Relevance

• Complex concept that has been studied for some time
  – Many factors to consider
  – People often disagree when making relevance judgments
• Retrieval models make various assumptions about relevance to simplify problem
  – e.g., topical vs. user relevance
  – e.g., binary vs. multi-valued relevance

Retrieval Model Overview

• Older models
  – Boolean retrieval
  – Overlap Measures
  – Vector Space model
• Probabilistic Models
  – BM25
  – Language models
• Combining evidence
  – Inference networks
  – Learning to Rank
Test Corpora

<table>
<thead>
<tr>
<th>Collection</th>
<th>Docs</th>
<th>Ds</th>
<th>Top</th>
<th>#Relev</th>
<th>TermDoc</th>
<th>Q &amp; D Relev</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>82</td>
<td>35</td>
<td></td>
<td>402</td>
<td>19,000</td>
<td></td>
</tr>
<tr>
<td>AAT</td>
<td>2309</td>
<td>14</td>
<td>2</td>
<td>450</td>
<td>19,000</td>
<td></td>
</tr>
<tr>
<td>CACIE</td>
<td>1004</td>
<td>64</td>
<td>3</td>
<td>34.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEL</td>
<td>1499</td>
<td>113</td>
<td>2</td>
<td>46.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodaGold</td>
<td>1499</td>
<td>113</td>
<td>2</td>
<td>55.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LISA</td>
<td>9372</td>
<td>70</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medline</td>
<td>1003</td>
<td>30</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NIST</td>
<td>11,029</td>
<td>193</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OBDMA</td>
<td>14,256</td>
<td>106</td>
<td>400</td>
<td>250</td>
<td>16,140</td>
<td></td>
</tr>
<tr>
<td>Reuters</td>
<td>21,678</td>
<td>672</td>
<td>28</td>
<td>139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREC</td>
<td>740,000</td>
<td>200</td>
<td>2000</td>
<td>89,360</td>
<td>110,000</td>
<td></td>
</tr>
</tbody>
</table>

Standard Benchmarks

- National Institute of Standards + Testing (NIST)
  - Has run large IR testbed for many years (TREC)
- Reuters and other benchmark sets used
- “Retrieval tasks” specified
  - sometimes as queries
- Human experts mark, for each query and for each doc, “Relevant” or “Not relevant”
  - or at least for subset that some system returned

Precision + Recall

- Precision: fraction of retrieved docs that are relevant = P(relevant|retrieved)
- Recall: fraction of relevant docs that are retrieved = P(retrieved|relevant)

\[
\text{Precision} = \frac{tp}{tp + fp}
\]

\[
\text{Recall} = \frac{tp}{tp + fn}
\]

Precision & Recall

Precision-Recall curve

Shows tradeoff

Precision-Recall Curves

- May return any # of results ordered by similarity
- By varying numbers of docs (levels of recall)
  - Produce a precision-recall curve

Precision/Recall

- Can get high recall (but low precision)
  - Retrieve all docs on all queries!
- Recall is a non-decreasing function of the number of docs retrieved
  - Precision usually decreases (in a good system)
- Difficulties in using precision/recall
  - Binary relevance
  - Should average over large corpus/query ensembles
  - Need human relevance judgements
  - Heavily skewed by corpus/authorship
A combined measure: F

- Combined measure assessing tradeoff is F measure (weighted harmonic mean):

\[ F = \frac{1}{\frac{1}{P} + (1 - \alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \]

- People usually use balanced F1 measure
  - i.e., with \( \beta = 1 \) or \( \alpha = \frac{1}{2} \)
- Harmonic mean is conservative average
  - See CJ van Rijsbergen, *Information Retrieval*

Other Measures

- Precision at fixed recall
  - This is perhaps the most appropriate thing for web search: all people want to know is how many good matches there are in the first one or two pages of results
- 11-point interpolated average precision
  - The standard measure in the TREC competitions: Take the precision at 11 levels of recall varying from 0 to 1 by tenths of the documents, using interpolation (the value for 0 is always interpolated!), and average them

Boolean Retrieval

- Two possible outcomes for query processing
  - TRUE and FALSE
  - “exact-match” retrieval
  - simplest form of ranking
- Query specified w/ Boolean operators
  - AND, OR, NOT
  - proximity operators also used

Query

- Which plays of Shakespeare contain the words *Brutus AND Caesar* but *NOT Calpurnia*?

