Introduction to
Information Retrieval

CS276: Information Retrieval and Web Search

Pandu Nayak and Prabhakar Raghavan

Lecture 3: Dictionaries and tolerant retrieval

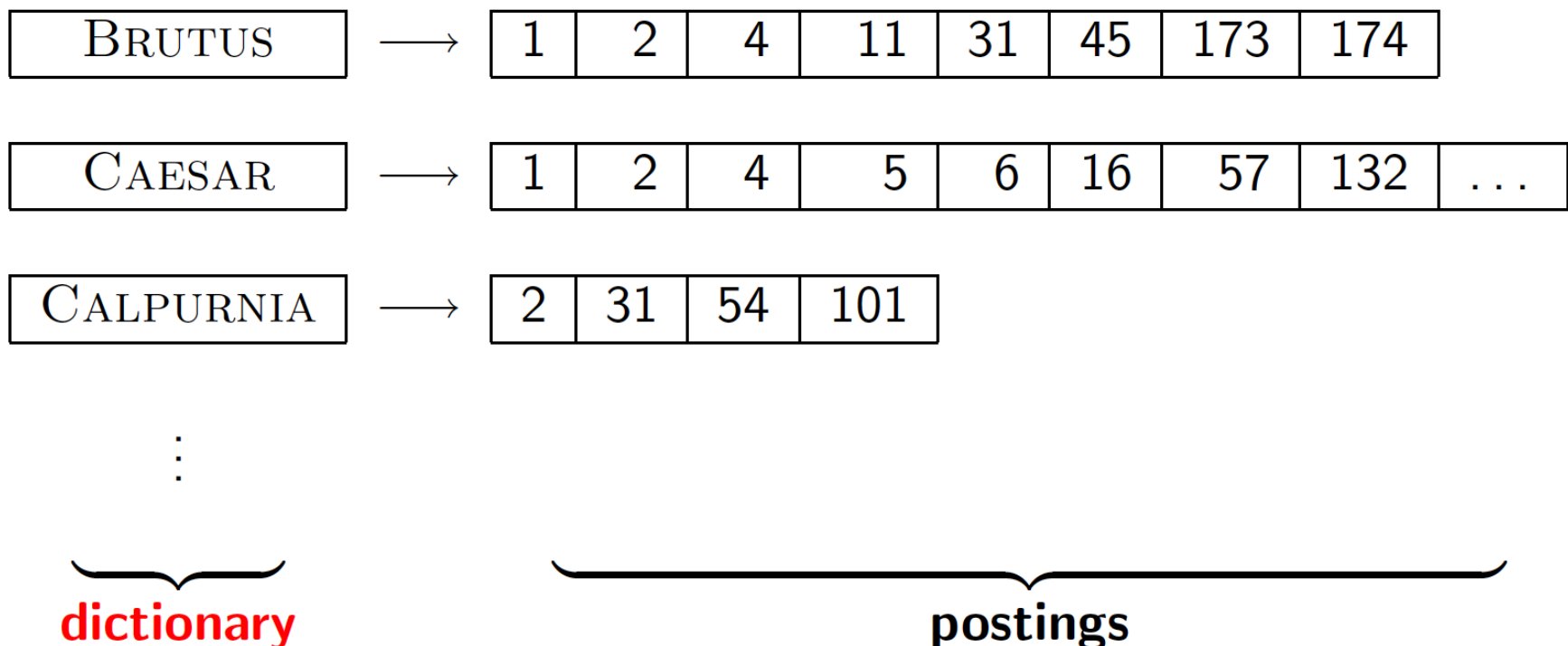# Recap of the previous lecture

- The type/token distinction
  - Terms are normalized types put in the dictionary
- Tokenization problems:
  - Hyphens, apostrophes, compounds, CJK
- Term equivalence classing:
  - Numbers, case folding, stemming, lemmatization
- Skip pointers
  - Encoding a tree-like structure in a postings list
- Biword indexes for phrases
- Positional indexes for phrases/proximity queries

