CS 572: Information Retrieval

Lecture 4: Tolerant Retrieval

Acknowledgment: Some slides in this lecture are adapted from Chris Manning (Stanford) and Doug Oard (Maryland)
Project 1 Updates

• CS572 partition created: /aut/proj/cs572
• Keep all your code/data in /aut/proj/cs572/Your_netid
• **The project just became easier. 😞** Read the API documentation:
  - [https://lucene.apache.org/core/5_4_0/core/org/apache/lucene/search/similarities/package-summary.html](https://lucene.apache.org/core/5_4_0/core/org/apache/lucene/search/similarities/package-summary.html)
  - Note: do **not** use the built-in TF*IDF and BM25 implementations, the point of the project is to re-implement them yourself.
  - Yes it sounds silly, but I **will** look for your own implementation/use of collection statistics, and ranking functions. Obviously you can look at how Lucene implements these for reference, but do not copy/paste.
Building an Index (conclusion)

- Phrase queries

Tolerant retrieval: positional indexing
Phrase queries

• Want to answer queries such as “\textit{emory university}” — as a phrase

• Thus the sentence “\textit{I went to university at Emory}” is not a match.
  – The concept of phrase queries has proven easily understood by users; about 10% of web queries are phrase queries

• No longer suffices to store only \texttt{<term : docs>} entries
Solution 1: Biword indexes

- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords
  - *friends romans*
  - *romans countrymen*
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.
Longer phrase queries

• Longer phrases are processed as we did with wild-cards:

• *emory university atlanta ga* can be broken into the Boolean query on biwords:

  \[\text{emory university AND university atlanta AND atlanta ga}\]

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.  

Can have false positives!
Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Call any string of terms of the form NX*N an extended biword.
  - Each such extended biword is now made a term in the dictionary.
- Example: catcher in the rye
  
  \[
  \begin{array}{ccc}
  & N & X \\
  X & X & N \\
  \end{array}
  \]
- Query processing: parse it into N’s and X’s
  - Segment query into enhanced biwords
  - Look up in index: catcher rye
Issues for biword indexes

• False positives, as noted before
• Index blowup due to bigger dictionary
  – Infeasible for more than biwords (e.g., triwords),
    ➔ Too big even for biwords!

• Biword indexes are not the standard solution (for all biwords) but can be part of a combination strategy
Solution 2: Positional indexes

- Store, for each *term*, entries of the form:
  
  `<number of docs containing term; doc1: position1, position2 ... ; doc2: position1, position2 ... ; etc.>`

- where position is *token* index in doc
Positional index example

<be: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, ...>

- Can compress position values/offsets
- Nevertheless, this expands postings storage substantially

Which of docs 1, 2, 4, 5 could contain “to be or not to be”?
Processing a phrase query

- Extract inverted index entries for each distinct term: *to, be, or, not*.
- Merge their *doc:position* lists to enumerate all positions with "*to be or not to be*".
  - *to*:
    - 2:1,17,74,222,551;
      4:8,16,190,429,433;
      7:13,23,191; ...
  - *be*:
    - 1:17,19;
      4:17,191,291,430,434;
      5:14,19,101; ...
- Same general method for proximity searches
Proximity queries

- **LIMIT! /3 STATUTE /3 FEDERAL /2 TORT** Here, \( /k \) means “within \( k \) words of”.

- Clearly, positional indexes can be used for such queries; biword indexes cannot.

- **Exercise**: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of \( k \)?
Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
  - Average web page has <1000 terms
  - SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Combination schemes

• These two approaches can be combined
  – For particular phrases ("Michael Jackson", "Britney Spears") it is inefficient to keep on merging positional postings lists
    • Even more so for phrases like "The Who"

• Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
  – A typical web query mixture was executed in ¼ of the time of using just a positional index
  – It required 26% more space than having a positional index alone
Tolerant Retrieval

- Searching the Index *with errors in queries*
  
  - Storing and searching dictionary entries
  
  - Supporting approximate queries
  
  - Support erroneous queries
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?

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132 ...</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>

...
Term Index Size

• **Heap’s “Law”** tells us about vocabulary size

\[ V = K n^{\beta} \]

- When adding new documents, the system is likely to have seen terms already
- Usually fits in RAM

\[ K \approx 20, \beta \approx 0.6 \]

\( V \) is vocabulary size
\( n \) is corpus size (number of documents)
\( K \) and \( \beta \) are constants

• But the postings file keeps growing!

