Information Retrieval

- Indexing
- Query Processing
- Document Retrieval

Development:
Moshe Friedman

Credits:
Yoav Goldberg, Ido Dagan, Reut Tsarfaty , Moshe Koppel, Wei Song, David Bamman, Ed Grefenstette, Chris Manning, Tsvi Kuflik, Hinrich Schütze, Christina Lioma and more

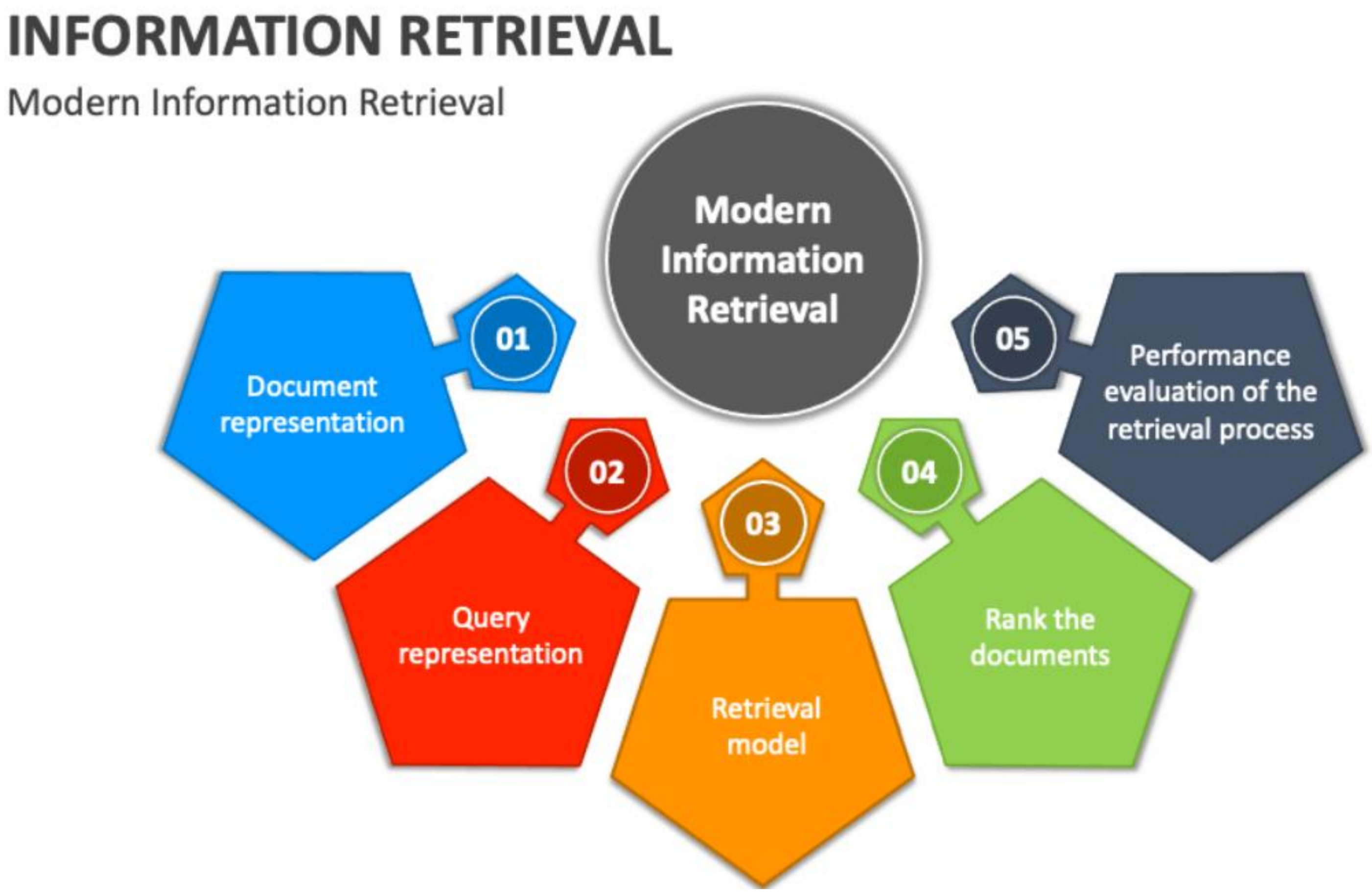
Information Retrieval - administration

Moshe Friedman

Email: moshefr.teach@gmail.com

Reception time: before/after lesson/zoom with coordination

Classic Information Retrieval Components



Introduction to

Information Retrieval

Language and Text - Recap

What do we mean by text?

Text (Wikipedia) - In literary theory, a text is any object that can be "read“.

String (Wikipedia) - In computer programming, a string is traditionally a sequence of characters.

Language is the use of a system of communication which consists of a set of sounds or written symbols.

A text string - We refer to text of (verbal) language saved as a string.

<u>We is included – (written) non-spoken language</u>:

- **Emoticons** - :-) :-(
- **Emoji -** 
- **Text messaging** – LOL ("laughing out loud"), "gr8" ("great")

And more

Text as data – properties and challenges

Sparse data

- Zipf's law, variability

Symbolic

- abstract symbol to meaning mapping, variability, ambiguity

Many levels of granularity

- document, paragraph, sentence, word, characters

How would these affect a classifier over text data?

Text data – properties and challenges
Variability - one meaning, many forms

he acquired it

he purchased it

he bought it

it was bought by him

it was sold to him

she sold it to him

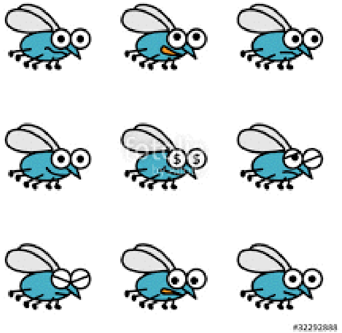
she sold him that

Text data – properties and challenges
Ambiguity - one form, many meanings

Time Flies

If time is a **noun**

If time is a **verb**

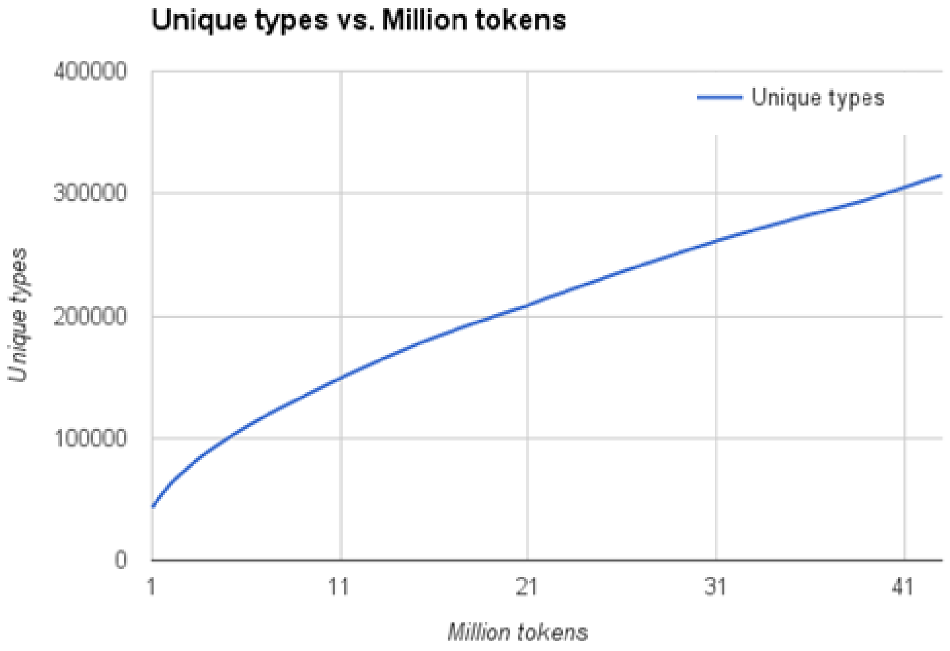


Text data – properties and challenges
Zipf law

Word frequencies follow a power-law distribution.
--> Long tail - most words will occur only few times if they occur

