NPFL103: Information Retrieval (2)

Dictionaries, Tolerant retrieval, Spelling correction

Pavel Pecina

pecina@ufal.mff.cuni.cz

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University

Based on slides by Hinrich Schütze, University of Stuttgart.

Contents

Dictionaries

Inverted index

For each term *t*, we store a list of all documents that contain *t*.

BRUTUS	$\longrightarrow$	1	2	4	11	31	45	173	174	
CAESAR	$\longrightarrow$	1	2	4	5	6	16	57	132	…
CALPURNIA	$\longrightarrow$	2	31	54	101					
⋮										

$\underbrace{\qquad\qquad}_{\text{dictionary}}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{postings}}$

Dictionary as array of fixed-width entries

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...
- Assume for the time being that we can store this information in a fixed-length entry.
- Assume that we store these entries in an array.

Dictionary as array of fixed-width entries

Dictionary:

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

Space needed: 20 bytes 4 bytes 4 bytes

1. How do we look up a query term q_i in this array at query time?
2. Which data structure do we use to locate the entry (row) in the array where q_i is stored?

Data structures for looking up term

- Two main classes of data structures: hashes and trees.
- Some IR systems use hashes, some use trees.
- Criteria for when to use hashes vs. trees:
  1. Is there a fixed number of terms or will it keep growing?
  2. What are the frequencies with which various keys will be accessed?
  3. How many terms are we likely to have?

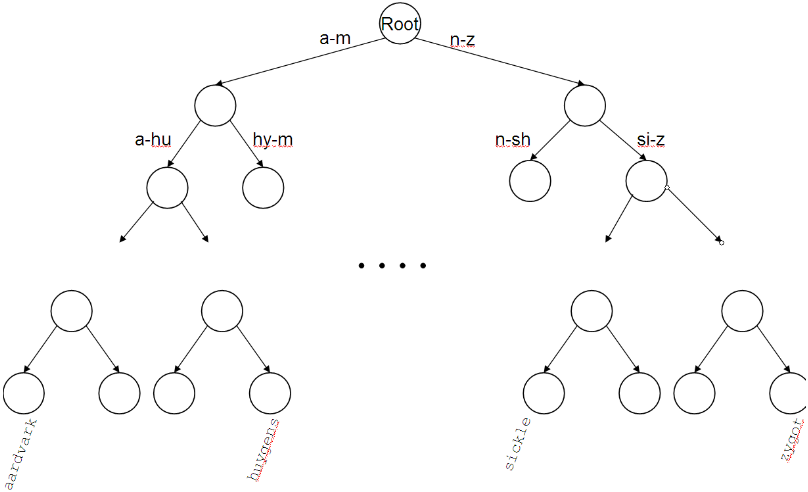
Hashes

- Each vocabulary term is hashed into an integer.
- Try to avoid collisions
- At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array
- Pros:
  1. Lookup in a hash is faster than lookup in a tree.
  2. Lookup time is constant.
- Cons:
  1. no way to find minor variants (*resume* vs. *résumé*)
  2. no prefix search (all terms starting with *automat*)
  3. need to rehash everything periodically if vocabulary keeps growing

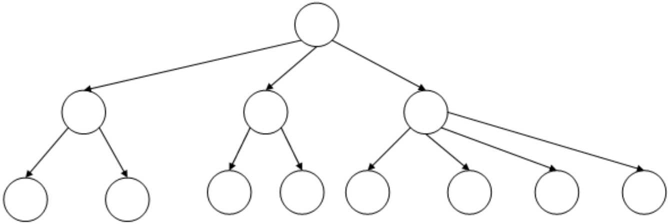
Trees

- Trees solve the prefix problem (e.g. find all terms starting with *auto*).
- Search is slightly slower than in hashes: $O(\log M)$, where M is the size of the vocabulary
- $O(\log M)$ only holds for balanced trees. Rebalancing is expensive.
- B-trees mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval $[a, b]$ where a, b are appropriate positive integers, e.g., $[2, 4]$.
- Simplest tree: binary tree

