Introduction to

# Information Retrieval

Introducing Information Retrieval and Web Search

# Information Retrieval

- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).
  - These days we frequently think first of web search, but there are many other cases:
    - E-mail search
    - Searching your laptop
    - Corporate knowledge bases
    - Legal information retrieval

# Unstructured (text) vs. structured (database) data in the mid-nineties

# Unstructured (text) vs. structured (database) data today

# Basic assumptions of Information Retrieval

- Collection: A set of documents
  - Assume it is a static collection for the moment
- Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

# The classic search model

# How good are the retrieved docs?

- *Precision* : Fraction of retrieved docs that are relevant to the user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved
  - More precise definitions and measurements to follow later

Introduction to

# Information Retrieval

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.

# Bigger collections

- Consider $N$ = 1 million documents, each with about 1000 words.
- Avg 6 bytes/word including spaces/punctuation
  - 6GB of data in the documents.
- Say there are $M$ = 500K *distinct* terms among these.

# Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
  - matrix is extremely sparse.

Why?

- What's a better representation?
  - We only record the 1 positions.

Introduction to

# Information Retrieval

The Inverted Index

The key data structure underlying modern IR

# Inverted index

- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each doc by a **docID**, a document serial number
- Can we used fixed-size arrays for this?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ***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?

# Inverted index

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

Sorted by docID (more later on why).

# Inverted index construction

# Initial stages of text processing

- Tokenization
  - Cut character sequence into word tokens
    - Deal with ***"John's", a state-of-the-art solution***
- Normalization
  - Map text and query term to same form
    - You want ***U.S.A.*** and ***USA*** to match
- Stemming
  - We may wish different forms of a root to match
    - ***authorize***, ***authorization***
- Stop words
  - We may omit very common words (or not)
    - ***the, a, to, of***

# 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
  - At least conceptually
    - 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.

# 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, macOS Spotlight

# Example: WestLaw http://www.westlaw.com/

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)
- Tens of terabytes of data; ~700,000 users
- Majority of users *still* use boolean queries
- Example query:
  - What is the statute of limitations in cases involving the federal tort claims act?
  - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
    - /3 = within 3 words, /S = in same sentence

# Example: WestLaw http://www.westlaw.com/

- Another example query:
  - Requirements for disabled people to be able to access a workplace
  - disabl! /p access! /s work-site work-place (employment /3 place
- Note that SPACE is disjunction, not conjunction!
- Long, precise queries; proximity operators; incrementally developed; not like web search
- Many professional searchers still like Boolean search
  - You know exactly what you are getting
- But that doesn't mean it actually works better….

# Boolean queries: More general merges

- Exercise: Adapt the merge for the queries:

  ***Brutus*** *AND NOT* ***Caesar***

  ***Brutus*** *OR NOT* ***Caesar***

- Can we still run through the merge in time O($x$+$y$)? What can we achieve?

# Merging

What about an arbitrary Boolean formula?

***(Brutus** OR **Caesar)** AND NOT*

***(Antony** OR **Cleopatra)***

- Can we always merge in "linear" time?
  - Linear in what?
- Can we do better?

# Query optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ***Brutus*** | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
| ***Caesar*** | 1 | 2 | 3 | 5 | 8 | 16 | 21 | 34 |
| ***Calpurnia*** | 13 | 16 | | | | | | |

Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

# Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*

This is why we kept document freq. in dictionary

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ***Brutus*** | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
| ***Caesar*** | 1 | 2 | 3 | 5 | 8 | 16 | 21 | 34 |
| ***Calpurnia*** | 13 | 16 | | | | | | |

Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

# Exercise

- Recommend a query processing order for

***(tangerine** OR **trees)** AND **(marmalade** OR **skies)** AND **(kaleidoscope** OR **eyes)***

- Which two terms should we process first?

| Term | Freq |
|---|---|
| eyes | 213312 |
| kaleidoscope | 87009 |
| marmalade | 107913 |
| skies | 271658 |
| tangerine | 46653 |
| trees | 316812 |

# More general optimization

- e.g., *(**madding** OR **crowd**) AND (**ignoble** OR **strife**)*
- Get doc. freq.'s for all terms.
- Estimate the size of each *OR* by the sum of its doc. freq.'s (conservative).
- Process in increasing order of *OR* sizes.

# Query processing exercises

- Exercise: If the query is ***friends*** *AND* ***romans*** *AND (NOT* ***countrymen****),* how could we use the freq of ***countrymen***?
- Exercise: Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?
- Hint: Begin with the case of a Boolean *formula* query: in this, each query term appears only once in the query.

# Exercise

- Try the search feature at http://www.rhymezone.com/shakespeare/
- Write down five search features you think it could do better

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

Structured vs. Unstructured Data

# What's ahead in IR? Beyond term search

- What about phrases?
  - ***Stanford University***
- Proximity: Find ***Gates*** *NEAR* ***Microsoft***.
  - Need index to capture position information in docs.
- Zones in documents: Find documents with (*author =* ***Ullman***) *AND* (text contains ***automata***).

# Evidence accumulation

- 1 vs. 0 occurrence of a search term
  - 2 vs. 1 occurrence
  - 3 vs. 2 occurrences, etc.
  - Usually more seems better
- Need term frequency information in docs

# Ranking search results

- Boolean queries give inclusion or exclusion of docs.
- Often we want to rank/group results
  - Need to measure proximity from query to each doc.
  - Need to decide whether docs presented to user are singletons, or a group of docs covering various aspects of the query.

# IR vs. databases: Structured vs unstructured data

- Structured data tends to refer to information in "tables"

| Employee | Manager | Salary |
|---|---|---|
| Smith | Jones | 50000 |
| Chang | Smith | 60000 |
| Ivy | Smith | 50000 |

Typically allows numerical range and exact match (for text) queries, e.g.,

*Salary < 60000 AND Manager = Smith*.

# Unstructured data

- Typically refers to free text
- Allows
  - Keyword queries including operators
  - More sophisticated “concept” queries e.g.,
    - find all web pages dealing with *drug abuse*
- Classic model for searching text documents

# Semi-structured data

- In fact almost no data is "unstructured"
- E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
  - … to say nothing of linguistic structure
- Facilitates "semi-structured" search such as
  - *Title* contains data AND *Bullets* contain search
- Or even
  - *Title* is about Object Oriented Programming AND *Author* something like stro*rup
  - where * is the wild-card operator

# Semi-structured data

- In fact almost no data is "unstructured"
- E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
- Facilitates "semi-structured" search such as
  - *Title* contains data AND *Bullets* contain search

… to say nothing of linguistic structure

# More sophisticated semi-structured search

- *Title* is about Object Oriented Programming AND *Author* something like stro*rup
- where * is the wild-card operator
- Issues:
  - how do you process "about"?
  - how do you rank results?
- The focus of XML search (*IIR* chapter 10)

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Basic inverted index construction

# Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?

# Recall index construction

- Documents are parsed to extract words and these are saved with the Document ID.

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 | Doc # |
|---|---|
| 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 |

# Key step

- After all documents have been parsed, the inverted file is sorted by terms.

We focus on this sort step.

| Term | Doc # |
|---|---|
| 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 | Doc # |
|---|---|
| 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 |

# RCV1: Our collection for this lecture

- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
  - This is one year of Reuters newswire (part of 1995 and 1996)
- The collection isn't really large enough, but it's publicly available and is a plausible example.

# A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

## Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

# Reuters RCV1 statistics

| symbol | statistic | value |
|---|---|---|
| N | documents | 800,000 |
| L | avg. # tokens per doc | 200 |
| M | terms (= word types) | 400,000 |
| | avg. # bytes per token (incl. spaces/punct.) | 6 |
| | avg. # bytes per token (without spaces/punct.) | 4.5 |
| | avg. # bytes per term | 7.5 |
| | non-positional postings | 100,000,000 |

4.5 bytes per word token vs. 7.5 bytes per word type: why?

# Sort-based index construction

- As we build the index, we parse docs one at a time.
  - The final postings for any term are incomplete until the end.
- At 8 bytes per *(termID, docID)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
  - So … we can do this in memory today, but typical collections are much larger. E.g., the *New York Times* provides an index of >150 years of newswire
- Thus: We need to store intermediate results on disk.

# Scaling index construction

- In-memory index construction does not scale
  - Can't stuff entire collection into memory, sort, then write back
- How can we construct an index for very large collections?
- Taking into account hardware constraints. . .
  - Memory, disk, speed, etc.
- Let's review some hardware basics

# Hardware basics

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.

# Hardware basics

- Access to data in memory is ***much*** faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.

# Hardware assumptions (circa 2007)

| symbol | statistic | value |
|---|---|---|
| s | average seek time | 5 ms = 5 x $10^{-3}$ s |
| b | transfer time per byte | 0.02 µs = 2 x $10^{-8}$ s |
| | processor's clock rate | $10^{9}$ $s^{-1}$ |
| p | low-level operation (e.g., compare & swap a word) | 0.01 µs = $10^{-8}$ s |
| | size of main memory | several GB |
| | size of disk space | 1 TB or more |

# Sort using disk as "memory"?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting T = 100,000,000 records on disk is too slow – too many disk seeks.
- We need an *external* sorting algorithm.

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

External memory indexing

# BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

- 8-byte records *(termID, docID)*
- These are generated as we parse docs
- Must now sort 100M such 8-byte records by *termID*
- Define a Block ~ 10M such records
  - Can easily fit a couple into memory
  - Will have 10 such blocks to start with
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk
  - Then merge the blocks into one long sorted order

```
BSBIndexConstruction()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block)
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_merged)
```

# Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes O($N$ ln $N$) expected steps
  - In our case $N=10M$
- 10 times this estimate – gives us 10 sorted *runs* of 10M records each.
- Done straightforwardly, need 2 copies of data on disk
  - But can optimize this

# How to merge the sorted runs?

- Can do binary merges, with a merge tree of $\log_2 10 = 4$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.

# How to merge the sorted runs?

- But it is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously
  - Open all block files simultaneously and maintain a read buffer for each one and a write buffer for the output file
  - In each iteration, pick the lowest termID that hasn't been processed using a priority queue
  - Merge all postings lists for that termID and write it out

- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks

# Remaining problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.

# SPIMI: Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

# SPIMI-Invert

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
   if term(token) ∉ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
   if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
   ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

- Merging of blocks is analogous to BSBI.

# SPIMI in action

| Input token | Dictionary | Sorted dictionary |
|---|---|---|
| Caesar d1 | brutus d1 d3 | brutus d1 d3 |
| with d1 | with d1 d2 d3 d5 | caesar d1 d2 d4 |
| Brutus d1 | noble d5 | noble d5 |
| Caesar d2 | caesar d1 d2 d4 | with d1 d2 d3 d5 |
| with d2 | | |
| Brutus d3 | | |
| with d3 | | |
| Caesar d4 | | |
| noble d5 | | |
| with d5 | | |

# SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
- More on this later …

Original publication on SPIMI: Heinz and Zobel (2003)

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Distributed indexing

# Distributed indexing

- For web-scale indexing (don't try this at home!):
  must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

# Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)

# Massive data centers

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
- Answer: 37% - meaning, 63% of the time one or more servers is down.
- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.

# Distributed indexing

- Maintain a *master* machine directing the indexing job – considered "safe".
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

# Parallel tasks

- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

# Data flow

assign
Master
assign
Postings
Parser
a-f
g-p
q-z
Inverter
a-f
Parser
a-f
g-p
q-z
Inverter
g-p
splits
Parser
a-f
g-p
q-z
Inverter
q-z
Map phase
Segment files
Reduce phase

# Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into *j* partitions
- Example: Each partition is for a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***) – here *j* = 3.
- Now to complete the index inversion

# Inverters

- An inverter collects all (term,doc) pairs (= postings) for <u>one</u> term-partition.
- Sorts and writes to postings lists

# Example for index construction

Map:

Caesar

conquered

d1 : C came, C c’ ed.

d2 : C died.

→

<C,d1>, <came,d1>, <C,d1>, <c’ ed, d1>, <C, d2>, <died,d2>

Reduce:

(<C,(d1,d1,d2)>, <died,(d2)>, <came,(d1)>, <c’ ed,(d1)>)

→

(<C,(d1:2,d2:1)><died,(d2:1)>, <came,(d1:1)>,<c’ ed,(d1:1)>)

# Index construction

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
  - *Term-partitioned:* one machine handles a subrange of terms
  - *Document-partitioned:* one machine handles a subrange of documents
- As we'll discuss in the web part of the course, most search engines use a document-partitioned index … better load balancing, etc.

