Information Retrieval

Lecture 3

Indexing (continued)and Compression

Plan

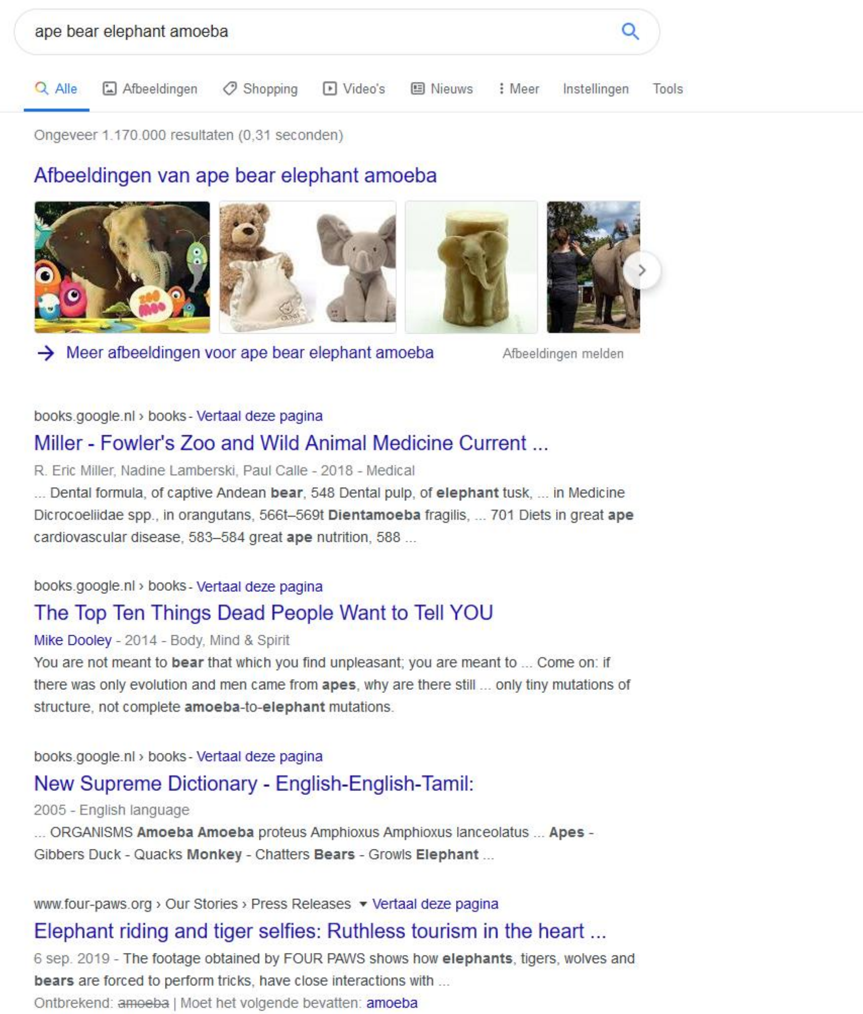
- Lecture 1:
  - Boolean retrieval
  - Inverted index
- Lecture 2:
  - Evaluation
- Today:
  - Ch2: phrase queries, stemming
  - Ch5: compression (contains extra material which is not in the book!)

- So, this year we skip (but interesting..):
  - Ch3: robust search (dealing with spelling errors)
  - Ch4: scalable index inversion

Does Google use the Boolean (exact match) model?

