Introduction to Information Retrieval

Lecture 3: Dictionaries and tolerant retrieval

Recap of the previous lecture

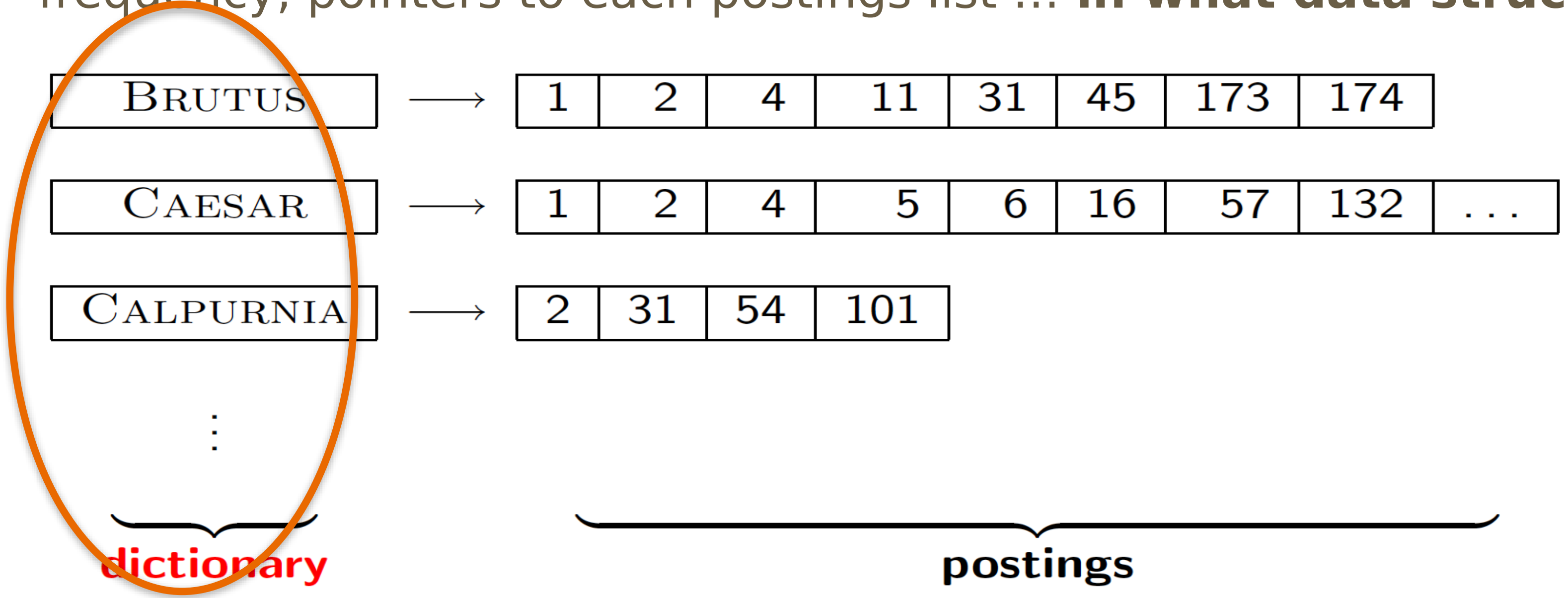
- The type/token distinction
  - Terms are normalized types put in the dictionary
- Tokenization problems:
  - Hyphens, apostrophes, compounds, CJK
- Term equivalence classing:
  - Numbers, case folding, stemming, lemmatization
- Skip pointers
  - Encoding a tree-like structure in a postings list

This lecture

- Dictionary data structures
- “Tolerant” retrieval
    - Wild-card queries
    - Spelling correction
    - Soundex