There are very likely to be word forms we did not see

In a 43M words text, there are:
- 316K unique words
- 144K words occur once
- 42K words occur twice
...
- 26K words occur >50 times



Morphological analysis

ספרו - ה-ספר-של-הוא

Lemma: the "dictionary entry" of a word.

create, created, creating, creator, creativity

create, create, create, creator, creativity

Lemmatization: reducing the inflected forms of a word into a single form for easy analysis.

Stem: a "base form", based on heuristics.

Stemming: cutting the inflected words to their root form.

create, created, creating, creator, creativity

creat, creat, creat, creat, creat

נרמול בעברית (ניקוד) - Word / token normalization

פיסוק – קיצורים, ראשי תיבות

- פְּרִי --> פרי
- צוּרָה --> צורה

סיכון:

- דֶּרֶךְ --> דרך (איבוד מידע, דרך --> דֶּרֶךְ או דָּרַךְ)

Basic useful preprocessing operation - summary

Tokenization (sometimes segmentation): dividing a text into tokens.

Input: “I love playing soccer with my friends, mostly on the weekends!”
Output: ["I", "love", "playing", "soccer", "with", "my", "friends", “mostly”, "on", "the", "weekends", "!"]

Stemming: cutting the inflected words to their root form.

Input: "The **mice** in the **fields were running** and **jumping** around."
Output: " The **mice** in the **field were run** and **jump** around .”

Lemmatization: reducing the inflected forms of a word into a single form for easy analysis.

Input: "The **mice** in the **fields were running** and **jumping** around.”
Output: "The **mouse** in the **field be run** and **jump** around”

<u>Other Useful operations:</u>

Sentence breaking: placing sentence boundaries on a text.

Word Normalizations and cleaning

Stop word removal: removing words, which are considered as such.

Part-of-speech tagging: identifying the part of speech for every word.

The bag of words (BOW) model

Each **word** is treated as a feature in a unit called **document**.

Each such word will become a feature

How do we measure their strength?

- Word Count
  - What about zipf law?

Vectorization:
extracting basic feature units

The bag of words (BOW) model:

Each **word** is treated as a feature in a unit called **document**.

Each such word will become a feature

- Alternative: **original tokens – no processing**
- Alternative: **normalized words – e.g., lemmas, stems**
- Alternative: **partial word (prefix, suffix)**
- Alternative: **ngrams – unigram, bigram, trigram**
- Alternative: **characters – we will usually use with ngrams**
- More complex alternatives ...

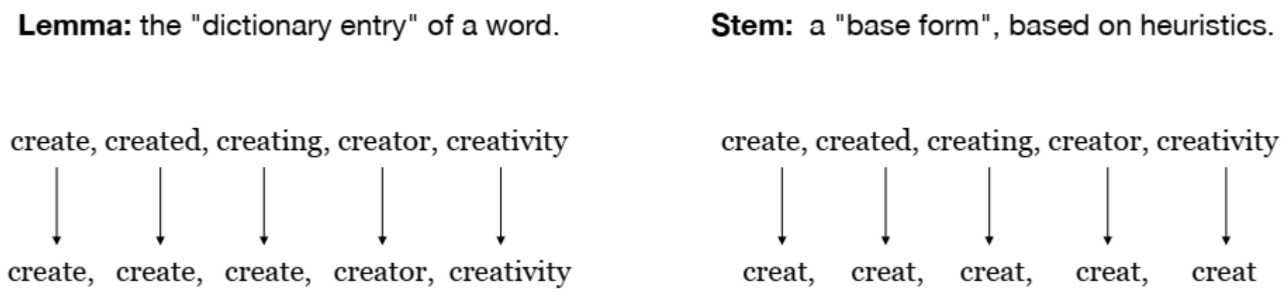
Vectorization:
extracting basic feature units

The bag of words (BOW) model:

Each **word** is treated as a feature in a unit called **document**.

Each such word will become a feature

- Alternative: **normalized words – e.g., lemmas, stems**



Vectorization:
extracting basic feature units

The bag of words (BOW) model:

Each **word** is treated as a feature in a unit called **document.**

Each such word will become a feature

	about	bird	heard	is	the	word	you
About the bird, the bird, bird bird bird	1	5	0	0	2	0	0
You heard about the bird	1	1	1	0	1	0	1
The bird is the word	0	1	0	1	2	1	0

Feature Vectors

(1, 5, 0, 0, 2, 0, 0)

(1, 1, 1, 0, 1, 0, 1)

(0, 1, 0, 1, 2, 1, 0)

Vectorization:
extracting basic feature units

The bag of words (BOW) model:

Each **word** is treated as a feature in a unit called **document**.

Each such word will become a feature

- Alternative: **ngrams – unigram, bigram, trigram**

ngrams: unigrams bigrams trigrams

['the','special', 'onion', 'soup', 'was', 'not', 'very', 'bad',
'the special', 'special onion', 'onion soup', 'soup was',
'was not', 'not very', 'very bad', 'the special onion', 'special onion soup',
'onion soup was', 'soup was not', 'was not very', 'not very bad']

Introduction to

Information Retrieval

Indexing:

Term-document incidence matrices

Unstructured data in 1620

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?
- One could `grep` all of Shakespeare's plays for ***Brutus*** and ***Caesar,*** then strip out lines containing ***Calpurnia***?
- Why is that not the answer?
  - Slow (for large corpora)
  - *NOT* ***Calpurnia*** is non-trivial
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

Term-document incidence matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if play contains word, 0 otherwise

Brutus *AND* ***Caesar*** *BUT NOT* ***Calpurnia***

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➔ bitwise *AND*.
  - 110100 *AND*
  - 110111 *AND*
  - 101111 =
  - **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Answers to query

- Antony and Cleopatra, Act III, Scene ii

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius ***Caesar*** dead,
He cried almost to roaring; and he wept
When at Philippi he found ***Brutus*** slain.

- Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius ***Caesar*** I was killed i' the
Capitol; ***Brutus*** killed me.

