Information Retrieval

Lecture 3

Recap: lecture 2

- Stemming, tokenization etc.
- Faster postings merges
- Phrase queries

This lecture

- Index compression
  - Space for postings
  - Space for the dictionary
  - Will only look at space for the basic inverted index here
- Wild-card queries

Corpus size for estimates

- Consider n = 1M documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data.
- Say there are m = 500K *distinct* terms among these.

Don’t build the matrix

- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- So we devised the inverted index
  - Devised query processing for it
- Where do we pay in storage?

- Where do we pay in storage?

Terms

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Pointers

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Storage analysis

- First will consider space for pointers
  - Devise compression schemes
- Then will do the same for dictionary
- No analysis for wildcards etc.

Pointers: two conflicting forces

- A term like *Calpurnia* occurs in maybe one doc out of a million - would like to store this pointer using $\log_2 1M \sim 20$ bits.
- A term like *the* occurs in virtually every doc, so 20 bits/pointer is too expensive.
  - Prefer 0/1 vector in this case.

Postings file entry

- Store list of docs containing a term in increasing order of doc id.
  - *Brutus*: 33,47,154,159,202 ...
- Consequence: suffices to store *gaps*.
  - 33,14,107,5,43 ...
- Hope: most gaps encoded with far fewer than 20 bits.

Variable encoding

- For *Calpurnia*, will use ~20 bits/gap entry.
- For *the*, will use ~1 bit/gap entry.
- If the average gap for a term is G, want to use ~$\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with ~ as few bits as needed for that integer.

γ codes for gap encoding

Length	Offset

- Represent a gap G as the pair *<length,offset>*
- *length* is in unary and uses $\lfloor \log_2 G \rfloor + 1$ bits to specify the length of the binary encoding of
- *offset* = $G - 2^{\lfloor \log_2 G \rfloor}$
- e.g., 9 represented as <1110,001>.
- Encoding G takes $2\lfloor \log_2 G \rfloor + 1$ bits.

Exercise

- Given the following sequence of γ–coded gaps, reconstruct the postings sequence:

1110001110101011111101101111011

From these γ–decode and reconstruct gaps, then full postings.

