Information Retrieval

BITS Pilani
Pilani Campus

Abhishek

February 2020

innovate achieve lead

BITS Pilani
Pilani Campus

CS F469, Information Retrieval

Lecture topics: Index Compression

innovate achieve lead

BITS Pilani
Pilani Campus

Most of these slides are based on:

https://web.stanford.edu/class/cs276/
https://www.inf.unibz.it/~ricci/ISR/
https://www.cis.uni-muenchen.de/~hs/teach/14s/ir/

This Lecture

- Index Compression
  - How big will the dictionary and postings be?
    - Dictionary compression
    - **Postings compression**

Zipf's Law

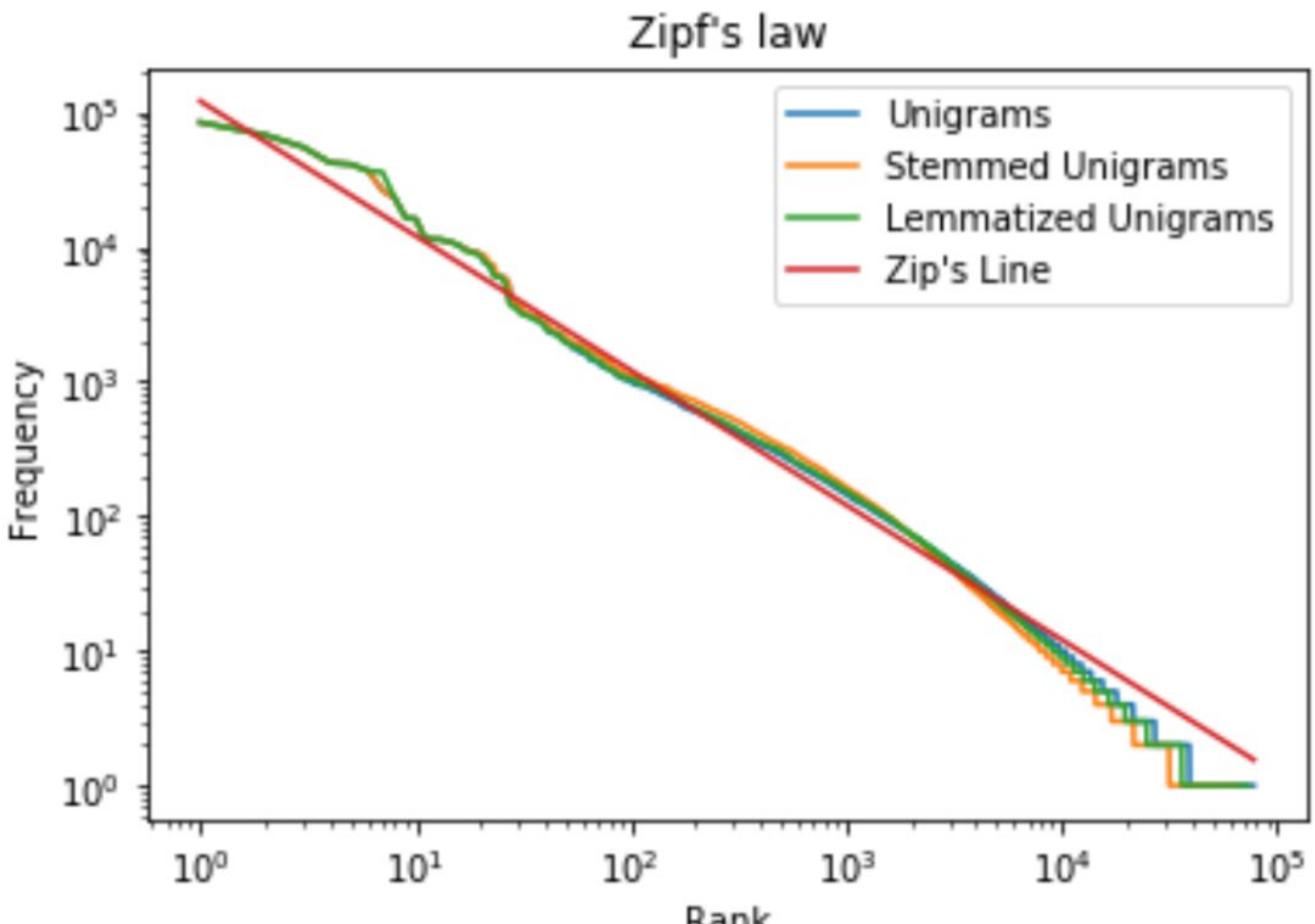
Zipf's Law

- In natural language, there are a few very frequent terms and very many very rare terms.
- **Zipf's law:** The **i**th most frequent term has frequency proportional to 1/**i**.
- $\mathbf{cf_i} \propto 1/\mathbf{i} = K/i$ where K is a normalizing constant.
- $\mathbf{cf_i}$ is collection frequency: the number of occurrences of the term t_i in the collection.