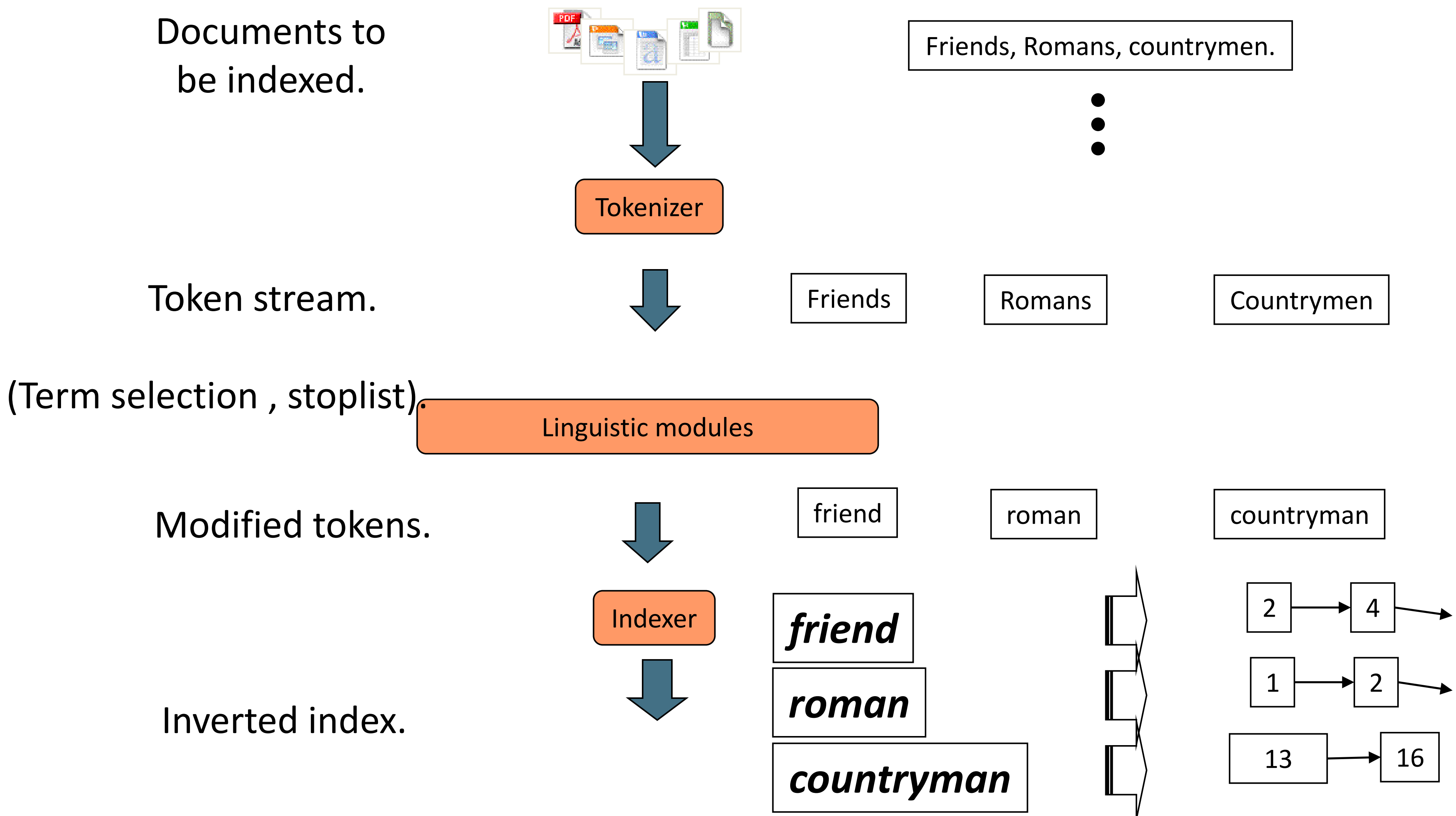
Challenge: reverse engineer a rough version of the Google query interpretation by just trying.

- On Google, the default interpretation of a query [w_1 w_2 . . .w_n] is w_1 AND w_2 AND . . .AND w_n
- Cases where you get hits that do not contain one of the *wi* :
  - anchor text
  - page contains variant of w_i (morphology, spelling correction, synonym)
  - boolean expression generates very few hits (coordination level matching)
- Simple Boolean vs. Ranking of result set
  - Simple Boolean retrieval returns matching documents in no particular order.
  - Google (and most well designed Boolean engines) rank the result set – they rank good hits (according to some estimator of relevance) higher than bad hits.



IMPROVING THE MATCH BETWEEN QUERY TERMS AND DOCUMENT TERMS

Recall the basic indexing pipeline



Normalizing terms

- Dealing with abbreviations
- Accents
- Dates
- Case folding

- Linguistic normalization: stemming / lemmatization

- Indexing: i) tokenizing ii) normalization iii) term selection (stopwording) iv) [term counting] v) inversion

Challenges for Normalizing natural language

- Morphological variation (inflection, derivation)
- Polysemy: word with different but related senses (e.g. get, good)
- Homonymy: word has different unrelated senses (e.g. java, bank)
- Synonymy: multiple words mean the exact same thing (car, automobile)

Normalizing: Lemmatization

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form

Normalizing: Stemming

- Reduce terms to their "roots" before indexing
- "Stemming" suggests crude affix chopping
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***

for example compressed and compression are both accepted as equivalent to compress.