Can't build the incidence matrix

- $M = 500{,}000 \times 10^6$ = half a trillion 0s and 1s.
- But the matrix has no more than one billion 1s.
  - Matrix is extremly sparse.
- What is a better representations?
  - We only record the 1s.

Introduction to

Information Retrieval

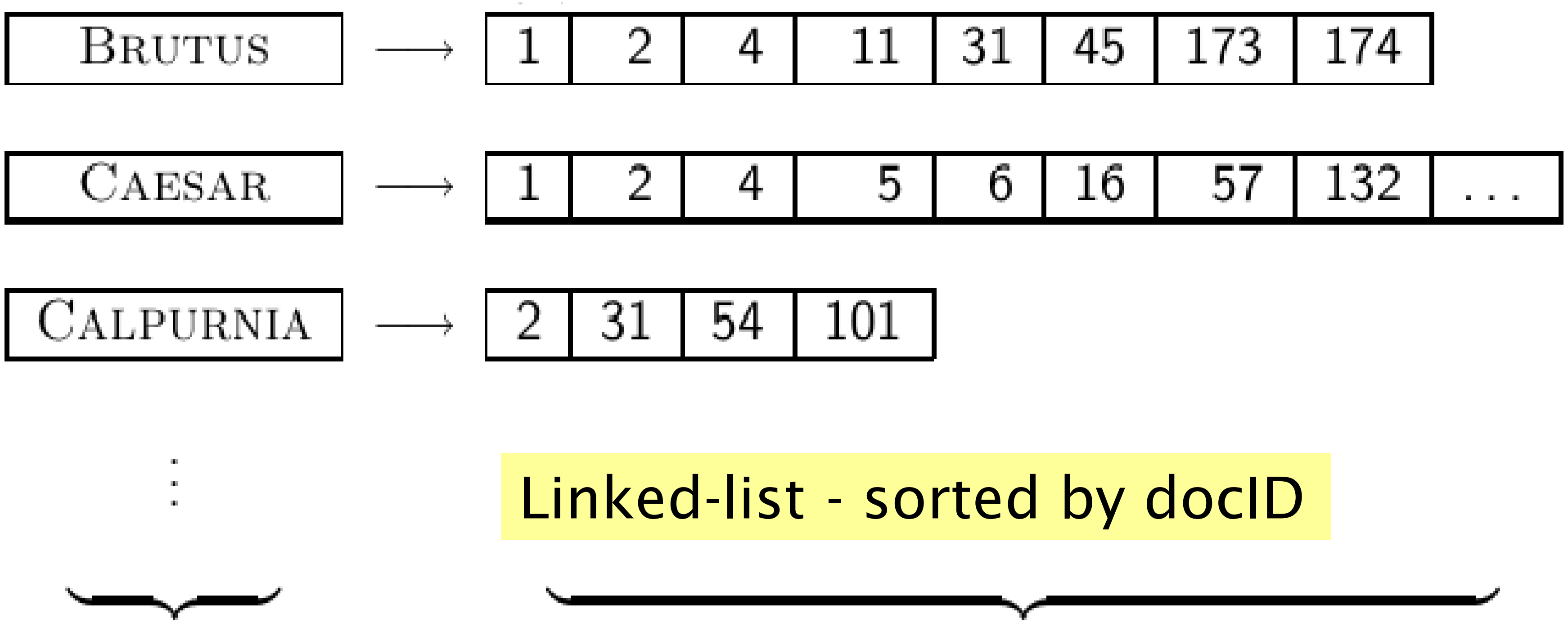
Indexing:

The Inverted Index

The key data structure underlying modern IR

Inverted Index

For each term *t*, we store a list of all documents that contain *t*.



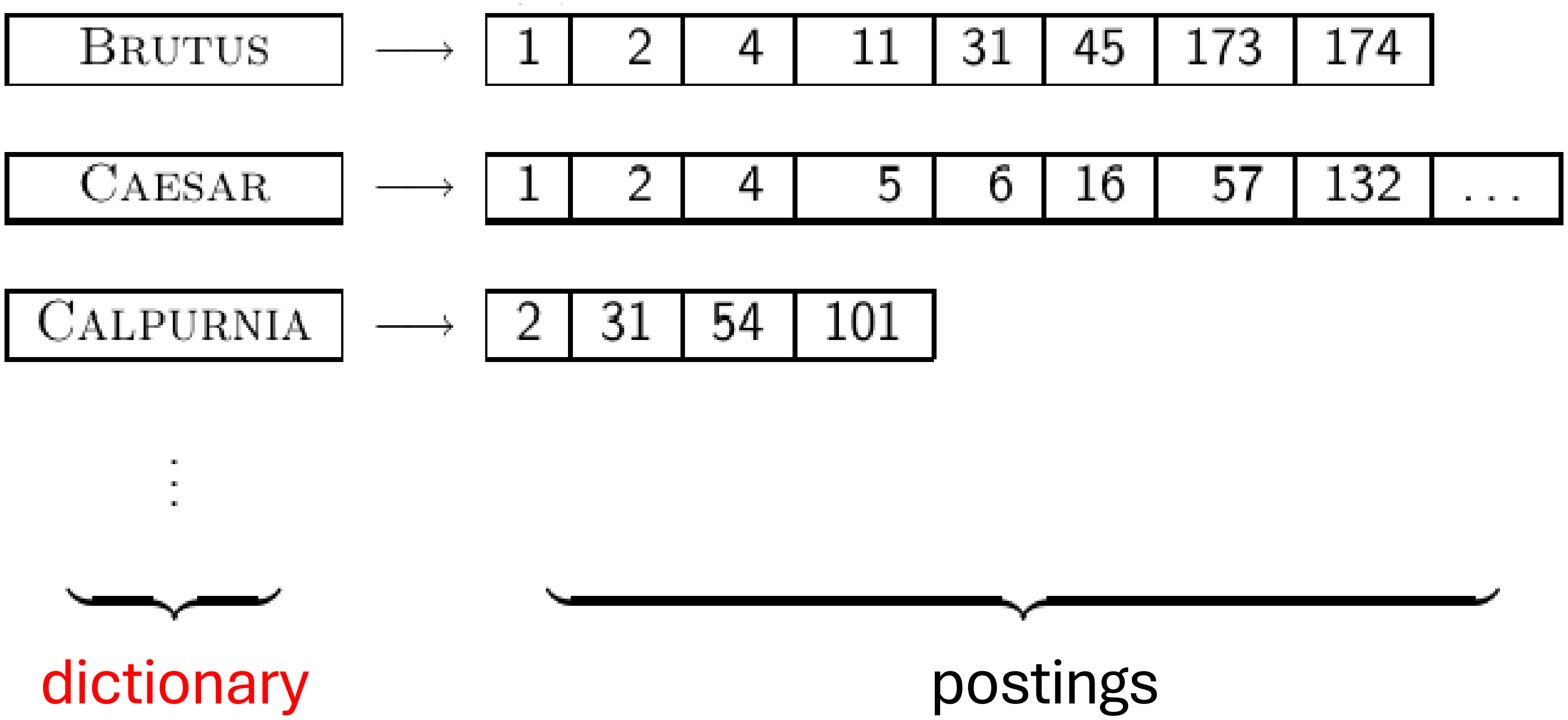
Linked-list - sorted by docID

dictionary

postings

Inverted Index

For each term *t*, we store a list of all documents that contain *t*.



