Information Retrieval and Organisation

Dell Zhang

Birkbeck, University of London

2016/17
Dictionaries and Tolerant Retrieval
Dictionaries

Dictionary: the data structure for storing the term vocabulary

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

...
Storing Dictionaries

- 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
  - we store these entries in an array
Storing Dictionaries

<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>

space needed: 20 bytes 4 bytes 4 bytes

- How do we look up an element in this array at query time?
- Remember: these dictionaries can be huge, scanning is not an option
Data Structures

- Two main classes of data structures: hash tables and trees
  - Some IR systems use hash tables, some use trees.
- Criteria for when to use hash tables 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?
Hash Tables

- 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 table is faster than in a tree.
- Cons:
  - 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.
- However, binary trees are problematic:
  - Only balanced trees allow efficient retrieval
  - Rebalancing binary trees is expensive
- Use B-trees (the index structure that you know from database lectures)
B-Tree

Taken from documentation for Oracle 10g
Wildcard Queries

- **mon**: find all docs containing any term beginning with *mon*
  - Easy with B-tree dictionary
  - retrieve all terms $t$ in the range: $\text{mon} \leq t < \text{moo}$

- ***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: $\text{nom} \leq t < \text{non}$
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.
Wildcards in Middle of Term

- Example: m*nchen
- We could look up m* and *nchen in the B-tree and intersect the two term sets.
  - Expensive (there are probably thousands and thousands of terms beginning with “m”)
- 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, o$hell, and $hello to the B-tree where $ is a special symbol
Permuterm Index

▶ 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$hel*$

▶ It’s really a tree and should be called permuterm tree

▶ But permuterm index is more common name.
Query Processing

- Once we modified the query (as shown on last slide), we can do a regular lookup on a B-tree.
- This is much faster than looking up X* and *Y and combining results (for query X*Y).
- Permuterm index also handles leading wildcards: *X.
- It has a disadvantage, though: quadruples the size of the dictionary compared to a regular B-tree (as every term is stored multiple times).
**k-gram Index**

- More space-efficient than permuterm index
- Enumerate all character $k$-grams (sequence of $k$ characters) occurring in a term
  - 2-grams are also called *bigrams*
  - 3-grams are also called *trigrams*
- Example:
  - from *April is the cruelest month*
  - we get the bigrams:
    - $a$ $a p$ $p r$ $r i$ $i l$ $l$ $i$ $i s$ $s$ $s$ $t$ $t$ $t$ $h$ $h$ $h$ $e$ $e$ $e$ $c$ $c$ $c$ $r$ $r$ $r$ $u$ $u$ $u$ $r$ $e$ $e$ $el$ $l$ $e$ $s$ $s$ $t$ $t$ $t$
  - $m$ $m$ $m$ $m$ $o$ $o$ $o$ $o$ $n$ $n$ $n$ $t$ $t$ $t$ $h$ $h$ $h$
  - $s$ is a special word boundary symbol.
- Maintain an inverted index from bigrams to the terms that contain the bigram
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 Wildcard Queries

- Query `mon*` can now be run as: $m \text{ AND } mo \text{ AND } on$
- Gets us all terms with the prefix `mon` ...
- ...but also many “false positives” like `moon`
- We must post-filter 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).
We must potentially execute a large number of Boolean queries for each enumerated, filtered term (on the term-document index)

- Recall the query: gen* AND universit*
- Most straightforward semantics: Conjunction of disjunctions
- Very expensive

Users hate to type

- If abbreviated queries like pyth* theo* for pythagoras’ theorem are legal, users will use them . . .
- . . . a lot
Spelling Correction

- Two principal uses
  - Correcting documents being indexed
  - Correcting user queries

- Two different methods
  - *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 the *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 (i.e. documents digitized via Optical Character Recognition)

✦ 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
Correcting Queries

- Can we use the term vocabulary of the inverted index as the list of correct words?
  - It can be very biased
  - It may be missing certain terms