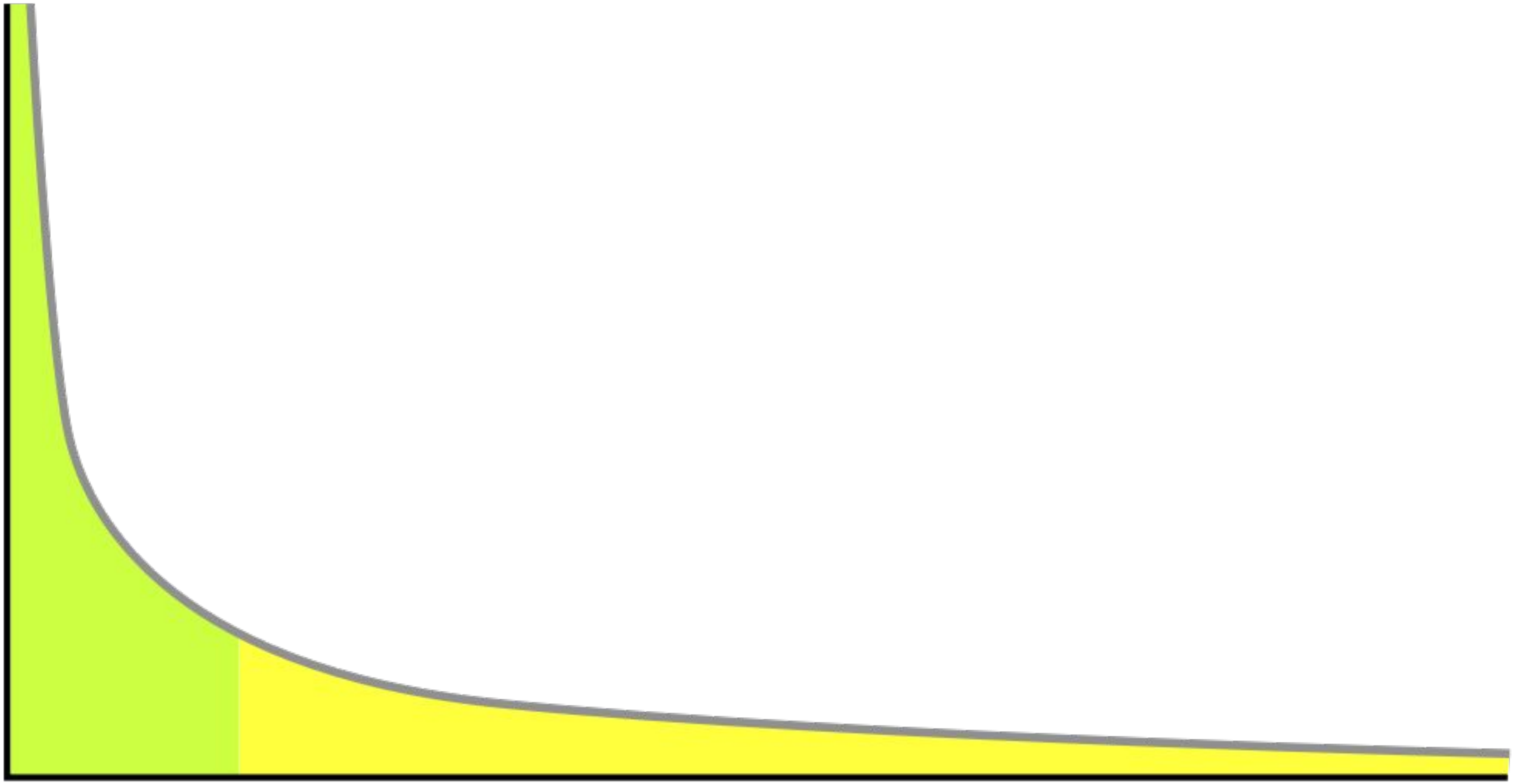
Zipf consequences

- If the most frequent term (**the**) occurs cf_1 times
- then the second most frequent term (**of**) occurs $cf_1/2$ times.
- the third most frequent term (**and**) occurs $cf_1/3$ times …
- Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
- $\log cf_i = \log K - \log i$
- Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship

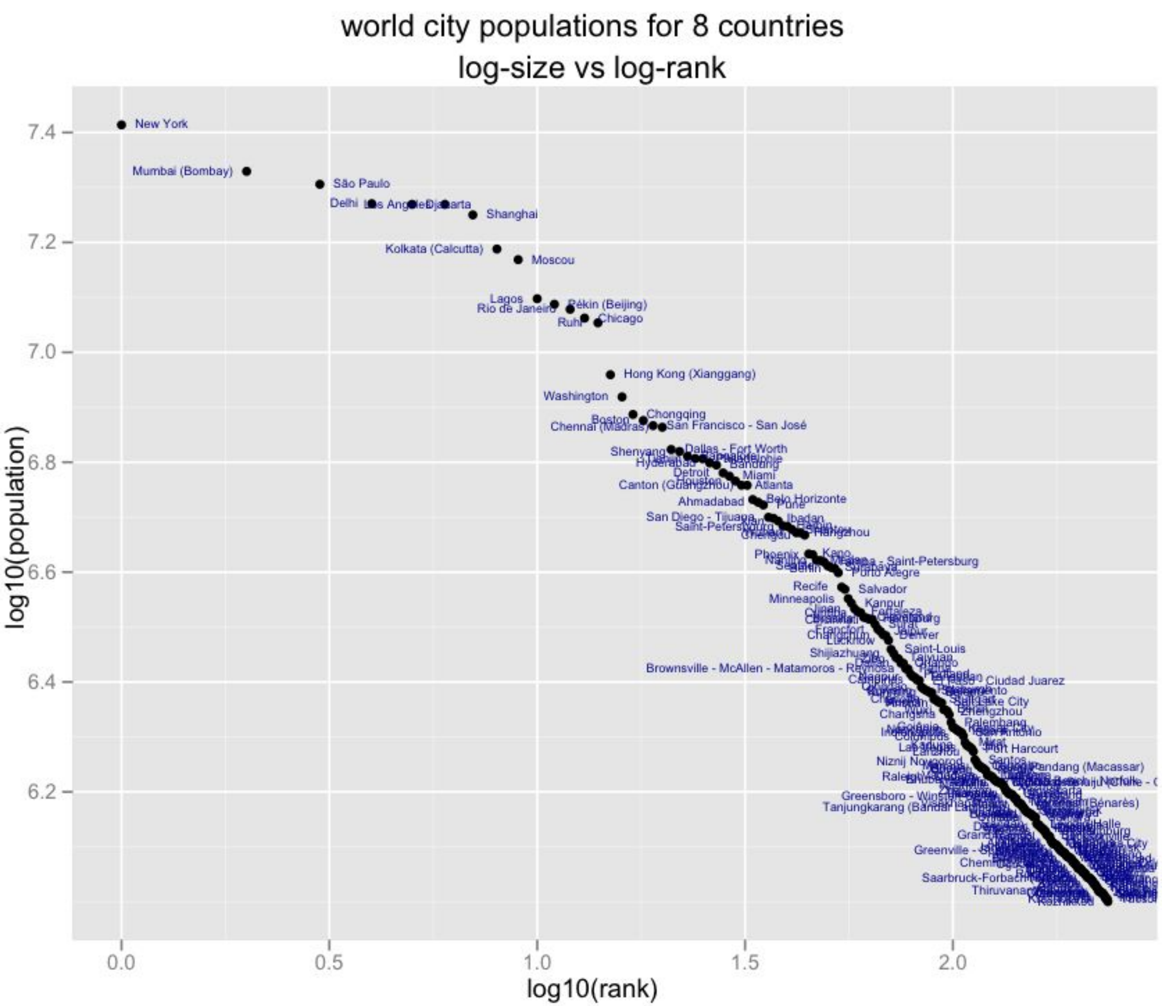
Zipf’s Law plot



Zipf’s Law plot



World city population



Source: https://brenocon.com/blog/2009/05/zipfs-law-and-world-city-populations/

Index Compression

Why Compression? (in General)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
- [read compressed data and decompress in memory]

  is faster than

  [read uncompressed data]
- Premise: Decompression algorithms are fast.

Why Compression in Inverted Index?

- **Dictionary**
  - Make it small enough to keep in main memory.
  - Make it so small that you can keep some postings lists in main memory too.
- **Postings file(s)**
  - Reduce disk space needed.
  - Decrease time needed to read postings lists from disk.
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory

Lossless vs. lossy compression

- **Lossless compression:** All information is preserved.
  - What we mostly do in IR.
- **Lossy compression:** Discard some information.
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- Chapter 7: Prune postings entries that are unlikely to turn up in the **top k** list for any query.
  - Almost no loss of quality in top k list

PNG vs JPEG

Posting Compression

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Need: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters corpus (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2(800{,}000) \approx 19.6 < 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID (on average).

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list:
  - COMPUTER: 283154, 283159, 283202, . . .
- It suffices to store gaps:
  - 283159 - 283154 = 5, 283202 - 283159 = 43
- Example postings list using gaps :
  - COMPUTER: 283154, 5, 43, . . .
- Gaps for frequent terms are small.
  - Thus: We can encode small gaps with fewer than 20 bits.

Gap Encoding

		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** and other rare terms, we will use about 20 bits per gap (= posting).
  - For **the** and other very frequent terms, we will use only a few bits per gap (= posting).

- In order to implement this, we need to devise some form of variable length encoding.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

Variable byte (VB) code

- Used by many commercial/research systems.
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit **c**.
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set **c** = 1.
- Else: encode lower-order 7 bits and then use one or more 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).

VB code examples

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

824 = 512 + 256 + 32 + 16 + 8

00000110 10111000 10000101 00001101 00001100 10110001

Other variable codes

- 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 on those.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent **n** as **n** 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111111110
  - Unary code for 70 is: 11111111111111111111111111111111111111111111111111 111111111111111111110

Gamma code

- Represent a gap G as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
- For example 13 → 1101 → 101 = offset
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

Gamma code examples

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Length of gamma code

- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits.
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- γ codes are always of odd length.

Gamma code: Properties

- Gamma code (like variable byte code) is prefix-free: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
- This result is independent of the distribution of gaps!
- We can use gamma codes for any distribution. Gamma code is universal.

Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits.
- Compressing and manipulating at granularity of bits can be slow.
- Variable byte encoding is aligned and thus potentially more efficient.
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost.

Compression of Reuters Corpus

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
T/D incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

Thank You!