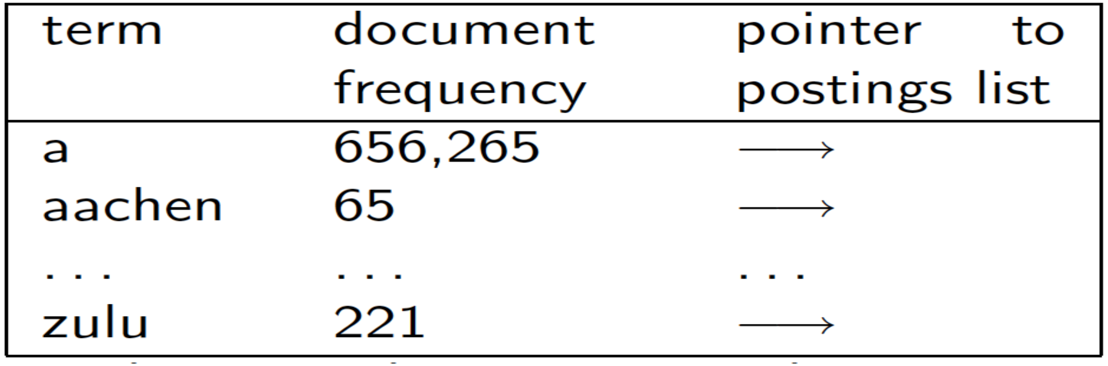
Dictionary data structures for inverted indexes

The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … **in what data structure**?



A naïve dictionary

- An array of struct:



term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

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?

Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

Dictionary data structures

- The choice of solution (hashing, or search trees) is governed by a number of questions:

  (1) How many keys are we likely to have?

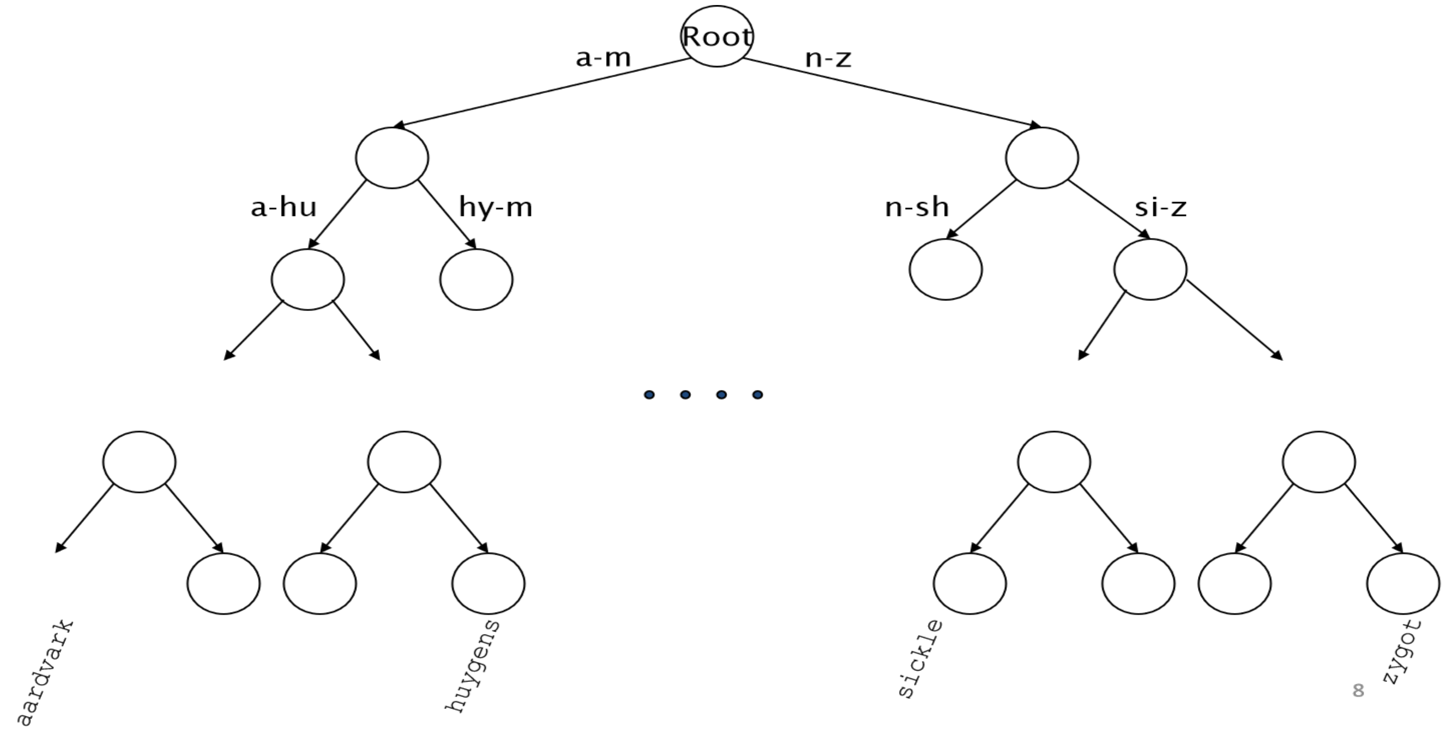
  (2) Is the number likely to remain static, or change a lot – and in the case of changes, are we likely to only have new keys inserted, or to also have some keys in the dictionary be deleted?

