Information Retrieval (CSD510)

Tolerant Retrieval and Spell-Checking

January 10, 2024

Lecture outline

1. Tolerant Retrieval
2. Spelling correction

Tolerant Retrieval

Wildcard queries

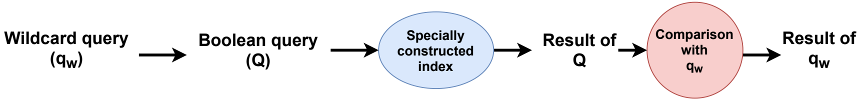
- Uncertain of the spelling of a query term (e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney)
- Seeking documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming (e.g., judicial vs. judiciary, leading to the wildcard query judicia*)
- Uncertain of the **correct rendition of a foreign word or phrase** (e.g., the query Universit* Stuttgart, Universität or University).

Wildcard queries

- **mon*** **(trailing wildcard query)**: find all docs containing any word beginning with "mon".
  - Easy with binary tree (or B-tree) lexicon: retrieve all words in the range: mon $\leq$ w $<$ moo
    ROOT $\rightarrow$ m $\rightarrow$ o $\rightarrow$ n
- ***mon** **(leading wildcard query)**: find all docs containing any word ending with "mon".
  - Maintain an additional B-tree (reverse B-tree)
  - **Reverse B-tree:** Each root-to-leaf path of the B-tree corresponds to a term in the dictionary written backwards.
  - Can retrieve all words in range: nom $\leq$ w $<$ non.
- How can we handle *'s in the middle of query term? **co*tion**
  1. Lookup for terms satisfying **co*** from B-tree
  2. Lookup for terms satisfying ***tion** from reverse B-tree
  3. Intersect the term sets

Wildcard queries

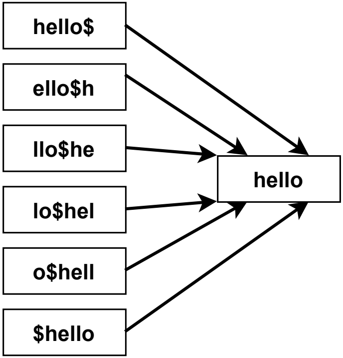
- Maintaining two B-trees and intersection operation are **expensive**.
- **Solutions:**
  1. **Permuterm** indexes
  2. *k*-**gram** indexes

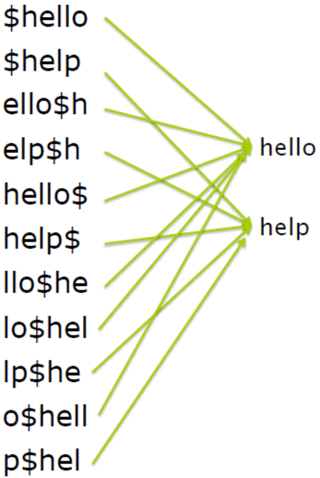


Permuterm index

- **Idea:** Transform wild-card queries so that the *'s occur at the end.
- **Steps for each term:**
  1. Introduce a special symbol **$** into the character set, to mark the end of a term.
  2. Generate the different possible rotations (**permutations**) of the character sequence of the term.
- Construct the *permuterm* index in which the different rotations of each term is linked to the original vocabulary term.
- **Permuterm vocabulary:** The set of rotated terms in the *permuterm index*.

Permuterm index





Permuterm index

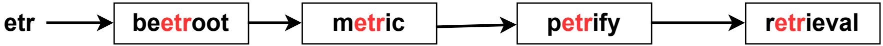
1. Consider the wildcard query m*n.
2. Rotate this wildcard query so that the * symbol appears at the end of the string
3. Rotated wildcard query becomes n$m*
4. Look up this string in the permuterm index (via a search tree) and get the terms (*man*, *moron*, *maintain*, *malfunction* etc.) in their rotated forms.
5. Map the rotations to the corresponding terms in the dictionary.
6. Search in the inverted index all the documents indexed by these terms

Permuterm index

- Wildcard query with multiple *s
- **Steps:**
  1. Consider the query: **fi*mo*er**
  2. Rotate to get the form with * at the end: **er$fi*mo***
  3. Find all terms that are in the permuterm index of **er$fi***
  4. Filter out the terms that contain **mo** in the middle by exhaustive enumeration
  5. Search in the inverted index all the documents indexed by these terms
- **Disadvantage:** The dictionary size becomes quite large due to inclusion of all rotations of the terms.