Inverted Index

For each term *t*, we store a list of all documents that contain *t*.

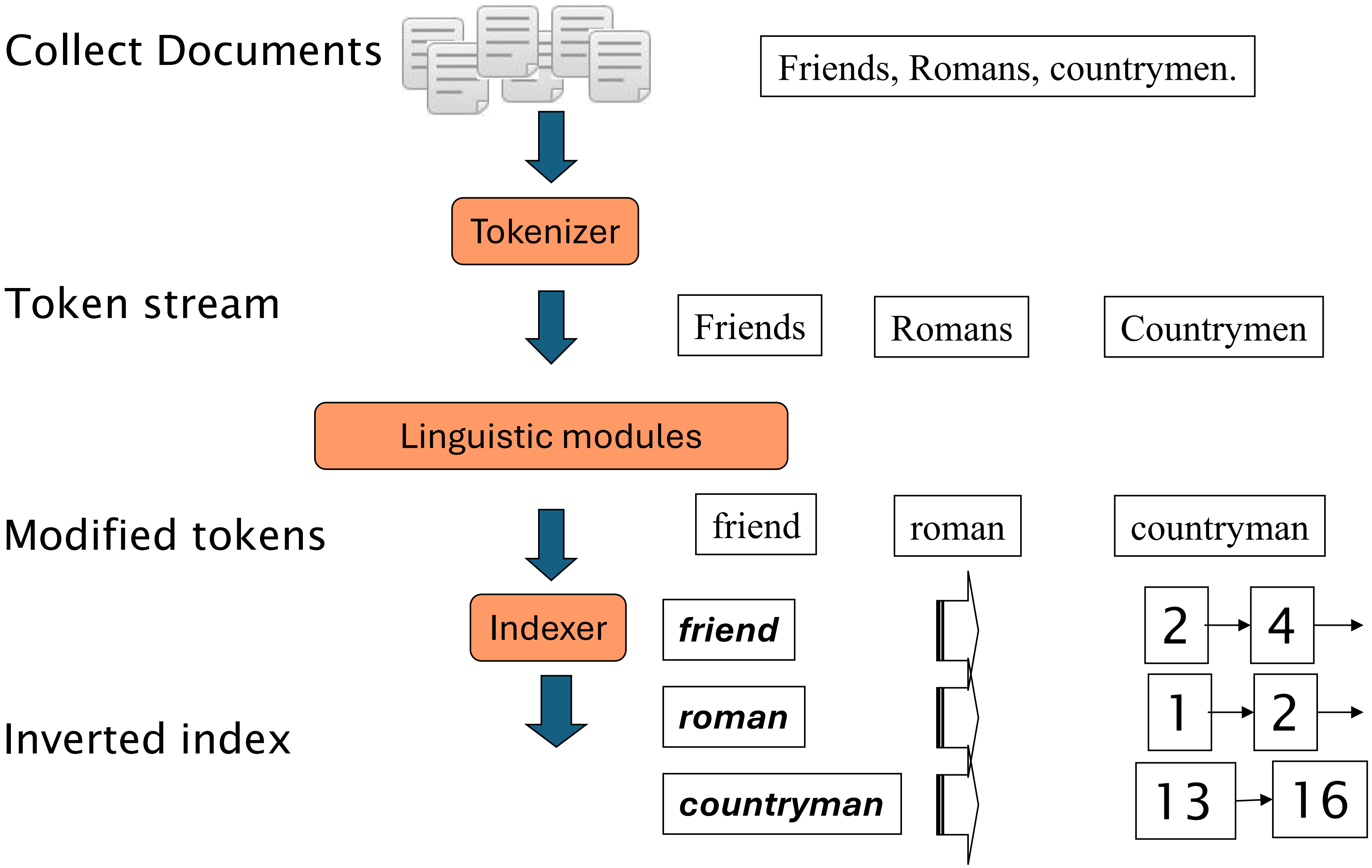
BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	...
CALPURNIA	→	2	31	54	101					
⋮										

What happens if the word ***Caesar*** is added to document 14?

dictionary

postings

Inverted index construction



Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID

Core indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

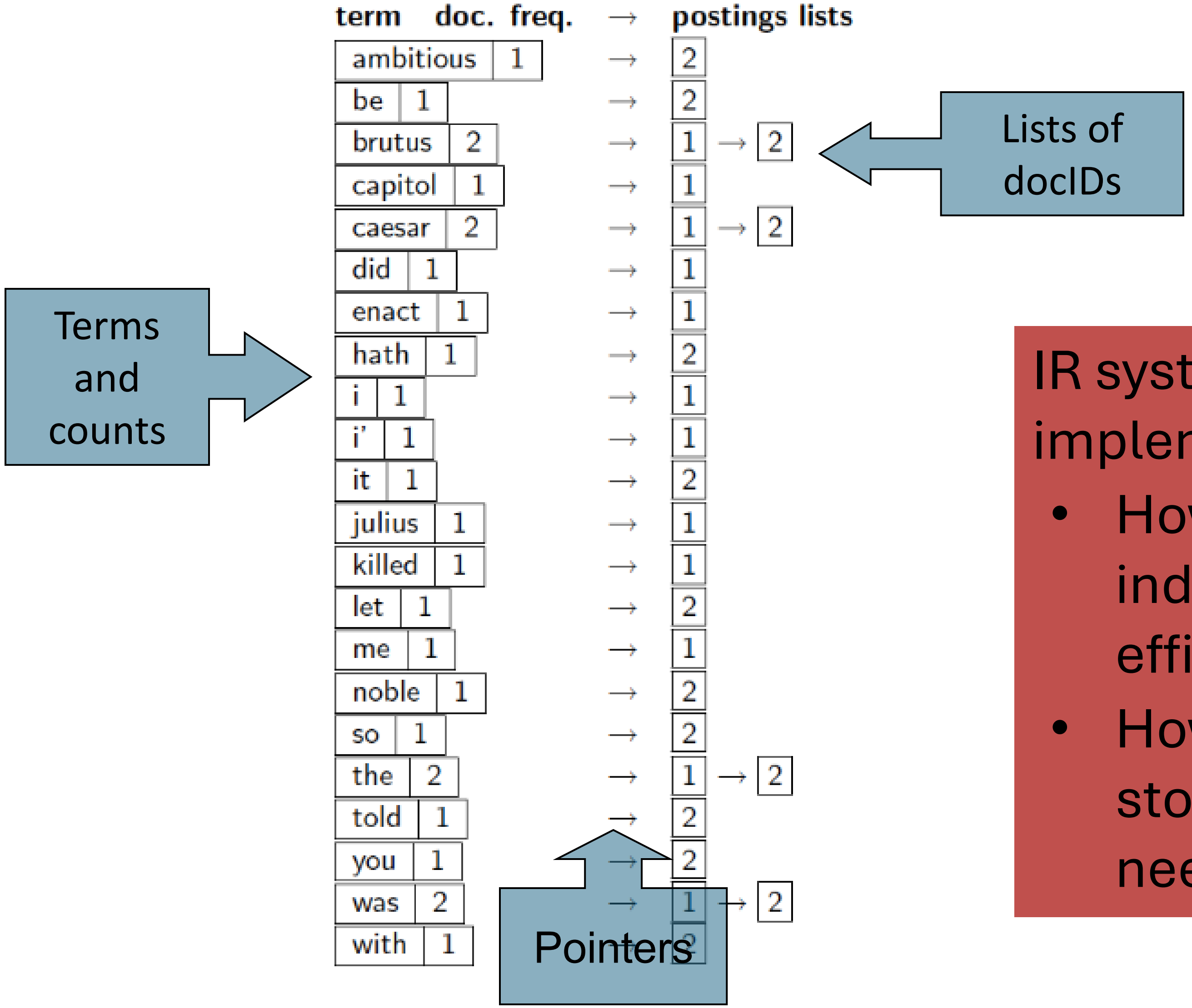
- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?