Term-document incidence

<table>
<thead>
<tr>
<th>Term</th>
<th>Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worse</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 if play contains word, 0 otherwise

Booleans over Incidence Vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for *Brutus, Caesar* and *Calpurnia* (complemented) \( \Rightarrow \) bitwise AND.
- \( 110100 \text{ AND } 110111 \text{ AND } 101111 = 100100 \).
### Boolean Retrieval

**Advantages**
- Results are predictable, relatively easy to explain
- Many different features can be incorporated
- Efficient processing since many documents can be eliminated from search

**Disadvantages**
- Effectiveness depends entirely on user
- Simple queries usually don’t work well
- Complex queries are difficult

---

### Interlude

**Better Models Coming Soon:**
- Vector Space model
- Probabilistic Models
- BM25
- Language models

**Shared Issues – What to Index**
- Punctuation
- Case Folding
- Stemming
- Stop Words
- Spelling
- Numbers

---

### Issues in what to index

Cooper’s concordance of Wordsworth was published in 1911. The applications of full-text retrieval are legion: they include résumé scanning, litigation support and searching published journals online.

- Cooper’s vs. Cooper vs. Coopers.
- Full-text vs. full text vs. (full, text) vs. fulltext.
- résumé vs. resume.

---

### Punctuation

- Ne’er: use language-specific, handcrafted “locale” to normalize.
- State-of-the-art: break up hyphenated sequence.
- U.S.A. vs. USA - use locale.
- a.out

---

### Numbers

- 3/12/91
- Mar. 12, 1991
- 55 B.C.
- B-52
- 100.2.86.144
- Generally, don’t index as text
- Creation dates for docs

---

### Case folding

- Reduce all letters to lower case
- Exception: upper case in mid-sentence
  - e.g., General Motors
  - Fed vs. fed
  - SAIL vs. sail
Thesauri and Soundex

• Handle synonyms and homonyms
  – Hand-constructed equivalence classes
    • e.g., car = automobile
    • your ≠ you’re

• Index such equivalences?
• Or expand query?

Spell Correction

• Look for all words within (say) edit distance 3 (Insert/Delete/Replace) at query time
  – e.g., Alanis Morisette
• Spell correction is expensive and slows the query (up to a factor of 100)
  – Invoke only when index returns zero matches?
  – What if docs contain mis-spellings?

Lemmatization

• Reduce inflectional/variant forms to base form
  – am, are, is → be
  – car, cars, car’s, cars’ → car

the boy's cars are different colors

→

the boy car be different color

Stemming

• Reduce terms to their “roots” before indexing
  – language dependent
  – e.g., automate(s), automatic, automation all reduced to automat.

Porter’s algorithm

• Common algorithm for stemming English
• Conventions + 5 phases of reductions
  – phases applied sequentially
  – each phase consists of a set of commands
  – sample convention: Of the rules in a compound command, select the one that applies to the longest suffix.
• Porter’s stemmer available:
  http://www.sims.berkeley.edu/~hearst/irbook/porter.html

Typical rules in Porter

• sses → ss
• ies → i
• ational → ate
• tional → tion
Challenges

• Sandy
• Sanded
• Sander

Beyond Term Search

• Phrases?
• Proximity: Find Gates NEAR Microsoft.
  – Index must capture position info in docs.
• Zones in documents: Find documents with (author = Ullman) AND (text contains automata).

Ranking search results

• Boolean queries give inclusion or exclusion of docs.
• Need to measure proximity from query to each doc.
• Whether docs presented to user are singletons, or a group of docs covering various aspects of the query.

