Natural Language Processing and Information Retrieval

Indexing and Vector Space Models

Alessandro Moschitti
Department of Computer Science and Information Engineering
University of Trento
Email: moschitti@disi.unitn.it
Outline

- Preprocessing for Inverted index production
- Vector Space
Stop words

- With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:
  - They have little semantic content: the, a, and, to, be
  - There are a lot of them: ~30% of postings for top 30 words

- But the trend is away from doing this:
  - Good compression techniques means the space for including stopwords in a system is very small
  - Good query optimization techniques mean you pay little at query time for including stop words.
  - You need them for:
    - Phrase queries: “King of Denmark”
    - Various song titles, etc.: “Let it be”, “To be or not to be”
    - “Relational” queries: “flights to London”
Normalization to terms

- We need to “normalize” words in indexed text as well as query words into the same form
  - We want to match **U.S.A.** and **USA**

- Result is terms: a **term** is a (normalized) word type, which is an entry in our IR system dictionary

- We most commonly implicitly define equivalence classes of terms by, e.g.,
  - deleting periods to form a term
    - **U.S.A.**, **USA** → **USA**
  - deleting hyphens to form a term
    - **anti-discriminatory**, **antidiscriminatory** → **antidiscriminatory**
Case folding

- Reduce all letters to lower case
  - exception: upper case in mid-sentence?
    - e.g., General Motors
    - Fed vs. fed
    - SAIL vs. sail
  - Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization...

- Google example:
  - Query C.A.T.
  - #1 result was for “cat” (well, Lolcats) not Caterpillar Inc.
Normalization to terms

- An alternative to equivalence classing is to do asymmetric expansion

- An example of where this may be useful
  - Enter: `window`  Search: `window, windows`
  - Enter: `windows`  Search: `Windows, windows, window`
  - Enter: `Windows`  Search: `Windows`

- Potentially more powerful, but less efficient
Lemmatization

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing “proper” reduction to dictionary headword form
Stemming

- Reduce terms to their “roots” before indexing
- “Stemming” suggest crude affix chopping
  - language dependent
  - e.g., *automate(s), automatic, automation* all reduced to *automat.*

for example compressed and compression are both accepted as equivalent to compress.

for example compress and compress are both accepted as equivalent to compress.
Porter’s algorithm

- Commonest algorithm for stemming English
  - Results suggest it’s at least as good as other stemming options

- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*
Typical rules in Porter

- *sses* → *ss*
- *ies* → *i*
- *ational* → *ate*
- *tional* → *tion*

- Rules sensitive to the *measure* of words

- *(m>1) EMENT* →
  - *replacement* → *replac*
  - *cement* → *cement*
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>Term</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>

...
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>

- 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*
Trees: binary tree

- Root
  - a-m
  - n-z
  - a-hu
  - hy-m
  - n-sh
  - si-z
  - ... 

- aardvark
- hyggen

- sickle
- zygo

Sec. 3.1
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].
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: \textit{mon} \leq w < \textit{moo}
- **mon**: find words ending in “mon”: harder
  - Maintain an additional B-tree for terms backwards.
  - Can retrieve all words in range: \textit{nom} \leq w < \textit{non}.

Exercise: from this, how can we enumerate all terms meeting the wild-card query \textit{pro*cent}?
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.
Bigram index example

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

\[
\begin{align*}
&m \\
&mo \\
&on \\
&\text{mace} \rightarrow \text{madden} \\
&\text{among} \rightarrow \text{amortize} \\
&\text{along} \rightarrow \text{among}
\end{align*}
\]
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., \textit{from} $\rightarrow$ \textit{form}
  - Context-sensitive
    - Look at surrounding words,
    - e.g., \textit{I flew form Heathrow to Narita.}
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  \(\text{ (Just 1 with transpose.)}\)
  - from *cat* to *dog* is 3.
- Generally found by dynamic programming.
- See [http://www.merriampark.com/ld.htm](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.
**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**)

![Diagram showing matching trigrams and example words](image)

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
- Hit-based spelling correction: 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?
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
    \[
    \text{argmax}_{corr} P(query \mid corr) \times P(corr)
    \]
    Noisy channel Language model
End Lecture