Introduction to Information Retrieval

http://informationretrieval.org

IIR 3: Dictionaries and tolerant retrieval

Hinrich Schütze

Institute for Natural Language Processing, Universität Stuttgart

2008.04.29

Overview

1. Recap
2. Dictionaries
3. Wildcard queries
4. Spelling correction
5. Soundex

Outline

1. Recap
2. Dictionaries
3. Wildcard queries
4. Spelling correction
5. Soundex

Type/token distinction

- Token – An instance of a word or term occurring in a document.
- Type – An equivalence class of tokens.
- *In June, the dog likes to chase the cat in the barn.*
- How many tokens? How many types?
- 12 tokens, 9 types

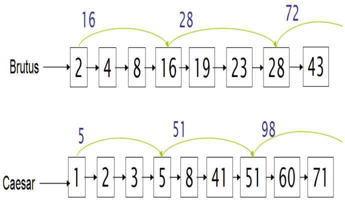
Problems in tokenization

- What are the delimiters? Space? Apostrophe? Hyphen?
- For each of these: sometimes they delimit, sometimes they don't.
- No whitespace in many languages! (e.g., Chinese)
- No whitespace in Dutch, German, Swedish compounds (*Lebensversicherungsgesellschaftsangestellter*)
- No whitespace in English: *database*, *whitespace*

Problems in "equivalence classing"

- A term is an equivalence class of tokens.
- How do we define equivalence classes?
- Numbers (3/12/91 vs. 12/3/91)
- Case folding
- Stemming, Porter stemmer
- Morphological analysis: inflectional vs. derivational
- Equivalence classing problems in other languages
  - More complex morphology than in English
  - Finnish: a single verb may have 12,000 different forms
  - Words written in different alphabets (Hiragana vs. Chinese characters)
  - Accents, umlauts

Skip pointers



Positional indexes

- Postings lists in a positional index: each posting is a docID and a list of positions
- Example: *to*$_1$ *be*$_2$ *or*$_3$ *not*$_4$ *to*$_5$ *be*$_6$

TO, 993427:
⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
2, 5: ⟨1, 17, 74, 222, 255⟩;
4, 5: ⟨8, 16, 190, 429, 433⟩;
5, 2: ⟨363, 367⟩;
7, 3: ⟨13, 23, 191⟩; . . . ⟩

BE, 178239:
⟨ 1, 2: ⟨17, 25⟩;
4, 5: ⟨17, 191, 291, 430, 434⟩;
5, 3: ⟨14, 19, 101⟩; . . . ⟩ Document 4 is a match.

Positional indexes

- With a positional index, we can answer phrase queries.
- With a positional index, we can answer proximity queries.

Outline

1. Recap
2. Dictionaries
3. Wildcard queries
4. Spelling correction
5. Soundex

Inverted index

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

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**

Inverted index

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

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**

Dictionaries

- The dictionary is the data structure for storing the term vocabulary.
- Term vocabulary: the data
- Dictionary: the data structure for storing the term vocabulary

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

	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

How do we look up an element in this array at query time?

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:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?

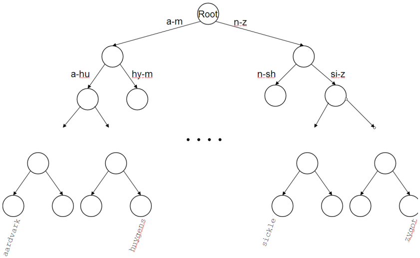
Hashes

- 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: Lookup in a hash is faster than lookup in a tree.
- Cons
  - no way to find minor variants (*resume* vs. *résumé*)
  - no prefix search (all terms starting with *automat*)
  - need to rehash everything periodically if vocabulary keeps growing

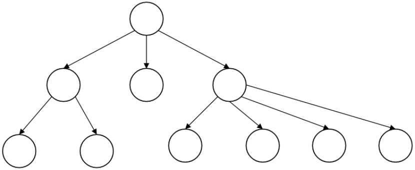
Trees

- Trees solve the prefix problem (find all terms starting with *automat*).
- Simplest tree: binary tree
- 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 binary trees 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., $[2, 4]$.
- Note that we need a standard ordering for characters in order to be able to use trees.