→

for exampl compress and compress ar both accept as equival to compress

Two Alternatives for working with equivalence classes (conflation: IIR 2.2.3)

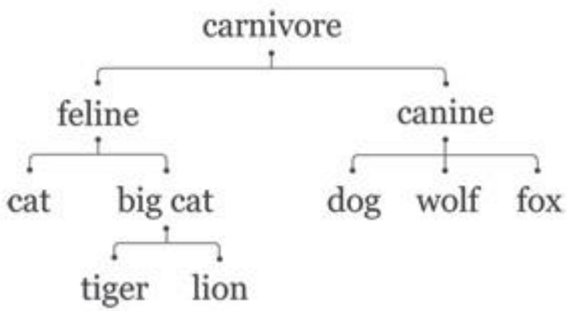
- A. Normalize documents and queries:
  - Tokens from the same equivalence class are replaced by a representative token to serve as index term
  - Dictionary size is reduced
  - Not so flexible

- B. Perform query expansion:
  - Create equivalence classes on the fly
  - Need proper run-time conflation back-end: how do we efficiently compute the document frequency of a run-time conflation class?
  - Structured Boolean queries help for non-ranked systems

- But: IR ranking models and conflation architectures are not independent! (Variations on Language Modeling for Information Retrieval, Kraaij(2004))

Normalizing: Synonyms

- Thesauri compile words and their relations for a certain language or domain (e.g. WordNet or MeSH)



- Typical relations are:
  - *Synonymy (car automobile)*
  - *Meronymy* part-of: wheel -> car
  - *Hyponymy* more specific term : snake<-reptile
  - *Hypernymy* (reverse: broader term)

Possible use of thesauri

- Using synonyms for query expansion, just as the morphological equivalence classes:
  - Q: Car sales in the USA in 2009
  - Q’: { car, automobile} & {sales,revenue} & { USA, US,…}&2009
- PUBMED:
  - Q: breast cancer
  - Q’: "breast neoplasms"[MeSH Terms] OR ("breast"[All Fields] AND "neoplasms"[All Fields]) OR "breast neoplasms"[All Fields] OR ("breast"[All Fields] AND "cancer"[All Fields]) OR "breast cancer"[All Fields]
- Mixed results of using thesauri:
  - No convincing results for Wordnet, domain specific resources seem to help (e.g. Stokes et al, Information Retrieval, 2009)
  - Hard to beat relevance feedback: mismatch resource-collection

Stop words

- With a stop list, you exclude from the dictionary entirely the most common words. Intuition:
  - They have little semantic content: *the, a, and, to, be*
  - There are a lot of them: ~30% of postings for top 30 words
- But the trend is away from doing this:
  - Good compression techniques means the space for including stopwords in a system is very small
  - Good query optimization techniques mean you pay little at query time for including stop words.
  - You need them for:
    - Phrase queries: “King of Denmark”
    - Various song titles, etc.: “Let it be”, “To be or not to be”
    - “Relational” queries: “flights to London”

Summary

- Overcoming the term mismatch
  - Normalization and query expansion techniques
  - Especially important for short queries, when recall is important
- Often conceived as “linguistic add-on” modules
- Integration in statistical IR models is not so straightforward
- Various more or less linguistic techniques have shown to be effective
- ➔ Neural IR / word embeddings is a recent method with a similar aim

PHRASE QUERIES AND POSITIONAL INDEXES

Phrase queries

- Want to be able to answer queries such as "***stanford university***" – as a phrase
- This will avoid matching the sentence *"I went to university at Stanford"*
  - The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works
  - Many more queries are *implicit phrase queries*
- How can we implement phrase search?

Solution 2: Positional indexes

- In the postings, store for each ***term*** the position(s) in which tokens of it appear:

<***term****,* number of docs containing ***term***;
doc1: position1, position2 … ;
doc2: position1, position2 … ;
etc.>

➔Of course this significantly increases the index size !

