CS276A

Information Retrieval

Lecture 4

## Recap of last time

- Index compression
- Space estimation

## This lecture

- "Tolerant" retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

## Wild-card queries

## Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning "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?
  - (Especially multiple *'s)
- The solution: transform every wild-card query 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***

  **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!

## 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.

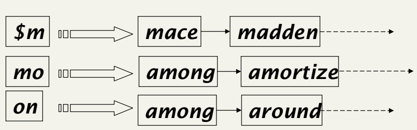
## Bigram indexes

- 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 an "inverted" index from bigrams to <u>*dictionary terms*</u> that match each bigram.

## Bigram index example

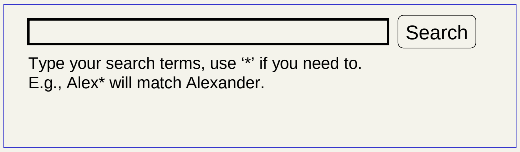



## Processing *n*-gram wild-cards

- Query ***mon**** can now be run as
  - ***$m*** *AND* ***mo*** *AND* ***on***
- Fast, space efficient.
- 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.

## 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
  - Avoid encouraging "laziness" in the UI:




## Advanced features

- Avoiding UI clutter is one reason to hide advanced features behind an "Advanced Search" button
- It also deters most users from unnecessarily hitting the engine with fancy queries

# Spelling correction

## Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Retrieve matching documents when query contains a spelling error
- 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.***

## Document correction

- Primarily for OCR'ed documents
  - Correction algorithms tuned for this
- Goal: the index (dictionary) contains fewer OCR-induced misspellings
- 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).

## 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
    - Google's *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
  - Weighted edit distance
  - *n*-gram overlap

## Edit distance

- Given two strings $S_1$ and $S_2$, the minimum number of basic operations to covert one to the other
- Basic operations are typically character-level
  - Insert
  - Delete
  - Replace
- E.g., the edit distance from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming.

## Edit distance

- Also called "Levenshtein distance"
- See http://www.merriampark.com/ld.htm for a nice example plus an applet to try on your own

## Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture keyboard errors, e.g. ***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***
  - (Same ideas usable for OCR, but with different weights)
- Require weight matrix as input
- Modify dynamic programming to handle weights

## Using edit distances

- Given query, first enumerate all dictionary terms within a preset (weighted) edit distance
- (Some literature formulates weighted edit distance as a probability of the error)
- Then look up enumerated dictionary terms in the term-document inverted index
  - Slow but no real fix
  - Tries help
- Better implementations – see Kukich, Zobel/Dart references.

## *n*-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Rank by number of matching *n*-grams
- Variants – weight by keyboard layout, etc.

## Example with trigrams

- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem,* *emb, mbe, ber*.
- The query is ***december***
  - Trigrams are *dec, ece, cem,* *emb, mbe, 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
- Let *X* and *Y* be two sets; then the J.C. is

$$|X \cap Y| / |X \cup Y|$$

- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint
- *X* and *Y* don't have to be of the same size
- 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

## Caveat

- Even for isolated-word correction, the notion of an index token is critical – what's the unit we're trying to correct?
- In Chinese/Japanese, the notions of spell-correction and wildcards are poorly formulated/understood

## 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.
  - NLP too heavyweight for 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***
  - ***etc.***
- Suggest the alternative that has lots of hits?

## Exercise

- Suppose that for ***"flew form 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 phrase matches and … rank them!

## General issue in spell correction

- Will enumerate multiple alternatives for "Did you mean"
- Need to figure out which one (or small number) to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries

## Computational cost

- Spell-correction is computationally expensive
- Avoid running routinely on every query?
- Run only on queries that matched few docs

## Thesauri

- Thesaurus: language-specific list of synonyms for terms likely to be queried
  - car → automobile, etc.
  - Machine learning methods can assist – more on this in later lectures.
- Can be viewed as hand-made alternative to edit-distance, etc.

## Query expansion

- Usually do query expansion rather than index expansion
  - No index blowup
  - Query processing slowed down
    - Docs frequently contain equivalences
  - May retrieve more junk
    - ***puma*** → ***jaguar*** retrieves documents on cars instead of on sneakers.

## Soundex

## Soundex

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

## 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 → 1
- C, G, J, K, Q, S, X, Z → 2
- D,T → 3
- L → 4
- M, N → 5
- R → 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?

## Exercise

- Using the algorithm described above, find the soundex code for your name
- Do you know someone who spells their name differently from you, but their name yields the same soundex code?

## Language detection

- Many of the components described above require language detection
  - For docs/paragraphs at indexing time
  - For query terms at query time – much harder
- For docs/paragraphs, generally have enough text to apply machine learning methods
- For queries, lack sufficient text
  - Augment with other cues, such as client properties/specification from application
  - Domain of query origination, etc.

## What queries can we process?

- We have
  - Basic inverted index with skip pointers
  - Wild-card index
  - Spell-correction
  - Soundex
- Queries such as

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

## Aside – results caching

- If 25% of your users are searching for

  ***britney** AND **spears***

  then you probably *do* need spelling correction, but you *don't* need to keep on intersecting those two postings lists
- Web query distribution is extremely skewed, and you can usefully cache results for common queries – more later.

## Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all this functionality
- 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?

## Exercise on previous slide

- Is the beginning of "what do we we need in our search engine?"
- Even if you're not building an engine (but instead use someone else's toolkit), it's good to have an understanding of the innards

## Resources

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

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