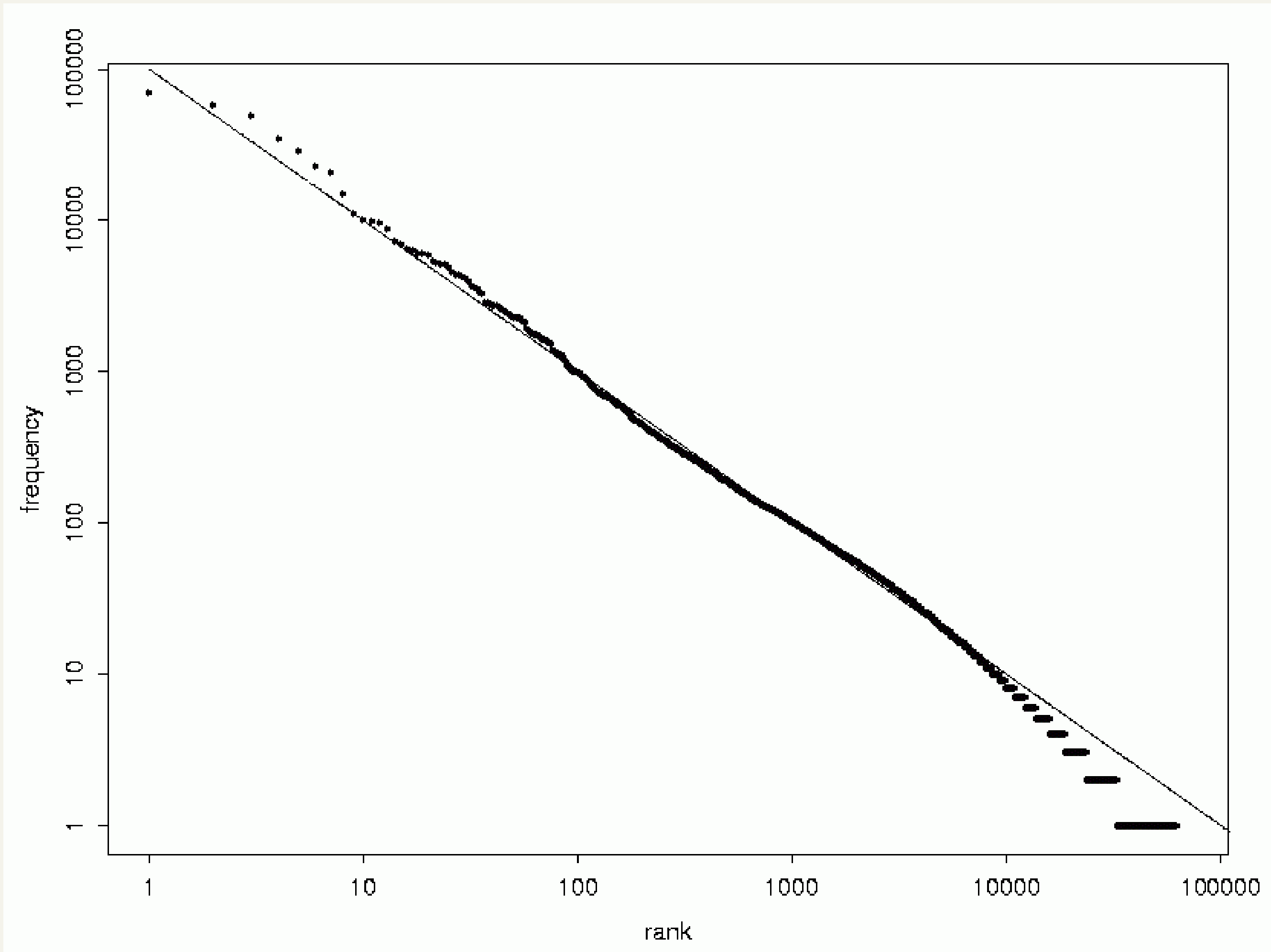
What we’ve just done

- Encoded each gap as tightly as possible, to within a factor of 2.
- For better tuning (and a simple analysis) - need a handle on the <u>distribution</u> of gap values.

Zipf's law

- The kth most frequent term has frequency proportional to $1/k$.
- Use this for a crude analysis of the space used by our postings file pointers.
  - Not yet ready for analysis of dictionary space.

Zipf’s law log-log plot



Rough analysis based on Zipf

- Most frequent term occurs in n docs
  - n gaps of 1 each.
- Second most frequent term in $n/2$ docs
  - $n/2$ gaps of 2 each …
- kth most frequent term in n/k docs
  - n/k gaps of k each - use $2\log_2 k + 1$ bits for each gap;
  - net of $\sim(2n/k).\log_2 k$ bits for kth most frequent term.

Sum over k from 1 to m=500K

- Do this by breaking values of k into groups: group i consists of $2^{i-1} \le k < 2^i$.
- Group i has 2^{i-1} components in the sum, each contributing at most $(2ni)/2^{i-1}$.
  - Recall n=1M
- Summing over i from 1 to 19, we get a net estimate of 340Mbits ~45MB for our index.

Work out calculation.

Caveats

- This is not the entire space for our index:
  - does not account for dictionary storage;
    - nor wildcards, etc.
  - as we get further, we’ll store even more stuff in the index.
- Assumes Zipf’s law applies to occurrence of terms in docs.
- All gaps for a term taken to be the same.
- Does not talk about query processing.

Dictionary and postings files

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Usually in memory

Gap-encoded, on disk

Inverted index storage

- Have estimate pointer storage
- Next up: Dictionary storage
  - Dictionary in main memory, postings on disk
    - This is common, especially for something like a search engine where high throughput is essential, but can also store most of it on disk with small, in-memory index
- Tradeoffs between compression and query processing speed
  - Cascaded family of techniques

How big is the lexicon V?

- Grows (but more slowly) with corpus size
- Empirically okay model:

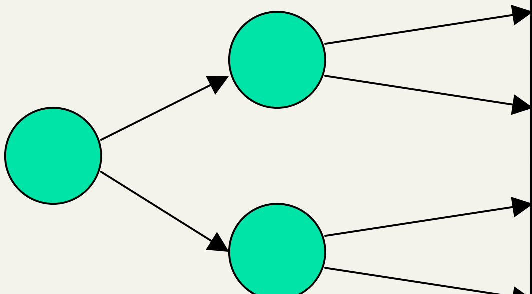
$$V = kN^b$$

Exercise: Can one derive this from Zipf's Law?

- where $b \approx 0.5$, $k \approx 30$–100; N = # tokens
- For instance TREC disks 1 and 2 (2 Gb; 750,000 newswire articles): ~ 500,000 terms
- V is decreased by case-folding, stemming
- Indexing all numbers could make it extremely large (so usually don't*)
- Spelling errors contribute a fair bit of size

Dictionary storage - first cut

- Array of fixed-width entries
  - 500,000 terms; 28 bytes/term = 14MB.



Terms	Freq.	Postings ptr.
a	999,712	
aardvark	71	
….	….	
zzzz	99	

Allows for fast binary search into dictionary

20 bytes

4 bytes each

Exercises

- Is binary search really a good idea?
- What are the alternatives?

