3}\pi_2 + \frac{1-\alpha}{3}\pi_3$
- $\pi_2 = \frac{\alpha}{5} + \frac{1-\alpha}{2}\pi_1 + \frac{1-\alpha}{2}\pi_4 + \frac{1-\alpha}{3}\pi_3$
- $\pi_4 = \frac{\alpha}{5} + \frac{1-\alpha}{2}\pi_5 + \frac{1-\alpha}{3}\pi_2$

$\pi_3 = \pi_2$, $\pi_5 = \pi_4$ → simmetry

- $\sum_1^5 \pi_i = 1$

① $\alpha = 0$, random walk on undirected graph (bidirectional)

$\forall v_i \quad \pi_i = \frac{d_i}{2m}$ m = # undirected arch

$\pi_1 = \frac{2}{2 \cdot 6} = 1/6 = \pi_4 = \pi_5$

$\pi_2 = \pi_3 = 1/4$

[What if? $\alpha = 1 \longrightarrow \pi_i = 1/N \quad \forall i$]

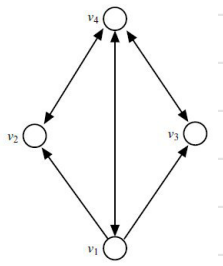
② Just compute π_1, π_2 and π_4 with $\alpha = 1/2$

- True or false
  - In a Boolean retrieval system, stemming never lower precision. **false**
    e.g. { house, housing } → stem. → haus ⟹ haus's postings list will contain both argument → lower precision
  - In a Boolean retrieval system, stemming never lowers recall **true**
    Stemming a term more documents are matched increasing the total number of relevant docs
  - Stemming increases the size of the vocabulary. **false**
    Because more terms are "unified" with the same stem.

- Write down all the necessary equations needed to calculate the pagerank for a general teleporting probability α

  - $\pi_v = \underbrace{\frac{\alpha}{N}}_{\text{teleport}} + \underbrace{(1-\alpha)\sum_{u:u\to v}\frac{\pi_u}{d_u^{out}}}_{\text{follow links}} \quad \forall_v$
  - $\sum_v \pi_v = 1$

- Given the following graph with teleport probability α



  - Write all necessary equation needed to calculate personalized pagerank with respect to personalization vector [1, 0, 0, 0]

$$\pi_1 = \alpha + (1-\alpha)\frac{\pi_4}{3}$$
$$\pi_2 = \pi_3 = (1-\alpha)\frac{\pi_1}{3} + (1-\alpha)\frac{\pi_4}{3}$$
$$\pi_4 = (1-\alpha)\left[\frac{\pi_1}{3} + \pi_2 + \pi_3\right]$$
$$\sum \pi_i = 1$$