IR system implementation

- How do we index efficiently?
- How much storage do we need?

Introduction to
Information Retrieval

Query processing with an inverted index

The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?

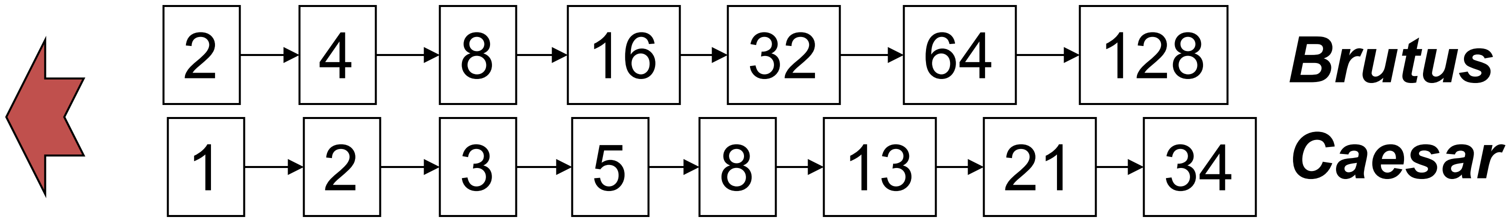
Our focus

Query processing: AND

- Consider processing the query:

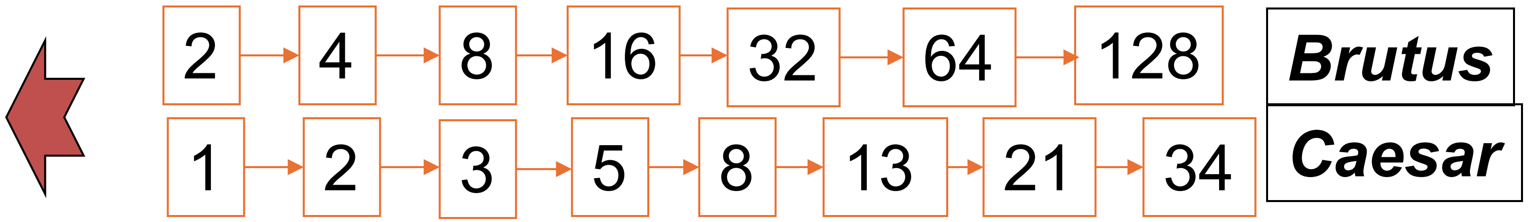
  Brutus *AND* ***Caesar***

  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings (intersect the document sets):



The merge

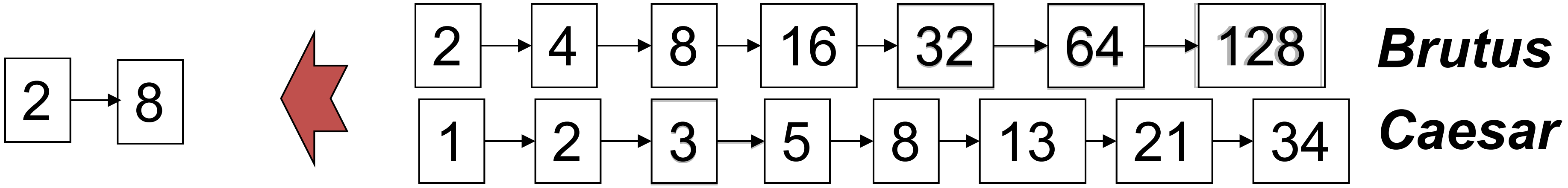
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are x and y, the merge takes $O(x+y)$ operations.
Crucial: postings sorted by docID.

The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.
Crucial: postings sorted by docID.

Intersecting two postings lists (a “merge” algorithm)

```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then ADD(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

Introduction to

Information Retrieval

The Boolean Retrieval Model
& Extended Boolean Models

Boolean queries: Exact match

- The Boolean retrieval model is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using *AND, OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
  - Email, library catalog, Mac OS X Spotlight

Query processing: Exercise

FRANCE ⟶ 1 → 2 → 3 → 4 → 5 → 7 → 8 → 9 → 11 → 12 → 13 → 14 → 15

PARIS ⟶ 2 → 6 → 10 → 12 → 14

LEAR ⟶ 12 → 15

Compute hit list for ((paris AND NOT france) OR lear)

Introduction to

Information Retrieval

Query optimization

Query optimization

- Consider a query that is an and of n terms, $n > 2$
- For each of the terms, get its postings list, then and them together
- Example query: BRUTUS AND CALPURNIA AND CAESAR
- What is the best order for processing this query?

43

Query optimization

- Example query: BRUTUS AND CALPURNIA AND CAESAR
- Simple and effective optimization: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further
- In this example, first CAESAR, then CALPURNIA, then BRUTUS

BRUTUS $\longrightarrow$ $\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{4} \rightarrow \boxed{11} \rightarrow \boxed{31} \rightarrow \boxed{45} \rightarrow \boxed{173} \rightarrow \boxed{174}$

CALPURNIA $\longrightarrow$ $\boxed{2} \rightarrow \boxed{31} \rightarrow \boxed{54} \rightarrow \boxed{101}$

CAESAR $\longrightarrow$ $\boxed{5} \rightarrow \boxed{31}$

44

Optimized intersection algorithm for conjuctive queries