Ranking models in IR

• Key idea:
  – We wish to return in order the documents most likely to be useful to the searcher
• To do this, we want to know which documents best satisfy a query
  – An obvious idea is that if a document talks about a topic more then it is a better match
• A query should then just specify terms that are relevant to the information need, without requiring that all of them must be present
  – Document relevant if it has a lot of the terms

Retrieval Model Overview

• Older models
  – Boolean retrieval
  – Overlap Measures
  – Vector Space model
• Probabilistic Models
  – BM25
  – Language models
• Combining evidence
  – Inference networks
  – Learning to Rank

Binary term presence matrices

• Record whether a document contains a word: document is binary vector in \( \{0,1\}^\gamma \)
• Idea: Query satisfaction = overlap measure:

\[
|X \cap Y|
\]
Overlap matching

• What are the problems with the overlap measure?
  • It doesn’t consider:
    – Term frequency in document
    – Term scarcity in collection
      • (How many documents mention term?)
    – Length of documents

Many Overlap Measures

\[
\begin{align*}
|Q \cap D| & \quad \text{Simple matching (coordination level match)} \\
2 \frac{|Q \cap D|}{|Q| + |D|} & \quad \text{Dice’s Coefficient} \\
\frac{|Q \cap D|}{|Q| + |D|} & \quad \text{Jaccard’s Coefficient} \\
\frac{|Q \cap D|}{(|Q| \times |D|)^{\frac{1}{2}}} & \quad \text{Cosine Coefficient} \\
\frac{|Q \cap D|}{\min(|Q|, |D|)} & \quad \text{Overlap Coefficient}
\end{align*}
\]

Documents as vectors

• Each doc \( j \) can be viewed as a vector of \( tf \) values, one component for each term
• So we have a vector space
  – terms are axes
  – docs live in this space
  – even with stemming, may have 20,000+ dimensions
• (The corpus of documents gives us a matrix, which we could also view as a vector space in which words live – transposable data)

Vector Space Representation

Documents that are close to query (measured using vector-space metric) => returned first.

TF x IDF

\[
w_{ik} = tf_{ik} \times \log\left(\frac{N}{n_k}\right)
\]

\( T_k \) = term \( k \) in document \( D \)
\( tf_{ik} \) = frequency of term \( T_k \) in document \( D \)
\( idf_k \) = inverse document frequency of term \( T_k \) in \( C \)
\( idf_k = \log\left(\frac{N}{n_k}\right) \)

\( N \) = total number of documents in the collection \( C \)
\( n_k \) = the number of documents in \( C \) that contain \( T_k \)

BM25

Popular and effective ranking algorithm based on binary independence model
– adds document and query term weights

\[
\sum_{i \in Q} \log \left( \frac{(R + r_i + 0.5) / (N - r_i - K + r_i + 0.5)}{(R - r_i + 0.5) / (N - r_i - R + r_i + 0.5)} \right) \cdot \frac{(k_1 + 1) f_i}{K + f_i} \cdot \frac{(k_2 + 1) qf_i}{K_2 + qf_i}
\]

– \( N \) = number of doc, \( n_i \) = num containing term \( I \)
– \( R, r_i \) = encode relevance info (if avail, otherwise = 0)
– \( f_i \) = freq of term \( I \) in doc; \( qf_i \) = freq in query
– \( k_1, k_2 \) and \( K \) are parameters, values set empirically
  • \( k_1 \) weights \( tf \) component as \( f_i \) increases
  • \( k_2 \) = weights query term weight
  • \( K \) normalizes

adapted from Croft, Metzler, Strohman. © Addison Wesley
**Simple Formulas**

**But How Process Efficiently?**

**Thinking about Efficiency**

- **Clock cycle: 4 GHz**
  - Typically completes 2 instructions / cycle
  - ~10 cycles / instruction, but pipelining & parallel execution
  - Thus: 8 billion instructions / sec
- **Disk access: 1-10ms**
  - Depends on seek distance, published average is 5ms
  - Thus perform 200 seeks / sec
  - (And we are ignoring rotation and transfer times)
- **Disk is 40 Million times slower !!!**

**Retrieval**

Document-term matrix

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>...</th>
<th>tj</th>
<th>...</th>
<th>tm</th>
<th>nf</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>w11</td>
<td>w12</td>
<td>...</td>
<td>w1j</td>
<td>...</td>
<td>w1m</td>
</tr>
<tr>
<td>d2</td>
<td>w21</td>
<td>w22</td>
<td>...</td>
<td>w2j</td>
<td>...</td>
<td>w2m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>di</td>
<td>wij</td>
<td>w2j</td>
<td>...</td>
<td>wii</td>
<td>...</td>
<td>wim</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dn</td>
<td>wni</td>
<td>wni</td>
<td>...</td>
<td>wnj</td>
<td>...</td>
<td>wnm</td>
</tr>
</tbody>
</table>

wij is the weight of term tj in document di

Most wij’s will be zero.