# MapReduce

- The index construction algorithm we just described is an instance of *MapReduce*.
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing …
- … without having to write code for the distribution part.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

# Schema for index construction in MapReduce

- **Schema of map and reduce functions**
- map: input → list(k, v)  reduce: (k,list(v)) → output
- **Instantiation of the schema for index construction**
- map: collection → list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Dynamic indexing

# Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

# Simplest approach

- Maintain "big" main index
- New docs go into "small" auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

# Issues with main and auxiliary indexes

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for OS.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

# Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one
  - At any time, some of these powers of 2 are instantiated
- Keep smallest ($Z_0$) in memory
- Larger ones ($I_0$, $I_1$, …) on disk
- If $Z_0$ gets too big (> *n*), write to disk as $I_0$
- or merge with $I_0$ (if $I_0$ already exists) as $Z_1$
- Either write merge $Z_1$ to disk as $I_1$ (if no $I_1$)
- Or merge with $I_1$ to form $Z_2$

# Logarithmic merge in action

```
LMergeAddToken(indexes, Z0, token)
  Z0 ← Merge(Z0, {token})
  if |Z0| = n
    then for i ← 0 to ∞
           do if Ii ∈ indexes
                then Zi+1 ← Merge(Ii, Zi)
                       (Zi+1 is a temporary index on disk.)
                     indexes ← indexes − {Ii}
                else Ii ← Zi     (Zi becomes the permanent index Ii.)
                     indexes ← indexes ∪ {Ii}
                     Break
           Z0 ← ∅
```

```
LogarithmicMerge()
  Z0 ← ∅     (Z0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMergeAddToken(indexes, Z0, getNextToken())
```

# Logarithmic merge

- Auxiliary and main index:
  - T/n merges where T is # of postings and n is size of auxiliary
  - Index construction time is $O(T^2/n)$ as in the worst case a posting is touched T/n times
- Logarithmic merge: Each posting is merged at most O(log (T/n)) times, so complexity is O(T log (T/n))
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of O(log (T/n)) indexes
  - Whereas it is O(1) if you just have a main and auxiliary index

# Further issues with multiple indexes

- Collection-wide statistics are hard to maintain
- E.g., when we speak of spell-correction: which of several corrected alternatives do we present to the user?
  - We may want to pick the one with the most hits
  - How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking

# Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is deleted

# Earlybird: Real-time search at Twitter

- Requirements for real-time search
  - Low latency, high throughput query evaluation
  - High ingestion rate and immediate data availability
  - Concurrent reads and writes of the index
  - Dominance of temporal signal

# Earlybird: Index organization

- Earlybird consists of multiple index segments
  - Each segment is relatively small, holding up to $2^{23}$ tweets
  - Each posting in a segment is a 32 bit word: 24 bits for the tweet id and 8 bits for the position in the tweet
- Only one segment can be written to at any given time
  - Small enough to be in memory
  - New postings are simply appended to the postings list
  - But the postings list is traversed backwards to prioritize newer tweets
- The remaining segments are optimized for read-only
  - Postings sorted in reverse chronological order (newest first)

# Other sorts of indexes

- Positional indexes
  - Same sort of sorting problem … just larger ← Why?
- Building character n-gram indexes:
  - As text is parsed, enumerate *n*-grams.
  - For each *n*-gram, need pointers to all dictionary terms containing it – the "postings"

# Resources for today's lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)
- Earlybird: Busch et al, ICDE 2012

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Christopher Manning and Pandu Nayak

Lecture 4: Index Compression

# Last lecture – index construction

- Sort-based indexing
  - Naïve in-memory inversion
  - Blocked Sort-Based Indexing (BSBI)
    - Merge sort is effective for hard disk–based sorting (avoid seeks!)
- Single-Pass In-Memory Indexing (SPIMI)
  - No global dictionary
    - Generate separate dictionary for each block
  - Don't sort postings
    - Accumulate postings in postings lists as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

# Today

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

- Collection statistics in more detail (with RCV1)
  - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression

# Why compression (in general)?

- Use less disk space
  - Save a little money; give users more space
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use

# Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

# Recall Reuters RCV1

| symbol | statistic | value |
|---|---|---|
| N | documents | 800,000 |
| L | avg. # tokens per doc | 200 |
| M | terms (= word types) | ~400,000 |
| | avg. # bytes per token (incl. spaces/punct.) | 6 |
| | avg. # bytes per token (without spaces/punct.) | 4.5 |
| | avg. # bytes per term | 7.5 |
| | non-positional postings | 100,000,000 |

# Index parameters vs. what we index

(details *IIR* Table 5.1, p.80)

| size of | word types (terms) | | | non-positional postings | | | positional postings | | |
|---|---|---|---|---|---|---|---|---|---|
| | dictionary | | | non-positional index | | | positional index | | |
| | Size (K) | Δ% | cumul % | Size (K) | Δ % | cumul % | Size (K) | Δ % | cumul % |
| Unfiltered | 484 | | | 109,971 | | | 197,879 | | |
| No numbers | 474 | -2 | -2 | 100,680 | -8 | -8 | 179,158 | -9 | -9 |
| Case folding | 392 | -17 | -19 | 96,969 | -3 | -12 | 179,158 | 0 | -9 |
| 30 stopwords | 391 | -0 | -19 | 83,390 | -14 | -24 | 121,858 | -31 | -38 |
| 150 stopwords | 391 | -0 | -19 | 67,002 | -30 | -39 | 94,517 | -47 | -52 |
| stemming | 322 | -17 | -33 | 63,812 | -4 | -42 | 94,517 | 0 | -52 |

Exercise: give intuitions for all the '0' entries. Why do some zero entries correspond to big deltas in other columns?

# Lossless vs. lossy compression

- Lossless compression: All information is preserved.
  - What we mostly do in IR.
- Lossy compression: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- Chapter 7: Prune postings entries that are unlikely to turn up in the top *k* list for any query.
  - Almost no loss of quality in top *k* list.

# Vocabulary size vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode ☺

# Vocabulary size vs. collection size

- Heaps' law: $M = kT^b$
- $M$ is the size of the vocabulary, $T$ is the number of tokens in the collection
- Typical values: $30 \leq k \leq 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size $M$ vs. $T$, Heaps' law predicts a line with slope about ½
  - It is the simplest possible (linear) relationship between the two in log-log space
    - $\log M = \log k + b \log T$
  - An empirical finding ("empirical law")

# Heaps' Law

Fig 5.1 p81

For RCV1, the dashed line

$\log_{10} M = 0.49 \log_{10} T + 1.64$

is the best least squares fit.

Thus, $M = 10^{1.64} T^{0.49}$ so $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Good empirical fit for Reuters RCV1 !

For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms

# Exercises

- What is the effect of including spelling errors, vs. automatically correcting spelling errors on Heaps' law?
- Compute the vocabulary size *M* for this scenario:
  - Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
  - Assume a search engine indexes a total of 20,000,000,000 ($2 \times 10^{10}$) pages, containing 200 tokens on average
  - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

# Zipf's law

- Heaps' law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The $i^{th}$ most frequent term has frequency proportional to $1/i$.
- $cf_i \propto 1/i = K/i$ where $K$ is a normalizing constant
- $cf_i$ is <u>collection frequency</u>: the number of occurrences of the term $t_i$ in the collection.

# Zipf consequences

- If the most frequent term (*the*) occurs $cf_1$ times
  - then the second most frequent term (*of*) occurs $cf_1/2$ times
  - the third most frequent term (*and*) occurs $cf_1/3$ times …
- Equivalent: $cf_i = K/i$ where $K$ is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$

- Another power law relationship

# Zipf's law for Reuters RCV1

# Compression

- Now, we will consider compressing the space for the dictionary and postings. We'll do:
  - Basic Boolean index only
  - No study of positional indexes, etc.
- But these ideas can be extended

- We will consider compression schemes

# DICTIONARY COMPRESSION

# Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

# Dictionary storage – naïve version

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.

# Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons.*
- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

# Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

| Freq. | Postings ptr. | Term ptr. |
|---|---|---|
| 33 | | |
| 29 | | |
| 44 | | |
| 126 | | |

Total string length = 400K x 8B = 3.2MB

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

# Space for dictionary as a string

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string

Now avg. 11 bytes/term, not 20.

- 400K terms x 19 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

# Blocking

- Store pointers to every *k*th term string.
  - Example below: *k*=4.
- Need to store term lengths (1 extra byte)

….7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo*….

| Freq. | Postings ptr. | Term ptr. |
|---|---|---|
| 33 | | |
| 29 | | |
| 44 | | |
| 126 | | |
| 7 | | |

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

# Blocking Net Gains

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes,

now we use 3 + 4 = 7 bytes.

Shaved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
We can save more with larger *k*.

Question: Why not go with larger *k*?

# Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = (1+2·2+4·3+4)/8 ~2.6

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?

# Dictionary search with blocking

- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = $(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3$ compares

# Exercises

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 4, 8$ and $16$.

- Estimate the impact on search performance (and slowdown compared to $k$=1) with blocking, for block sizes of $k = 4, 8$ and $16$.

# Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last *k-1* in a block of *k*)

8*automata*8*automate*9*automatic*10*automation*

→8*automat***a*1◊*e*2◊*ic*3◊*ion*

Encodes prefix ***automat***

Extra length beyond ***automat.***

Begins to resemble general string compression.

# RCV1 dictionary compression summary

| Technique | Size in MB |
|---|---|
| Fixed width | 11.2 |
| Dictionary-as-String with pointers to every term | 7.6 |
| + blocking, $k = 4$ | 7.1 |
| + blocking + front coding | 5.9 |

# POSTINGS COMPRESSION

# Postings compression

- The postings file is much larger than the dictionary, factor of at least 10, often over 100 times larger
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

# Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ≈ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting ≈ 2MB is too expensive.
  - Prefer 0/1 bitmap vector in this case (≈100K)

# Gap encoding of postings file entries

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 …
- Consequence: it suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.
  - Especially for common words

# Three postings entries

| | encoding | postings list | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| THE | docIDs | ... | | 283042 | | 283043 | | 283044 | | 283045 | ... |
| | gaps | | | | 1 | | 1 | | 1 | | ... |
| COMPUTER | docIDs | ... | | 283047 | | 283154 | | 283159 | | 283202 | ... |
| | gaps | | | | 107 | | 5 | | 43 | | ... |
| ARACHNOCENTRIC | docIDs | 252000 | | 500100 | | | | | | | |
| | gaps | 252000 | 248100 | | | | | | | | |

# Variable length encoding

- Aim:
  - For ***arachnocentric***, we will use ~20 bits/gap entry.
  - For ***the***, we will use ~1 bit/gap entry.
- If the average gap for a term is *G*, we want to use ~$\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers

# Unary code

- Represent $n$ as $n$ 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is

11111111111111111111111111111111111111110 .

- Unary code for 80 is:

11111111111111111111111111111111111111111111
1111111111111111111111111111111110

- This doesn't look promising, but….
  - Optimal if $P(n) = 2^{-n}$
  - We can use it as part of our solution

# Gamma codes

- We can compress better with bit-level codes
  - The Gamma code is the best known of these.
- Represent a gap *G* as a pair *length* and *offset*
- *offset* is *G* in binary, with the leading bit cut off
  - For example 13 → 1101 → 101
- *length* is the length of offset
  - For 13 (offset 101), this is 3.
- We encode *length* with *unary code*: 1110.
- Gamma code of 13 is the concatenation of *length* and *offset*: 1110101

# Gamma code examples

| number | length | offset | γ-code |
|---|---|---|---|
| 0 | | | none |
| 1 | 0 | | 0 |
| 2 | 10 | 0 | 10,0 |
| 3 | 10 | 1 | 10,1 |
| 4 | 110 | 00 | 110,00 |
| 9 | 1110 | 001 | 1110,001 |
| 13 | 1110 | 101 | 1110,101 |
| 24 | 11110 | 1000 | 11110,1000 |
| 511 | 111111110 | 11111111 | 111111110,11111111 |
| 1025 | 11111111110 | 0000000001 | 11111111110,0000000001 |

# Reminder: bitwise operations

- For compression, you need to use bitwise operators

- Python (and most everything else):
  - & bitwise and; | bitwise or; ^ bitwise xor; ~ ones complement
  - << left shift bits, >> right shift; LACKS >>> zero fill right shift
  - Recipes:
    - Extract 7 bits: a & 0x7f00 >> 8 ; if take high-order bit add: & 0x7f
    - Combine 3 5-bit numbers: a | (b << 5) | (c << 10)
    - Lookup tables rather than decoding can be faster, yet still small

# Gamma code properties

- $G$ is encoded using $2\lfloor \log G \rfloor + 1$ bits
  - Length of offset is $\lfloor \log G \rfloor$ bits
  - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$

- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
  - Optimal for $P(n) \approx 1/(2n^2)$
- Gamma code is parameter-free

# Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be too slow
- All modern practice is to use byte or word aligned codes
  - Variable byte encoding is a faster, conceptually simpler compression scheme, with decent compression

# Variable Byte (VB) codes

- For a gap value *G*, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store *G* and dedicate 1 bit in it to be a continuation bit *c*
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.