More on this when we get to LMs
A naïve dictionary

• An array of struct:

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td>→</td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td>→</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td>→</td>
</tr>
</tbody>
</table>

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?
• How long does *lookup* take, in the worst case?
  – Expected/average?

• How long to *update* the dictionary, in the worst case?
  – Expected/average?
Dictionary data structures

• Two main choices:
  – Hash table
  – Tree

• Some IR systems use hashes, some trees
Hashes

• Each vocabulary term is hashed to an integer

• Pros:
  – Lookup is faster than for a tree: O(1)

• Cons:
  – No easy way to find minor variants:
    • judgment/judgement
  – No prefix search [uhm... tolerant retrieval?]
  – If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing everything
Tree Index

• Tree structures allow easy insertion
  – But the worst case lookup time is $O(n)$

• Balanced trees provide the best of both
  – Fast lookup [$O(\log n)$] and easy insertion [$O(\log n)$]
  – But they require 45% more disk space
Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings

Pros:
- Solves the prefix problem (terms starting with \textit{hyp})

Cons:
- Slower: $O(\log M)$ [and this requires \textit{balanced} tree]
  - Rebalancing binary trees is expensive
AVL Trees

- $n = 2^{30} = 10^9$ (approx).
- $30 \leq \text{height} \leq 43$.
- When the AVL tree resides on a disk, up to 43 disk access are made for a search.
- This takes up to (approx) 4 seconds.
- Not acceptable.
B-trees (m-way search tree)

Node may have [a,b] children
Each branch represents range of next character values
“Collapses” multiple levels of binary search tree into one (avoid re-balancing)
4-Way Search Tree

- k < 10
- 10 < k < 30
- 30 < k < 35
- k > 35
Choice Of $m$ (branching factor)

- Worst-case search time.
  - $\text{(time to fetch a node + time to search node)} \times \text{height}$
  - $(a + b \times m + c \times \log_2 m) \times h$

where $a$, $b$, and $c$ are constants.
B-Tree Example (1)
m=5, k=4 (5 children, 4 keys/node)

Insert keys F, W, L, and T
B-Tree Example (2)

Insert key Z: split rightmost leaf
move median item T up to parent
B-Tree Example (3)

Insert key D: the new key D is the new *median* so it is moved into parent.
B-Tree Example (4, fin)

Insert key S: node with N, P, Q, and R splits; median Q moves up to parent. But, parent node is full, so it splits $\rightarrow$ median M is new root.
Assume enough memory to hold all $h$ nodes accessed on way down.

- $h$ read accesses on way down.
- $h - 1$ sibling read accesses on way up.
- $h - 2$ writes of combined nodes on way up.
- 3 writes of root and level 2 nodes for sibling borrowing at level 2.
  - Total is $3h$.

Cache access time vs main memory access time.

Reduce main memory accesses using a B-tree.
Complexity Of B-Tree Node Operations

• Search a B-tree node … $O(\log m)$.  
• Find middle pair … $O(\log m)$.  
• Insert a pair … $O(\log m)$.  
• Delete a pair … $O(\log m)$.  
• Split a B-tree node … $O(\log m)$.  
• Join 2 B-tree nodes … $O(m)$.  
  – Need to copy indexed red-black tree that represents one B-tree node into the array space of the other B-tree node.
Wild Card Queries

- Killer App: word completion for mobile search
Supporting 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: \( \text{mon} \leq w < \text{moo} \)

- **mon**: find words ending in “mon”? 
Supporting 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: \( mon \leq w < moo \)

- **mon**: find words ending in “mon”:
  - Maintain an additional B-tree for terms *backwards*.
  - Now can retrieve all words in range: \( nom \leq w < non \).
Supporting 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: $\text{mon} \leq w < \text{moo}$

- **mon**: find words ending in “mon”:
  - Maintain an additional B-tree for terms backwards.
  - Now can retrieve all words in range: $\text{nom} \leq w < \text{non}$.