Binary tree



B-tree



Outline

1. Recap
2. Dictionaries
3. Wildcard queries
4. Spelling correction
5. Soundex

Wildcard queries

- mon*: find all docs containing any term beginning with *mon*
- Easy with B-tree dictionary: retrieve all terms t in the range: mon $\leq t <$ moo
- *mon: find all docs containing any term ending with *mon*
  - Maintain an additional tree for terms *backwards*
  - Then retrieve all terms t in the range: nom $\leq t <$ non

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wildcard query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query: gen* AND universit*
- This may result in the execution of many Boolean AND queries.

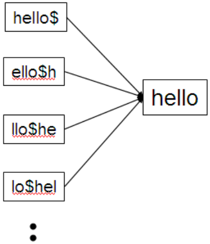
How to handle * in the middle of a term

- Example: m*nchen
- We could look up m* and *nchen in the B-tree and intersect the two term sets.
- Expensive
- Alternative: permuterm index
- Basic idea: Rotate every wildcard query, so that the * occurs at the end.

Permuterm index

- For term HELLO: add *hello$*, *ello$h*, *llo$he*, *lo$hel*, and *o$hell* to the B-tree where $ is a special symbol
- Queries

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*
  - How do we handle X*Y*Z?
- It's really a tree and should be called permuterm tree.
- But permuterm index is more common name.

Processing a lookup in the permuterm index

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

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.
- Maintain an inverted index from bigrams to the terms that contain the bigram

Postings list in a 3-gram index

etr → BEETROOT → METRIC → PETRIFY → RETRIEVAL

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

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 the term-document inverted index.
- *k*-gram indexes are fast and space efficient (compared to permuterm indexes).

Processing wildcard queries in the term-document index

- As before, we must potentially execute a large number of Boolean queries for each enumerated, filtered term.
- Recall the query: gen* AND universit*
- Most straightforward semantics: Conjunction of disjunctions
- Very expensive
- Does Google allow wildcard queries?
- Why?
- Users hate to type.
- If abbreviated queries like pyth* theo* for pythagoras' theorem are legal, users will use them ...
- ... a lot

Outline

1. Recap
2. Dictionaries
3. Wildcard queries
4. Spelling correction
5. Soundex

Spelling correction

- Two principal uses
  - Correcting documents being indexed
  - Correcting user queries
- Two different methods for spelling correction
- 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*
- Context-sensitive spelling correction
  - Look at surrounding words
  - Can correct *form*/*from* error above

Correcting documents

- We're not interested in interactive spelling correction of documents (e.g., MS Word) in this class.
- In IR, we use document correction primarily for OCR'ed documents.
- The general philosophy in IR is: don't change the documents.

Correcting queries

- First: isolated word spelling correction
- Fundamental premise 1: There is a list of "correct words" from which the correct spellings come.
- Fundamental premise 2: We have a way of computing the distance between a misspelled word and a correct word.
- Simple spelling correction algorithm: return the "correct" word that has the smallest distance to the misspelled word.
- Example: *informaton* → *information*
- We can use the term vocabulary of the inverted index as the list of correct words.
- 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 study several alternatives.
- Edit distance
- Levenshtein distance
- Weighted edit distance
- *k*-gram overlap

Edit distance

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

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 $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
    else $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
return $m[|s_1|, |s_2|]$

Operations: insert, delete, replace, copy

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 $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
    else $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
return $m[|s_1|, |s_2|]$

Operations: insert, delete, replace, copy

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 $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
    else $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
return $m[|s_1|, |s_2|]$

Operations: insert, delete, replace, copy

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 $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
    else $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
return $m[|s_1|, |s_2|]$

Operations: insert, delete, replace, copy

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 $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
    else $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
return $m[|s_1|, |s_2|]$

Operations: insert, delete, replace, copy

Levenshtein distance: Example

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

Each cell of Levenshtein matrix

cost of getting here from my upper left neighbor (copy or 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

Dynamic programming (Cormen et al.)