# Example

| docIDs | 824 | 829 | 215406 |
|---|---|---|---|
| gaps | | 5 | 214577 |
| VB code | 00000110<br>10111000 | 10000101 | 00001101<br>00001100<br>10110001 |

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

# RCV1 compression

| Data structure | Size in MB |
|---|---|
| dictionary, fixed-width | 11.2 |
| dictionary, term pointers into string | 7.6 |
| with blocking, k = 4 | 7.1 |
| with blocking & front coding | 5.9 |
| collection (text, xml markup etc) | 3,600.0 |
| collection (text) | 960.0 |
| Term-doc incidence matrix | 40,000.0 |
| postings, uncompressed (32-bit words) | 400.0 |
| postings, uncompressed (20 bits) | 250.0 |
| postings, variable byte encoded | 116.0 |
| postings, $\gamma$–encoded | 101.0 |

# Other variable unit codes

- Variable byte codes are used by many real systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches
- Byte alignment wastes space if you have many small gaps – as gap encoding often makes
- More modern work mainly uses the ideas:
  - Be word aligned (32 or 64 bits; even faster)
  - Encode several gaps at the same time
  - Often assume a maximum gap size, perhaps with an escape

# Group Variable Integer code

- Used by Google around turn of millennium….
  - Jeff Dean, keynote at WSDM 2009 and presentations at CS276
- Encodes 4 integers in blocks of size 5–17 bytes
- First byte: four 2-bit binary length fields
- | $L_1$ | $L_2$ | $L_3$ | $L_4$ | , $L_j \in \{1,2,3,4\}$
- Then, L1+L2+L3+L4 bytes (between 4–16) hold 4 numbers
  - Each number can use 8/16/24/32 bits. Max gap length ~4 billion
- It was suggested that this was about twice as fast as VB encoding
  - Decoding gaps is much simpler – no bit masking
  - First byte can be decoded with lookup table or switch

# Simple-9 [Anh & Moffat, 2004]

A word-aligned, multiple number encoding scheme

How can we store several numbers in 32 bits with a format selector?

# Simple9 Encoding Scheme [Anh & Moffat, 2004]

- Encoding block: 4 bytes (32 bits)
- Most significant nibble (4 bits) describe the layout of the 28 other bits as follows:

| Layout (4 bits) | n numbers of b bits each<br>$n * b \leq 28$ |
|---|---|

  - 0: a single 28-bit number
  - 1: two 14-bit numbers
  - 2: three 9-bit numbers (and one spare bit)
  - 3: four 7-bit numbers
  - 4: five 5-bit numbers (and three spare bits)
  - 5: seven 4-bit numbers
  - 6: nine 3-bit numbers (and one spare bit)
  - 7: fourteen two-bit numbers
  - 8: twenty-eight one-bit numbers
- Simple16 is a variant with 5 additional (uneven) configurations
- Efficiently decoded with hand-coded decoder, using bit masks
- Extended Simple Family – idea applies to 64-bit words, etc.

# Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection

- We've ignored positional information
- Hence, space savings are less for indexes used in practice
  - But techniques substantially the same

# Resources for today's lecture

- *IIR* 5
- *MG* 3.3, 3.4.
- F. Scholer, H.E. Williams and J. Zobel. 2002. Compression of Inverted Indexes For Fast Query Evaluation. *Proc. ACM-SIGIR 2002*.
  - Variable byte codes
- V. N. Anh and A. Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval* 8: 151–166.
  - Word aligned codes

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$h, llo$he, lo$hel, o$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 <u>*second*</u> inverted index <u>*from bigrams to dictionary terms*</u> 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!

Search

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.

# 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
  - 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*

# Terminology

- We just discussed *character bigrams and k-grams*:
  - *st, pr, an …*
- We can also have *word bigrams and n-grams*:
  - *palo alto, flying from, road repairs*

The Noisy Channel Model of Spelling

# INDEPENDENT WORD SPELLING CORRECTION

# Noisy Channel Intuition

# Noisy Channel = Bayes' Rule

- We see an observation *x* of a misspelled word
- Find the correct word $\hat{w}$

$$\hat{w} = \underset{w \square V}{\operatorname{argmax}} P(w \mid x)$$

$$= \underset{w \square V}{\operatorname{argmax}} \frac{P(x \mid w) P(w)}{P(x)}$$

Bayes

$$= \underset{w \square V}{\operatorname{argmax}} P(x \mid w) P(w)$$

Prior

Noisy channel model

# History: Noisy channel for spelling proposed around 1990

- **IBM**
  - Mays, Eric, Fred J. Damerau and Robert L. Mercer. 1991. Context based spelling correction. *Information Processing and Management*, 23(5), 517–522
- **AT&T Bell Labs**
  - Kernighan, Mark D., Kenneth W. Church, and William A. Gale. 1990. A spelling correction program based on a noisy channel model. Proceedings of COLING 1990, 205-210

# Non-word spelling error example

acress

# Candidate generation

- Words with similar spelling
  - Small *edit distance* to error
- Words with similar pronunciation
  - Small distance of pronunciation to error

# Candidate Testing: Damerau-Levenshtein edit distance

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

- See *IIR* sec 3.3.3 for edit distance

# 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

# Let's say we've generated candidates: Now back to Bayes' Rule

- We see an observation *x* of a misspelled word
- Find the correct word $\hat{w}$

$$\hat{w} = \underset{w \square V}{\operatorname{argmax}} P(w \mid x)$$

$$= \underset{w \square V}{\operatorname{argmax}} \frac{P(x \mid w) P(w)}{P(x)}$$

$$= \underset{w \square V}{\operatorname{argmax}} P(x \mid w) P(w)$$

What's *P(w)*?

# Language Model

- Take a big supply of words (your document collection with *T* tokens); let *C(w)* = # occurrences of *w*

$$P(w) = \frac{C(w)}{T}$$

- In other applications – you can take the supply to be typed queries (suitably filtered) – when a static dictionary is inadequate

# Unigram Prior probability

Counts from 404,253,213 words in Corpus of Contemporary English (COCA)

| word | Frequency of word | *P(w)* |
|---|---|---|
| actress | 9,321 | .0000230573 |
| cress | 220 | .0000005442 |
| caress | 686 | .0000016969 |
| access | 37,038 | .0000916207 |
| across | 120,844 | .0002989314 |
| acres | 12,874 | .0000318463 |

# Channel model probability

- **Error model probability, Edit probability**
- *Kernighan, Church, Gale 1990*

- *Misspelled word* $x = x_1, x_2, x_3 \ldots x_m$
- *Correct word* $w = w_1, w_2, w_3, \ldots, w_n$

- $P(x|w)$ = probability of the edit
  - (deletion/insertion/substitution/transposition)

# Computing error probability: confusion "matrix"

```
del[x,y]:    count(xy typed as x)
ins[x,y]:    count(x typed as xy)
sub[x,y]:    count(y typed as x)
trans[x,y]:  count(xy typed as yx)
```

Insertion and deletion conditioned on previous character

# Confusion matrix for substitution

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

| X | Y (correct) | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| a | 0 | 0 | 7 | 1 | 342 | 0 | 0 | 2 | 118 | 0 | 1 | 0 | 0 | 3 | 76 | 0 | 0 | 1 | 35 | 9 | 9 | 0 | 1 | 0 | 5 | 0 |
| b | 0 | 0 | 9 | 9 | 2 | 2 | 3 | 1 | 0 | 0 | 0 | 5 | 11 | 5 | 0 | 10 | 0 | 0 | 2 | 1 | 0 | 0 | 8 | 0 | 0 | 0 |
| c | 6 | 5 | 0 | 16 | 0 | 9 | 5 | 0 | 0 | 0 | 1 | 0 | 7 | 9 | 1 | 10 | 2 | 5 | 39 | 40 | 1 | 3 | 7 | 1 | 1 | 0 |
| d | 1 | 10 | 13 | 0 | 12 | 0 | 5 | 5 | 0 | 0 | 2 | 3 | 7 | 3 | 0 | 1 | 0 | 43 | 30 | 22 | 0 | 0 | 4 | 0 | 2 | 0 |
| e | 388 | 0 | 3 | 11 | 0 | 2 | 2 | 0 | 89 | 0 | 0 | 3 | 0 | 5 | 93 | 0 | 0 | 14 | 12 | 6 | 15 | 0 | 1 | 0 | 18 | 0 |
| f | 0 | 15 | 0 | 3 | 1 | 0 | 5 | 2 | 0 | 0 | 0 | 3 | 4 | 1 | 0 | 0 | 0 | 6 | 4 | 12 | 0 | 0 | 2 | 0 | 0 | 0 |
| g | 4 | 1 | 11 | 11 | 9 | 2 | 0 | 0 | 0 | 1 | 1 | 3 | 0 | 0 | 2 | 1 | 3 | 5 | 13 | 21 | 0 | 0 | 1 | 0 | 3 | 0 |
| h | 1 | 8 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 12 | 14 | 2 | 3 | 0 | 3 | 1 | 11 | 0 | 0 | 2 | 0 | 0 | 0 |
| i | 103 | 0 | 0 | 0 | 146 | 0 | 1 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 49 | 0 | 0 | 0 | 2 | 1 | 47 | 0 | 2 | 1 | 15 | 0 |
| j | 0 | 1 | 1 | 9 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| k | 1 | 2 | 8 | 4 | 1 | 1 | 2 | 5 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 4 | 0 | 0 | 3 |
| l | 2 | 10 | 1 | 4 | 0 | 4 | 5 | 6 | 13 | 0 | 1 | 0 | 0 | 14 | 2 | 5 | 0 | 11 | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| m | 1 | 3 | 7 | 8 | 0 | 2 | 0 | 6 | 0 | 0 | 4 | 4 | 0 | 180 | 0 | 6 | 0 | 0 | 9 | 15 | 13 | 3 | 2 | 2 | 3 | 0 |
| n | 2 | 7 | 6 | 5 | 3 | 0 | 1 | 19 | 1 | 0 | 4 | 35 | 78 | 0 | 0 | 7 | 0 | 28 | 5 | 7 | 0 | 0 | 1 | 2 | 0 | 2 |
| o | 91 | 1 | 1 | 3 | 116 | 0 | 0 | 0 | 25 | 0 | 2 | 0 | 0 | 0 | 0 | 14 | 0 | 2 | 4 | 14 | 39 | 0 | 0 | 0 | 18 | 0 |
| p | 0 | 11 | 1 | 2 | 0 | 6 | 5 | 0 | 2 | 9 | 0 | 2 | 7 | 6 | 15 | 0 | 0 | 1 | 3 | 6 | 0 | 4 | 1 | 0 | 0 | 0 |
| q | 0 | 0 | 1 | 0 | 0 | 0 | 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 14 | 0 | 30 | 12 | 2 | 2 | 8 | 2 | 0 | 5 | 8 | 4 | 20 | 1 | 14 | 0 | 0 | 12 | 22 | 4 | 0 | 0 | 1 | 0 | 0 |
| s | 11 | 8 | 27 | 33 | 35 | 4 | 0 | 1 | 0 | 1 | 0 | 27 | 0 | 6 | 1 | 7 | 0 | 14 | 0 | 15 | 0 | 0 | 5 | 3 | 20 | 1 |
| t | 3 | 4 | 9 | 42 | 7 | 5 | 19 | 5 | 0 | 1 | 0 | 14 | 9 | 5 | 5 | 6 | 0 | 11 | 37 | 0 | 0 | 2 | 19 | 0 | 7 | 6 |
| u | 20 | 0 | 0 | 0 | 44 | 0 | 0 | 0 | 64 | 0 | 0 | 0 | 0 | 2 | 43 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 8 | 0 |
| v | 0 | 0 | 7 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 8 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| w | 2 | 2 | 1 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 6 | 3 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| x | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| y | 0 | 0 | 2 | 0 | 15 | 0 | 1 | 7 | 15 | 0 | 0 | 0 | 2 | 0 | 6 | 1 | 0 | 7 | 36 | 8 | 5 | 0 | 0 | 1 | 0 | 0 |
| z | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 5 | 0 | 0 | 0 | 0 | 2 | 21 | 3 | 0 | 0 | 0 | 0 | 3 | 0 |

# Nearby keys

# Generating the confusion matrix

- Peter Norvig's list of errors
- Peter Norvig's list of counts of single-edit errors
  - All Peter Norvig's ngrams data links: http://norvig.com/ngrams/

