OUTLINE

• Documents

• Terms
  • General + Non-English
  • English

• Skip pointers

• Phrase queries
Phrase queries

- We want to answer a query such as [stanford university] – as a phrase.
- Thus *The inventor Stanford Ovshinsky never went to university* should **not** be a match.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists.
- Two ways of extending the inverted index:
  - biword index
  - positional index
Biword indexes

• Index every consecutive pair of terms in the text as a phrase.
• For example, *Friends, Romans, Countrymen* would generate two biwords: “*friends romans*” and “*romans countrymen*”
• Each of these biwords is now a vocabulary term.
• Two-word phrases can now easily be answered.
Longer phrase queries

• A long phrase like “stanford university palo alto” can be represented as the Boolean query “STANFORD UNIVERSITY” AND “UNIVERSITY PALO” AND “PALO ALTO”

• We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase.
Extended biwords

• Parse each document and perform part-of-speech tagging
• Bucket the terms into (say) nouns (N) and articles/prepositions (X)
• Now deem any string of terms of the form NX*N to be an extended biword
• Examples: catcher in the rye
  N X X N
  king of Denmark
  N X N
• Include extended biwords in the term vocabulary
• Queries are processed accordingly
Issues with biword indexes

• **Why are biword indexes rarely used?**
• False positives, as noted above
• Index blowup due to very large term vocabulary
Positional indexes

- Positional indexes are a more efficient alternative to biword indexes.
- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions
Positional indexes: Example

Query: “to\textsubscript{1} be\textsubscript{2} or\textsubscript{3} not\textsubscript{4} to\textsubscript{5} be\textsubscript{6}”

TO, 993427:

\langle 1, \langle 7, 18, 33, 72, 86, 231 \rangle; \\
2: \langle 1, 17, 74, 222, 255 \rangle; \\
4: \langle 8, 16, 190, 429, 433 \rangle; \\
5: \langle 363, 367 \rangle; \\
7: \langle 13, 23, 191 \rangle; \ldots \rangle

BE, 178239:

\langle 1, \langle 17, 25 \rangle; \\
4: \langle 17, 191, 291, 430, 434 \rangle; \\
5: \langle 14, 19, 101 \rangle; \ldots \rangle

Document 4 is a match!
Proximity search

- We just saw how to use a positional index for phrase searches.
- We can also use it for proximity search.
- For example: employment /4 place
- Find all documents that contain EMPLOYMENT and PLACE within 4 words of each other.
- Employment agencies that place healthcare workers are seeing growth is a hit.
- Employment agencies that have learned to adapt now place healthcare workers is not a hit.
Proximity search

- Use the positional index
- Simplest algorithm: look at cross-product of positions of (i) EMPLOYMENT in document and (ii) PLACE in document
- Very inefficient for frequent words, especially stop words
- Note that we want to return the actual matching positions, not just a list of documents.
- This is important for dynamic summaries etc.
“Proximity” intersection

```python
def positional_intersect(p1, p2, k):
    answer = [
        docID(p1) == docID(p2)
        then l ← ⟨ ⟩
        pp1 ← positions(p1)
        pp2 ← positions(p2)
        while pp1 ≠ NIL
        do while pp2 ≠ NIL
            do if |pos(pp1) − pos(pp2)| ≤ k
                then ADD(l, pos(pp2))
                else if pos(pp2) > pos(pp1)
                    then break
            pp2 ← next(pp2)
        while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
        do DELETE(l[0])
        for each ps ∈ l
            do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
        pp1 ← next(pp1)
    p1 ← next(p1)
    p2 ← next(p2)
    else if docID(p1) < docID(p2)
        then p1 ← next(p1)
        else p2 ← next(p2)
    return answer
```
Combination scheme

- Biword indexes and positional indexes can be profitably combined.
- Many biwords are extremely frequent: Michael Jackson, Britney Spears etc.
- For these biwords, increased speed compared to positional postings intersection is substantial.
- Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme. Faster than a positional index, at a cost of 26% more space for index.
“Positional” queries on Google