- Optimal substructure: The optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems: The optimal solutions to subproblems ("subsolutions") overlap. These subsolutions are computed over and over again when computing the global optimal solution.
- Optimal substructure: We compute minimum distance of substrings in order to compute the minimum distance of the entire string.
- Overlapping subproblems: Need most distances of substrings 3 times (moving right, diagonally, down)

`http://ifnlp.org/lehre/teaching/2008-SS/ir/editdist2.pdf`

Exercise

- Given: *cat* and *catcat*
- Compute the matrix of Levenshtein distances
- Read out the editing operations that transform *cat* into *catcat*

Weighted edit distance

- As above, but weight of an operation depends 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*.
- We now require a weight matrix as input.
- Modify dynamic programming to handle weights.

Using edit distance

- Given query, first enumerate all character sequences within a preset (possibly weighted) edit distance
- Intersect this set with list of “correct” words
- Then suggest terms you found to the user.
- Or do automatic correction – but this is potentially expensive and disempowers the user.

k-gram indexes for spelling correction

- Enumerate all *k*-grams in the query term
- 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
- Example: bigram index, misspelled word *bordroom*
- Bigrams: *bo, or, rd, dr, ro, oo, om*

k-gram indexes for spelling correction: *bordroom*

BO → aboard → about → boardroom → border

OR → border → lord → morbid → sordid

RD → aboard → ardent → boardroom → border

Example with trigrams

- Issue: Fixed number of *k*-grams that differ does not work for words of differing length.
- Suppose the correct word is NOVEMBER
- Trigrams: *nov*, *ove*, *vem*, *emb, mbe, ber*
- And the query term is DECEMBER
- Trigrams: *dec*, *ece*, *cem*, *emb, mbe, ber*
- So 3 trigrams overlap (out of 6 in each term)
- How can we turn this into a normalized measure of overlap?

Jaccard coefficient

- A commonly used measure of overlap of two sets
- Let A and B be two sets
- Jaccard coefficient:

$$\frac{|A \cap B|}{|A \cup B|}$$

- Values if A and B have the same elements? If they are disjoint?
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1.
- december/november example: Jaccard coefficient?
- Application to spelling correction: declare a match if the coefficient is, say, > 0.8.

Context-sensitive spelling correction

- Our example was: *an asteroid that fell form the sky*
- How can we correct *form* here?
- Ideas?
- One idea: hit-based spelling correction
  - Retrieve "correct" terms close to each query term
  - for *flew form munich*: *flea* for *flew*, *from* for *form*, *munch* for *munich*
  - Now try all possible resulting phrases as queries with one word "fixed" at a time
  - Try query *"flea form munich"*
  - Try query *"flew from munich"*
  - Try query *"flew form munch"*
  - The correct query *"flew from munich"* has the most hits.
- Suppose we have 7 alternatives for *flew*, 19 for *form* and 3 for *munich*, how many "corrected" phrases will we enumerate?

Context-sensitive spelling correction

- The "hit-based" algorithm we just outlined is not very efficient.
- More efficient alternative: look at "collection" of queries, not documents

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.

Peter Norvig's complete spelling corrector in only 21 lines of code!

Outline

1 Recap

2 Dictionaries

3 Wildcard queries

4 Spelling correction

5 Soundex

Soundex

- Soundex is the basis for finding phonetic (as opposed to orthographic) alternatives.
- Example: *chebyshev* / *tchebyscheff*
- Algorithm:
  - Turn every token to be indexed into a 4-character reduced form
  - Do the same with query terms
  - 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*

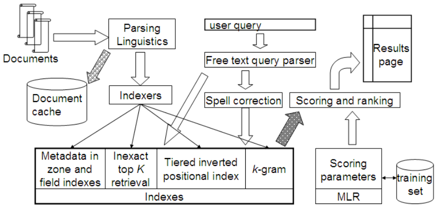
- Retain H
- *ERMAN* → *0RM0N*
- *0RM0N* → *06505*
- *06505* → *06505*
- *06505* → *655*
- Return *H655*
- Will *HERMANN* generate the same code?

Compute soundex code of your last name.

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.

The complete search system



Resources

- Chapter 3 of IIR
- Resources at `http://ifnlp.org/ir`
- Soundex demo
- Levenshtein distance demo
- Levenshtein distance slides
- Peter Norvig's spelling corrector