# Channel model

Kernighan, Church, Gale 1990

$$P(x|w) = \begin{cases} \dfrac{\text{del}[w_{i-1}, w_i]}{\text{count}[w_{i-1} w_i]}, & \text{if deletion} \\ \dfrac{\text{ins}[w_{i-1}, x_i]}{\text{count}[w_{i-1}]}, & \text{if insertion} \\ \dfrac{\text{sub}[x_i, w_i]}{\text{count}[w_i]}, & \text{if substitution} \\ \dfrac{\text{trans}[w_i, w_{i+1}]}{\text{count}[w_i w_{i+1}]}, & \text{if transposition} \end{cases}$$

# Smoothing probabilities: Add-1 smoothing

- But if we use the confusion matrix example, unseen errors are impossible!
- They'll make the overall probability 0. That seems too harsh
  - e.g., in Kernighan's chart q➔a and a➔q are both 0, even though they're adjacent on the keyboard!
- A simple solution is to add 1 to all counts and then if there is a |A| character alphabet, to normalize appropriately:

$$\text{If substitution, } P(x \mid w) = \frac{\text{sub}[x,w]+1}{\text{count}[w]+A}$$

# Channel model for acress

| Candidate Correction | Correct Letter | Error Letter | *x/w* | *P(x/w)* |
|---|---|---|---|---|
| actress | t | - | c\|ct | .000117 |
| cress | - | a | a\|# | .00000144 |
| caress | ca | ac | ac\|ca | .00000164 |
| access | c | r | r\|c | .000000209 |
| across | o | e | e\|o | .0000093 |
| acres | - | s | es\|e | .0000321 |
| acres | - | s | ss\|s | .0000342 |

| Candidate Correction | Correct Letter | Error Letter | *x/w* | *P(x/w)* | *P(w)* | $10^9$ * *P(x/w)** *P(w)* |
|---|---|---|---|---|---|---|
| actress | t | - | c\|ct | .000117 | .0000231 | 2.7 |
| cress | - | a | a\|# | .00000144 | .000000544 | .00078 |
| caress | ca | ac | ac\|ca | .00000164 | .00000170 | .0028 |
| access | c | r | r\|c | .000000209 | .0000916 | .019 |
| across | o | e | e\|o | .0000093 | .000299 | 2.8 |
| acres | - | s | es\|e | .0000321 | .0000318 | 1.0 |
| acres | - | s | ss\|s | .0000342 | .0000318 | 1.0 |

| Candidate Correction | Correct Letter | Error Letter | *x/w* | *P(x/w)* | *P(w)* | $10^9 * P(x/w)P(w)$ |
|---|---|---|---|---|---|---|
| actress | t | - | c\|ct | .000117 | .0000231 | 2.7 |
| cress | - | a | a\|# | .00000144 | .000000544 | .00078 |
| caress | ca | ac | ac\|ca | .00000164 | .00000170 | .0028 |
| access | c | r | r\|c | .000000209 | .0000916 | .019 |
| **across** | **o** | **e** | **e\|o** | **.0000093** | **.000299** | **2.8** |
| acres | - | s | es\|e | .0000321 | .0000318 | 1.0 |
| acres | - | s | ss\| | .0000342 | .0000318 | 1.0 |

# Evaluation

- Some spelling error test sets
  - Wikipedia's list of common English misspelling
  - Aspell filtered version of that list
  - Birkbeck spelling error corpus
  - Peter Norvig's list of errors (includes Wikipedia and Birkbeck, for training or testing)

Context-Sensitive Spelling Correction

# SPELLING CORRECTION WITH THE NOISY CHANNEL

# Real-word spelling errors

- …leaving in about fifteen ***minuets*** to go to her house.
- The design ***an*** construction of the system…
- Can they ***lave*** him my messages?
- The study was conducted mainly ***be*** John Black.

- 25-40% of spelling errors are real words Kukich 1992

# Context-sensitive spelling error fixing

- For each word in sentence (phrase, query …)
  - Generate *candidate set*
    - the word itself
    - all single-letter edits that are English words
    - words that are homophones
    - (all of this can be pre-computed!)
- Choose best candidates
    - Noisy channel model

# Noisy channel for real-word spell correction

- Given a sentence $x_1, x_2, x_3, \ldots, x_n$
- Generate a set of candidates for each word $x_i$
  - Candidate($x_1$) = {$x_1$, $w_1$ , $w'_1$ , $w''_1$ ,…}
  - Candidate($x_2$) = {$x_2$, $w_2$ , $w'_2$ , $w''_2$ ,…}
  - Candidate($x_n$) = {$x_n$, $w_n$ , $w'_n$ , $w''_n$ ,…}
- Choose the sequence W that maximizes P(W | $x_1, \ldots, x_n$)

$$\hat{w} = \underset{w \square V}{\operatorname{argmax}} P(w \mid x)$$

$$= \underset{w \square V}{\operatorname{argmax}} P(x \mid w) P(w)$$

# Incorporating context words: Context-sensitive spelling correction

- Determining whether **actress** or **across** is appropriate will require looking at the context of use
- We can do this with a better **language model**
  - You learned/can learn a lot about language models in CS124 or CS224N
  - Here we present just enough to be dangerous/do the assignment
- A **bigram language model** conditions the probability of a word on (just) the previous word

$$P(w_1 \ldots w_n) = P(w_1)P(w_2 \mid w_1)\ldots P(w_n \mid w_{n-1})$$

# Incorporating context words

- For unigram counts, $P(w)$ is always non-zero
  - if our dictionary is derived from the document collection
- This won't be true of $P(w_k | w_{k-1})$. We need to **smooth**
- We could use add-1 smoothing on this conditional distribution
- But here's a better way – interpolate a unigram and a bigram:

$$P_{li}(w_k | w_{k-1}) = \lambda P_{uni}(w_k) + (1-\lambda) P_{bi}(w_k | w_{k-1})$$

  - $P_{bi}(w_k | w_{k-1}) = C(w_{k-1}, w_k) / C(w_{k-1})$

# All the important fine points

- Note that we have several probability distributions for words
  - Keep them straight!
- You might want/need to work with log probabilities:
  - $\log P(w_1 \ldots w_n) = \log P(w_1) + \log P(w_2 | w_1) + \ldots + \log P(w_n | w_{n-1})$
  - Otherwise, be very careful about floating point underflow
- Our query may be words anywhere in a document
  - We'll start the bigram estimate of a sequence with a unigram estimate
  - Often, people instead condition on a start-of-sequence symbol, but not good here
  - Because of this, the unigram and bigram counts have different totals – not a problem

# Using a bigram language model

- "a stellar and versatile **acress** whose combination of sass and glamour…"
- Counts from the Corpus of Contemporary American English with add-1 smoothing
- P(actress|versatile)=.000021
- P(across|versatile) =.000021

P(whose|actress) = .0010
P(whose|across) = .000006

- P("versatile actress whose") = .000021*.0010 = 210 x$10^{-10}$
- P("versatile across whose") = .000021*.000006 = 1 x$10^{-10}$

# Using a bigram language model

- "a stellar and versatile **acress** whose combination of sass and glamour…"
- Counts from the Corpus of Contemporary American English with add-1 smoothing
- P(actress|versatile)=.000021 P(whose|actress) = .0010
- P(across|versatile) =.000021 P(whose|across) = .000006

- **P("versatile actress whose") = .000021*.0010 = 210 x10$^{-10}$**
- P("versatile across whose") = .000021*.000006 = 1 x10$^{-10}$

# Noisy channel for real-word spell correction

# Noisy channel for real-word spell correction

# Simplification: One error per sentence

- Out of all possible sentences with one word replaced
  - $w_1$, $\mathbf{w''_2}$,$w_3$,$w_4$ two **off** thew
  - $w_1$,$w_2$,$\mathbf{w'_3}$,$w_4$ two of **the**
  - $\mathbf{w'''_1}$,$w_2$,$w_3$,$w_4$ **too** of thew
  - …
- Choose the sequence W that maximizes P(W)

# Where to get the probabilities

- Language model
  - Unigram
  - Bigram
  - etc.
- Channel model
  - Same as for non-word spelling correction
  - Plus need probability for no error, $P(w/w)$

# Probability of no error

- What is the channel probability for a correctly typed word?
- P("the"|"the")
  - If you have a big corpus, you can estimate this percent correct
- But this value depends strongly on the application
  - .90 (1 error in 10 words)
  - .95 (1 error in 20 words)
  - .99 (1 error in 100 words)

# Peter Norvig's "thew" example

| x | w | x\|w | P(x\|w) | P(w) | $10^9$ P(x\|w)P(w) |
|---|---|---|---|---|---|
| thew | the | ew\|e | 0.000007 | 0.02 | 144 |
| thew | thew | | 0.95 | 0.00000009 | 90 |
| thew | thaw | e\|a | 0.001 | 0.0000007 | 0.7 |
| thew | threw | h\|hr | 0.000008 | 0.000004 | 0.03 |
| thew | thwe | ew\|we | 0.000003 | 0.00000004 | 0.0001 |

# State of the art noisy channel

- We never just multiply the prior and the error model
- Independence assumptions→probabilities not commensurate
- Instead: Weight them

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(x \mid w) P(w)^{\lambda}$$

- Learn λ from a development test set

# 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

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Pandu Nayak and Prabhakar Raghavan

Lecture 6: Scoring, Term Weighting and the Vector Space Model

# This lecture; IIR Sections 6.2-6.4.3

- Ranked retrieval
- Scoring documents
- Term frequency
- Collection statistics
- Weighting schemes
- Vector space scoring

# Ranked retrieval

- Thus far, our queries have all been Boolean.
  - Documents either match or don't.
- Good for expert users with precise understanding of their needs and the collection.
  - Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
  - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
  - Most users don't want to wade through 1000s of results.
    - This is particularly true of web search.

# Problem with Boolean search: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: "*standard user dlink 650*" → 200,000 hits
- Query 2: "*standard user dlink 650 no card found*": 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
  - AND gives too few; OR gives too many

# Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in ranked retrieval, the system returns an ordering over the (top) documents in the collection for a query
- Free text queries: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language
- In principle, there are two separate choices here, but in practice, ranked retrieval has normally been associated with free text queries and vice versa

# Feast or famine: not a problem in ranked retrieval

- When a system produces a ranked result set, large result sets are not an issue
  - Indeed, the size of the result set is not an issue
  - We just show the top $k$ ( ≈ 10) results
  - We don't overwhelm the user

- Premise: the ranking algorithm works

# Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query "match".

# Take 1: Jaccard coefficient

- A common measure of overlap of two sets $A$ and $B$
- jaccard$(A,B) = |A \cap B| \ / \ |A \cup B|$
- jaccard$(A,A) = 1$
- jaccard$(A,B) = 0$ if $A \cap B = 0$
- $A$ and $B$ don't have to be the same size.
- Always assigns a number between 0 and 1.

# Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- Query: *ides of march*
- Document 1: *caesar died in march*
- Document 2: *the long march*

# Issues with Jaccard for scoring

- It doesn't consider *term frequency* (how many times a term occurs in a document)
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length

# Query-document matching scores

- We need a way of assigning a score to a query/document pair
- Let's start with a one-term query
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

# Recall (Lecture 2): Binary term-document incidence matrix

| | 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 |

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

# Term-document count matrices

- Consider the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^v$: a column below

| | Antony and Cleopatra | Julius Caesar | The Tempest | Hamlet | Othello | Macbeth |
|---|---|---|---|---|---|---|
| Antony | 157 | 73 | 0 | 0 | 0 | 0 |
| Brutus | 4 | 157 | 0 | 1 | 0 | 0 |
| Caesar | 232 | 227 | 0 | 2 | 1 | 1 |
| Calpurnia | 0 | 10 | 0 | 0 | 0 | 0 |
| Cleopatra | 57 | 0 | 0 | 0 | 0 | 0 |
| mercy | 2 | 0 | 3 | 5 | 5 | 1 |
| worser | 2 | 0 | 1 | 1 | 1 | 0 |

# *Bag of words* model

- Vector representation doesn't consider the ordering of words in a document
- *John is quicker than Mary* and *Mary is quicker than John* have the same vectors
- This is called the bag of words model.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.

# Term frequency tf

- The term frequency $tf_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
  - Note: Frequency means count in IR
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

# Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.
- Score for a document-query pair: sum over terms *t* in both *q* and *d*:
- score $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$

- The score is 0 if none of the query terms is present in the document.