Fixed-width terms are wasteful

- Most of the bytes in the Term column are wasted – we allot 20 bytes for 1 letter terms.
  - And still can't handle *supercalifragilisticexpialidocious.*
- Written English averages ~4.5 characters.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
  - Short words dominate token counts.
- Average word in English: ~8 characters.

Explain this.

What are the corresponding numbers for Italian text?

Compressing the term list

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

Total string length = 500KB x 8 = 4MB

Pointers resolve 4M positions: $\log_2 4M$ = 22bits = 3bytes

Binary search these pointers

Total space for compressed list

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 500K terms $\Rightarrow$ 9.5MB

Now avg. 11 bytes/term, not 20.

Blocking

- Store pointers to every *k*th on term string.
  - Example below: $k=4$.
- Need to store term lengths (1 extra byte)

….*7systile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9szomo*….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Net

- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes for k=4 pointers,

now we use 3+4=7 bytes for 4 pointers.

Shaved another ~0.5MB; can save more with larger k.

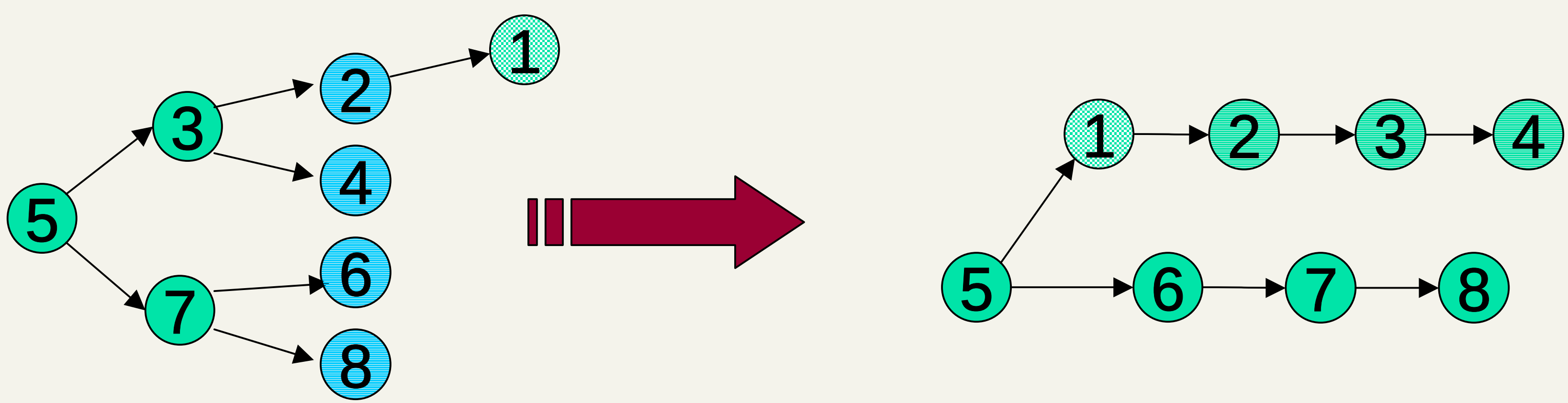
Why not go with larger k?

Exercise

- Estimate the space usage (and savings compared to 9.5MB) with blocking, for block sizes of $k = 4, 8$ and 16.

Impact on search

- Binary search down to 4-term block;
- Then linear search through terms in block.
- 8 documents: binary tree ave. = 2.6 compares
- Blocks of 4 (binary tree), ave. = 3 compares



$= (1+2\cdot2+4\cdot3+4)/8$

$=(1+2\cdot2+2\cdot3+2\cdot4+5)/8$

Exercise

- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4$, 8 and 16.

Total space

- By increasing k, we could cut the pointer space in the dictionary, at the expense of search time; space 9.5MB $\rightarrow$ ~8MB
- Adding in the 45MB for the postings, total 53MB for the simple Boolean inverted index

Some complicating factors

- Accented characters
  - Do we want to support accent-sensitive as well as accent-insensitive characters?
  - E.g., query *resume* expands to *resume* as well as *résumé*
  - But the query *résumé* should be executed as only *résumé*
  - Alternative, search application specifies
- If we store the accented as well as plain terms in the dictionary string, how can we support both query versions?

Index size

- Stemming/case folding cut
  - number of terms by ~40%
  - number of pointers by 10-20%
  - total space by ~30%
- Stop words
  - Rule of 30: ~30 words account for ~30% of all term occurrences in written text
  - Eliminating 150 commonest terms from indexing will cut almost 25% of space

Extreme compression (see *MG*)

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last *k-1* in a block of *k*)

8*automata*8*automate*9*automatic*10*automation*

→8{*automat*}*a*1◊*e*2◊*ic*3◊*ion*

Encodes *automat*

Extra length beyond *automat.*

Begins to resemble general string compression.