Exercise: from this, how can we enumerate all terms meeting the wild-card query $\text{pro*cent}$?
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*Y** lookup on **Y$X**
  – **X*Y*Z** ???
  – **X*$** lookup on **X*$**
  – ***X*$** lookup on **X*$**
  – **X*Y*Z** ???
Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem:* $\approx$ *quadruples lexicon size*

Empirical observation for English.
Another Solution: 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)

\[
\text{$a, ap, pr, ri, il$, $i, is, s$, $t, th, he, e$, $c, cr, ru, ue, el, le, es, st, t$, $m, mo, on, nt, h$}
\]

- $\$ is a special word boundary symbol
- Maintain a second inverted index from bigrams to dictionary terms that match each bigram.
Bigram index example

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

```
  $m$
  mo
  on

  mace -> madden
  among -> amortize
  among -> around
```
Processing wild-cards

• Query *mon* can now be run as
  – $m \text{ AND } mo \text{ AND } on$

• 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).
Spelling correction

It's the British version

SPELL CHEQUE
Kinds of Spelling Mistakes: Typos

• Typos are wrong characters by mistake
• Insertions
  – “appellate” as “appellare”, “prejudice” as “prejudsice”
• Deletions
  – “plaintiff” as “paintiff”, “judgement” as “judment”, “liability” as “liabilty”, “discovery” as “dicovery”, “fourth amendment” as “fourthamendment”
• Substitutions
  – “habeas” as “haceas”
• Transpositions
  – “fraud” as “fruad”, “bankruptcy” as “banrkuptcy
  – “subpoena” as “subpeona”
  – “plaintiff” as “plaitniff”
Kinds of Spelling Mistakes: Brainos

- Brainos are wrong characters “on purpose”
- The kinds of mistakes found in lists of “common” misspellings
- Very common in general web queries
- Derive from either pronunciation or spelling or deep semantic confusions
- English is particularly bad due to irregularity
- Probably (?) common in other languages importing words
Brainos: Soundalikes

• Latinates
  – “subpoena” as “supena”, “judicata” as “judicada”, “voir” as “voire”

• Consonant Clusters & Flaps
  – “privelege” as “priveledge”, “rescission” as “recision”, “collateral” as “colaterall”, “latter” as “ladder”, “estoppel” as “estopple”, “withholding” as “witholding”, “recission” as “recision”

• Vowel Reductions
  – “collateral” as “collaterel”, “punitive” as “punative”
Brainos: Confusions

• Substitute more common or just plain different
  – Names: “Opperman” as “Oppenheimer”; “Eisenstein” as “Einstein”

• Pronunciation Confusions
  – Transpositions; “preclusion” as “perclusion”, “meruit” as “meriut”

• Irregular word forms
  – “juries” as “jurys” or “jureys”; “men” as “mans”
  – English is particularly bad for this, too

• Tokenization issues
  – Correct variant (if unique) depends on search engine’s notion of “word”

• Word Boundaries
  – “in camera” as “incamera”, “qui tam” as “quitam”, “injunction” as “in junction”, “foreclosure” as “for closure”, “dramshop” as “dram shop”
Decoding L33T-speak

• “L33T” is l33t-speak for “elite”
• Used by gamers (pwn4g3) and spammers (med|catiOn)
• Substitute numbers (e.g. ‘E’ to ‘3’, ‘A’ to ‘4’, ‘O’ to ‘0’, ‘l’ to ‘1’)
• Substitute punctuation (e.g. ‘\’ for ‘A’, ‘|’ for ‘L’, ‘\|\’ for ‘W’)
• Some standard typos (e.g. ‘p’ for ‘o’)
• De-duplicate or duplicate characters freely
• Delete characters relatively freely
• Insert/delete space or punctuation freely
• Get creative
• Examples from Spam:
  • VÀLIUM CíAL1SS ViÁGRRA; MACR0MEDIA, M1CR0S0FT, SYMANNTEC $20 EACH; univers.ty de-gree online; H0t penny pick fue|ed by high demand; Fwd; cials-tabs, 24 hour sale online; HOw 1s yOur health; Your C A
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 form Heathrow to Narita.
Document correction

• Primarily for OCR’ed or Speech→Text 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).
  – Speech→Text can confuse similar sounding words ("right" vs. "write" vs. "rite" vs. "wright"). More later.
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”?
- 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
  - Can use Trie data structure or BK-tree: [https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/](https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/)
Edit distance to all dictionary terms?

• Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  – Expensive and slow

• How do we cut the set of candidate dictionary terms?

• Can use $n$-gram overlap for this
**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
- Threshold 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

$$\frac{|X \cap Y|}{|X \cup Y|}$$

• 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
Matching trigrams

• Consider the query *lord* – we wish to identify words matching 2 of its 3 bigrams (*lo, or, rd*)

Standard postings “merge” will enumerate …

Adapt this to using Jaccard (or another) measure.
Context-sensitive spell correction

• Text: *I flew from Heathrow to Narita.*
• Consider the phrase query *“flew form 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
  – etc.
• Suggest the alternative that has lots of hits?
Query Spelling Correction

We have included grand copthorne hotel results - Show only grand copthorn hotel

Grand Copthorne Hotel Singapore
Five-star hotel in the city with guaranteed great rates for online...
www.asiatravel.com

Luxury Hotel Singapore | Official Site Grand Copthorne Waterfront Hotel...
Luxury hotel Singapore, experience the great comfort and the relaxing environment that matches your life style at Grand Copthorne Waterfront Hotel nestled next to...