Positional index example

<***be***:
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just docid equality

Processing a phrase query

- Extract inverted index entries for each distinct term: ***to, be, or, not.***
- Merge their *doc:position* lists to enumerate all positions with "***to be or not to be***". First check docid, than positions.
  - ***to**:*
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - ***be**:*
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...
- Same general method for proximity searches

CH 5. INDEX COMPRESSION

Zipf’s law

- Zipf's law is named after the Harvard linguistic professor George Kingsley Zipf (1902-1950)
  - *Human Behavior and the Principle of the Least Effort* from 1949
  - empirical law on word frequencies in natural language speech and texts.
- In some fields where Zipf's law plays an important role:
  - quantitative linguistics
  - urban growth
  - internet

Principle of least effort

- The **principle of least effort** is a broad theory that covers diverse fields from evolutionary biology to webpage design.
- It postulates that animals, people, even well designed machines will naturally choose the path of least resistance or "effort".
- It is closely related to many other similar principles (such as Principle of least action).
- This is perhaps best known or at least documented among researchers in the field of library and information science.

- Zipf's law states that
  - while only a few words are used very often,
  - many or most are used rarely.

- Zipf's law states that in a tabulation of the occurrence of all words in a sufficiently comprehensive text (concatenated), ranked by their frequency, will the product of rank number and frequency make up a constant.

http://www.iva.dk/bh/Core%20Concepts%20in%20LIS/articles%20a-z/zipfs_law.htm

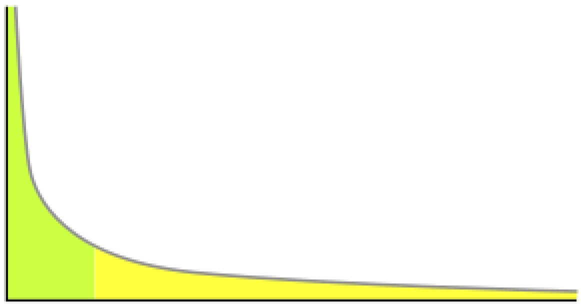
- Zipf's law: The *k*-th most frequent term has frequency proportional to 1/*k* .

- cf_k is collection frequency: the number of occurrences of the term t_k in the collection.

$$cf_k = \frac{c}{k} = \frac{cf_1}{k} = cf_1 \cdot k^{-1}$$

where *c* is the normalizing constant

This is a power law (power = -1)



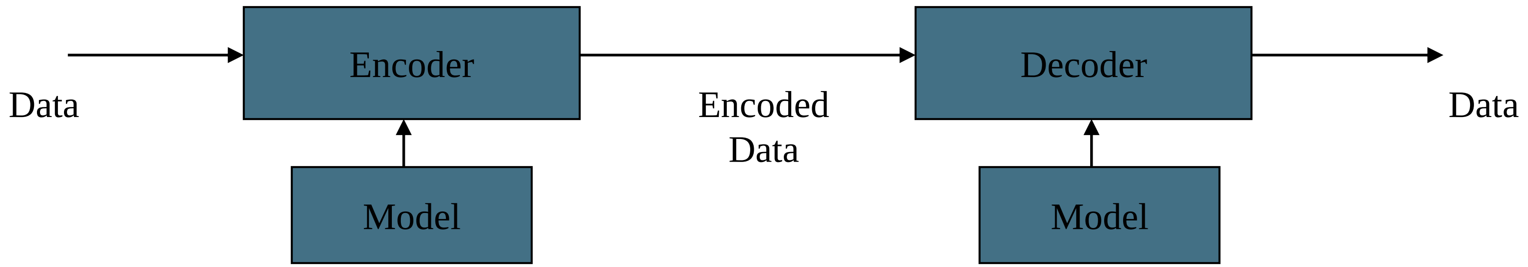
BASICS OF (TEXT) COMPRESSION, NOT INCLUDED IN IIR BOOK

Why compression?

- Search engines
  1. Keep more stuff in memory (increases speed)
  2. Increase data transfer from disk to memory
     - [read compressed data and decompress] is faster than [read uncompressed data]
  - Premise: Decompression algorithms are fast
    - True of the decompression algorithms we use
- Video distribution application
  - maximum quality given limited bandwidth
- In the previous century
  - maximize capacity of storage media

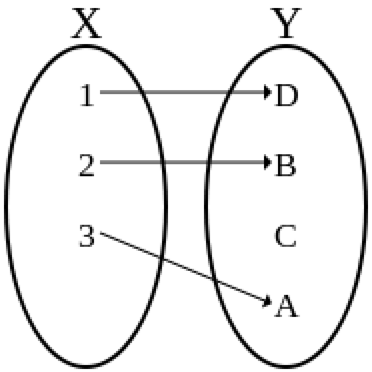
Compression characteristics

- Lossless vs. lossy compression
  - text
  - image / audio / video
  - Typical compression rates 10-50%, depends on..
- Compression: removing redundancy
- Compression is typically achieved by coding
- Other applications of coding are
  - error correcting codes (transmission, noisy channel)
  - cryptography



Text compression

- ths lctr s bt cmprssn
- this lecture is about compression
  - data compression ratio: 33:21
  - problem: decoding process is not unique
- Desiderata for a lossless compression function c(X)
  - c(X) is *injective*: c(X) = c(X') → X = X'
  - inverse function is fast
  - C(X') is (on average) shorter than X



- Typical rates for text: 3:1
- Some strings are easier to compress, can we quantify?