This lecture

- Dictionary data structures
- "Tolerant" retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?

| | 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174 |
| BRUTUS → | | | | | | | | |

| | 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132 | … |
| CAESAR → | | | | | | | | | |

| | 2 | 31 | 54 | 101 |
| CALPURNIA → | | | | |

dictionary   postings

4

A naïve dictionary

- An array of struct:

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

char[20]   int   Postings *

20 bytes   4/8 bytes   4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

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
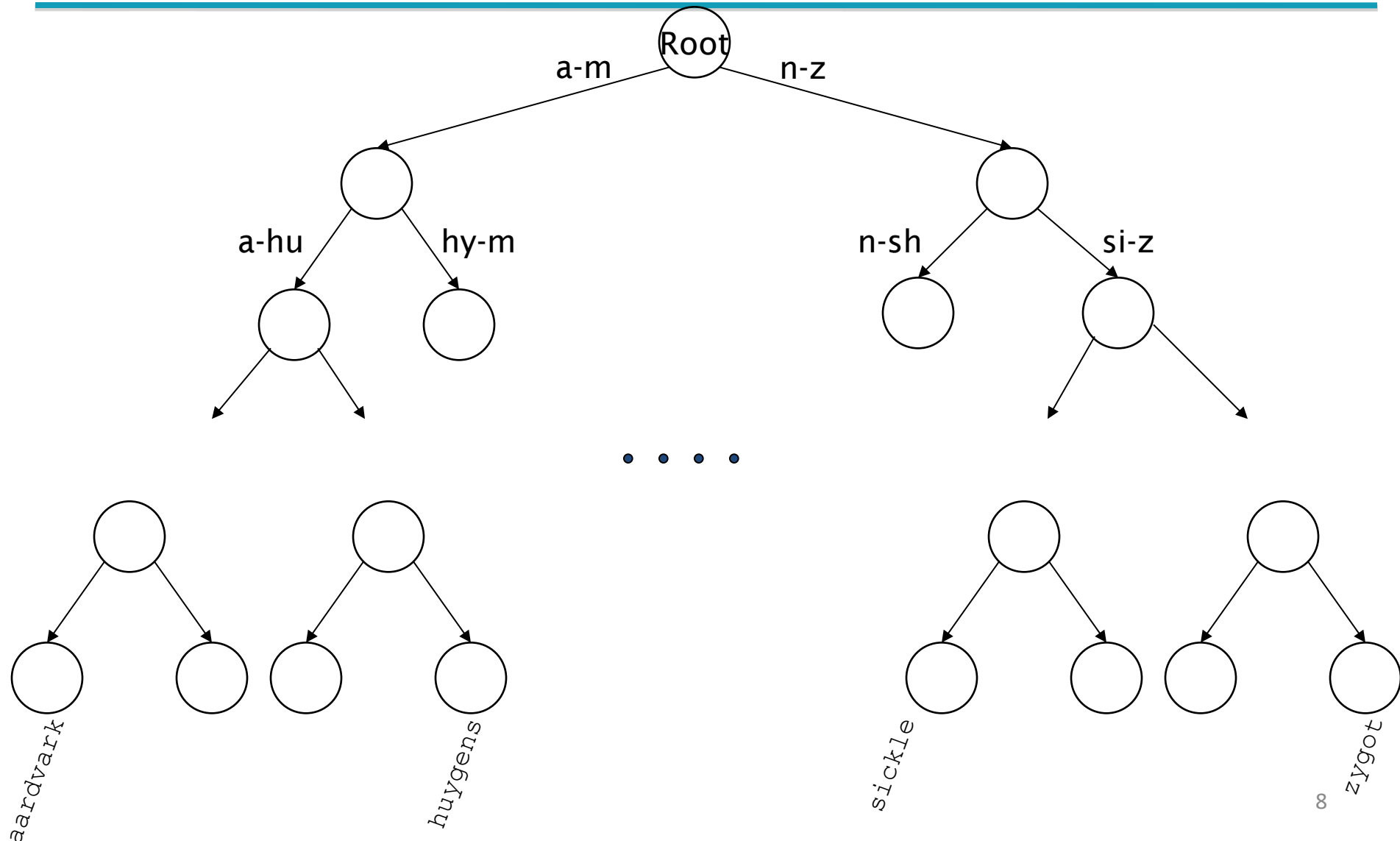
- Strings starting from a or b or c to m
- All strings starting from n to z are stored in right subtree.
- Everything of the left of this node corresponds to string string from a and going up to string starting with hu
- At leaf level actual term . Extract posting list
- B tree is used for this purpose