```
INTERSECT(⟨t1, ..., tn⟩)
terms ← SORTBYINCREASINGFREQUENCY(⟨t1, ..., tn⟩)
result ← postings(first(terms))
terms ← rest(terms)
while terms ≠ NIL and result ≠ NIL
do result ← INTERSECT(result, postings(first(terms)))
    terms ← rest(terms)
return result
```


45

Introduction to

Information Retrieval

Phrase queries and positional indexes

Phrase queries

- We want to be able to answer queries such as ***"stanford university"*** – as a phrase
- Thus, the sentence *"I went to university at Stanford"* is not a match.
  - The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only

  <*term* : *docs*> entries

A first attempt: 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 can be processed by breaking them down
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

stanford university *AND* ***university palo*** *AND* ***palo alto***

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

Can have false positives!

Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

- Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy

Solution 2: Positional indexes

- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:

<***term**,* number of docs containing ***term***;
doc1: position1, position2 … ;
doc2: position1, position2 … ;
etc.>

Positional index example

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

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

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
  - Again, 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*?
  - This is a little tricky to do correctly and efficiently
  - See Figure 2.12 of *IIR*

Positional index size

- A positional index expands postings storage *substantially*
  - Even though indices can be compressed
- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system.

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%

Why?

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
  - Caveat: all of this holds for “English-like” languages

Combination schemes

- These two approaches can be profitably 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

Introduction to

Information Retrieval

Back to Preprocessing - Terms

The things indexed in an IR system

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 (IIR 5) means the space for including stop words in a system is very small
  - Good query optimization techniques (IIR 7) 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 may 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***

Normalization: other languages

- Accents: e.g., French ***résumé*** vs. ***resume*.**
- Umlauts: e.g., German: ***Tuebingen*** vs. ***Tübingen***
  - Should be equivalent
- Most important criterion:
  - How are your users like to write their queries for these words?

- Even in languages that standardly have accents, users often may not type them
  - Often best to normalize to a de-accented term
    - ***Tuebingen, Tübingen, Tubingen* → *Tubingen***

Normalization: other languages

- Normalization of things like date forms
  - ***7月30日 vs. 7/30***
  - ***Japanese use of kana vs. Chinese characters***

- Tokenization and normalization may depend on the language and so is intertwined with language detection

Is this German “mit”?

Morgen will ich in MIT …

- Crucial: Need to “normalize” indexed text as well as query terms identically

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…

- Longstanding Google example: [fixed in 2011…]
  - Query C.A.T.
  - #1 result is for "cats" (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

Thesauri and soundex

- Do we handle synonyms and homonyms?
  - E.g., by hand-constructed equivalence classes
    - ***car*** = ***automobile*** ***color*** = ***colour***
  - We can rewrite to form equivalence-class terms
    - When the document contains ***automobile***, index it under ***car-automobile*** (and vice-versa)
  - Or we can expand a query
    - When the query contains ***automobile***, look under ***car*** as well
- What about spelling mistakes?
  - One approach is Soundex, which forms equivalence classes of words based on phonetic heuristics
- More in IIR 3 and IIR 9

Introduction to

Information Retrieval

Preprocessing - Terms

The things indexed in an IR system

Introduction to

Information Retrieval

Preprocessing –
Stemming and Lemmatization

Lemmatization

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

Stemming

- Reduce terms to their "roots" before indexing
- "Stemming" suggests 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 exampl compress and compress ar both accept as equival 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* $\rightarrow$ *ss*
- *ies* $\rightarrow$ *i*
- *ational* $\rightarrow$ *ate*
- *tional* $\rightarrow$ *tion*

- Weight of word sensitive rules
- *(m>1) EMENT* $\rightarrow$
  - *replacement* $\rightarrow$ *replac*
  - *cement* $\rightarrow$ *cement*

Other stemmers

- Other stemmers exist:
  - Lovins stemmer
    - http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
    - Single-pass, longest suffix removal (about 250 rules)
  - Paice/Husk stemmer
  - Snowball
- Full morphological analysis (lemmatization)
  - At most modest benefits for retrieval

Language-specificity

- The above methods embody transformations that are
  - Language-specific, and often
  - Application-specific
- These are “plug-in” addenda to the indexing process
- Both open source and commercial plug-ins are available for handling these

Does stemming help?

- English: very mixed results. Helps recall for some queries but harms precision on others
  - E.g., operative (dentistry) ⇒ oper
- Definitely useful for Spanish, German, Finnish, …
  - 30% performance gains for Finnish!

Introduction to

Information Retrieval

Preprocessing –
Stemming and Lemmatization

Introduction to

Information Retrieval

Preprocessing –
Stemming and Lemmatization

Lemmatization

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

Stemming

- Reduce terms to their "roots" before indexing