- For web search engines, positional queries are much more expensive than regular Boolean queries.
- Let’s look at the example of phrase queries.
- Why are they more expensive than regular Boolean queries?
- Can you demonstrate on Google that phrase queries are more expensive than Boolean queries?
Take-away

- Understanding of the basic unit of classical information retrieval systems: **words** and **documents**: What is a document, what is a term?
- Tokenization: how to get from raw text to words (or tokens)
- More complex indexes: skip pointers and phrases
Resources

- Chapter 1 and 2 of IIR
- Resources at [https://tartarus.org/martin/PorterStemmer/](https://tartarus.org/martin/PorterStemmer/)
  - Porter stemmer
D ICTIONARY & TOLERANT RETRIEVAL
THIS LECTURE

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex
The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list ... in what data structure?

- Brutus → [1 2 4 11 31 45 173 174]
- Caesar → [1 2 4 5 6 16 57 132 ...]
- Calpurnia → [2 31 54 101]

...
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>

- 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
    - Due to bucket overflow!
TREE: BINARY TREE

![Binary Tree Diagram]
**Tree: B-tree**

- Definition: Every internal node has a number of children in the interval \([a, b]\) where \(a, b\) are appropriate natural numbers, e.g., \([2, 4]\).
- The range has to do with the size of a disk block or memory page.
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: *

- **mon**: find all docs containing any word beginning with “mon”.
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: $mon \leq w < moo$
- ***mon**: find words ending in “mon”: harder
  - Maintain an additional B-tree for terms backwards. Can retrieve all words in range: $nom \leq w < non$. 
QUIZ: ENUMERATION

From the last slide, how can we enumerate all terms satisfying the wild-card query *pro*nal?
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*, *llo*, *lo*, *o*, *hell*, *hell* where * is a special symbol (end of a term).

- Queries:
  - **X** lookup on **X**
  - **X** lookup on **X**
  - **X** lookup on **X**
  - **X** lookup on **X**

- Query = *hel* o
  - **X** = *hel*, **Y** = o
  - Lookup o*$hel$
QUIZ: PERMUTERM

- How do we handle query 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.
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 second inverted index from bigrams to dictionary terms that match each bigram.
The \( k \)-gram index finds terms based on a query consisting of \( k \)-grams (here \( k=2 \)).
PROCESSING WILD-CARDS

- Query *mon* can now be run as
  - $m$ AND $mo$ AND $on$
- Gets terms that match and AND them.
- 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!

Which web search engines allow wildcard queries?
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 form Heathrow to Narita.
DOCUMENT CORRECTION

- Especially needed for OCR’ed documents
  - Correction algorithms are tuned for this: rn vs. 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.
QUIZ

Considering only insertion, deletion and replacement, what is the edit distance:

1) goat → toad

2) gap → apply
**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?
    - Alternative is to generate everything up to edit distance k and then intersect.
    - Fine for distance 1; okay for distance 2.

- 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.
**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)

The amount overlap indicates the similarity between query and the text

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 from 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 “corrected” at a time
  - *flew from heathrow*
  - *fled form heathrow*
  - *flea form heathrow*

- Hit-based spelling correction: Suggest the alternative that has lots of hits.
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 (Previous lecture).
- 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

\[ \arg\max_{corr} P(\text{corr} | \text{query}) \]

• From Bayes rule, this is equivalent to
  \[ \arg\max_{corr} P(\text{query} | \text{corr}) \times P(\text{corr}) \]
**Soundex**

- Class of heuristics to expand a query into **phonetic** equivalents
  - Language specific – mainly for names
  - E.g., *chebyshev* → *tchebycheff*
- Invented for the U.S. census … in 1918
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)

Details can be found:
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 (vowels and alike) to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows (equivalence classes):
   - 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
4. Retain the first digit if two identical digits are side-by-side
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* $\rightarrow$ H06505 $\rightarrow$ H655.
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

  \[(SPELL(moriset) /3 toron*to) OR SOUNDSEX(chaikofski)\]
RESOURCES

- IIR 3, MG 4.2
- Efficient spell retrieval:
- Nice, easy reading on spell correction:
  - Peter Norvig: How to write a spelling corrector http://norvig.com/spell-correct.html