Example: creating a coding scheme

- aabbbccccdddddeeeeeefffffffgggggggg
- length: 35 characters, 8 bits per char, 270 bit
- Coding table
  - a : 1001 e: 111
  - b : 1000 f: 110
  - c : 011 g: 00
  - d : 010

  results in code of 117 bits:

  1001100110001000011011011010010010010111111111111111110110110110110110110000000000000000

- *Frequent letters get short codes*

- *What about 'natural language' ?*

Complexity of a string

1. 010101…01 (million times pattern 01)
2. concatenation of lotto/toto results since 1900
3. first million decimals of π
4. measurement of motion of elementary particles

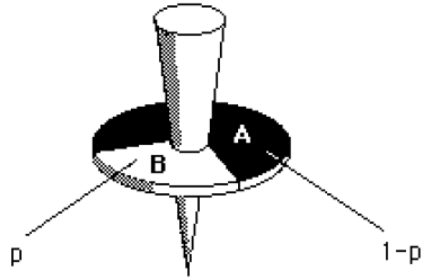
1. print pattern *n* times
2. difficult to compress
3. π: many formulas available http://en.wikipedia.org/wiki/Pi
4. Even more difficult to compress

the less complex, the better to compress

kolmogorov complexity:
length of the shortest program
(and required data)
that can regenerate the original string

Maximum attainable compression (1)

- If we have a model of the data, we can compute the maximum compression ratio
- Simple model: generative model without memory



- Spinning top emits symbol strings like: AABBAAAAABAAABABAA
- Law of the large numbers: “in the limit” pN symbols are equal to B

-continued- (2)

- A symbol string of N symbols with pN B's can have $\binom{N}{pN}$ forms.
  - how to code all these possible outcomes?
- Coding scheme:
  - Sort all symbol strings in a systematic fashion (lexicographic)
  - Each possible outcome is encoded as its rank number

dictio nary

- A rank number of M can be encoded by log_2(M) bits
- The outcomes can thus be encoded by $log_2\binom{N}{pN}$ bits
  - If *p=0.1 and N = 1000* ➔ *|encoded string|=* $log_2\binom{1000}{100}$ bits

Background info (not for exam) -continued- (3a)

$$\log(N!) \approx \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = n\log n - n\log e + \tfrac{1}{2}\log(2\pi n) \approx n\log n$$

- Using Stirling's approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ we derive

$$\log\binom{N}{pN} = \log\left(\frac{N!}{(pN)!(N-pN)!}\right)$$

$$= \log(N!) - \log((pN)!) - \log((N-pN)!)$$

$$\cong N\log(N) \boxed{- pN\log(pN)} \boxed{-(1-p)N\log((1-p)N}\,)$$

$$= N\cdot\left[\cancel{\log(N)} \boxed{-p\log(p) - p\cancel{\log(N)}} \boxed{-(1-p)\log(1-p) - (1-p)\cancel{\log(N)}}\right]$$

$$= N\cdot\left(-p\log(p) - (1-p)\log(1-p)\right)$$

$$= -N\sum_{x\in X} P(x)\log P(x) = N\cdot H(P) \qquad \mathit{for}\ \ X = \{A, B\}$$

- H(P) is the entropy function for a distribution (Shannon) where X = {A, B} and P(B) = p and P(A) =1-p

-continued- (3)