- "Stemming" suggests 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 exampl compress and compress ar both accept as equival 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* $\rightarrow$ *ss*
- *ies* $\rightarrow$ *i*
- *ational* $\rightarrow$ *ate*
- *tional* $\rightarrow$ *tion*

- Weight of word sensitive rules
- *(m>1) EMENT →*
  - *replacement → replac*
  - *cement → cement*

Other stemmers

- Other stemmers exist:
  - Lovins stemmer
    - http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
    - Single-pass, longest suffix removal (about 250 rules)
  - Paice/Husk stemmer
  - Snowball
- Full morphological analysis (lemmatization)
  - At most modest benefits for retrieval

Language-specificity

- The above methods embody transformations that are
  - Language-specific, and often
  - Application-specific
- These are “plug-in” addenda to the indexing process
- Both open source and commercial plug-ins are available for handling these

Does stemming help?

- English: very mixed results. Helps recall for some queries but harms precision on others
  - E.g., operative (dentistry) ⇒ oper
- Definitely useful for Spanish, German, Finnish, …
  - 30% performance gains for Finnish!

Introduction to

Information Retrieval

Preprocessing –
Stemming and Lemmatization

Introduction to

Information Retrieval

CS276: Information Retrieval and Web Search

Christopher Manning and Pandu Nayak

Wildcard queries and Spelling Correction

Wild-card queries

Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning with "mon".
- Easy with binary tree (or B-tree) dictionary: 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***.

From this, how can we enumerate all terms meeting the wild-card query ***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.

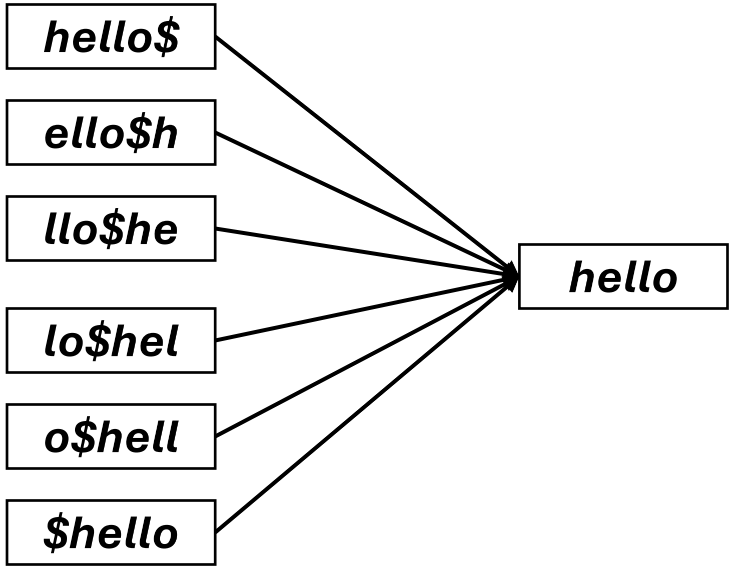
B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - ***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

- Add a ***$*** to the end of each term
- Rotate the resulting term and index them in a B-tree
- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello***

  where $ is a special symbol.



Empirically, dictionary quadruples in size

Permuterm query processing

- (Add ***$***), rotate * to end, lookup in permuterm index
- Queries:
  - **X** — lookup on **X$** ***hello$*** for ***hello***
  - **X*** — lookup on $**X*** ***$hel**** for ***hel****
  - ***X** — lookup on **X$*** ***llo$**** for ****llo***
  - ***X*** — lookup on **X*** ***ell**** for ****ell****
  - **X*Y** — lookup on **Y$X*** ***lo$h*** for ***h*lo***
  - **X*Y*Z** — treat as a search for **X*Z** and post-filter

    For ***h*a*o***, search for ***h*o*** by looking up ***o$h****
    and post-filter ***hello*** and retain ***halo***

Bigram (*k*-gram) indexes

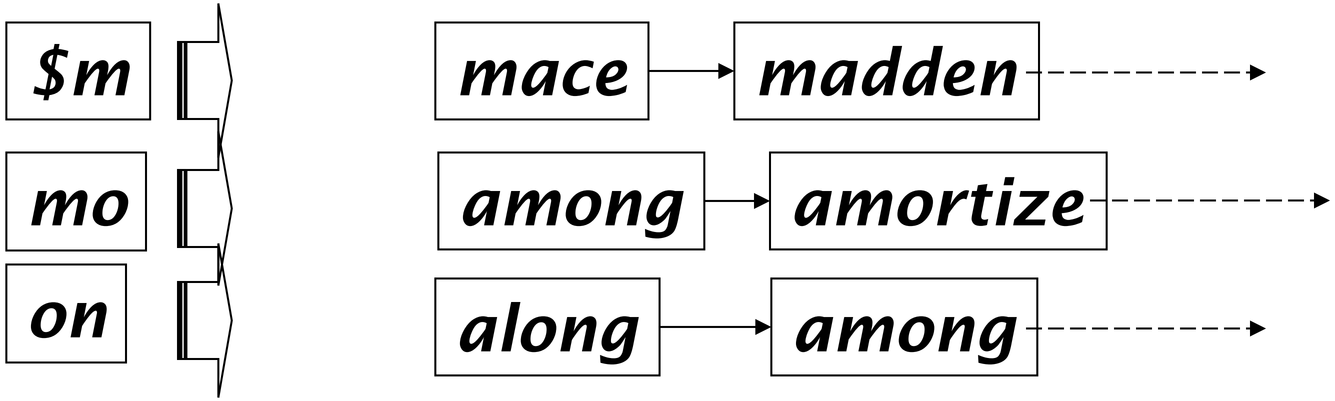
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.*, from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

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).

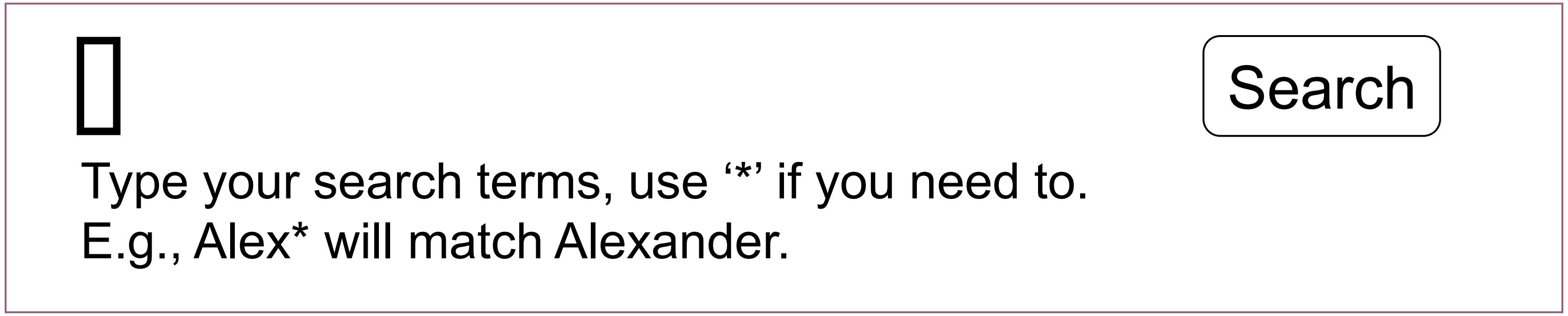


Processing wild-cards

- 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).

Processing wild-card queries

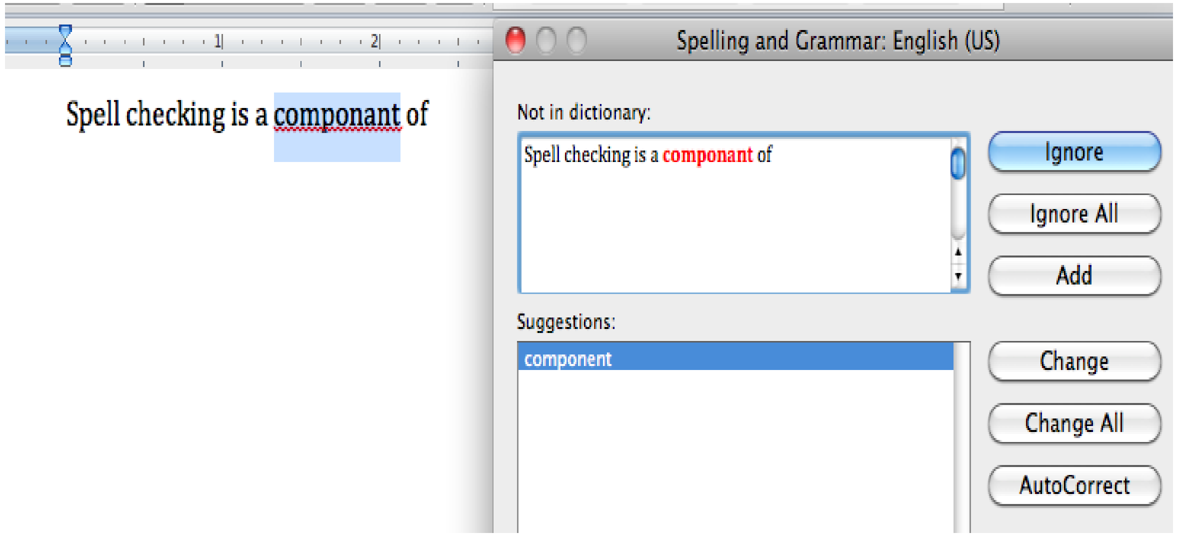
- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!



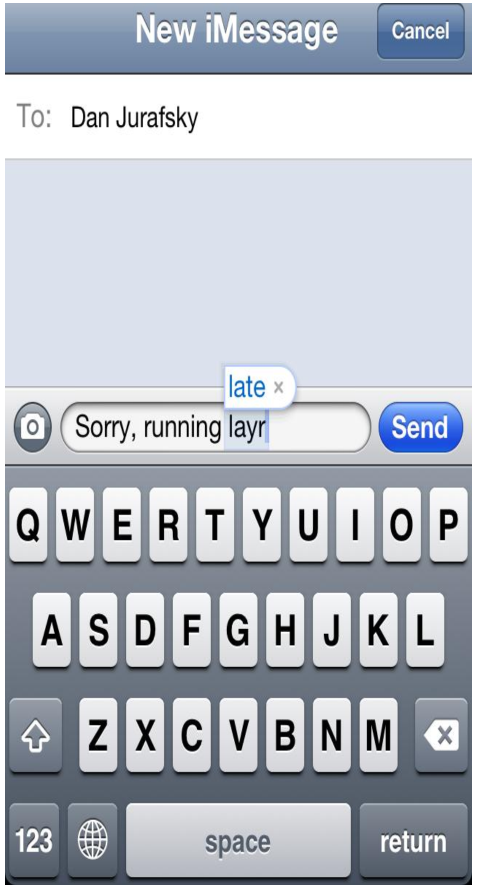
Spelling correction

Applications for spelling correction

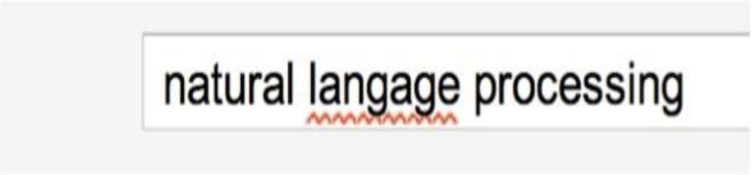
Word processing



Phones



Web search



Showing results for natural ***language*** processing

Search instead for natural langage processing

Rates of spelling errors

Depending on the application, ~1–20% error rates

26%: Web queries Wang *et al.* 2003

13%: Retyping, no backspace: Whitelaw *et al.* English&German

7%: Words corrected retyping on phone-sized organizer

2%: Words uncorrected on organizer Soukoreff &MacKenzie 2003

1-2%: Retyping: Kane and Wobbrock 2007, Gruden et al. 1983

Spelling Tasks

- Spelling Error Detection
- Spelling Error Correction:
  - Autocorrect
    - hte→the
  - Suggest a correction
  - Suggestion lists

Types of spelling errors

- Non-word Errors
  - *graffe →giraffe*
- Real-word Errors
  - Typographical errors
    - *three →there*
  - Cognitive Errors (homophones)
    - *piece→peace,*
    - *too → two*
    - *your →you're*

- Non-word correction was historically mainly context insensitive
- Real-word correction almost needs to be context sensitive

Non-word spelling errors

- Non-word spelling error detection:
  - Any word not in a ***dictionary*** is an error
  - The larger the dictionary the better … up to a point
  - (The Web is full of mis-spellings, so the Web isn't necessarily a great dictionary …)
- Non-word spelling error correction:
  - Generate ***candidates***: real words that are similar to error
  - Choose the one which is best:
    - Shortest weighted edit distance
    - Highest noisy channel probability

Real word & non-word spelling errors

- For each word *w*, generate candidate set:
  - Find candidate words with similar ***pronunciations***
  - Find candidate words with similar ***spellings***
  - Include *w* in candidate set
- Choose best candidate
  - <u>Noisy Channel</u> view of spell errors
  - Context-sensitive – so have to consider whether the surrounding words "make sense"
  - *Flying <u>form</u> Heathrow to LAX* → *Flying <u>from</u> Heathrow to LAX*

Real word & non-word spelling errors

- For each word *w*, generate candidate set:
  - Find candidate words with similar ***pronunciations***
  - Find candidate words with similar ***spellings***
  - Include *w* in candidate set
- Choose best candidate
  - Noisy Channel view of spell errors
  - Context-sensitive – so have to consider whether the surrounding words "make sense"
  - *Flying form Heathrow to LAX* → *Flying from Heathrow to LAX*

Candidate Testing:
Damerau-Levenshtein edit distance

- Minimal edit distance between two strings, where edits are:
  - Insertion
  - Deletion
  - Substitution
  - Transposition of two adjacent letters

Damerau-Levenshtein edit distance

