Dictionaries and tolerant retrieval

CE-324: Modern Information Retrieval
Sharif University of Technology

M. Soleymani
Fall 2014

Most slides have been adapted from: Profs. Nayak & Raghavan (CS-276, Stanford)
Topics

- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex
Typical IR system architecture
The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?

### Dictionary data structures for inverted indexes

- **Brutus**: 1 2 4 11 31 45 173 174
- **Caesar**: 1 2 4 5 6 16 57 132 ...
- **Calpurnia**: 2 31 54 101

---

**dictionary**

**postings**
Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees
Hashtables

- Each vocabulary term is hashed to an integer
  - (We assume you’ve seen hashtables before)

- Pros:
  - Lookup is faster than for a tree: $O(1)$

- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*
Tree: binary tree

The diagram illustrates a binary tree with the root node labeled as Root. The tree branches into two main branches labeled a-m and n-z. Each branch further splits into smaller nodes labeled a-hu, hy-m, n-sh, and si-z. The nodes proceed downward in a hierarchical structure, indicating the organization of elements in a binary tree.
Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one

Pros:
- Solves the prefix problem (terms starting with hyp)

Cons:
- Slower: $O(\log M)$ [and this requires balanced tree]
- Rebalancing binary trees is expensive
  - But B-trees mitigate the rebalancing problem
Wild-card queries: *

- Query: *mon*
  - Any word beginning with “mon”.
  - Easy with binary tree (or B-tree) lexicon: retrieve all words in range: \( mon \leq w < moo \)

- Query: *mon*
  - Find words ending in “mon” (harder)
  - Maintain an additional tree for terms backwards. Can retrieve all words in range: \( nom \leq w < non \).

- How can we enumerate all terms matching pro*cent ?
B-trees handle *’s at the end of a term

- How can we handle *’s in the middle of query term?
  \[ \text{co}^*\text{tion} \]
  \[ \text{co}^*\text{AND }\text{*tion} \]

- Look up in the regular tree (for finding terms with the specified prefix) and the reverse tree (for finding terms with the specified suffix) and intersect these sets
  - Expensive

- **Solutions:**
  - *permuterm* index
  - *k-gram* index
Permuterm index

- For term **hello**, index under:
  - **hello$**, **ello$h**, **llo$he**, **lo$hel**, **o$hell**
    where $ is a special symbol.

- Transform wild-card queries so that the *’s occur at the end

- Query: **m*n**
  - **m*n** → **n$m*$**
  - Lookup **n$m*$** in the permutation index
  - Lookup the matched terms in the standard inverted index
Permuterm query processing

- **Permuterm problem**: \( \approx \) quadruples lexicon size

Empirical observation for English.
Bigram (\(k\)-gram) indexes

- Enumerate all \(k\)-grams (sequence of \(k\) chars)
  - e.g., “April is the cruelest month” into 2-grams (bigrams)
    - $ is a special word boundary symbol
    - \(sa, ap, pr, ri, il, l$, $i, is, s$, $t, th, he, e$, $c, cr, ru, ue, el, le, es, st, t$, $m, mo, on, nt, h$\)

- Maintain a second inverted index
  - \textit{from bigrams to dictionary terms} that match each bigram.
Bigram index example

- **k-gram index**: finds terms based on a query consisting of $k$-grams (here $k=2$).

```
$m\rightarrow mace\rightarrow madden$

$mo\rightarrow among\rightarrow amortize$

$on\rightarrow along\rightarrow among$
```
Bigram index: Processing wild-cards

- **Query:** `mon*`
  - $\rightarrow$ $m$ AND $mo$ AND $on$
  - But we’d enumerate *moon* (false positive).

- Must **post-filter** these terms against query.

- Run surviving ones through term-document inverted index.

- Fast, space efficient (compared to permuterm).
Processing wild-card queries

- Wild-cards can result in expensive query execution
  - `pyth* AND prog*`
  - As before, a Boolean query for each enumerated, filtered term (conjunction of disjunctions).