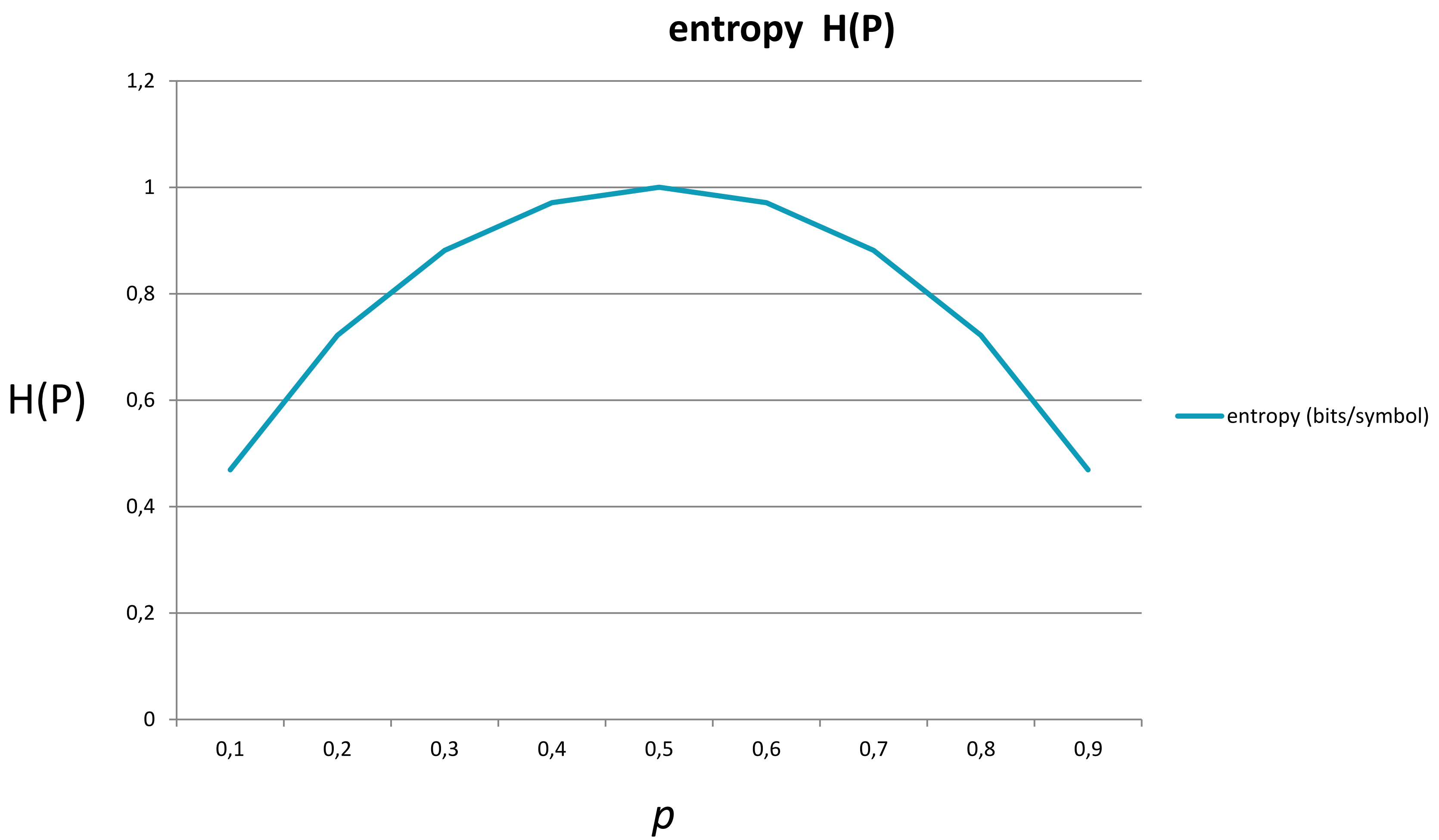
$$\log\binom{N}{pN} = N \cdot H(P)$$

- So the average length of a compressed string of length N requires per symbol the following number of bits:

$$\frac{1}{N}\log\binom{N}{pN} \cong -\sum_{x \in X} P(x)\log P(x) = H(P)$$

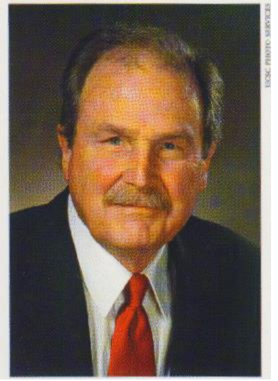
- e.g. entropy is 0.467 bit/symbol for p=0.1 (1 for p=0.5)
  - ➔ take logarithm base=2 for bit units!

Entropy of spinning top (function of p)



Example of efficient compression scheme

Huffman coding (1952)



- In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.
- In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffman avoided the major flaw of the suboptimal Shannon-Fano coding by building the tree from the bottom up instead of from the top down.