Binary tree example



B-tree example



Wildcard queries

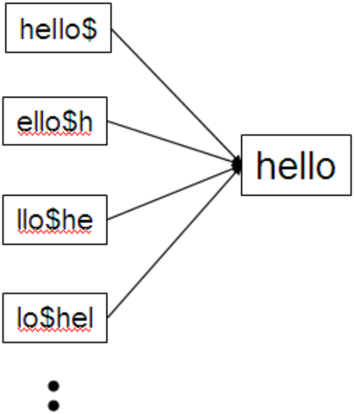
Wildcard queries

- *mon**: find all docs containing any term beginning with *mon*
- With B-tree dictionary: find all terms t in the range $mon \leq t < moo$
- **mon*: find all docs containing any term ending with *mon*
  1. Maintain an additional tree for terms *backwards*
  2. Retrieve all terms t in the range: nom $\leq t <$ non
- Result: A set of terms that are matches for wildcard query
- Then retrieve documents that contain any of these terms

How to handle * in the middle of a term

- Example: *m*nchen*
- Simple approach: We look up *m** and **nchen* in the backward B-tree and intersect the two sets of terms (expensive).
- Alternative: permuterm index
  - Basic idea: Rotate every wildcard query so that * occurs at the end.
  - Store each of these rotations in the dictionary, say, in a B-tree
  - For term HELLO: add *hello$*, *ello$h*, *llo$he*, *lo$hel*, and *o$hell* to the B-tree where $ is a special symbol

Permuterm → term mapping



Permuterm index

- For HELLO, we've stored: *hello$*, *ello$h*, *llo$he*, *lo$hel*, and *o$hell*
- Queries:
  - For X, look up X$
  - For X*, look up $X*
  - For *X, look up X$*
  - For *X*, look up X*
  - For X*Y, look up Y$X*
- Example: For *hel*o*, look up *o$hel**

Processing a lookup in the permuterm index

- Rotate query wildcard to the right
- Use B-tree lookup as before
- Problem: Permuterm more than quadruples the size of the dictionary compared to a regular B-tree (empirical estimation).

k-gram indexes

- More space-efficient than permuterm index
- Enumerate all character *k*-grams (sequence of *k* characters) occurring in a term (2-grams are called bigrams).
- Example: from "*April is the cruelest month*" we get the bigrams: *$a ap pr ri il l$ $i is s$ $t th he e$ $c cr ru ue el le es st t$ $m mo on nt h$*
- $ is a special word boundary symbol, as before.
- Maintain an inverted index from bigrams to the terms that contain the bigram.

Postings list in a 3-gram inverted index

k-gram (bigram, trigram, ...) indexes

- Note that we now have two different types of inverted indexes
- The term-document inverted index for finding documents based on a query consisting of terms

BRUTUS $\longrightarrow$ | 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174 |

- The *k*-gram index for finding terms based on a query *k*-grams

etr → BEETROOT → METRIC → PETRIFY → RETRIEVAL

Processing wildcarded terms in a bigram index

- Query *mon** can now be run as: *$m* AND *mo* AND *on*
- Gets us all terms with the prefix *mon* ...

  ...but also many "false positives" like MOON.
- We must postfilter these terms against query.
- Surviving terms are then looked up in term-document inverted index.
- *k*-gram index vs. permuterm index
  - *k*-gram index is more space efficient.
  - Permuterm index doesn't require postfiltering.

Exercise

- Google has very limited support for wildcard queries.
- Query example which doesn't work well on Google: *[gen* universit*]*
  - Intention: you are looking for the University of Geneva, but don't know which accents to use for the French words for university and Geneva.
- According to Google search basics, 2010-04-29: "Note that the * operator works only on whole words, not parts of words."
- But this is not entirely true. Try e.g. *[pythag*]*
- Exercise: Why doesn't Google fully support wildcard queries?

Processing wildcard queries in the term-document index

- Problem 1: Potential execution of a large number of Boolean queries.
  - Most straightforward semantics: Conjunction of disjunctions
  - For *[gen* universit*]*: *geneva university* OR *geneva université* OR *genève university* OR *genève université* OR *general universities* OR ...
  - Very expensive
- Problem 2: Users hate to type.
  - If abbreviated queries like *[pyth* theo*]* for *[pythagoras' theorem]* are allowed, users will use them a lot.
  - This would significantly increase the cost of answering queries.
  - Somewhat alleviated by Google Suggest

Spelling correction

Spelling correction

- Two principal uses:
  1. Correcting documents being indexed
  2. Correcting user queries at query time
- Two different methods for spelling correction:

1. Isolated word spelling correction
   - Check each word on its own for misspelling
   - Will not catch typos resulting in correctly spelled words, e.g., *an asteroid that fell form the sky*
2. Context-sensitive spelling correction
   - Look at surrounding words
   - Can correct *form*/*from* error above

Correcting documents vs. correcting queries

- We're not interested in interactive spelling correction of documents.
- In IR, we use document correction primarily for OCR'ed documents. (OCR = optical character recognition)
- The general philosophy in IR is: don't change the documents.
- Spelling errors in queries are much more frequent

Isolated word spelling correction

- Premises:
  1. There is a list of “correct words” from which the correct spellings come.
  2. We have a way of computing the distance between a misspelled word and a correct word.
- Simple algorithm: return the “correct” word that has the smallest distance to the misspelled word.
- Example: *informaton* $\rightarrow$ *information*
- For the list of correct words, we can use the vocabulary of all words that occur in our collection.
- Why is this problematic?

Alternatives to using the term vocabulary

- A standard dictionary (Webster's, OED etc.)
- An industry-specific dictionary (for specialized IR systems)
- The term vocabulary of the collection, appropriately weighted

Distance between misspelled word and "correct" word

We will discuss several alternatives:

1. Edit distance and Levenshtein distance
2. Weighted edit distance
3. *k*-gram overlap

Edit distance

- The edit distance between string s_1 and string s_2 is the minimum number of basic operations that convert s_1 to s_2.
- Levenshtein: The basic operations are insert, delete, and replace.
- Examples:
  - Levenshtein distance *dog*-*do*: 1
  - Levenshtein distance *cat*-*cart*: 1
  - Levenshtein distance *cat*-*cut*: 1
  - Levenshtein distance *cat*-*act*: 2
- Damerau-Levenshtein: transposition as a fourth possible operation.
- Example:
  - Damerau-Levenshtein distance *cat*-*act*: 1

Levenshtein distance

Levenshtein distance: Computation

		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

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 $m[|s_1|, |s_2|]$

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein distance: Example

			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*

Each cell of Levenshtein matrix

	↘	↓
	cost of getting here from my upper left neighbor → copy/replace	cost of getting here from my upper neighbor → delete
→	cost of getting here from my left neighbor → insert	the minimum of the three possible "movements"; the cheapest way of getting here

Example: Levenshtein distance OSLO – SNOW

		s	n	o	w
	0	1 1	2 2	3 3	4 4
o	1 1	1 2 2 1	2 3 2 2	2 4 3 2	4 5 3 3
s	2 2	1 2 3 1	2 3 2 2	3 3 3 3	3 4 4 3
l	3 3	3 2 4 2	2 3 3 2	3 4 3 3	4 4 4 4
o	4 4	4 3 5 3	3 3 4 3	2 4 4 2	4 5 3 3

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

Example: Levenshtein distance CAT – CATCAT

		c	a	t	c	a	t
	0	1 1	2 2	3 3	4 4	5 5	6 6
c	1 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4	6 7 5 5
a	2 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4
t	3 3	3 2 4 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3

cost	operation	input	output
1	insert	*	c
1	insert	*	a
1	insert	*	t
0	(copy)	c	c
0	(copy)	a	a
0	(copy)	t	t

Example: Levenshtein distance CAT – CATCAT

		c	a	t	c	a	t
	0	1 1	2 2	3 3	4 4	5 5	6 6
c	1 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4	6 7 5 5
a	2 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4
t	3 3	3 2 4 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3

cost	operation	input	output
0	(copy)	c	c
1	insert	*	a
1	insert	*	t
1	insert	*	c
0	(copy)	a	a
0	(copy)	t	t

Example: Levenshtein distance CAT – CATCAT

		c	a	t	c	a	t
	 0	 **1** **1**	 **2** **2**	 **3** **3**	 **4** **4**	 **5** **5**	 **6** **6**
c	**1** **1**	**0** 2 2 **0**	2 3 **1** **1**	3 4 **2** **2**	**3** 5 **3** **3**	5 6 **4** **4**	6 7 **5** **5**
a	**2** **2**	2 **1** 3 **1**	**0** 2 2 **0**	2 3 **1** **1**	3 4 **2** **2**	**3** 5 **3** **3**	5 6 **4** **4**
t	**3** **3**	3 **2** 4 **2**	2 **1** 3 **1**	**0** 2 2 **0**	2 3 **1** **1**	3 4 **2** **2**	**3** 5 **3** **3**

cost	operation	input	output
0	(copy)	c	c
0	(copy)	a	a
1	insert	*	t
1	insert	*	c
1	insert	*	a
0	(copy)	t	t