Millennium & Copthorne Hotels
... Official Site of Millennium and Copthorne Hotels offering a magnitude of exceptional ...
Reformulations from Bad to Good Spellings

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-rewrite</td>
<td>mic amps -&gt; create taxi</td>
<td>53.2%</td>
</tr>
<tr>
<td>insertions</td>
<td>game codes -&gt; video game codes</td>
<td>9.1%</td>
</tr>
<tr>
<td>substitutions</td>
<td>john wayne bust -&gt; john wayne statue</td>
<td>8.7%</td>
</tr>
<tr>
<td>deletions</td>
<td>skateboarding pics -&gt; skateboarding</td>
<td>5.0%</td>
</tr>
<tr>
<td>spell correction</td>
<td>real eastate -&gt; real estate</td>
<td>7.0%</td>
</tr>
<tr>
<td>mixture</td>
<td>huston's restaurant -&gt; houston's</td>
<td>6.2%</td>
</tr>
<tr>
<td>specialization</td>
<td>jobs -&gt; marine employment</td>
<td>4.6%</td>
</tr>
<tr>
<td>generalization</td>
<td>gm reabtes -&gt; show me all the current auto rebates</td>
<td>3.2%</td>
</tr>
<tr>
<td>other</td>
<td>thanksgiving -&gt; dia de acconde gracias</td>
<td>2.4%</td>
</tr>
</tbody>
</table>

[Jones & Fain, 2003]
Spelling Correction: Noisy Channel Model

Platonic concept of query

Correct Spelling

Typing quickly
Distracted
Forgot how to spell

Typos/spelling errors

Reconstruct original query by “reversing this process”
Spelling Correction: Iterative Approach
[Cucerzan and Brill, EMNLP 2004]

• Main idea:
  – Iteratively transform the query into other strings that correspond to more likely queries.
  – Use statistics from query logs to determine likelihood.
    • Despite the fact that many of these are misspelled
    • Assume that the less wrong a misspelling is, the more frequent it is, and correct > incorrect

• Example:
  – ditroitigers ->
    • detroittigers ->
      – detroit tigers

<table>
<thead>
<tr>
<th>Query</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>albert einstein</td>
<td>4834</td>
</tr>
<tr>
<td>albert einstien</td>
<td>525</td>
</tr>
<tr>
<td>albert einstine</td>
<td>149</td>
</tr>
<tr>
<td>albert einsten</td>
<td>27</td>
</tr>
<tr>
<td>albert einsteins</td>
<td>25</td>
</tr>
<tr>
<td>albert einstain</td>
<td>11</td>
</tr>
<tr>
<td>albert einstin</td>
<td>10</td>
</tr>
<tr>
<td>albert eintein</td>
<td>9</td>
</tr>
<tr>
<td>albeart einstein</td>
<td>6</td>
</tr>
<tr>
<td>aolbert einstein</td>
<td>6</td>
</tr>
<tr>
<td>alber einstein</td>
<td>4</td>
</tr>
<tr>
<td>albert einseint</td>
<td>3</td>
</tr>
<tr>
<td>albert einsteirn</td>
<td>3</td>
</tr>
<tr>
<td>albert einterin</td>
<td>3</td>
</tr>
<tr>
<td>albert eintien</td>
<td>3</td>
</tr>
<tr>
<td>alberto einstein</td>
<td>3</td>
</tr>
<tr>
<td>albrecht einstein</td>
<td>3</td>
</tr>
<tr>
<td>alvert einstein</td>
<td>3</td>
</tr>
</tbody>
</table>
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
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)

• [Link](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
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?
Summary: What queries can we process?

• We have
  – Basic inverted index (optimization: skip pointers)
    ➢ Wild-card index
    ➢ Spell-correction
    ➢ Soundex

• Queries such as

\[
(SPELL(moriset) /3 toron*to) \text{ OR } SOUNDEX(chaikofski)
\]
Next Finish Index Construction+Ranking

- Consider overall index size
- Term/document distributions $\rightarrow$ LMs
- Distributed index construction
- (Chapter 4 in MRS +)
- Vector Space models