**Naïve Retrieval**

Consider query Q = (q1, q2, …, qj, …, qm), nf = 1/|q|.

How evaluate Q?

(i.e., compute the similarity between q and every document)?

**Method 1: Compare Q with every doc.**

Document data structure:

- d1 : ((t1, w1i), (t2, w2i), …, (tj, wij), …, (tm, wim), 1/|d1|)
- Only terms with positive weights are kept.
- Terms are in alphabetic order.

Query data structure:

- Q : ((t1, q1), (t2, q2), …, (tj, qj), …, (tm, qmn), 1/|q|)

**Naïve Retrieval (continued)**

Method 1: Compare q with documents directly

initialize all sim(q, di) = 0;

for each document di (i = 1, …, n)

{ for each term tj (j = 1, …, m)

  if tj appears in both q and di

    sim(q, di) += qj * wij;

    sim(q, di) = sim(q, di) * (1/|q|) * (1/|di|); }

sort documents in descending similarities;

display the top k to the user;

**Observation**

- **Method 1 is not efficient**
  - Needs to access most non-zero entries in doc-term matrix.
- **Solution: Use Index (Inverted File)**
  - Data structure to permit fast searching.
- **Like an Index in the back of a text book.**
  - Key words --- page numbers.
  - E.g. "Etzioni, 40, 55, 60-63, 89, 220"
  - Lexicon
  - Occurrences
Search Processing (Overview)

1. Lexicon search
   - E.g. looking in index to find entry
2. Retrieval of occurrences
   - Seeing where term occurs
3. Manipulation of occurrences
   - Going to the right page

Simple Index for One Document

- A file is a list of words by position
- First entry is the word in position 1 (first word)
- Entry 4562 is the word in position 4562 (4562nd word)
- Last entry is the last word
- An inverted file is a list of positions by word!

Requirements for Search

- Need index structure
  - Must handle multiple documents
  - Must support phrase queries
  - Must encode TF/IDF values
  - Must minimize disk seeks & reads

How Store Index?

Lexicon

Oracle Database?

Unix File System?

The Solution

- Inverted Files for Multiple Documents
  - Broken into Two Files
- Lexicon
  - Hashtable on disk (one read)
  - Nowadays: stored in main memory
- Occurrence List
  - Stored on Disk
  - “Google Filesystem”
Using Inverted Files

Some data structures:

- **Lexicon**: a hash table for all terms in the collection.
- **Inverted file lists previously stored on disk.**
- **Now fit in main memory.**

**Using Inverted Files**

Several data structures:

1. For each term \( t_j \), create a list (occurrence file list) that contains all document ids that have \( t_j \).
   
   \[ l(t_j) = \{ (d_1, w_{ij}), (d_2, w_{ij}), \ldots \} \]

   - \( d_i \) is the document id number of the \( i \)th document.
   - Weights come from freq of term in doc
   - Only entries with non-zero weights are kept.

**The Lexicon**

- **Grows Slowly (Heap’s law)**
  - \( O(n^\beta) \) where \( n=\)text size; \( \beta \) is constant \(-0.4 \sim 0.6 \)
  - E.g. for 1GB corpus, lexicon = 5Mb
  - Can reduce with stemming (Porter algorithm)

- **Store lexicon in file in lexicographic order**
  - Each entry points to loc in occurrence file (aka inverted file list)
More Elaborate Inverted File

Several data structures:

2. For each term $t_j$, create a list (occurrence file list) that contains all document ids that have $t_j$.
   $l(t_j) = \{(d_1, freq, pos_1, \ldots, pos_k),
   (d_2, \ldots, \ldots)\}$
   - $d_i$ is the document id number of the $i^{th}$ document.
   - Weights come from freq of term in doc
   - Only entries with non-zero weights are kept.