# Rare terms are more informative

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

# Collection vs. Document frequency

- Collection frequency of *t* is the number of occurrences of *t* in the collection
- Document frequency of t is the number of documents in which t occurs
- Example:

| Word | Collection frequency | Document frequency |
|---|---|---|
| *insurance* | 10440 | 3997 |
| *try* | 10422 | 8760 |

- Which word is for better search (gets higher weight)

# idf weight

- $\mathrm{df}_t$ is the <u>document</u> frequency of $t$: the number of documents that contain $t$
  - $\mathrm{df}_t$ is an inverse measure of the informativeness of $t$
  - $\mathrm{df}_t \leq N$
- We define the idf (inverse document frequency) of $t$ by

$$\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$$

  - We use $\log(N/\mathrm{df}_t)$ instead of $N/\mathrm{df}_t$ to "dampen" the effect of idf.

# idf example, suppose $N$ = 1 million

| term | $df_t$ | $idf_t$ |
|---|---|---|
| calpurnia | 1 | 6 |
| animal | 100 | 4 |
| sunday | 1,000 | 3 |
| fly | 10,000 | 2 |
| under | 100,000 | 1 |
| the | 1,000,000 | 0 |

$$\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$$

There is one idf value for each term $t$ in a collection.

# Effect of idf on ranking

- Does idf have an effect on ranking for one-term queries, like
  - iPhone
- idf has no effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
- For the query capricious person, idf weighting makes occurrences of capricious count for much more in the final document ranking than occurrences of person.

# tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = \log(1 + \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

- Best known weighting scheme in information retrieval
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

# Score for a document given a query

$$\text{Score}(q,d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

- There are many variants
  - How "tf" is computed (with/without logs)
  - Whether the terms in the query are also weighted
  - …

# Binary → count → weight matrix

| | Antony and Cleopatra | Julius Caesar | The Tempest | Hamlet | Othello | Macbeth |
|---|---|---|---|---|---|---|
| Antony | 5.25 | 3.18 | 0 | 0 | 0 | 0.35 |
| Brutus | 1.21 | 6.1 | 0 | 1 | 0 | 0 |
| Caesar | 8.59 | 2.54 | 0 | 1.51 | 0.25 | 0 |
| Calpurnia | 0 | 1.54 | 0 | 0 | 0 | 0 |
| Cleopatra | 2.85 | 0 | 0 | 0 | 0 | 0 |
| mercy | 1.51 | 0 | 1.9 | 0.12 | 5.25 | 0.88 |
| worser | 1.37 | 0 | 0.11 | 4.15 | 0.25 | 1.95 |

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

# Documents as vectors

- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

# Queries as vectors

- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity ≈ inverse of distance

# Formalizing vector space proximity

- First cut: distance between two points
  - ( = distance between the end points of the two vectors)
- Euclidean distance?
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.

# Why distance is a bad idea

The Euclidean distance between $\vec{q}$ and $\vec{d_2}$ is large even though the distribution of terms in the query $\vec{q}$ and the distribution of terms in the document $\vec{d_2}$ are very similar.

# Use angle instead of distance

- Thought experiment: take a document *d* and append it to itself. Call this document *d′*.
- "Semantically" d and d′ have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.

- Key idea: Rank documents according to angle with query.

# From angles to cosines

- The following two notions are equivalent.
  - Rank documents in <u>decreasing</u> order of the angle between query and document
  - Rank documents in <u>increasing</u> order of cosine(query,document)
- Cosine is a monotonically decreasing function for the interval [$0^o$, $180^o$]

# From angles to cosines

- But how should we be computing cosines?

# Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the $L_2$ norm:

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

- Dividing a vector by its $L_2$ norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents d and d′ (d appended to itself) from earlier slide: they have identical vectors after length-normalization.
  - Long and short documents now have comparable weights

# cosine(query,document)

Dot product

Unit vectors

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

$q_i$ is the weight of term $i$ in the query
$d_i$ is the weight of term $i$ in the document

$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

# Cosine for length-normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(\vec{q},\vec{d}) = \vec{q} \bullet \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

for q, d length-normalized.

# Cosine similarity illustrated

# Cosine similarity amongst 3 documents

How similar are the novels

SaS: *Sense and Sensibility*

PaP: *Pride and Prejudice*, and

WH: *Wuthering Heights*?

| term | SaS | PaP | WH |
|---|---|---|---|
| affection | 115 | 58 | 20 |
| jealous | 10 | 7 | 11 |
| gossip | 2 | 0 | 6 |
| wuthering | 0 | 0 | 38 |

Term frequencies (counts)

Note: To simplify this example, we don't do idf weighting.

# 3 documents example contd.

**Log frequency weighting**

| term | SaS | PaP | WH |
|---|---|---|---|
| affection | 3.06 | 2.76 | 2.30 |
| jealous | 2.00 | 1.85 | 2.04 |
| gossip | 1.30 | 0 | 1.78 |
| wuthering | 0 | 0 | 2.58 |

**After length normalization**

| term | SaS | PaP | WH |
|---|---|---|---|
| affection | 0.789 | 0.832 | 0.524 |
| jealous | 0.515 | 0.555 | 0.465 |
| gossip | 0.335 | 0 | 0.405 |
| wuthering | 0 | 0 | 0.588 |

dot(SaS,PaP) ≈ 12.1
dot(SaS,WH) ≈ 13.4
dot(PaP,WH) ≈ 10.1

cos(SaS,PaP) ≈ 0.94
cos(SaS,WH) ≈ 0.79
cos(PaP,WH) ≈ 0.69

# Computing cosine scores

```
CosineScore(q)
  float Scores[N] = 0
  float Length[N]
  for each query term t
  do calculate w_{t,q} and fetch postings list for t
      for each pair(d, tf_{t,d}) in postings list
      do Scores[d] += w_{t,d} × w_{t,q}
  Read the array Length
  for each d
  do Scores[d] = Scores[d]/Length[d]
  return Top K components of Scores[]
```

# Computing cosine scores

- Previous algorithm scores term-at-a-time (TAAT)
- Algorithm can be adapted to scoring document-at-a-time (DAAT)
- Storing $w_{t,d}$ in each posting could be expensive
  - …because we'd have to store a floating point number
  - For tf-idf scoring, it suffices to store $tf_{t,d}$ in the posting and $idf_t$ in the head of the postings list
- Extracting the top K items can be done with a priority queue (e.g., a heap)

# tf-idf weighting has many variants

| Term frequency | | Document frequency | | Normalization | |
|---|---|---|---|---|---|
| n (natural) | $\mathrm{tf}_{t,d}$ | n (no) | 1 | n (none) | 1 |
| l (logarithm) | $1+\log(\mathrm{tf}_{t,d})$ | t (idf) | $\log \frac{N}{\mathrm{df}_t}$ | c (cosine) | $\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$ |
| a (augmented) | $0.5+\frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$ | p (prob idf) | $\max\{0, \log\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$ | u (pivoted unique) | $1/u$ |
| b (boolean) | $\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\ 0 & \text{otherwise}\end{cases}$ | | | b (byte size) | $1/CharLength^{\alpha}$, $\alpha<1$ |
| L (log ave) | $\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$ | | | | |

# Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents
- SMART Notation: denotes the combination in use in an engine, with the notation *ddd.qqq,* using the acronyms from the previous table
- A very standard weighting scheme is: lnc.ltc
- Document: logarithmic tf (l as first character), no idf and cosine normalization
- Query: logarithmic tf (l in leftmost column), idf (t in second column), cosine normalization …

# tf-idf example: lnc.ltc

Document: *car insurance auto insurance*
Query: *best car insurance*

| Term | Query | | | | | | Document | | | | Prod |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | tf-raw | tf-wt | df | idf | wt | n'lize | tf-raw | tf-wt | wt | n'lize | |
| auto | 0 | 0 | 5000 | 2.3 | 0 | 0 | 1 | 1 | 1 | 0.52 | 0 |
| best | 1 | 1 | 50000 | 1.3 | 1.3 | 0.34 | 0 | 0 | 0 | 0 | 0 |
| car | 1 | 1 | 10000 | 2.0 | 2.0 | 0.52 | 1 | 1 | 1 | 0.52 | 0.27 |
| insurance | 1 | 1 | 1000 | 3.0 | 3.0 | 0.78 | 2 | 1.3 | 1.3 | 0.68 | 0.53 |

Doc length $=\sqrt{1^2+0^2+1^2+1.3^2} \approx 1.92$

Score = 0+0+0.27+0.53 = 0.8

# Summary – vector space ranking

- Represent the query as a weighted tf-idf vector
- Represent each document as a weighted tf-idf vector
- Compute the cosine similarity score for the query vector and each document vector
- Rank documents with respect to the query by score
- Return the top $K$ (e.g., $K = 10$) to the user

# Resources for today's lecture

- IIR 6.2 – 6.4.3
- http://www.miislita.com/information-retrieval-tutorial/cosine-similarity-tutorial.html
  - Term weighting and cosine similarity tutorial for SEO folk!

Introduction to

# Information Retrieval

Evaluation

Chris Manning and Pandu Nayak

CS276 – Information Retrieval and Web Search

# Situation

- Thanks to your stellar performance in CS276, you quickly rise to VP of Search at internet retail giant nozama.com. Your boss brings in her nephew Sergey, who claims to have built a better search engine for nozama. Do you
  - Laugh derisively and send him to rival Tramlaw Labs?
  - Counsel Sergey to go to Stanford and take CS276?
  - Try a few queries on his engine and say “Not bad”?
  - … ?

# What could you ask Sergey?

- How fast does it index?
  - Number of documents/hour
  - Incremental indexing – nozama adds 10K products/day
- How fast does it search?
  - Latency and CPU needs for nozama's 5 million products
- Does it recommend related products?
- This is all good, but it says nothing about the *quality* of Sergey's search
  - You want nozama's users to be happy with the search experience

# How do you tell if users are happy?

- Search returns products relevant to users
  - How do you assess this at scale?
- Search results get clicked a lot
  - Misleading titles/summaries can cause users to click
- Users buy after using the search engine
  - Or, users spend a lot of $ after using the search engine
- Repeat visitors/buyers
  - Do users leave soon after searching?
  - Do they come back within a week/month/… ?

# Happiness: elusive to measure

- Most common proxy: relevance of search results
  - Pioneered by Cyril Cleverdon in the Cranfield Experiments

- But how do you measure relevance?

# Measuring relevance

- Three elements:
  1. A benchmark document collection
  2. A benchmark suite of queries
  3. An assessment of either Relevant or Nonrelevant for each query and each document

# So you want to measure the quality of a new search algorithm?

- Benchmark documents – nozama's products
- Benchmark query suite – more on this
- Judgments of document relevance for each query

# Relevance judgments

- Binary (relevant vs. non-relevant) in the simplest case
  - More nuanced relevance levels also used(0, 1, 2, 3 …)
- What are some issues already?
- 5 million times 50K takes us into the range of a quarter trillion judgments
  - If each judgment took a human 2.5 seconds, we'd still need $10^{11}$ seconds, or nearly $300 million if you pay people $10 per hour to assess
  - 10K new products per day

# Crowd source relevance judgments?

- Present query-document pairs to low-cost labor on online crowd-sourcing platforms
  - Hope that this is cheaper than hiring qualified assessors
- Lots of literature on using crowd-sourcing for such tasks
  - You get fairly good signal, but the variance in the resulting judgments is quite high

# What else?

- Still need test queries
  - Must be germane to docs available
  - Must be representative of actual user needs
  - Random query terms from the documents are not a good idea
  - Sample from query logs if available
- Classically (non-Web)
  - Low query rates – not enough query logs
  - Experts hand-craft “user needs”

# Early public test Collections (20th C)

TABLE 4.3 Common Test Corpora

| *Collection* | *NDocs* | *NQrys* | *Size (MB)* | *Term/Doc* | *Q-D RelAss* |
|---|---|---|---|---|---|
| ADI | 82 | 35 | | | |
| AIT | 2109 | 14 | 2 | 400 | >10,000 |
| CACM | 3204 | 64 | 2 | 24.5 | |
| CISI | 1460 | 112 | 2 | 46.5 | |
| Cranfield | 1400 | 225 | 2 | 53.1 | |
| LISA | 5872 | 35 | 3 | | |
| Medline | 1033 | 30 | 1 | | |
| NPL | 11,429 | 93 | 3 | | |
| OSHMED | 34,8566 | 106 | 400 | 250 | 16,140 |
| Reuters | 21,578 | 672 | 28 | 131 | |
| TREC | 740,000 | 200 | 2000 | 89-3543 | » 100,000 |

Typical TREC

Recent datasets: 100s of million web pages (GOV, ClueWeb, …)

# Now we have the basics of a benchmark

- Let's review some evaluation measures
  - *Precision*
  - *Recall*
  - DCG
  - …