- If you encourage “laziness” people will respond!

Type your search terms, use ‘*’ if you need to. E.g., Alex* will match Alexander.
Spelling correction
Spell correction

- Two principal uses
  - Correcting doc(s) being indexed
  - Correcting user queries to retrieve “right” answers

- Two main flavors:
  - **Isolated word:**
    - Check each word on its own for misspelling. Will not catch typos resulting in correctly spelled words (e.g., *from* → *form*)
  - **Context-sensitive:**
    - Look at surrounding words,
      - e.g., *I flew* *form* Heathrow to Narita.
Document correction

- Especially needed for OCR’ed docs
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (that are adjacent on the keyboard and so more likely interchanged in typing query).

- But also: web pages

- Goal: the dictionary contains fewer misspellings

- But often we don’t change docs and instead fix query-doc mapping
Lexicon

- Fundamental premise – there is a lexicon from which the correct spellings come

- Two basic choices for this
  - A standard lexicon such as
    - Webster’s English Dictionary
    - An “industry-specific” lexicon (hand-maintained)
  - The lexicon of the indexed corpus (Including mis-spellings)
    - E.g., all words on the web
    - All names, acronyms etc.
Basic principles for spelling correction

- From correct spellings for a misspelled query choose the “nearest” one.

- When two correctly spelled queries are tied, select the one that is more common.
  - Query: grnt
  - Correction: grunt? grant?
Query mis-spellings

- We can either
  - Retrieve docs indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - Did you mean … ?
Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q

- We’ll study several alternatives for closeness
  - Edit distance (Levenshtein distance)
    - Weighted edit distance
  - n-gram overlap
Edit distance

- Given two strings $S_1$ and $S_2$, the minimum number of operations to convert one to the other

- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)

- E.g., the edit distance from **dof** to **dog** is 1
  - From **cat** to **act** is 2  (Just 1 with transpose.)
  - from **cat** to **dog** is 3.
Edit distance

- Generally found by dynamic programming.

```plaintext
EDITDISTANCE(s1, s2)
1 int m[i, j] = 0
2 for i ← 1 to |s1|
3   do m[i, 0] = i
4 for j ← 1 to |s2|
5   do m[0, j] = j
6 for i ← 1 to |s1|
7   do for j ← 1 to |s2|
8     do m[i, j] = min\{m[i - 1, j - 1] + if (s1[i] = s2[j]) then 0 else 1: m[i - 1, j] + 1, m[i, j - 1] + 1\}
9 return m[|s1|, |s2|]
```
Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - keyboard errors
    - Example: $m$ more likely to be mis-typed as $n$ than as $q$
      - $\Rightarrow$ replacing $m$ by $n$ is a smaller edit distance than by $q$
  - This may be formulated as a probability model

- Requires weight matrix as input
  - Modify dynamic programming to handle weights
Using edit distances

- A way: Given query,
  - enumerate all character sequences within a preset edit distance
  - Intersect this set with the list of “correct” words
  - Show terms you found to user as suggestions

- Alternatively,
  - We can look up all possible corrections in our inverted index and return all docs … slow
  - We can run with a single most likely correction
    - disempower the user, but save a round of interaction with the user
Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow

- How do we cut the set of candidate dictionary terms?
  - One possibility is to use $n$-gram overlap for this
    - This can also be used by itself for spelling correction.
$n$-gram overlap

- Enumerate all $n$-grams in the query

- Use the $n$-gram index for the lexicon to retrieve all terms matching an $n$-gram

- Threshold by number of matching $n$-grams
  - Variants – weights are also considered according to the keyboard layout, etc.

\[ \text{sm} \rightarrow \text{mace} \rightarrow \text{madden} \]
\[ \text{mo} \rightarrow \text{among} \rightarrow \text{amortize} \]
\[ \text{on} \rightarrow \text{along} \rightarrow \text{among} \]
Example with trigrams