Extreme compression

- Using perfect hashing to store terms "within" their pointers
  - not good for vocabularies that change.
- Partition dictionary into pages
  - use B-tree on first terms of pages
  - pay a disk seek to grab each page
  - if we're paying 1 disk seek anyway to get the postings, "only" another seek/query term.

Compression: Two alternatives

- Lossless compression: all information is preserved, but we try to encode it compactly
  - What IR people mostly do
- Lossy compression: discard some information
  - Using a stoplist can be thought of in this way
  - Techniques such as Latent Semantic Indexing (later) can be viewed as lossy compression
  - One could prune from postings entries unlikely to turn up in the top *k* list for query on word
    - Especially applicable to web search with huge numbers of documents but short queries (e.g., Carmel et al. *SIGIR 2002)*

Top *k* lists

- Don’t store all postings entries for each term
  - Only the “best ones”
  - Which ones are the best ones?
- More on this subject later, when we get into ranking

Wild-card queries

Wild-card queries: *

- *mon*:* find all docs containing any word beginning "mon".
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: $mon \leq w < moo$
- **mon:* find words ending in "mon": harder
  - Maintain an additional B-tree for terms *backwards.*

  Now retrieve all words in range: $nom \leq w < non$.

Exercise: from this, how can we enumerate all terms meeting the wild-card query *pro*cent*?

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

  *se*ate AND fil*er*

  This may result in the execution of many Boolean *AND* queries.

Permuterm index

- For term *hello* index under:
  - *hello$, ello$h, llohe, lohel, o$hell*

  where $ is a special symbol.
- Queries:
  - X lookup on X$ X* lookup on X*$
  - *X lookup on X$*   *X* lookup on X*
  - X*Y lookup on Y$X*   X*Y*Z ???

  Exercise!

Bigram indexes

- *Permuterm problem:* ≈ *quadruples lexicon size*
- Another way: index all *k*-grams occurring in any word (any sequence of *k* chars)
- *e.g.,* from text "*April is the cruelest month*" we get the 2-grams (*bigrams*)

  a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
  ue,el,le,es,st,t$, m,mo,on,nt,h

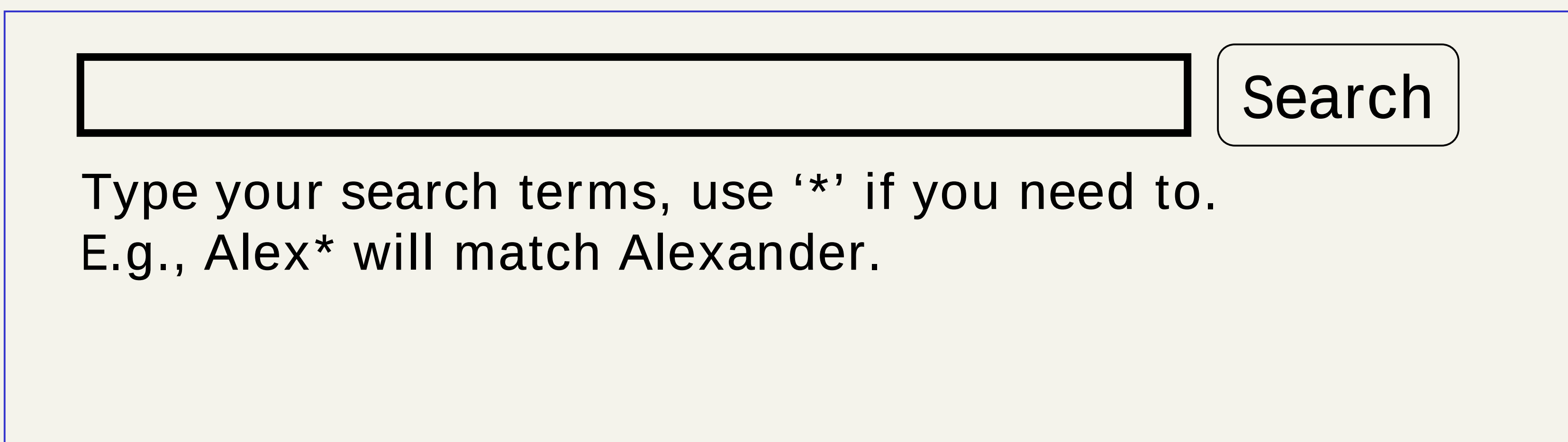
  - $ is a special word boundary symbol

Processing *n*-gram wild-cards

- Query _mon*_ can now be run as
  - _$m AND mo AND on_
- Fast, space efficient.
- But we'd enumerate *moon*.
- Must post-filter these terms against query.

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution
  - Avoid encouraging “laziness” in the UI:



Resources for this lecture

- MG 3.3, 3.4, 4.2