# Evaluating an IR system

- Note: **user need** is translated into a **query**
- Relevance is assessed relative to the **user need**, *not* the **query**
- E.g., Information need: *My swimming pool bottom is becoming black and needs to be cleaned.*
- Query: ***pool cleaner***
- Assess whether the doc addresses the underlying need, not whether it has these words

# Unranked retrieval evaluation: Precision and Recall – recap from IIR 8/video

- **Binary assessments**

**Precision**: fraction of retrieved docs that are relevant = P(relevant|retrieved)

**Recall**: fraction of relevant docs that are retrieved = P(retrieved|relevant)

| | Relevant | Nonrelevant |
|---|---|---|
| Retrieved | tp | fp |
| Not Retrieved | fn | tn |

- Precision P = tp/(tp + fp)
- Recall R = tp/(tp + fn)

# Rank-Based Measures

- Binary relevance
  - Precision@K (P@K)
  - Mean Average Precision (MAP)
  - Mean Reciprocal Rank (MRR)
- Multiple levels of relevance
  - Normalized Discounted Cumulative Gain (NDCG)

# Precision@K

- Set a rank threshold K
- Compute % relevant in top K
- Ignores documents ranked lower than K
- Ex:
  - Prec@3 of 2/3
  - Prec@4 of 2/4
  - Prec@5 of 3/5

- In similar fashion we have Recall@K

# A precision-recall curve

Lots more detail on this in the Canvas video

# Mean Average Precision

- Consider rank position of each ***relevant*** doc
  - $K_1, K_2, \ldots K_R$
- Compute Precision@K for each $K_1, K_2, \ldots K_R$
- Average precision = average of P@K
- Ex: has AvgPrec of $\frac{1}{3}\cdot\left(\frac{1}{1}+\frac{2}{3}+\frac{3}{5}\right)\approx 0.76$
- MAP is Average Precision across multiple queries/rankings

# Average Precision

= the relevant documents

Ranking #1

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Recall | 0.17 | 0.17 | 0.33 | 0.5 | 0.67 | 0.83 | 0.83 | 0.83 | 0.83 | 1.0 |
| Precision | 1.0 | 0.5 | 0.67 | 0.75 | 0.8 | 0.83 | 0.71 | 0.63 | 0.56 | 0.6 |

Ranking #2

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Recall | 0.0 | 0.17 | 0.17 | 0.17 | 0.33 | 0.5 | 0.67 | 0.67 | 0.83 | 1.0 |
| Precision | 0.0 | 0.5 | 0.33 | 0.25 | 0.4 | 0.5 | 0.57 | 0.5 | 0.56 | 0.6 |

$$\text{Ranking \#1: } (1.0 + 0.67 + 0.75 + 0.8 + 0.83 + 0.6)/6 = 0.78$$

$$\text{Ranking \#2: } (0.5 + 0.4 + 0.5 + 0.57 + 0.56 + 0.6)/6 = 0.52$$

# MAP

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Recall | 0.2 | 0.2 | 0.4 | 0.4 | 0.4 | 0.6 | 0.6 | 0.6 | 0.8 | 1.0 |
| Precision | 1.0 | 0.5 | 0.67 | 0.5 | 0.4 | 0.5 | 0.43 | 0.38 | 0.44 | 0.5 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Recall | 0.0 | 0.33 | 0.33 | 0.33 | 0.67 | 0.67 | 1.0 | 1.0 | 1.0 | 1.0 |
| Precision | 0.0 | 0.5 | 0.33 | 0.25 | 0.4 | 0.33 | 0.43 | 0.38 | 0.33 | 0.3 |

$$average\ precision\ query\ 1 = (1.0 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62$$
$$average\ precision\ query\ 2 = (0.5 + 0.4 + 0.43)/3 = 0.44$$

$$mean\ average\ precision = (0.62 + 0.44)/2 = 0.53$$

# Mean average precision

- If a relevant document never gets retrieved, we assume the precision corresponding to that relevant doc to be zero
- MAP is macro-averaging: each query counts equally
- Now perhaps most commonly used measure in research papers
- Good for web search?
- MAP assumes user is interested in finding many relevant documents for each query
- MAP requires many relevance judgments in text collection

# BEYOND BINARY RELEVANCE

YAHOO!

Web Images Video Local Shopping More

Toyota safety

Search

Options

Search Pad

SearchScan - On

108,000,000 results for
Toyota safety:

Show All

Toyota

Motor Trend

CarsDirect

Shopping Sites

Also try: toyota safety ratings, toyota safety recall, More...

Sponsored Results

Toyota Recall
Toyota Takes Care of its Customers. Read the FAQs at Toyota.com.
www.Toyota.com/Recall

Toyota Safety
& Latest Prices. Free Info. Toyota Research, Reviews.
www.Toyota.Edmunds.com

fair

TOYOTA | Car Safety Innovation and Technology
Toyota home page for car safety and car technology Prius model.
www.safetytoyota.com - Cached

fair

Toyota home page for car safety and car technology ...
We are presenting Toyota's safety technologies for cars. We clearly explain about car safety and car technology using movies and more.
www.safetytoyota.com/en-gb - Cached

Good

Toyota Safety Ratings - Toyota Safety Features - Motor Trend ...
MotorTrend offers Toyota safety ratings, comprehensive auto safety reports, and more. View a all of the standard Toyota safety features. ...
motortrend.com/new_cars/07/toyota/safety_ratings/index.html - 149k - Cached

Toyota Motor Europe Corporate Site Safety
Our approach. Toyota believes that all stakeholders in the road safety equation share a responsibility to reduce the frequency of road accidents. ...
www.toyota.eu/Safety - Cached

[PDF] pdf European Safety Brochure 2005
4047k - Adobe PDF - View as html
not guarantee that all accidents or injuries will be avoided when driving a Toyota and/or Lexus brand motor vehicle equipped with the safety systems ...
www.toyota.no/Images/Safety_Brochure_tcm308-344461.pdf

Toyota - Star Safety System
Star Safety System ... Toyota Mobility Program. Careers. Contact Us. Home. contact us. site map. your privacy rights. legal terms. Toyota Newsroom. sign up for info ...
www.toyota.com/vehicles/demos/star-safety.html - 58k - Cached

Toyota Prius Safety Ratings - CarsDirect
Get overall safety ratings and NHTSA crash test results for the Toyota Prius at CarsDirect.

Sponsored Results

Safety for a Toyota
Research Safety Ratings and Reviews For New Car at Kelley Blue Book.
www.kbb.com

Toyota Safety
Find Toyota Safety dealers, new cars, prices, and photos.
www.NewCars.org

Toyota Safety
Toyota safety Discount Prices Save Money Shopping Online Today.
www.smarter.com

Saftey Toyoto
Explore 5,000+ Pro Sports Choices. Save On Saftey Toyoto.
BaseballGear.Shopzilla.com

See your message here...

# Discounted Cumulative Gain

- Popular measure for evaluating web search and related tasks

- Two assumptions:
  - Highly relevant documents are more useful than marginally relevant documents
  - the lower the ranked position of a relevant document, the less useful it is for the user, since it is less likely to be examined

# Discounted Cumulative Gain

- Uses *graded relevance* as a measure of usefulness, or *gain,* from examining a document
- Gain is accumulated starting at the top of the ranking and may be reduced, or *discounted*, at lower ranks
- Typical discount is 1/log *(rank)*
  - With base 2, the discount at rank 4 is 1/2, and at rank 8 it is 1/3

# Summarize a Ranking: DCG

- What if relevance judgments are in a scale of [0,r]? r>2
- Cumulative Gain (CG) at rank n
  - Let the ratings of the n documents be $r_1, r_2, \ldots r_n$ (in ranked order)
  - $CG = r_1 + r_2 + \ldots r_n$
- Discounted Cumulative Gain (DCG) at rank n
  - $DCG = r_1 + r_2/\log_2 2 + r_3/\log_2 3 + \ldots r_n/\log_2 n$
    - We may use any base for the logarithm

# Discounted Cumulative Gain

- *DCG* is the total gain accumulated at a particular rank *p*:

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i}$$

- Alternative formulation:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log(1+i)}$$

  - used by some web search companies
  - emphasis on retrieving highly relevant documents

# DCG Example

- 10 ranked documents judged on 0–3 relevance scale:

  3, 2, 3, 0, 0, 1, 2, 2, 3, 0

- discounted gain:

  3, 2/1, 3/1.59, 0, 0, 1/2.59, 2/2.81, 2/3, 3/3.17, 0

  = 3, 2, 1.89, 0, 0, 0.39, 0.71, 0.67, 0.95, 0

- DCG:

  3, 5, 6.89, 6.89, 6.89, 7.28, 7.99, 8.66, 9.61, 9.61

# NDCG for summarizing rankings

- Normalized Discounted Cumulative Gain (NDCG) at rank *n*
  - Normalize DCG at rank *n* by the DCG value at rank *n* of the ideal ranking
  - The ideal ranking would first return the documents with the highest relevance level, then the next highest relevance level, etc
- Normalization useful for contrasting queries with varying numbers of relevant results
- NDCG is now quite popular in evaluating Web search

# NDCG - Example

## 4 documents: $d_1$, $d_2$, $d_3$, $d_4$

| i | Ground Truth | | Ranking Function$_1$ | | Ranking Function$_2$ | |
|---|---|---|---|---|---|---|
| | Document Order | $r_i$ | Document Order | $r_i$ | Document Order | $r_i$ |
| 1 | d4 | 2 | d3 | 2 | d3 | 2 |
| 2 | d3 | 2 | d4 | 2 | d2 | 1 |
| 3 | d2 | 1 | d2 | 1 | d4 | 2 |
| 4 | d1 | 0 | d1 | 0 | d1 | 0 |
| | $NDCG_{GT}$=1.00 | | $NDCG_{RF1}$=1.00 | | $NDCG_{RF2}$=0.9203 | |

$$DCG_{GT} = 2 + \left( \frac{2}{\log_2 2} + \frac{1}{\log_2 3} + \frac{0}{\log_2 4} \right) = 4.6309$$

$$DCG_{RF1} = 2 + \left( \frac{2}{\log_2 2} + \frac{1}{\log_2 3} + \frac{0}{\log_2 4} \right) = 4.6309$$

$$DCG_{RF2} = 2 + \left( \frac{1}{\log_2 2} + \frac{2}{\log_2 3} + \frac{0}{\log_2 4} \right) = 4.2619$$

$$MaxDCG = DCG_{GT} = 4.6309$$

# What if the results are not in a list?

- Suppose there's only one Relevant Document
- Scenarios:
  - known-item search
  - navigational queries
  - looking for a fact
- Search duration ~ Rank of the answer
  - measures a user's effort

# Mean Reciprocal Rank

- Consider rank position, K, of first relevant doc
  - Could be – only clicked doc
- Reciprocal Rank score = $\frac{1}{K}$
- MRR is the mean RR across multiple queries

# Human judgments are

- Expensive
- Inconsistent
  - Between raters
  - Over time
- Decay in value as documents/query mix evolves
- Not always representative of "real users"
  - Rating vis-à-vis query, don't know underlying need
  - May not understand meaning of terms, etc.
- So – what alternatives do we have?

# USING USER CLICKS

# User Behavior

Taken with slight adaptation from Fan Guo and Chao Liu's 2009/2010 CIKM tutorial: Statistical Models for Web Search: Click Log Analysis

- Search Results for "CIKM" (in 2009!)

# User Behavior

- Adapt ranking to user clicks?

# What do clicks tell us?

- Tools needed for non-trivial cases

Strong position bias, so absolute click rates unreliable

# Eye-tracking User Study

# Click Position-bias

- Higher positions receive more user attention (eye fixation) and clicks than lower positions.
- This is true even in the extreme setting where the order of positions is reversed.
- "Clicks are informative but biased".

[Joachims+07]

# Relative vs absolute ratings

User's click sequence

Hard to conclude Result1 > Result3

Probably can conclude Result3 > Result2

# Evaluating pairwise relative ratings

- Pairs of the form: DocA better than DocB for a query
  - Doesn't mean that DocA relevant to query
- Now, rather than assess a rank-ordering wrt per-doc relevance assessments …
- Assess in terms of conformance with historical pairwise preferences recorded from user clicks
- BUT!
- Don't learn and test on the same ranking algorithm
  - I.e., if you learn historical clicks from nozama and compare Sergey vs nozama on this history …

# Comparing two rankings via clicks (Joachims 2002)

Query: [support vector machines]

| Ranking A | Ranking B |
|---|---|
| Kernel machines | Kernel machines |
| SVM-light | SVMs |
| Lucent SVM demo | Intro to SVMs |
| Royal Holl. SVM | Archives of SVM |
| SVM software | SVM-light |
| SVM tutorial | SVM software |

# Interleave the two rankings

This interleaving starts with B

| |
|---|
| Kernel machines |
| Kernel machines |
| SVMs |
| SVM-light |
| Intro to SVMs |
| Lucent SVM demo |
| Archives of SVM |
| Royal Holl. SVM |
| SVM-light |

…

# Remove duplicate results

| |
|---|
| Kernel machines |
| Kernel machines |
| SVMs |
| SVM-light |
| Intro to SVMs |
| Lucent SVM demo |
| Archives of SVM |
| Royal Holl. SVM |
| SVM-light |

…

# Count user clicks

Ranking A: 3
Ranking B: 1

| | Clicks |
|---|---|
| Kernel machines | A, B |
| Kernel machines | |
| SVMs | |
| SVM-light | A |
| Intro to SVMs | |
| Lucent SVM demo | A |
| Archives of SVM | |
| Royal Holl. SVM | |
| SVM-light | |

...