Tree: B-tree
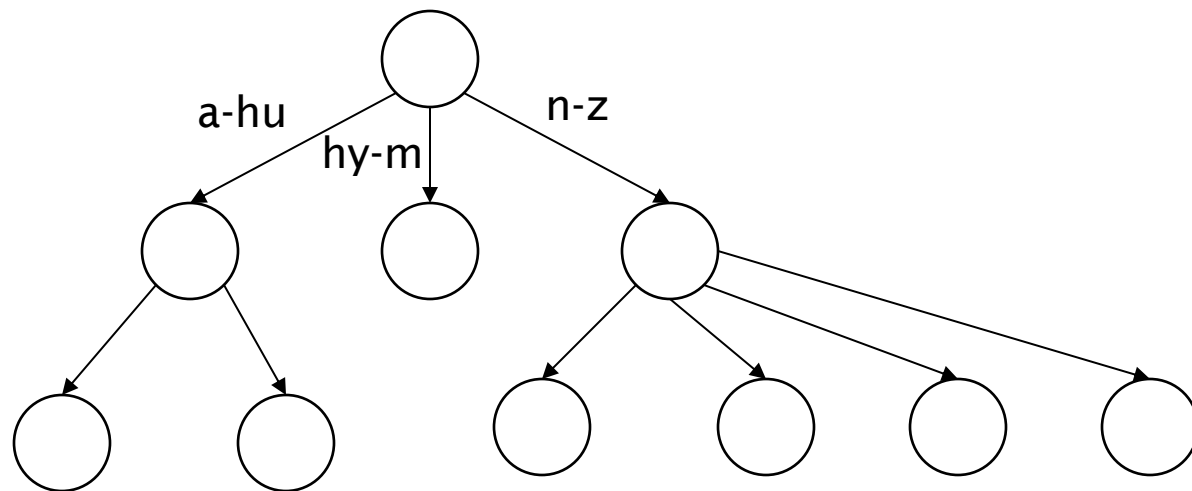
- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].

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

Wildcard queries are used in any of the following situations

(1) the user is uncertain of the spelling of a query term (e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney);

(2) the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants (e.g., color vs. colour);

(3) the user seeks 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*);

(4) the user is uncertain of the correct rendition of a foreign word or phrase (e.g., the query Universit* Stuttgart).

Wild-card queries: *

- ***mon*:** find all docs containing any word beginning with "mon".
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ***mon*:** find words ending in "mon": harder
  - Maintain an additional B-tree for terms *backwards.*

  Can retrieve all words in range: ***nom ≤ w < non.***

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.

- We still have to look up the postings for each enumerated term.

- E.g., consider the query:
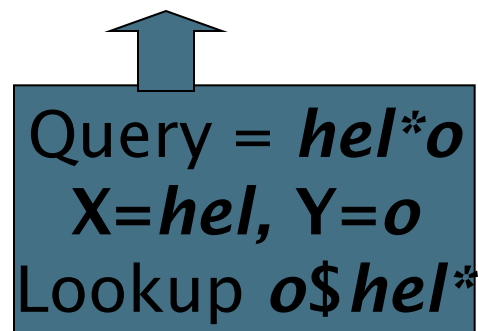
  se*ate *AND* **fil*er**

  This may result in the execution of many Boolean *AND* queries.