  (3) What are the relative frequencies with which various keys will be accessed?

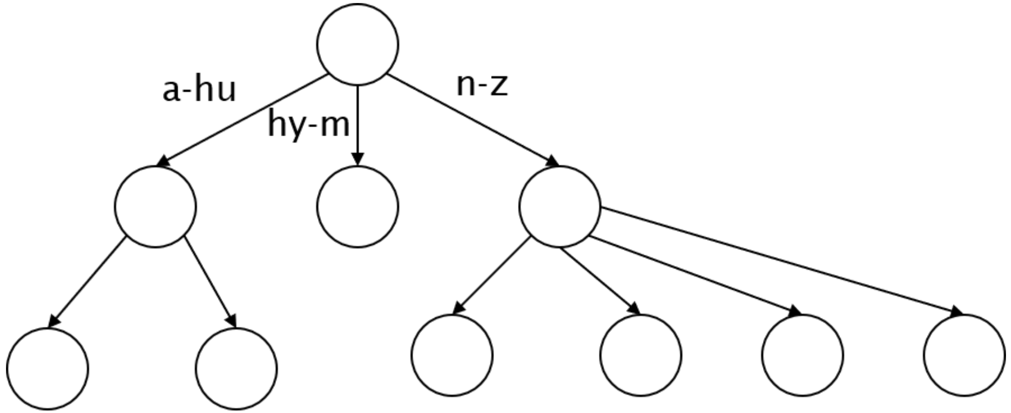
Hashtables

- 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

Tree: binary tree



Tree: B-tree



Definition: Every internal nodel 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)

Trees

- Cons:
  - Slower: O(log M) [and this requires **balanced** tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Introduction to Information Retrieval

Wild-card Queries

Introduction

1. the user is uncertain of the spelling of a query term

   e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney

2. the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants

   e.g., color vs. colour

Introduction

3. the user seeks documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming

   e.g., judicial vs. judiciary , leading to the wildcard query judicia*

4. the user is uncertain of the correct rendition of a foreign word or phrase

   e.g., the query Universit* Stuttgart

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 ≤ 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** ? => pro* and *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?
  - **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

co*tion → tion$co*

- This gives rise to the **Permuterm** Index.

Permuterm index

- For term **hello**, index under:
    - **hello$, ello$h, llohe, lohel, o$hell, $hello**

  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!

Query = ***hel*o***
X=***hel***, Y=***o***
Lookup ***o$hel****

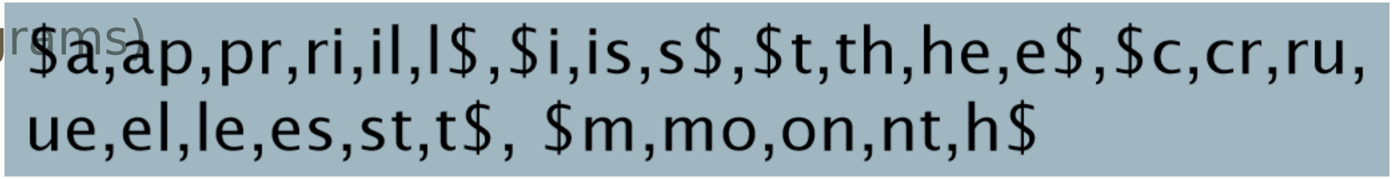
Permuterm query processing

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

Empirical observation for English.