```
algorithm DL-distance is
    input: strings a[1..length(a)], b[1..length(b)]
    output: distance, integer

    da := new array of |Σ| integers
    for i := 1 to |Σ| inclusive do
        da[i] := 0

    let d[−1..length(a), −1..length(b)] be a 2-d array of integers, dimensions length(a)+2, length(b)+2
    // note that d has indices starting at −1, while a, b and da are one-indexed.

    maxdist := length(a) + length(b)
    d[−1, −1] := maxdist
    for i := 0 to length(a) inclusive do
        d[i, −1] := maxdist
        d[i, 0] := i
    for j := 0 to length(b) inclusive do
        d[−1, j] := maxdist
        d[0, j] := j

    for i := 1 to length(a) inclusive do
        db := 0
        for j := 1 to length(b) inclusive do
            k := da[b[j]]
            ℓ := db
            if a[i] = b[j] then
                cost := 0
                db := j
            else
                cost := 1
            d[i, j] := minimum(d[i−1, j−1] + cost,  //substitution
                               d[i,   j−1] + 1,     //insertion
                               d[i−1, j  ] + 1,     //deletion
                               d[k−1, ℓ−1] + (i−k−1) + cost + (j-ℓ−1)) //transposition
        da[a[i]] := i
    return d[length(a), length(b)]
```

Words within 1 of acress

Error	Candidate Correction	Correct Letter	Error Letter	Type
acress	actress	t	-	deletion
acress	cress	-	a	insertion
acress	caress	ca	ac	transposition
acress	access	c	r	substitution
acress	across	o	e	substitution
acress	acres	-	s	insertion

Candidate generation

- 80% of errors are within edit distance 1
- Almost all errors within edit distance 2

- Also allow insertion of **space** or **hyphen**
  - `thisidea` → `this idea`
  - `inlaw` → `in-law`
- Can also allow merging words
  - `data base` → `database`
  - For short texts like a query, can just regard whole string as one item from which to produce edits

How do you generate the candidates?

1. Run through dictionary, check edit distance with each word
2. Generate all words within edit distance ≤ *k* (e.g., *k* = 1 or 2) and then intersect them with dictionary
3. Use a character *k*-gram index and find dictionary words that share "most" *k*-grams with word (e.g., by Jaccard coefficient)
   - see *IIR* sec 3.3.4
4. Compute them fast with a Levenshtein finite state transducer
5. Have a precomputed map of words to possible corrections

A paradigm …

- We want the best spell corrections
- Instead of finding the very best, we
  - Find a subset of pretty good corrections
    - (say, edit distance at most 2)
  - Find the best amongst them
- *These may not be the actual best*
- This is a recurring paradigm in IR including finding the best docs for a query, best answers, best ads …
  - Find a good candidate set
  - Find the top *K amongst them* and return them as the best

Improvements to channel model

- Allow richer edits (Brill and Moore 2000)
  - ent→ant
  - ph→f
  - le→al
- Incorporate pronunciation into channel (Toutanova and Moore 2002)
- Incorporate device into channel
  - Not all Android phones need have the same error model
  - But spell correction may be done at the system level

Until the next time ☺