- Alternatives:
  - A standard dictionary (Webster’s, Encyclopædia Britannica, etc.)
  - An industry-specific dictionary (for specialized IR systems)
  - The term vocabulary of the collection, appropriately weighted
Computing Distance

▶ How can we compute the distance between words?
▶ We’ll look at some alternatives:
  ▶ edit distance (Levenshtein distance)
  ▶ weighted edit distance
  ▶ $k$-gram overlap
The (minimum) edit distance between two strings $s_1$ and $s_2$ 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 $dog \rightarrow do$: 1 (deletion)
- Levenshtein distance $cat \rightarrow cart$: 1 (insertion)
- Levenshtein distance $cat \rightarrow cut$: 1 (replacement)
- Levenshtein distance $cat \rightarrow act$: 2
  (2 replacements or 1 insertion and 1 deletion)
Computing Distance

- Getting from *cats* to *fast*

<table>
<thead>
<tr>
<th></th>
<th>“”</th>
<th>f</th>
<th>a</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>“” → “”</td>
<td>“” → f</td>
<td>“” → fa</td>
<td>“” → fas</td>
<td>“” → fast</td>
</tr>
<tr>
<td>c</td>
<td>c → “”</td>
<td>c → f</td>
<td>c → fa</td>
<td>c → fas</td>
<td>c → fast</td>
</tr>
<tr>
<td>a</td>
<td>ca → “”</td>
<td>ca → f</td>
<td>ca → fa</td>
<td>ca → fas</td>
<td>ca → fast</td>
</tr>
<tr>
<td>t</td>
<td>cat → “”</td>
<td>cat → f</td>
<td>cat → fa</td>
<td>cat → fas</td>
<td>cat → fast</td>
</tr>
<tr>
<td>s</td>
<td>cats → “”</td>
<td>cats → f</td>
<td>cats → fa</td>
<td>cats → fas</td>
<td>cats → fast</td>
</tr>
</tbody>
</table>

- Each cell will contain the (cheapest) cost of getting from the string on the left-hand side to the string on the right-hand side.
Computing Distance

- We know the costs for the uppermost row and the leftmost column:
  - we have to get from “” to \textit{fast} by inserting characters
  - we have to get from \textit{cats} to “” by deleting characters

\begin{tabular}{|c|c|c|c|c|}
\hline
 & “” & f & a & s & t \\
\hline
“” & 0 & 1 & 2 & 3 & 4 \\
\hline
c & 1 & & & & \\
\hline
a & 2 & & & & \\
\hline
\textit{t} & 3 & & & & \\
\hline
\textit{s} & 4 & & & & \\
\hline
\end{tabular}
Computing Distance

- For other cells, take the minimum of costs
  - Coming from (a):
    - add 1 to cost in (a) — insertion
  - Coming from (b):
    - add 1 to cost in (b) — deletion
  - Coming from (c):
    - if characters in row and column are equal, copy cost from (c)
    - otherwise, add 1 to cost in (c) — replacement
Resulting Matrix

Computing the costs for all cells results in the following matrix:

<table>
<thead>
<tr>
<th></th>
<th>“”</th>
<th>f</th>
<th>a</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

So the Levenshtein distance is 3
Algorithm

EDITDISTANCE($s_1, s_2$)
1   int $m[i, j] = 0$
2   for $i \leftarrow 1$ to $|s_1|$
3       do $m[i, 0] = i$
4   for $j \leftarrow 1$ to $|s_2|$
5       do $m[0, j] = j$
6   for $i \leftarrow 1$ to $|s_1|$
7       do for $j \leftarrow 1$ to $|s_2|$
8           do $m[i, j] = \min\{m[i-1, j-1] + \text{if } (s_1[i] = s_2[j]) \text{ then } 0 \text{ else } 1,\$
9               \quad m[i-1, j] + 1,\$
10              \quad m[i, j-1] + 1\}$
11   return $m[|s_1|, |s_2|]$
Weighted Edit Distance