Bigram (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)

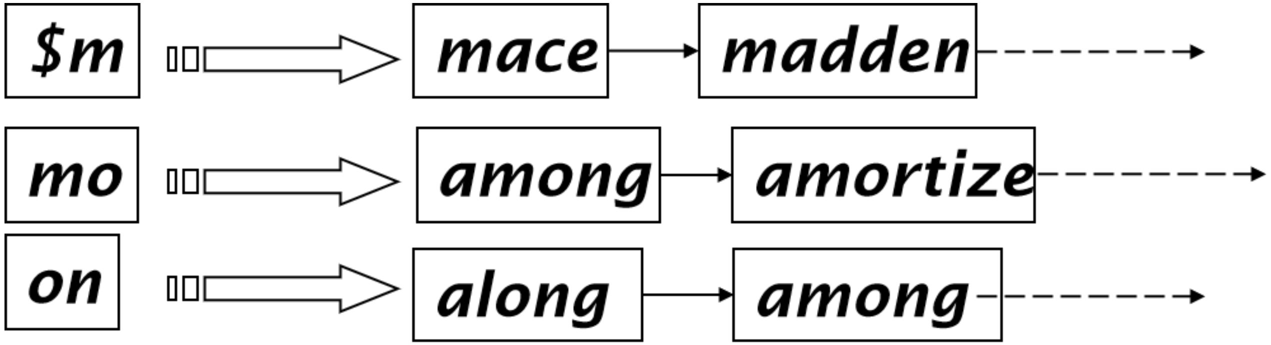


a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
ue,el,le,es,st,t$, m,mo,on,nt,h

  - $ 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).



Processing wild-cards

- Query mon* can now be run as
  - $m AND mo 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).

Introduction to Information Retrieval

Spelling Correction

Spell correction

- Two principal uses
  - Correcting document(s) being indexed (document docment)
  - Correcting user queries to retrieve “right” answers (docment)
- 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

Spell correction

- 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/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).

Document correction

- 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 **Ryan Goslang** (Gosling)
- 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

Isolated word correction

- The lexicon of the indexed corpus
  - E.g., all words on the web
  - All names, acronyms etc.
  - (Including the mis-spellings)

Isolated word correction

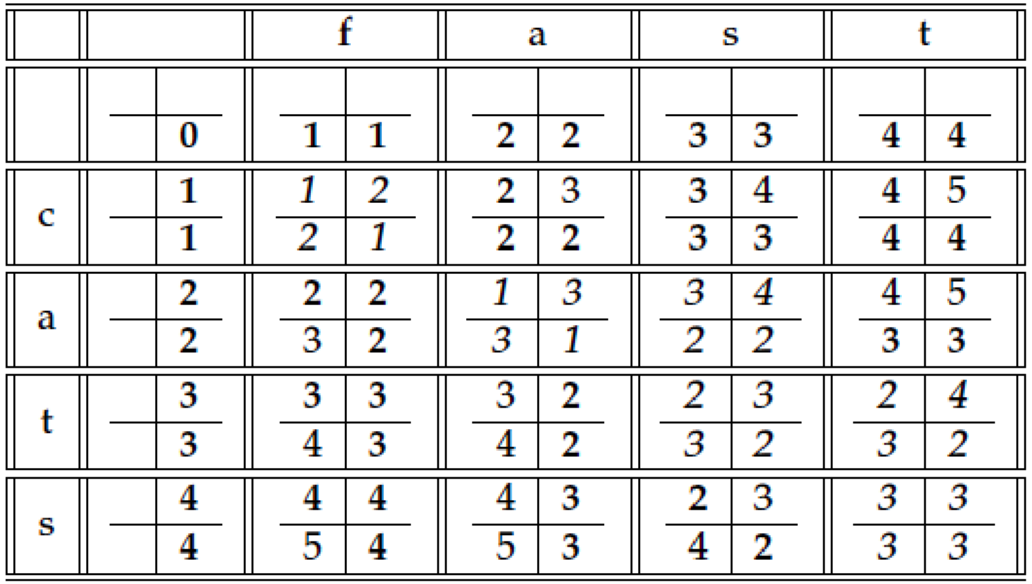
- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- **What’s “closest**”?
- We’ll study several alternatives
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - n-gram overlap

Edit distance

- Given two strings S1 and S2, 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 → dat (replacement) → dot (r) → dog (r).
    - (at → dat) deletion, insertion (2)

Edit distance

- Generally found by dynamic programming.
- See http://www.merriampark.com/ld.htm for a nice example plus an applet.