Example: Levenshtein distance CAT – CATCAT

		c	a	t	c	a	t
	0	1 1	2 2	3 3	4 4	5 5	6 6
c	1 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4	6 7 5 5
a	2 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3	5 6 4 4
t	3 3	3 2 4 2	2 1 3 1	0 2 2 0	2 3 1 1	3 4 2 2	3 5 3 3

cost	operation	input	output
0	(copy)	c	c
0	(copy)	a	a
0	(copy)	t	t
1	insert	*	c
1	insert	*	a
1	insert	*	t

Weighted edit distance

- As above, but operation weights depend on the characters involved.
- Meant to capture keyboard errors
  (e.g., *m* more likely to be mistyped as *n* than as *q*).
- Therefore, replacing *m* by *n* is a smaller edit distance than by *q*.
- Requires a weight matrix as input.
- The dynamic programming need to be modified to handle weights.

Using edit distance for spelling correction

- Given a query, first enumerate all character sequences within a preset (possibly weighted) edit distance.
- Intersect this set with our list of "correct" words.
- Then suggest terms in the intersection to the user.
- $\rightarrow$ exercise in a few slides.

k-gram indexes for spelling correction

- 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 "correct" words that match query term *k*-grams
- Threshold by number of matching *k*-grams (e.g., only vocabulary terms that differ by at most 3 *k*-grams)

k-gram indexes for spelling correction: *bordroom*

BO → aboard → about → boardroom → border

OR → border → lord → morbid → sordid

RD → aboard → ardent → boardroom → border

Context-sensitive spelling correction

- Our example was: *an asteroid that fell form the sky*
- How can we correct *form* here?
- One idea: hit-based spelling correction (hit = retrieved document)
  1. Retrieve "correct" terms close to each query term
     for *flew form munich*: *flea* for *flew*, *from* for *form*, *munch* for *munich*
  2. Try all possible phrases as queries with one word "fixed" at a time:
     "flea form munich", *"flew from munich"*, *"flew form munch"*
  3. The correct query *"flew from munich"* has the most hits.
- Suppose we have 7 alternatives for *flew*, 20 for *form* and 3 for *munich*, how many "corrected" phrases will we enumerate?

Context-sensitive spelling correction cont'd.

- The "hit-based" algorithm we just outlined is not very efficient.
- More efficient alternative: look at "collection" of queries (query logs), not documents.
- Another alternative: learn corrections from the users (mine query logs for sequences of a *incorrect query* followed by a *corrected query*).

General issues in spelling correction

- User interface:
  - automatic vs. suggested correction
  - *Did you mean* only works for one suggestion.
  - What about multiple possible corrections?
  - Tradeoff: simple vs. powerful UI
- Cost:
  - Spelling correction is potentially expensive.
  - Avoid running on every query?
  - Maybe just on queries that match few documents.
  - Guess: Spelling correction of major search engines is efficient enough to be run on every query.

Soundex

Soundex

- Soundex is the basis for finding phonetic (as opposed to orthographic) alternatives (in English).
- Example: *chebyshev* / *tchebyscheff*
- Algorithm:
  1. Turn every token to be indexed into a 4-character reduced form
  2. Do the same with query terms
  3. Build and search an index on the reduced forms

Soundex algorithm

1. Retain the first letter of the term.
2. Change all occurrences of the following letters to '0' (zero): A, E, I, O, U, H, W, Y
3. Change letters to digits as follows:
   - B, F, P, V to 1
   - C, G, J, K, Q, S, X, Z to 2
   - D,T to 3
   - L to 4
   - M, N to 5
   - R to 6
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all zeros from the resulting string; pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

Example: Soundex of *HERMAN*

- Retain H
- *ERMAN* $\rightarrow$ *0RM0N*
- *0RM0N* $\rightarrow$ *06505*
- *06505* $\rightarrow$ *06505*
- *06505* $\rightarrow$ *655*
- Return *H655*
- Note: *HERMANN* will generate the same code

How useful is Soundex?

- Not very – for information retrieval
- OK for "high recall" tasks in other applications (e.g., Interpol)
- Zobel and Dart (1996) suggest better alternatives for phonetic matching in IR.