Inverted files continued

More data structures:

3. Normalization factors of documents are pre-computed and stored similarly to lexicon
   $nf[i]$ stores $1/|d_i|$.

Retrieval Using Inverted Files

initialize all $sim(q, d_i) = 0$
for each term $t_j$ in $q$
   find $l(t)$ using the hash table
   for each $(d_i, w_{ij})$ in $l(t)$
      $sim(q, d_i) += q_j * w_{ij}$
   for each (relevant) document $d_i$
      $sim(q, d_i) = sim(q, d_i) * nf[i]$
sort documents in descending similarities and display the top $k$ to the user;

Observations about Method 2

- If doc $d$ doesn’t contain any term of query $q$, then $d$ won’t be considered when evaluating $q$.
- Only non-zero entries in the columns of the document-term matrix which correspond to query terms … are used to evaluate the query.
- Computes the similarities of multiple documents simultaneously (w.r.t. each query word)

Example (Method 2): Suppose

$q = \{(t_1, 1), (t_3, 1)\}$, $1/|q| = 0.7071$
$d_1 = \{(t_1, 2), (t_2, 1), (t_3, 1)\}$, $nf[1] = 0.4082$
$d_2 = \{(t_2, 2), (t_3, 1), (t_4, 1)\}$, $nf[2] = 0.4082$
$d_3 = \{(t_1, 1), (t_3, 1), (t_4, 1)\}$, $nf[3] = 0.5774$
$d_4 = \{(t_1, 2), (t_2, 1), (t_3, 2), (t_4, 2)\}$, $nf[4] = 0.2774$
$d_5 = \{(t_2, 2), (t_4, 1), (t_5, 2)\}$, $nf[5] = 0.3333$

After $t_1$ is processed:
   $sim(q, d_1) = 2, sim(q, d_2) = 0, sim(q, d_3) = 1, sim(q, d_4) = 0, sim(q, d_5) = 0$

Efficient Retrieval

Example (Method 2): Suppose

$q = \{(t_1, 1), (t_3, 1)\}$, $1/|q| = 0.7071$
$d_1 = \{(t_1, 2), (t_2, 1), (t_3, 1)\}$, $nf[1] = 0.4082$
$d_2 = \{(t_2, 2), (t_3, 1), (t_4, 1)\}$, $nf[2] = 0.4082$
$d_3 = \{(t_1, 1), (t_3, 1), (t_4, 1)\}$, $nf[3] = 0.5774$
$d_4 = \{(t_1, 2), (t_2, 1), (t_3, 2), (t_4, 2)\}$, $nf[4] = 0.2774$
$d_5 = \{(t_2, 2), (t_4, 1), (t_5, 2)\}$, $nf[5] = 0.3333$

After $t_1$ is processed:
   $sim(q, d_1) = 2, sim(q, d_2) = 0, sim(q, d_3) = 1, sim(q, d_4) = 0, sim(q, d_5) = 0$

Efficient Retrieval

Example (Method 2): Suppose

$q = \{(t_1, 1), (t_3, 1)\}$, $1/|q| = 0.7071$
$d_1 = \{(t_1, 2), (t_2, 1), (t_3, 1)\}$, $nf[1] = 0.4082$
$d_2 = \{(t_2, 2), (t_3, 1), (t_4, 1)\}$, $nf[2] = 0.4082$
$d_3 = \{(t_1, 1), (t_3, 1), (t_4, 1)\}$, $nf[3] = 0.5774$
$d_4 = \{(t_1, 2), (t_2, 1), (t_3, 2), (t_4, 2)\}$, $nf[4] = 0.2774$
$d_5 = \{(t_2, 2), (t_4, 1), (t_5, 2)\}$, $nf[5] = 0.3333$

After $t_1$ is processed:
   $sim(q, d_1) = 2, sim(q, d_2) = 0, sim(q, d_3) = 1, sim(q, d_4) = 0, sim(q, d_5) = 0$