- As Levenshtein distance, 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 Distances

- Comparing query term $q$ to all terms in the vocabulary is too expensive

- Solution: use heuristics to determine subset
  - Only compare to terms beginning with the same letter (doesn’t work for typos at beginning)
  - Generate set of rotations for $q$ and use a permuterm index (doesn’t work well for replacements)
  - For each rotation, omit a suffix of $l$ characters before doing lookup in permuterm index
    - Ensures that each term in query rotation shares a substring with retrieved terms
    - The value of $l$ could be fixed to a constant length (e.g. 2), or depend on the length of $q$
Using a \textit{k-gram} Index

- Enumerate all \textit{k-grams} in the query term
- Use the \textit{k-gram} index to retrieve “correct” words that match query term \textit{k-grams}
- Threshold by number of matching \textit{k-grams}
  - e.g., only vocabulary terms that differ by at most 3 \textit{k-grams}
Example with 2-grams

▶ Suppose the misspelled word is “bordroom”:
  $b, bo, or, rd, dr, ro, oo, om, m$

```
bo → aboard → about → boardroom → border
or → border → lord → morbid → sordid
rd → aboard → ardent → boardroom → border
```
Example with 3-grams

- Suppose the correct word is “november”:
  $$n, no, nov, ove, vem, emb, ber, er$$, r$$

- And the query term is “december”:
  $$d, de, dec, ece, cem, emb, mbe, ber, er$$, r$$

- So 5 trigrams overlap (out of 10 in each term)

- Issue: Fixed number of k-grams that differ does not work for words of differing length.

- How can we turn this into a normalized measure of overlap?
Jaccard Coefficient

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

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

- $A$ and $B$ don’t have to be the same size.
- Always assigns a number between 0 and 1.

- Application to spelling correction: declare a match if the coefficient is, say, $> 0.8$. 
Our example was:
“an asteroid that fell form the sky”

How can we correct form here?

One idea: hit-based spelling correction

We’ll return back to this idea when we talk about the probabilistic approach to spelling correction, in the second half of the module.
Context-Sensitive Correction

- Given query “flew form munich”
- Retrieve the correct terms close to each query term
  - 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” should have the most hits.
The hit-based algorithm we just outlined is not very efficient.

Suppose we have 7 alternatives for flew, 19 for form and 3 for munich

Then we have to test $7 \times 19 \times 3$ different variants

More efficient alternative: look at the collection of queries, not documents

This assumes that we log queries
General Issues

- User interface
  - Automatic or 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.
Phonetic Matching

- Soundex is the basis for finding *phonetic* (as opposed to orthographic) alternatives.
  - e.g., 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 ⇒ 1
   - C, G, J, K, Q, S, X, Z ⇒ 2
   - D, T ⇒ 3
   - L ⇒ 4
   - M, N ⇒ 5
   - R ⇒ 6
4. Repeatedly remove one out of each pair of consecutive identical digits
5. Remove all 0s from the resulting string; pad the resulting string with trailing 0s, and return the first four positions, which will consist of a letter followed by three digits
Soundex Algorithm

Example

<table>
<thead>
<tr>
<th></th>
<th>difficulty</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>steps 1 and 2</td>
<td>d0ff0c0lt0</td>
<td>d0ff0r0nc0</td>
</tr>
<tr>
<td>step 3</td>
<td>d011020430</td>
<td>d011060520</td>
</tr>
<tr>
<td>step 4</td>
<td>d01020430</td>
<td>d01060520</td>
</tr>
<tr>
<td>step 5</td>
<td>d124</td>
<td>d165</td>
</tr>
</tbody>
</table>

- Vowels are viewed as being interchangeable
- Consonants with similar sounds (e.g. D and T) are put in equivalence classes
- Works fairly well for European languages
Summary

- How to organize a dictionary of an inverted index
- How to do imprecise searches on this dictionary handling
  - wildcards
  - spelling mistakes