k-gram indexes

- A k-gram is a sequence of k characters.
- **castle**: *cas, ast, stl, tle* (3-grams)
- Full set of 3-grams for *castle*: *\$ca, cas, ast, stl, tle, le\$*
- In a k-gram index, the dictionary contains all k-grams that occur in any term in the vocabulary
- Each postings list points from a k-gram to all vocabulary terms containing that k-gram



k-gram indexes

Example 1

- 3-gram index; **Query:** re*ve
- **Boolean query:** $re AND ve$
- **Matching terms:** *relieve, retrieve, revive, remove* etc.
- Lookup the matching terms in the standard inverted index.

Example 1

- 3-gram index; **Query:** red*
- **Boolean query:** $re AND red
- **False positive:** **retired** (Does not match the query)
- **Post-filtering step:** String matching operation to remove the terms that do not match the query.

Spelling correction

Spelling correction

- Among the various alternative correct spellings for a misspelled query, choose the "nearest" one from the vocabulary.
  - We try to formalize the notions of *proximity* among the query terms.
  - Efficient computation
- When two correctly spelled queries are tied (or nearly tied)

  grnt: grunt and *grant*

  1. Choose the term that occurs more frequently in the collection
  2. Choose the term that is most common in the queries typed by other users.
- Two forms of correction
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words

      e.g., *from* $\rightarrow$ *form*
  - Context-sensitive
    - Takes surrounding words into account
    - **I flew <u>form</u> Heathrow to Narita**

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come.
  1. A standard lexicon such *a dictionary* or *a hand-maintained lexicon*
  2. The lexicon of the indexed corpus such as *all words on the web* and *all names, acronyms etc.*
- Given a lexicon and a character sequence Q, return the words in the lexicon *closest* to Q
- The two techniques are:
  1. Edit distance
  2. *k*-gram index

Edit distance

1. How to compute edit distance?
2. How to apply it in finding the vocabulary term closes to the query term?

- **Idea:** 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 and Replace
  - From **dof** to **dog** is **1**
  - From **cat** to **act** is **2**
  - From **cat** to **dog** is **3**
- **Levenshtein distance** is a measure for computing edit distance
- Levenshtein distance is a dynamic programming based algorithm.

Levenshtein distance

- S_1: Length m
- S_2: Length n
- **Initialize:** Fill the entries in a matrix M whose two dimensions equals the length of the two strings.
- The (i, j) entry of M will hold the edit distance between the sub-strings consisting of the first i characters of S_1 and the first j characters of S_2.

Levenshtein distance algorithm

LevenshteinDistance(s_1, s_2)

for $i \leftarrow 0$ **to** $|s_1|$
do $m[i, 0] = i$
for $j \leftarrow 0$ **to** $|s_2|$
do $m[0, j] = j$
for $i \leftarrow 1$ **to** $|s_1|$
do for $j \leftarrow 1$ **to** $|s_2|$
  do if $s_1[i] = s_2[j]$
    then $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]\}$
    else $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]+1\}$
return $m[|s_1|, |s_2|]$

Levenshtein distance algorithm

Cost of getting here from my upper left neighbour (by copy or replace)	Cost of getting here from my upper neighbour (by inserting a character in s_1)
Cost of getting here from my left neighbour (by inserting a character in s_2)	Minimum cost out of the three

Levenshtein distance algorithm

			f		a		s		t	
		0	1	1	2	2	3	3	4	4
c		1	1	2	2	3	3	4	4	5
		1	2	1	2	2	3	3	4	4
a		2	2	2	1	3	3	4	4	5
		2	3	2	3	1	2	2	3	3
t		3	3	3	3	2	2	3	2	4
		3	4	3	4	2	3	2	3	2
s		4	4	4	4	3	2	3	3	3
		4	5	4	5	3	4	2	3	3

Levenshtein distance algorithm

		f	a	s	t
	0	1	2	3	4
c	1	1	2	3	4
a	2	2	1	2	3
t	3	3	2	2	2
s	4	4	3	2	3

Using edit distances

- Given query, enumerate all character sequences within a preset edit distance (e.g., 2)
- Intersect this set with list of "correct" words
- Show the terms found to user as suggestions
- Alternatively,
  1. We can look up all possible corrections in our inverted index and return all docs $\cdots$ slow
  2. We can run with a single most likely correction
- The alternatives disempower the user, but save a round of interaction with the user
- Given a (misspelled) query, computation of edit distance to every dictionary term is slow and expensive
- Reduce the number of **candidate correct words** by *k*-gram overlap

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors
    Example: m more likely to be mistyped as n than as q
  - Therefore, 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 the Levenshtein's distance implementation to incorporate the *weight handling*.