- [wikipedia:Huffman coding, February 2016]
- cf. http://www.huffmancoding.com/david-huffman/huffman-algorithm

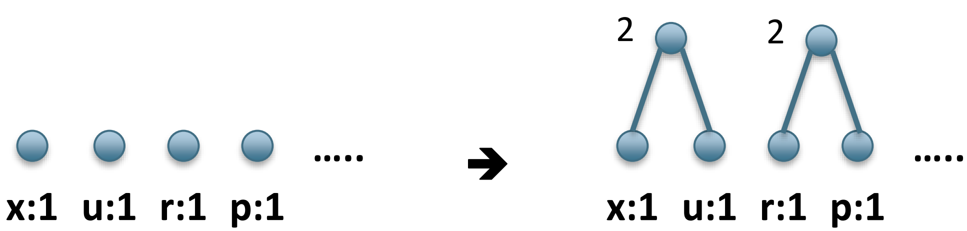
Huffman coding (1)

this is an example of a Huffman tree

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in queue:
    1. Remove two nodes of lowest probability from queue
    2. Create new internal node
        1. these two nodes as children
        2. probability = sum of the two nodes' probabilities.
    3. Add new node to queue.

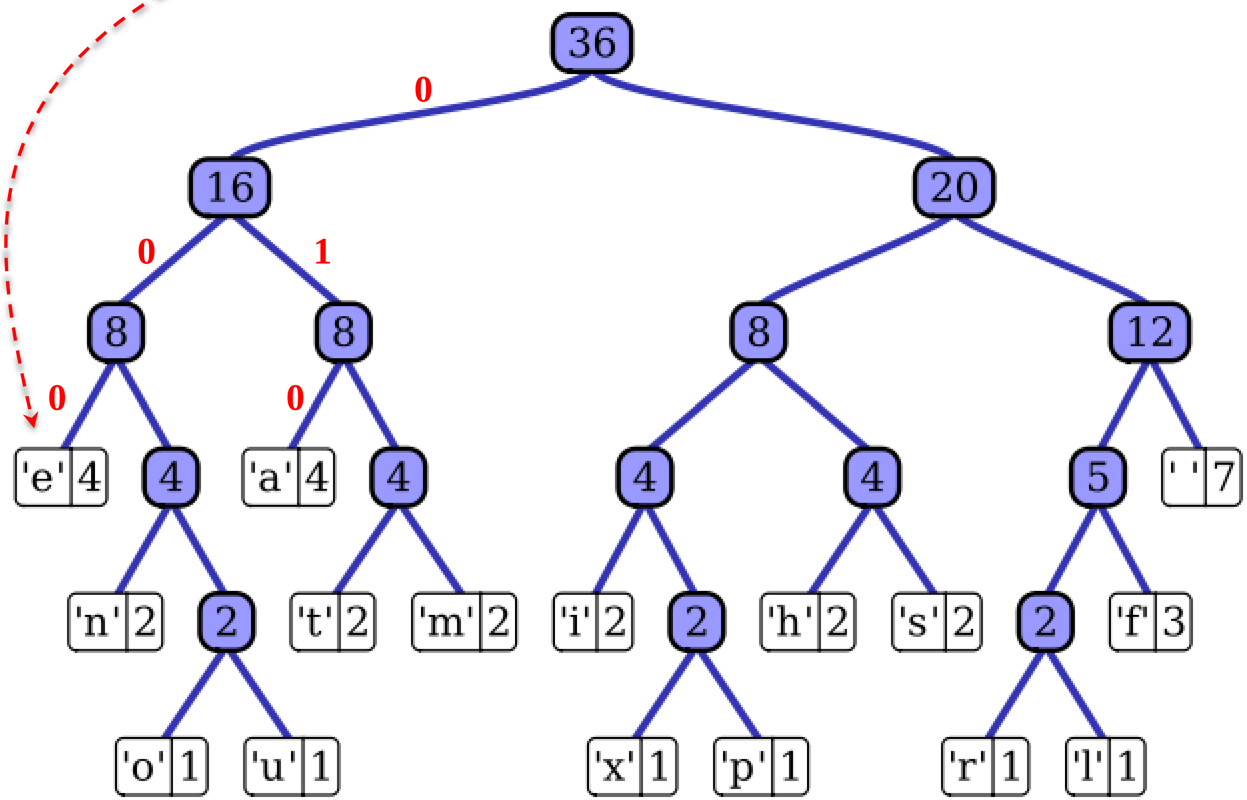
- Tree is complete.