B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ****tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
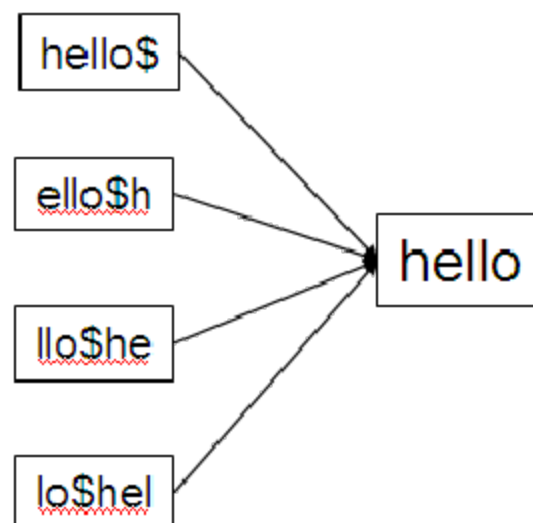- This gives rise to the **Permuterm** Index.

Permuterm index

- For term ***hello***, index under:
    - ***hello\$, ello\$h, llo\$he, lo\$hel, o\$hell, \$hello***

    where \$ is a special symbol.

- Queries:
    - **X**   lookup on **X\$**   **X***   lookup on   \$**X***
    - ***X**   lookup on **X\$***   ***X*** lookup on   **X***
    - **X*Y** lookup on **Y\$X***   **X*Y*Z**   ??? Exercise!

Query = ***hel*o***
X=***hel***, Y=***o***
Lookup ***o\$hel****

- Our first special index for general wildcard queries is the *permuterm index* into our character set, to mark the end of a term. Thus, the term hello is shown here as the augmented term hello$.

- Next, we construct a permuterm index, in which the various rotations of each term (augmented with $) all link to the original vocabulary term.

- permuterm index entry for the term hello.

- We refer to the set of rotated terms in the permuterm index as the *permuterm vocabulary*.

- How does this index help us with wildcard queries? Consider the wildcard query m*n. The key is to *rotate* such a wildcard query so that the * symbol appears at the end of the string - thus the rotated wildcard query becomes n$m*.

- Next, we look up this string in the permuterm index, where seeking n$m* (via a search tree) leads to rotations of (among others) the terms man and moron.

- Now that the permuterm index enables us to identify the original vocabulary terms matching a wildcard query, we look up these terms in the standard

- We can thus handle any wildcard query with a single * symbol. But what about a query such as fi*mo*er?

- In this case we first enumerate the terms in the dictionary that are in the permuterm index of er$fi*. Not all such dictionary terms will have the string mo in the middle - we filter these out by exhaustive enumeration, checking each candidate to see if it contains mo.

- In this example, the term fishmonger would survive this filtering but filibuster would not. We then run the surviving terms through the standard inverted index for document retrieval.

- One disadvantage of the permuterm index is that its dictionary becomes quite large, including as it does

Permuterm query processing

- Rotate query wild-card to the right

- Now use B-tree lookup as before.

- *Permuterm problem: ≈ quadruples lexicon size*

 Empirical observation for English.

Bigram (*k*-gram) indexes

- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term

- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*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 a <u>*second*</u> inverted index <u>*from bigrams to dictionary terms*</u> that match each bigram.