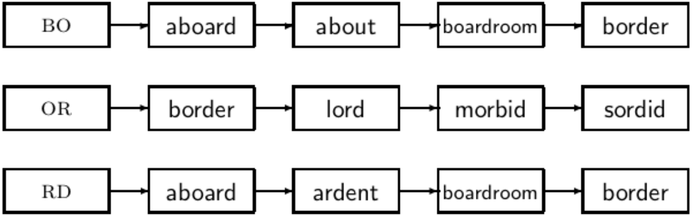
k-gram index

- Enumerate all *k*-grams in the query term

  Example: bigram index, misspelled word **bordroom**
- Bigrams: **bo, or, rd, dr, ro, oo, om**
- Use the k-gram index to retrieve vocabulary terms that have many k-grams in common with the query

k-gram index (Query term: **bord**)

1. **2-grams:** *bo, or, rd*
2. **Heuristic:** Find terms that contain at least $n - 1$ of the k-grams. $(n = 3)$
3. aboard, boardroom, border



- Implausible correction "boardroom" gets listed.
- Sophisticated measure of overlap between *query term* and *vocabulary term* required to filter candidates.

Jaccard co-efficient

- Jaccard coefficient $= \frac{|A \cap B|}{|A \cup B|}$
- Compute Jaccard coefficient between the
  1. set of bigrams in the query term (Q)
  2. set of bigrams in the vocabulary term (T)
- Select the vocabulary terms with *co-efficient value* greater than a threshold
- Enumeration of bigrams on-the-fly is expensive
- Use string lengths for computation

1. **bord:** String length - 4, 2-grams: 3
2. **boardroom:** String length - 9, 2-grams: 8
3. Overlap of two 2-grams.
4. Jaccard co-efficient $= \frac{2}{8+3-2}$

Context-sensitive spell correction

- Take the Example : *flew* **form** form *munich*
  - How can we correct **form** here?
- One idea: hit-based spelling correction

1. Retrieve vocabulary terms close to each query term for "*flew form munich*":

   flea for flew, from for form, munch for munich

2. Now try all possible resulting phrases as queries with one word "fixed" at a time. Try the queries
   - *flea form munich*
   - *flew from munich*
   - *flew form munch*
3. The correct query "*flew from munich*" has the most hits

- Not efficient.
- Better source of information: large corpus of queries, not documents

Phonetic correction

- Misspellings that arise because the user types a query that sounds like the target term
- The **main idea** is to generate, for each term, a "**phonetic hash**" so that similar-sounding terms hash to the same value.
- Algorithms for such phonetic hashing are *collectively* known as **soundex algorithms**.
- Use class of heuristics to expand a query into *phonetic equivalents*.

  e.g., **chebyshev** $\rightarrow$ **tchebycheff**

1. How to encode a query term?
2. How to use the encoding to map the **query term** to similar sounding **vocabulary term**?

Soundex - algorithm

1. Retain the first letter of the word
2. Change all occurrences of the letters to digits based on the following rules
    1. A, E, I, O, U, H, W, Y $\rightarrow$ 0
    2. B, F, P, V $\rightarrow$ 1
    3. C, G, J, K, Q, S, X, Z $\rightarrow$ 2
    4. D,T $\rightarrow$ 3
    5. L $\rightarrow$ 4
    6. M, N $\rightarrow$ 5
    7. R $\rightarrow$ 6
3. Remove all pairs of consecutive digits
4. Remove all zeros from the resulting string.
5. Pad the resulting string with trailing zeros and return the first four positions

    <**uppercase letter**> <**digit**> <**digit**> <**digit**>

Soundex-based mapping

- Turn every token to be indexed into a **4-character reduced form**.
- **Soundex index:** Build an inverted index from these reduced forms to the original terms
- Repeat the same with query terms
- When the query calls for a soundex match, search this soundex index