# Interleaved ranking

- Present interleaved ranking to users
  - Start randomly with ranking A or ranking B to even out presentation bias
- Count clicks on results from A versus results from B
- Better ranking will (on average) get more clicks

# A/B testing at web search engines

- Purpose: Test a single innovation
- Prerequisite: You have a large search engine up and running.
- Have most users use old system
- Divert a small proportion of traffic (e.g., 0.1%) to an experiment to evaluate an innovation
  - Interleaved experiment
  - Full page experiment

# Facts/entities (what happens to clicks?)

Chrome File Edit View History Bookmarks Window Help 59% Sat 9:30 PM

https://www.google.com/search?q=mount+everest+height&aq=0&oq=mount+everest+he&aqs=chrome.1.57j0l3.6626j0&sourceid=chrome&ie=UTF-8

+Prabhakar Search Images Mail Drive Calendar Sites Groups Contacts More

Google mount everest height pragh@google.com 0 + Share

Web Images Maps Shopping News More Search tools

About 1,300,000 results (0.39 seconds)

29,029' (8,848 m)

Mount Everest, Elevation

**Mount Everest** - Wikipedia, the free encyclopedia

https://en.wikipedia.org/wiki/**Mount_Everest**

By the same measure of base to summit, **Mount** McKinley, in Alaska, is also taller than **Everest**. Despite its **height** above sea level of only 6,193.6 m (20,320 ft), ...

List of deaths on eight - List of people who died ... - Timeline of climbing Mount

Facts About **Mt. Everest** - Scholastic

teacher.scholastic.com/activities/hillary/archive/evefacts.htm

Number of people to successfully climb **Mt. Everest**: 660. Number of

Mount Everest

Mountain

Mount Everest is the Earth's highest mountain, with a peak at 8,848 metres above sea level and the 5th tallest mountain measured from the centre of the Earth. It is located in the Mahalangur section of the Himalayas.

Wikipedia

**Elevation:** 29,029' (8,848 m)

**First ascent:** May 29, 1953

# Recap

- Benchmarks consist of
  - Document collection
  - Query set
  - Assessment methodology
- Assessment methodology can use raters, user clicks, or a combination
  - These get quantized into a *goodness measure* – Precision/NDCG etc.
  - Different engines/algorithms compared on a <u>benchmark</u> together with a <u>goodness measure</u>

# User behavior

- User behavior is an intriguing source of relevance data
  - Users make (somewhat) informed choices when they interact with search engines
  - Potentially a lot of data available in search logs
- But there are significant caveats
  - User behavior data can be very noisy
  - Interpreting user behavior can be tricky
  - Spam can be a significant problem
  - Not all queries will have user behavior

# Incorporating user behavior into ranking algorithm

- Incorporate user behavior features into a ranking function like BM25F
  - But requires an understanding of user behavior features so that appropriate $V_j$ functions are used
- Incorporate user behavior features into *learned* ranking function
- Either of these ways of incorporating user behavior signals improve ranking

Introduction to

# Information Retrieval

CS276

Information Retrieval and Web Search

Chris Manning and Pandu Nayak

**Systems issues**

# Background

- Score computation is a large (10s of %) fraction of the CPU work on a query
  - Generally, we have a tight budget on latency (say, 250ms)
  - CPU provisioning doesn't permit exhaustively scoring every document on every query
- Today we'll look at ways of cutting CPU usage for scoring, without compromising the quality of results (much)
- Basic idea: avoid scoring docs that won't make it into the top *K*

# Safe vs non-safe ranking

- The terminology "safe ranking" is used for methods that guarantee that the *K* docs returned are the *K* absolute highest scoring documents
- Is it ok to be non-safe?

# Ranking function is only a proxy

- User has a task and a query formulation
- Ranking function matches docs to query
- Thus the ranking function is anyway a proxy for user happiness
- If we get a list of *K* docs "close" to the top *K* by the ranking function measure, should be ok

# Recap: Queries as vectors

- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors, measured by cosine similarity

# Efficient cosine ranking

- Find the $K$ docs in the collection "nearest" to the query $\Rightarrow$ $K$ largest query-doc cosines.
- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the $K$ largest cosine values efficiently.
    - Can we do this without computing all $N$ cosines?

# Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - not to totally order all docs in the collection
- Can we pick off docs with *K* highest cosines?
- Let *J* = number of docs with nonzero cosines
  - We seek the *K* best of these *J*

# Use heap for selecting top *K*

- Binary tree in which each node's value > the values of children
- Takes *2J* operations to construct, then each of *K* "winners" read off in 2log *J* steps.
- For *J*=1M, *K*=100, this is about 10% of the cost of sorting.

# Bottlenecks

- Primary computational bottleneck in scoring: cosine computation
- Can we avoid all this computation?
- Yes, but may sometimes get it wrong
  - a doc *not* in the top *K* may creep into the list of *K* output docs
  - As noted earlier, this may not be a bad thing

# SPEEDING COSINE COMPUTATION BY PRUNING

# Generic approach

- Find a set $A$ of *contenders*, with $K < |A| << N$
  - $A$ does not necessarily contain the top $K$, but has many docs from among the top $K$
  - Return the top $K$ docs in $A$
- Think of $A$ as pruning non-contenders
- The same approach is also used for other (non-cosine) scoring functions
- Will look at several schemes following this approach

# Index elimination

- Basic cosine computation algorithm only considers docs containing at least one query term
- Take this further:
  - Only consider high-idf query terms
  - Only consider docs containing many query terms

# High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: ***in*** and ***the*** contribute little to the scores and so <u>don't alter rank-ordering much</u>
- Benefit:
  - Postings of low-idf terms have many docs → these (many) docs get eliminated from set *A* of contenders

# Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms
  - Say, at least 3 out of 4
  - Imposes a "soft conjunction" on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

# 3 of 4 query terms

Scores only computed for docs 8, 16 and 32.

# Champion lists

- Precompute for each dictionary term *t*, the *r* docs of highest weight in *t*'s postings
  - Call this the <u>champion list</u> for *t*
  - (aka <u>fancy list</u> or <u>top docs</u> for *t*)
- Note that *r* has to be chosen at index build time
  - Thus, it's possible that $r < K$
- At query time, only compute scores for docs in the champion list of some query term
  - Pick the *K* top-scoring docs from amongst these

# Exercises

- How can Champion Lists be implemented in an inverted index?

# QUERY-INDEPENDENT DOCUMENT SCORES

# Static quality scores

- We want top-ranking documents to be both *relevant* and *authoritative*
- *Relevance* is being modeled by cosine scores
- *Authority* is typically a query-independent property of a document
- Examples of authority signals
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - Many bitlys, likes, or bookmarks
  - Pagerank

Quantitative

# Modeling authority

- Assign to each document a *query-independent* <u>quality score</u> in [0,1] to each document *d*
  - Denote this by *g(d)*
- Thus, a quantity like the number of citations is scaled into [0,1]
  - Exercise: suggest a formula for this.

# Net score

- Consider a simple total score combining cosine relevance and authority
- net-score($q,d$) = $g(d)$ + cosine($q,d$)
  - Can use some other linear combination
  - Indeed, any function of the two "signals" of user happiness
- Now we seek the top $K$ docs by <u>net score</u>

# Top *K* by net score – fast methods

- First idea: Order all postings by *g(d)*
- Key: this is a common ordering for all postings
- Thus, can concurrently traverse query terms' postings for
  - Postings intersection
  - Cosine score computation
- Exercise: write pseudocode for cosine score computation if postings are ordered by *g(d)*

# Why order postings by *g(d)*?

- Under *g(d)*-ordering, top-scoring docs likely to appear early in postings traversal
- In time-bound applications (say, we have to return whatever search results we can in 50 ms), this allows us to stop postings traversal early
  - Short of computing scores for all docs in postings

# Champion lists in *g(d)*-ordering

- Can combine champion lists with *g(d)*-ordering
- Maintain for each term a champion list of the *r* docs with highest $g(d) + \text{tf-idf}_{td}$
- Seek top-*K* results from only the docs in these champion lists

# CLUSTER PRUNING

# Cluster pruning: preprocessing

- Pick $\sqrt{N}$ *docs* at random: call these *leaders*
- For every other doc, pre-compute nearest leader
  - Docs attached to a leader: its *followers;*
  - <u>Likely</u>: each leader has ~ $\sqrt{N}$ followers.

# Cluster pruning: query processing

- Process a query as follows:
  - Given query *Q*, find its nearest *leader L.*
  - Seek *K* nearest docs from among *L*'s followers.

# Visualization

# Why use random sampling

- Fast
- Leaders reflect data distribution

# General variants

- Have each follower attached to *b1*=3 (say) nearest leaders.
- From query, find *b2*=4 (say) nearest leaders and their followers.
- Can recurse on leader/follower construction.

# TIERED INDEXES

# High and low lists

- For each term, we maintain two postings lists called *high* and *low*
  - Think of *high* as the champion list
- When traversing postings on a query, only traverse *high* lists first
  - If we get more than *K* docs, select the top *K* and stop
  - Else proceed to get docs from the *low* lists
- Can be used even for simple cosine scores, without global quality *g(d)*
- A means for segmenting index into two <u>tiers</u>

# Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important
  - …
  - Least important
- Can be done by *g(d)* or another measure
- Inverted index thus broken up into tiers of decreasing importance
- At query time use top tier unless it fails to yield *K* docs
  - If so drop to lower tiers

# Example tiered index

# Impact-ordered postings

- We only want to compute scores for docs for which $wf_{t,d}$ is high enough
- We sort each postings list by $wf_{t,d}$
- Now: not all postings in a common order!
- How do we compute scores in order to pick off top *K?*
  - Two ideas follow

# 1. Early termination

- When traversing $t$'s postings, stop early after either
  - a fixed number of $r$ docs
  - $wf_{t,d}$ drops below some threshold
- Take the union of the resulting sets of docs
  - One from the postings of each query term
- Compute only the scores for docs in this union

# 2. idf-ordered terms

- When considering the postings of query terms
- Look at them in order of decreasing idf
  - High idf terms likely to contribute most to score
- As we update score contribution from each query term
  - Stop if doc scores relatively unchanged
- Can apply to cosine or some other net scores

# SAFE RANKING

# *Safe* vs non-safe ranking

- The terminology "safe ranking" is used for methods that guarantee that the *K* docs returned are the *K* absolute highest scoring documents
  - (Not necessarily just under cosine similarity)

# Safe ranking

- When we output the top $K$ docs, we have a proof that these are indeed the top $K$
- Does this imply we always have to compute all $N$ cosines?
  - We'll look at pruning methods
  - So we only fully score some $J$ documents

# WAND scoring

- An instance of DAAT scoring
- Basic idea reminiscent of branch and bound
  - We maintain a running *threshold* score – e.g., the $K^{th}$ highest score computed so far
  - We prune away all docs whose cosine scores are guaranteed to be below the threshold
  - We compute exact cosine scores for only the un-pruned docs

Broder et al. Efficient Query Evaluation using a Two-Level Retrieval Process.