Bigram index example

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

$m	⇒	mace	→	madden	---→
mo	⇒	among	→	amortize	---→
on	⇒	along	→	among	---→
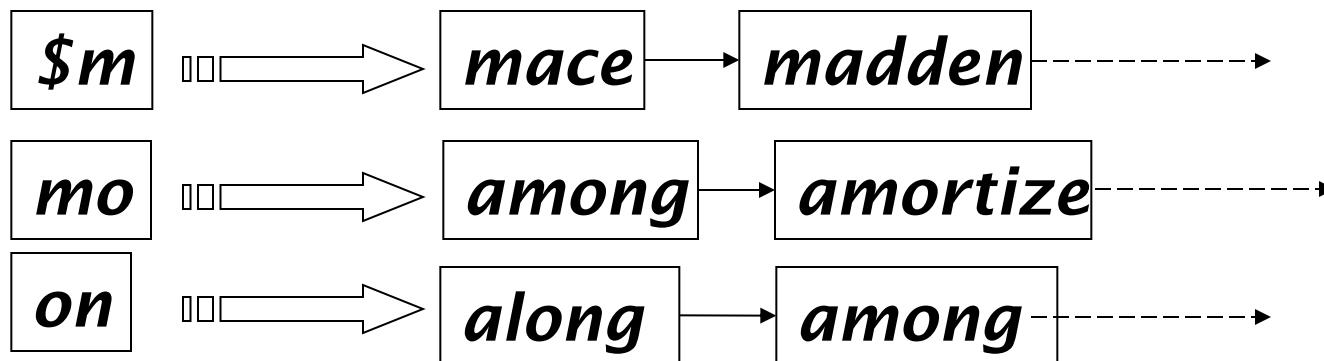
Processing wild-cards

- Query **mon*** can now be run as
  - **$m** *AND* **mo** *AND* **on**
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate **moon**.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.

- Wild-cards can result in expensive query execution (very large disjunctions…)

  - pyth* AND prog*

- If you encourage "laziness" people will respond!

	Search

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.

- Which web search engines allow wildcard queries?

26

SPELLING CORRECTION

Spell correction

- Two principal uses
    - Correcting document(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 <u>form</u> Heathrow to Narita.*

28

Document correction

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material have typos
- Goal: the dictionary contains fewer misspellings
- But often we don't change the documents and instead fix the query-document mapping

Query mis-spellings

- Our principal focus here
  - E.g., the query **Alanis Morisett**
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*

Isolated word correction

- 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

    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Isolated word correction

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

- What's "closest"?

- We'll study several alternatives

  - **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.

- Generally found by dynamic programming.

- See http://www.merriampark.com/ld.htm for a nice example plus an applet.

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 mis-typed 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 dynamic programming to handle weights

Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with 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
- The alternatives 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
  - Alternative?
- 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.

Spelling correction using k-gram overlap

- While the minimum edit distance discussed provides a good list of possibly correct words, there are far too many words in the English dictionary to consider finding the edit distance between all pairs.

- To simplify the list of candidate words, the k-gram overlap is used in typical IR and NLP systems.

- K-grams are k-length subsequences of a string. Here, k can be 1, 2, 3 and so on. For k=1, each resulting subsequence is called a "unigram"; for k=2, a "bigram"; and for k=3, a "trigram". These are the most widely used k-grams for spelling correction, but the value of k really depends on the situation and context.

- As an example, consider the string "catastrophic". In this case,

- **Unigrams:** ["c", "a", "t", "a", "s", "t", "r", "o", "p", "h", "i", "c"]

- **Bigrams:** ["ca", "at", "ta", "as", "st", "tr", "ro", "op", "ph", "hi", "ic"]

- **Trigrams:** ["cat", "ata", "tas", "ast", "str", "tro", "rop", "oph", "phi", "hic"]

- **K-Gram Index**
  A k-gram index maps a k-gram to a postings list of all possible vocabulary terms that contain it. The figure below shows the k-gram postings list corresponding to the bigram "ur".

- It is noteworthy that the postings list is sorted alphabetically.

ur	blurry	burn	cure	curfew	surf	yogurt	yourself

- **Spelling Correction**

- While creating the candidate list of possible corrected words, we can use the "k-gram overlap" to find the most likely corrections.