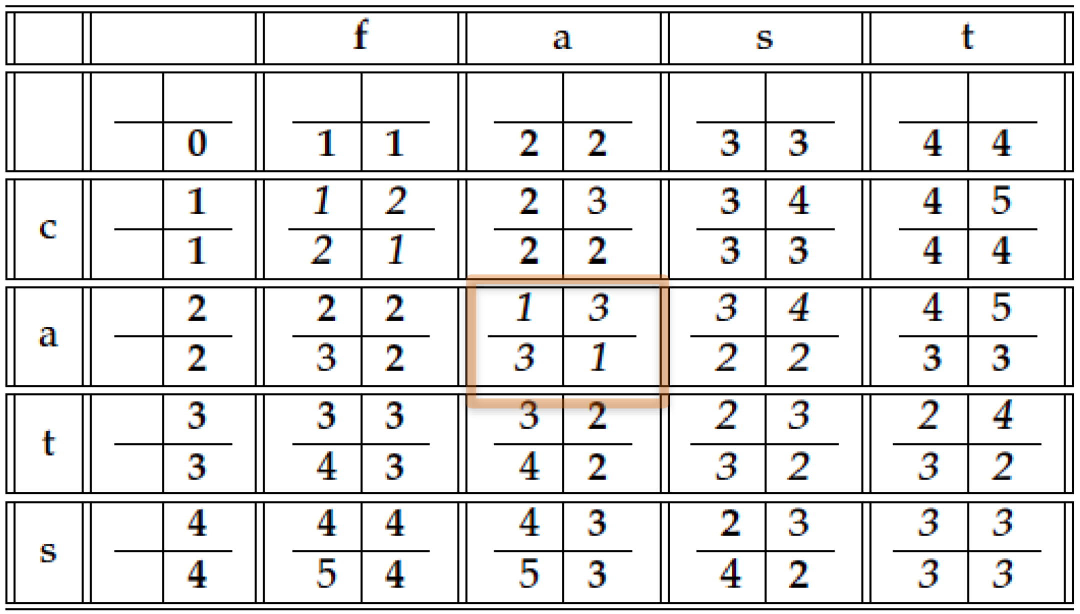
			f		a		s		t	
		0	1	1	2	2	3	3	4	4
c		1	*1*	*2*	2	3	3	4	4	5
		1	*2*	*1*	2	2	3	3	4	4
a		2	2	2	*1*	*3*	*3*	*4*	4	5
		2	3	2	*3*	*1*	*2*	*2*	3	3
t		3	3	3	3	2	*2*	*3*	*2*	*4*
		3	4	3	4	2	*3*	*2*	*3*	*2*
s		4	4	4	4	3	2	3	*3*	*3*
		4	5	4	5	3	4	2	*3*	*3*

‣ **Figure 3.6** Example Levenshtein distance computation. The 2×2 cell in the $[i, j$
ntry of the table shows the three numbers whose minimum yields the fourth. Th
ells in italics determine the edit distance in this example.

```
EditDistance($s_1, s_2$)
  $int\ m[i,j] = 0$
  for $i \leftarrow 1$ to $|s_1|$
  do $m[i,0] = i$
  for $j \leftarrow 1$ to $|s_2|$
  do $m[0,j] = j$
  for $i \leftarrow 1$ to $|s_1|$
  do for $j \leftarrow 1$ to $|s_2|$
      do $m[i,j] = \min\{m[i-1,j-1] +$ if $(s_1[i] = s_2[j])$ then 0 else 1fi,
                           $m[i-1,j] + 1,$
                           $m[i,j-1] + 1\}$
  return $m[|s_1|,|s_2|]$
```

► **Figure 3.5** Dynamic programming algorithm for computing the edit distance
tween strings s_1 and s_2.

```
do $m[i,j] = \min\{m[i-1,j-1] + \text{if } (s_1[i] = s_2[j]) \text{ then } 0 \text{ else } 1 \text{fi},$
                   $m[i-1,j]+1,$
                   $m[i,j-1]+1\}$
return $m[|s_1|, |s_2|]$
```



		f	a	s	t
	— / 0	— / 1 1	— / 2 2	— / 3 3	— / 4 4
c	1 / 1	*1* *2* / *2* *1*	2 3 / 2 2	3 4 / 3 3	4 5 / 4 4
a	2 / 2	2 2 / 3 2	*1* *3* / *3* *1*	*3* *4* / *2* *2*	4 5 / 3 3
t	3 / 3	3 3 / 4 3	3 2 / 4 2	*2* *3* / *3* *2*	*2* *4* / *3* *2*
s	4 / 4	4 4 / 5 4	4 3 / 5 3	2 3 / 4 2	*3* *3* / *3* *3*

igure 3.6 Example Levenshtein distance computation. The 2 × 2 cell in the $[i, j$
ry of the table shows the three numbers whose minimum yields the fourth. Th
s in italics determine the edit distance in this example.