- Suppose the text is *november*
  - Trigrams are *nov*, *ove*, *vem*, *emb*, *mbc*, *ber*.
- The query is *december*
  - Trigrams are *dec*, *ece*, *cem*, *emb*, *mbc*, *ber*.
- So 3 trigrams overlap (of 6 in each term)

- How can we turn this into a normalized measure of overlap?
One option – Jaccard coefficient

- A commonly-used measure of overlap between two sets:

\[ JC(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \]

- Properties
  - X and Y don’t have to be of the same size
  - Equals 1 when X and Y have the same elements and zero when they are disjoint
  - Always assigns a number between 0 and 1
    - Now threshold to decide if you have a match
    - E.g., if J.C. > 0.8, declare a match
Matching trigrams

- Consider the query **lord** – we wish to identify words matching 2 of its 3 bigrams (**lo, or, rd**)

Standard postings “merge” will enumerate …

Adapt this to using Jaccard (or another) measure.
Context-sensitive spell correction

- Phrase query: “flew form Heathrow”

- Text: I flew from Heathrow to Narita.

- We’d like to respond
  Did you mean “flew from Heathrow”? because no docs matched the query phrase.
Context-sensitive correction

- Need surrounding context to catch this.

- First idea: retrieve dictionary terms close to each query term

- Now try all possible resulting phrases with one word “fixed” at a time
  - flew from heathrow
  - fled form heathrow
  - flea form heathrow

- Hit-based spelling correction: Suggest the alternative that has lots of hits.
Another approach

- Break phrase query into a conjunction of biwords
- Look for biwords that need only one term be corrected.
- Enumerate only phrases containing “common” biwords.
General issues in spell correction

- Enumerate multiple alternatives for “Did you mean?”
- Need to figure out which to present to the user
  - The alternative hitting most docs
  - Query log analysis
- More generally, rank alternatives probabilistically
  \[ \text{argmax}_{\text{corr}} P(\text{corr} | \text{query}) \]
**Probabilistic Approach**

\[ \text{argmax}_{\text{corr}} P(\text{corr} \mid \text{query}) \]

- Bayes rule:

\[ \text{argmax}_{\text{corr}} P(\text{query} \mid \text{corr}) \ast P(\text{corr}) \]

Noisy channel

Language model
Soundex Algorithm
Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific (mainly for names)
  - E.g., chebyshev $\rightarrow$ tchebycheff

- Invented for the U.S. census … in 1918
Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form

- Do the same with query terms

- Soundex index: Build and search an index on the reduced forms
  - Used when the query calls for a soundex match

http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top
Soundex algorithm

1. Retain the first letter of the word.

2. Change all occurrences of the following letters to ‘0’: 'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y' → 0

3. Change letters to digits as follows:
   - B, F, P, V → 1
   - C, G, J, K, Q, S, X, Z → 2
   - D, T → 3
   - L → 4
   - M, N → 5
   - R → 6
Soundex algorithm

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions of the form:
   
   \(<\text{uppercase letter}> <\text{digit}> <\text{digit}> <\text{digit}>.\)

Example: \textit{Herman} → H655.

Will \textit{hermann} generate the same code?
Soundex

- Soundex is the classic algorithm
  - provided by most databases (Oracle, Microsoft, …)

- How useful is soundex?
  - Not very – for information retrieval
  - Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities

- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR
What queries can we process?

- We have
  - Positional inverted index with skip pointers
  - Wild-card index
  - Spell-correction
  - Soundex

- Queries such as
  \[(\text{SPELL(moriset)}/3 \text{ toron}*\text{to}) \text{ OR SOUNDEX(chaikofski)}\]
Resources

- IIR 3, MG 4.2
- Efficient spell retrieval:
- Nice, easy reading on spell correction:
  - Peter Norvig: How to write a spelling corrector http://norvig.com/spell-correct.html