Char	Freq
space	7
a	4
e	4
f	3
h	2
I	2
m	2
n	2
s	2
t	2
l	1
o	1
p	1
r	1
u	1
x	1



Huffman coding (2)

- "this is an example of a Huffman tree"
  |text|=36 chars
- Construction is bottom up!



Char	Freq	Code
Space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
l	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

- 135/36 bits per character (<4)

Inverted Index

- Each index term is associated with an *inverted list*
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

Example "Collection"

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index

and	1
aquarium	3
are	3 4
around	1
as	2
both	1
bright	3
coloration	3 4
derives	4
due	3
environments	1
fish	1 2 3 4
fishkeepers	2
found	1
fresh	2
freshwater	1 4
from	4
generally	4
in	1 4
include	1
including	1
iridescence	4
marine	2
often	2 3

only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1 4
saltwater	2
species	1
term	2
the	1 2
their	3
this	4
those	2
to	2 3
tropical	1 2 3
typically	4
use	2
water	1 2 4
while	4
with	2
world	1

- Inverted Index
  - with counts

- supports better ranking algorithms

Term	Postings			
and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

- Inverted Index
- with positions
  - supports
- proximity matches

Term	Postings
and	1,15
aquarium	3,5
are	3,3 4,14
around	1,9
as	2,21
both	1,13
bright	3,11
coloration	3,12 4,5
derives	4,7
due	3,7
environments	1,8
fish	1,2 1,4 2,7 2,18 2,23 3,2 3,6 4,3 4,13
fishkeepers	2,1
found	1,5
fresh	2,13
freshwater	1,14 4,2
from	4,8
generally	4,15
in	1,6 4,1
include	1,3
including	1,12
iridescence	4,9
marine	2,22
often	2,2 3,10
only	2,10
pigmented	4,16
popular	3,4
refer	2,9
referred	2,19
requiring	2,12
salt	1,16 4,11
saltwater	2,16
species	1,18
term	2,5
the	1,10 2,4
their	3,9
this	4,4
those	2,11
to	2,8 2,20 3,8
tropical	1,1 1,7 2,6 2,17 3,1
typically	4,6
use	2,3
water	1,17 2,14 4,12
while	4,10
with	2,15
world	1,11

Why Index Compression?

- Inverted lists are very large
  - e.g., 25-50% of collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good *compression ratios* and are easy to decompress
- *Lossless* compression – no information lost

Compression in inverted indexes

- First, we will consider compresing the dictionary → read book
  - Make it small enough to keep in main memory
- Then the postings
  - Reduce disk space needed, decrease time to read from disk
  - Large search engines keep a significant part of postings in memory
- (Each postings entry is a docID)

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer single bit per document

Delta Encoding

- Word count data is good candidate for compression
  - many small numbers and few larger numbers
  - encode the frequent small numbers with short codes
- Document id's are less predictable
  - but differences between docid's in an ordered list are smaller and more predictable
- *Delta encoding*:
  - encoding differences between document numbers (*d-gaps*)

Delta Encoding

- Inverted list of doc-id's (without counts)

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

- Differences between adjacent numbers

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

- Differences for a high-frequency word are easier to compress, e.g.,

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$$

- Differences for a low-frequency word are large, e.g.,

$$109, 3766, 453, 1867, 992, \ldots$$

- Hope: most gaps can be encoded/stored with far fewer than 20 bits.

Three postings entries

	encoding	postings list									
THE	docIDs	...		283042		283043		283044		283045	...
	gaps				1		1		1		...
COMPUTER	docIDs	...		283047		283154		283159		283202	...
	gaps				107		5		43		...
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

- Aim:
  - For ***arachnocentric***, we will use ~20 bits/gap entry.
  - For ***the***, we will use ~1 bit/gap entry.
- If the average gap for a term is *G*, we want to use ~$\log_2 G$ bits/gap entry.
- <u>Key challenge</u>: encode every integer (gap) with ~ as few bits as needed for that integer.
- Variable length codes achieve this by using short codes for small numbers

Variable Byte (VB) codes (variable length)

- For a gap value *G*, use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store *G* and dedicate 1 bit in it to be a continuation bit *c*
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) and of the other bytes to 0 ($c = 0$).

Example with MSB first

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Continuation bit

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a single byte.

Other variable length codes (FYI)

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc.
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.

I00041

As usual: assignment on Brightspace!

END of lecture