After $t_1$ is processed:
   $sim(q, d_1) = 3, sim(q, d_2) = 1, sim(q, d_3) = 2, sim(q, d_4) = 0, sim(q, d_5) = 0$

After $t_3$ is processed:
   $sim(q, d_1) = .87, sim(q, d_2) = .29, sim(q, d_3) = .82, sim(q, d_4) = .78, sim(q, d_5) = 0$
Efficiency versus Flexibility

- Storing computed document weights is good for efficiency, but bad for flexibility.
  - Recomputation needed if TF and IDF formulas change and/or TF and DF information changes.
- Flexibility improved by storing raw TF, DF information, but efficiency suffers.
- A compromise
  - Store pre-computed TF weights of documents.
  - Use IDF weights with query term TF weights instead of document term TF weights.
Lexicon Construction

- Build Trie (or hash table)

This is a text. A text has many words. Words are made from letters.

Stop lists

- Language-based stop list:
  - words that bear little meaning
  - 20-500 words
  - http://www.dcs.gla.ac.uk/sandor/ir_resources/linguistic_utils/stop_words

- Subject-dependent stop lists

- Removing stop words
  - From document
  - From query

Stemming

- Are there different index terms?
  - retrieve, retrieving, retrieval, retrieved, retrieves...

- Stemming algorithm:
  - (retrieve, retrieving, retrieval, retrieved, retrieves) ⇒ retriev
  - Strips prefixes of suffixes (-s, -ed, -ly, -ness)
  - Morphological stemming

Stemming Continued

- Can reduce vocabulary by ~ 1/3
- C, Java, Perl versions, python, c#
  www.tartarus.org/~martin/PorterStemmer

- Criterion for removing a suffix
  - Does "a document is about \( w_1 \)" mean the same as
  - a "a document about \( w_2 \)"

- Problems: sand / sander & wand / wander

- Commercial SEs use giant in-memory tables

Memory Too Small?

- Merging
  - When word is shared in two lexicons
  - Concatenate occurrence lists
  - \( O(n_1 + n_2) \)

- Overall complexity
  - \( O(n \log(n/M)) \)

Compression

- What Should We Compress?
  - Repository
  - Lexicon
  - Inv Index

- What properties do we want?
  - Compression ratio
  - Compression speed
  - Decompression speed
  - Memory requirements
  - Pattern matching on compressed text
  - Random access
Inverted File Compression

Each inverted list has the form $<f_1, d_1, d_2, d_3, \ldots, d_k>$.

A naïve representation results in a storage overhead of $(f + n) \cdot [\log N]$.

This can also be stored as $<f_1|d_1|d_1 - d_0|d_2 - d_1|d_3 - d_2|\ldots|$.

Each difference is called a d-gap. Since $\sum (d - gap) \leq N$,

each pointer requires fewer than $[\log N]$ bits.

Trick is encoding …. since worst case …. Assume d-gap representation for the rest of the talk, unless stated otherwise.

Text Compression

Two classes of text compression methods

- Symbolwise (or statistical) methods
  - Estimate probabilities of symbols - modeling step
  - Use shorter code for the most likely symbol
  - Usually based on either arithmetic or Huffman coding
- Dictionary methods
  - Replace fragments of text with a single code word
  - Typically an index to an entry in the dictionary.
  - eg: Ziv-Lempel coding: replaces strings of characters with a pointer to a previous occurrence of the string.
  - No probability estimates needed

Symbolwise methods are more suited for coding d-gaps

Classifying d-gap Compression Methods:

- **Global:** each list compressed using same model
  - non-parameterized: probability distribution for d-gap sizes is predetermined.
  - parameterized: probability distribution is adjusted according to certain parameters of the collection.
- **Local:** model is adjusted according to some parameter, like the frequency of the term
- By definition, local methods are parameterized.

Conclusion

- **Local methods best**
  - Parameterized global models - non-parameterized
    - Pointers not scattered randomly in file
  - In practice, best index compression algorithm is:
    - Local Bernoulli method (using Golomb coding)
  - Compressed inverted indices usually faster+smaller than
    - Signature files
    - Bitmaps

Local $<$ Parameterized Global $<$ Non-parameterized Global

Not by much