M[2,3] – a string of length 2 and a string of length 3. – I am finding the distance. M[I,j] – it stores that distance.

D(c,f); D(ca,f), D(c, fa) D(ca,fa)

ca fa, , [c,f]

c fa + 1
ca f + 1
c f + 0

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

Weighted edit distance

- 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

- Cat rat, mat, fat, sat, qat, uat

Using edit distances

- 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.
- (Generating words at edit distance) huge number of words. Second – have dictionary words and try to get edit distance, but then that would also involve

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

$$|\boldsymbol{X} \cap \boldsymbol{Y}| / |\boldsymbol{X} \cup \boldsymbol{Y}|$$

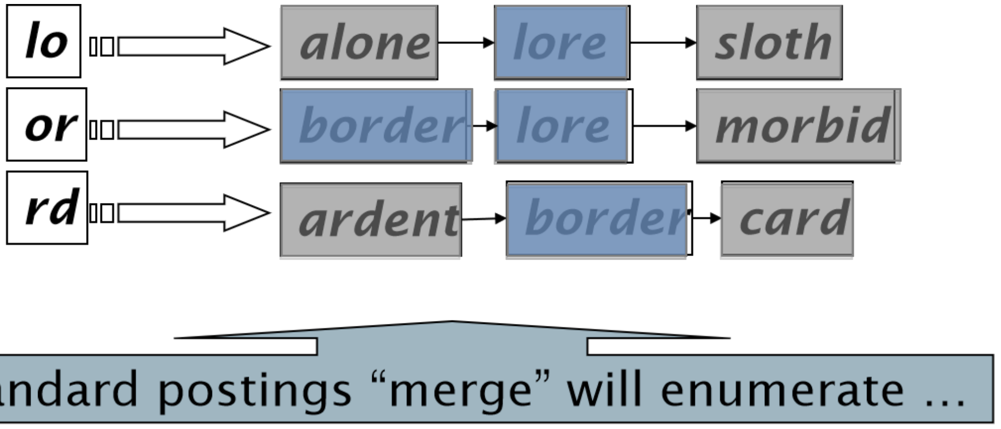
- Equals 1 when X and Y have the same elements and zero when they are disjoint

One option – Jaccard coefficient

- 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 bigrams

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



Adapt this to using Jaccard (or another) measure.

Shortlisting Candidates

One method that has some empirical support is to first use the **k -gram** index to enumerate a set of candidate vocabulary terms that are potential corrections of **q** .

We then compute the edit distance from **q** to each term in this set, selecting terms from the set with small edit distance to **q** .(edit distance = 2)

Summarizing

- Generate word □ Too many

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

General issues in spell correction

- More generally, rank alternatives probabilistically

$$\text{argmaxcorr } P(\text{corr} \mid \text{query})$$

  - From Bayes rule, this is equivalent to

$$\text{argmaxcorr } \underbrace{P(\text{query} \mid \text{corr})}_{\text{Noisy channel}} * \underbrace{P(\text{corr})}_{\text{Language model}}$$

Introduction to Information Retrieval

Soundex

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)

http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

- Retain the first letter of the word.
- Change all occurrences of the following letters to '0' (zero):
  'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
- 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

- Remove all pairs of consecutive digits.
- Remove all zeros from the resulting string.
- 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?

Soundex – typical algorithm

- Retain the first letter of the word.
- Change all occurrences of the following letters to '0' (zero):
  'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
- 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
- Remove all pairs of consecutive digits.
- Remove all zeros from the resulting string.
- Pad the resulting string with trailing zeros and return the first four positions.

Herman
H06505
H655

Hermann
H065055
H06505
H655

Barman
B655

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

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 SOUNDEX(**chaikofski**)

Thank you for listening.
Any questions?