- Consider the misspelt word: "appe". The postings lists for the bigrams contained in it are shown below.

ap	ape	apple	chapel

pp	apple	applet	frappe

pe	ape	apple	pea

- To find the k-gram overlap between two postings list, we use the Jaccard coefficient. Here, A and B are two sets (postings lists), A for the misspelt word and B for the corrected word.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- Now, consider some candidate terms for spelling correction, namely "ape" and "apple".

- To find the Jaccard coefficient, simply scan through the postings lists of all bigrams of "appe" and count the instances where "ape" appears.

- In the first postings list, "ape" appears 1 time. In the second postings list, "ape" appears 0 times. In the third postings list, "ape" appears 1 time.

$$A \cap B = 2$$

- Now, the no. of bigrams in "appe" is 3, and the no. of bigrams in "ape" is 2. Therefore,

$$A \cup B = 3 + 2 - 2 = 3$$

- J(A, B) = 2/3 = 0.67.

Example

- **"apple"** $A \cap B = 3$

- Now, the no. of bigrams in "appe" is 3, and the no. of bigrams in "apple" is 4. Therefore, $A \cup B = 3 + 4 - 3 = 4$

- J(A, B) = 3/4 = 0.75.

- This suggests that "apple" is a more plausible correction. Practically, this method is used to filter out unlikely corrections.

- The steps involved for spelling correction are:

- Find the k-grams of the misspelled word.

- For each k-gram, linearly scan through the postings list in the k-gram index.

- Find k-gram overlaps after having linearly scanned the lists (no extra time complexity because we are finding the Jaccard coefficient).

- Return the terms with the maximum k-gram overlaps.

Context-sensitive spell correction

- Text: ***I flew <u>from</u> Heathrow to Narita.***

- Consider the phrase query ***"flew <u>form</u> Heathrow"***

- 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 (in weighted edit distance) 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.

Exercise

■ Suppose that for ***"flew <u>form</u> Heathrow"*** we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many "corrected" phrases will we enumerate in this scheme?

Another approach

- Break phrase query into a conjunction of biwords (Lecture 2).

- Look for biwords that need only one term corrected.

- Enumerate only phrases containing "common" biwords.

General issues in spell correction

- We 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

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

  - From Bayes rule, this is equivalent to

$$\text{argmax}_{corr}\ P(query \mid corr) * P(corr)$$

Noisy channel   Language model

SOUNDEX

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** → ***tchebycheff***

During the linguistic phase there will be a Soundex module that will take every token that needs to be indexed and map it or transform it into a 4-character term. Do the same thing for the words in the document and for the words in the query and then the term that actually forms an inverted index will be this 4 character of the token

Soundex – typical 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
  - (when the query calls for a soundex match)

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

Soundex – typical algorithm

1. Retain the first letter of the word.

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 $\rightarrow$ 1

- C, G, J, K, Q, S, X, Z $\rightarrow$ 2

- D,T $\rightarrow$ 3

- L $\rightarrow$ 4

- M, N $\rightarrow$ 5

- R $\rightarrow$ 6

Soundex continued

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, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., **Herman** becomes H655.

Will **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

(SPELL(moriset) /3 toron*to) OR *SOUNDEX***(chaikofski)**

Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about

- Identify some of the key design choices in the index pipeline:

  - Does stemming happen before the Soundex index?

  - What about *n*-grams?

- Given a query, how would you parse and dispatch sub-queries to the various indexes?

Resources

- IIR 3, MG 4.2

- Efficient spell retrieval:

  - K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4), Dec 1992.

  - J. Zobel and P. Dart. Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995. http://citeseer.ist.psu.edu/zobel95finding.html

  - Mikael Tillenius: Efficient Generation and Ranking of Spelling Error Corrections. Master's thesis at Sweden's Royal Institute of Technology. http://citeseer.ist.psu.edu/179155.html

- **Nice, easy reading on spell correction:**

  - Peter Norvig: How to write a spelling corrector

  http://norvig.com/spell-correct.html