# Index structure for WAND

- Postings ordered by docID
- Assume a special iterator on the postings of the form "go to the first docID greater than or equal to *X*"
- Typical state: we have a "finger" at some docID in the postings of each query term
  - Each finger moves only to the right, to larger docIDs
- Invariant – all docIDs lower than any finger have already been *processed,* meaning
  - These docIDs are either pruned away or
  - Their cosine scores have been computed

# Upper bounds

- At all times for each query term *t*, we maintain an *upper bound* $UB_t$ on the score contribution of any doc to the right of the finger
  - Max (over docs remaining in *t*'s postings) of $w_t$(doc)

$UB_t = w_t(38)$

As finger moves right, *UB* drops

# Pivoting

- Query: *catcher in the rye*
- Let's say the current finger positions are as below

# Prune docs that have no hope

- Terms sorted in order of finger positions
- Move fingers to 589 or right

Update *UB*'s

# Compute 589's score if need be

- If 589 is present in enough postings, compute its full cosine score – else some fingers to right of 589
- Pivot again …

# WAND summary

- In tests, WAND leads to a 90+% reduction in score computation
  - Better gains on longer queries
- Nothing we did was specific to cosine ranking
  - We need scoring to be *additive* by term
- WAND and variants give us <u>safe ranking</u>
  - Possible to devise "careless" variants that are a bit faster but not safe (see summary in Ding+Suel 2011)
  - Ideas combine some of the non-safe scoring we considered

# FINISHING TOUCHES FOR A COMPLETE SCORING SYSTEM

# Query term proximity

- Free text queries: just a set of terms typed into the query box – common on the web
- Users prefer docs in which query terms occur within close proximity of each other
- Let *w* be the smallest window in a doc containing all query terms, e.g.,
- For the query *strained mercy* the smallest window in the doc *The quality of mercy is not strained* is 4 (words)
- Would like scoring function to take this into account – how?

# Query parsers

- Free text query from user may in fact spawn one or more queries to the indexes, e.g., query *rising interest rates*
  - Run the query as a phrase query
  - If <*K* docs contain the phrase *rising interest rates*, run the two phrase queries *rising interest* and *interest rates*
  - If we still have <*K* docs, run the vector space query *rising interest rates*
  - Rank matching docs by vector space scoring
- This sequence is issued by a <u>query parser</u>

# Aggregate scores

- We've seen that score functions can combine cosine, static quality, proximity, etc.
- How do we know the best combination?
- Some applications – expert-tuned
- Increasingly common: machine-learned

# Putting it all together

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Text Classification 1

Chris Manning and Pandu Nayak

# Standing queries

- The path from IR to text classification:
  - You have an information need to monitor, say:
    - Unrest in the Niger delta region
  - You want to rerun an appropriate query periodically to find new news items on this topic
  - You will be sent new documents that are found
    - I.e., it's not ranking but classification (relevant vs. not relevant)
- Such queries are called **standing queries**
  - Long used by "information professionals"
  - A modern mass instantiation is **Google Alerts**
- Standing queries are (hand-written) text classifiers

**From:** Google Alerts
**Subject:** **Google Alert - stanford -neuro-linguistic nlp OR "Natural Language Processing" OR parser OR tagger OR ner OR "named entity" OR segmenter OR classifier OR dependencies OR "core nlp" OR corenlp OR phrasal**
**Date:** May 7, 2012 8:54:53 PM PDT
**To:** Christopher Manning

---

**Web**

3 new results for **stanford -neuro-linguistic nlp OR "Natural Language Processing" OR parser OR tagger OR ner OR "named entity" OR segmenter OR classifier OR dependencies OR "core nlp" OR corenlp OR phrasal**

Twitter / **Stanford NLP** Group: @Robertoross If you only n ...
@Robertoross If you only need tokenization, java -mx2m edu.**stanford.nlp**. process.PTBTokenizer file.txt runs in 2MB on a whole file for me.... 9:41 PM Apr 28th ...
twitter.com/stanfordnlp/status/196459102770171905

[Java] LexicalizedParser lp = LexicalizedParser.loadModel("edu ...
loadModel("edu/**stanford/nlp**/models/lexparser/englishPCFG.ser.gz");. String[] sent = { "This", "is", "an", "easy", "sentence", "." };. Tree parse = lp.apply(Arrays.
pastebin.com/az14R9nd

More Problems with Statistical **NLP** || kuro5hin.org
Tags: nlp, ai, coursera, **stanford**, **nlp**-class, cky, nltk, reinventing the wheel, ... Programming Assignment 6 for **Stanford's nlp**-class is to implement a CKY parser .
www.kuro5hin.org/story/2012/5/5/11011/68221

---

Tip: Use quotes ("like this") around a set of words in your query to match them exactly. Learn more.

Delete this alert.
Create another alert.
Manage your alerts.

# Spam filtering
# Another text classification task

From: "" <takworlld@hotmail.com>

Subject: real estate is the only way... gem  oalvgkay

Anyone can buy real estate with no money down

Stop paying rent TODAY !

There is no need to spend hundreds or even thousands for similar courses

I am 22 years old and I have already purchased 6 properties using the methods outlined in this truly INCREDIBLE ebook.

Change your life NOW !

==========

Click Below to order:

http://www.wholesaledaily.com/sales/nmd.htm

# Categorization/Classification

- Given:
  - A representation of a document *d*
    - Issue: how to represent text documents.
    - Usually some type of high-dimensional space – bag of words
  - A fixed set of classes:

    $C = \{c_1, c_2, \ldots, c_J\}$
- Determine:
  - The category of *d*: $\gamma(d) \in C$, where $\gamma(d)$ is a classification function
  - We want to build classification functions ("classifiers").

# Classification Methods (1)

- Manual classification
  - Used by the original Yahoo! Directory
  - Looksmart, about.com, ODP, **PubMed**
  - Accurate when job is done by experts
  - Consistent when the problem size and team is small
  - Difficult and expensive to scale
    - Means we need automatic classification methods for big problems

# Classification Methods (2)

- Hand-coded rule-based classifiers
  - One technique used by news agencies, intelligence agencies, etc.
  - Widely deployed in government and enterprise
  - Vendors provide “IDE” for writing such rules

# Classification Methods (2)

- Hand-coded rule-based classifiers
  - Commercial systems have complex query languages
  - Accuracy can be high if a rule has been carefully refined over time by a subject expert
  - Building and maintaining these rules is expensive

# A Verity topic

## A complex classification rule: art

| | |
|---|---|
| comment line | # Beginning of art topic definition |
| top-level topic | art ACCRUE |
| topic definition modifiers | /author = "fsmith" |
| | /date = "30-Dec-01" |
| | /annotation = "Topic created by fsmith" |
| subtopic topic | * 0.70 performing-arts ACCRUE |
| evidence topic | ** 0.50 WORD |
| topic definition modifier | /wordtext = ballet |
| evidence topic | ** 0.50 STEM |
| topic definition modifier | /wordtext = dance |
| evidence topic | ** 0.50 WORD |
| topic definition modifier | /wordtext = opera |
| evidence topic | ** 0.30 WORD |
| topic definition modifier | /wordtext = symphony |
| subtopic | * 0.70 visual-arts ACCRUE |
| | ** 0.50 WORD |
| | /wordtext = painting |
| | ** 0.50 WORD |
| | /wordtext = sculpture |
| subtopic | * 0.70 film ACCRUE |
| | ** 0.50 STEM |
| | /wordtext = film |
| subtopic | ** 0.50 motion-picture PHRASE |
| | *** 1.00 WORD |
| | /wordtext = motion |
| | *** 1.00 WORD |
| | /wordtext = picture |
| | ** 0.50 STEM |
| | /wordtext = movie |
| subtopic | * 0.50 video ACCRUE |
| | ** 0.50 STEM |
| | /wordtext = video |
| | ** 0.50 STEM |
| | /wordtext = vcr |
| | # End of art topic |

- Note:
  - maintenance issues (author, etc.)
  - Hand-weighting of terms

[Verity was bought by Autonomy in 2005, which was bought by HP in 2011 – a mess; I think it no longer exists ...]

At the heart of eContext is its taxonomy of the commercial and social web. This taxonomy provides a consistent framework that groups conversations, behavior, and digital interactions into topical hierarchies up to 21 tiers, with clear connections between those topics. Clients push large amounts of text data from multiple sources through eContext, creating a single data set for measurement and analytics. With this holistic perspective, they can distill more comprehensive, meaningful insights, in less time.

As clients feed information through our API, our custom pre-processing parses the text into linguistically appropriate units that eContext can work with efficiently.

Then eContext checks each unit against its 55 million positive and negative rules to determine the correct topic, or topics, that content exhibits. This happens hundreds of thousands of times per second, as we enrich the highest velocity data streams available in our industry — everything up to and including the full firehose of Tweets. Thanks to the rules that govern eContext's decision-making, and the 450,000 different concepts it recognizes, eContext is able to deliver deep, broad, and accurate results.

Schedule a Demo

Enhancing Virtual Agents with Structured Knowledge

Download White Paper

# Classification Methods (3): Supervised learning

- Given:
  - A document *d*
  - A fixed set of classes:

    $C = \{c_1, c_2, \ldots, c_J\}$
  - A <u>training set</u> *D* of documents each with a label in *C*
- Determine:
  - A learning method or algorithm which will enable us to learn a classifier $\gamma$
  - For a test document *d*, we assign it the class

$\gamma(d) \in C$

# Classification Methods (3)

- Supervised learning
  - Naive Bayes (simple, common) – see video, cs229
  - k-Nearest Neighbors (simple, powerful)
  - Support-vector machines (newer, generally more powerful)
  - Decision trees → random forests → gradient-boosted decision trees (e.g., xgboost)
  - … plus many other methods
  - No free lunch: need hand-classified training data
  - But data can be built up by amateurs
- Many commercial systems use a mix of methods

# Features

- Supervised learning classifiers can use any sort of feature
  - URL, email address, punctuation, capitalization, dictionaries, network features
- In the simplest bag of words view of documents
  - We use **only** word features
  - we use **all** of the words in the text (not a subset)

# The bag of words representation

Y( I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun… It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet. )=c

# The bag of words representation

$\gamma($

| | |
|---|---|
| great | 2 |
| love | 2 |
| recommend | 1 |
| laugh | 1 |
| happy | 1 |
| ... | ... |

$) = c$

# Feature Selection: Why?

- Text collections have a large number of features
  - 10,000 – 1,000,000 unique words … and more
- Selection may make a particular classifier feasible
  - Some classifiers can't deal with 1,000,000 features
- Reduces training time
  - Training time for some methods is quadratic or worse in the number of features
- Makes runtime models smaller and faster
- Can improve generalization (performance)
  - Eliminates noise features
  - Avoids overfitting

# Feature Selection: Frequency

- The simplest feature selection method:
  - Just use the commonest terms
  - No particular foundation
  - But it make sense why this works
    - They're the words that can be well-estimated and are most often available as evidence
  - In practice, this is often 90% as good as better methods
- Smarter feature selection:
  - chi-squared, etc.

# Naïve Bayes: See *IIR* 13 or cs124 lecture on Coursera or cs229

- Classify based on prior weight of class and conditional parameter for what each word says:

$$c_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \left[ \log P(c_j) + \sum_{i \in positions} \log P(x_i \mid c_j) \right]$$

- Training is done by counting and dividing:

$$P(c_j) \leftarrow \frac{N_{c_j}}{N} \qquad P(x_k \mid c_j) \leftarrow \frac{T_{c_j x_k} + \alpha}{\sum_{x_i \in V} [T_{c_j x_i} + \alpha]}$$

- Don't forget to smooth

# SpamAssassin

- Naïve Bayes has found a home in spam filtering
  - Paul Graham's A Plan for Spam
  - Widely used in spam filters
  - But many features beyond words:
    - black hole lists, etc.
    - particular hand-crafted text patterns

# SpamAssassin Features:

- Basic (Naïve) Bayes spam probability
- Mentions: Generic Viagra
- Regex: millions of (dollar) ((dollar) NN,NNN,NNN.NN)
- Phrase: impress ... girl
- Phrase: ‘Prestigious Non-Accredited Universities’
- From: starts with many numbers
- Subject is all capitals
- HTML has a low ratio of text to image area
- Relay in RBL, http://www.mail-abuse.com/enduserinfo_rbl.html
- RCVD line looks faked
- http://spamassassin.apache.org/tests_3_3_x.html

# Naive Bayes is Not So Naive

- Very fast learning and testing (basically just count words)
- Low storage requirements
- Very good in domains with many <u>equally important</u> features
- More robust to irrelevant features than many learning methods
  - Irrelevant features cancel out without affecting results

# Naive Bayes is Not So Naive

- More robust to concept drift (changing class definition over time)
- Naive Bayes won 1st and 2nd place in KDD-CUP 97 competition out of 16 systems

  Goal: Financial services industry direct mail response prediction: Predict if the recipient of mail will actually respond to the advertisement – 750,000 records.
- A good dependable baseline for text classification (but not the best)!

# Evaluating Categorization

- Evaluation must be done on test data that are independent of the training data
  - Sometimes use cross-validation (averaging results over multiple training and test splits of the overall data)
- Easy to get good performance on a test set that was available to the learner during training (e.g., just memorize the test set)

# Evaluating Categorization

- Measures: precision, recall, F1, classification accuracy
- Classification accuracy: $r/n$ where n is the total number of test docs and $r$ is the number of test docs correctly classified

# Remember: Vector Space Representation

- Each document is a vector, one component for each term (= word).
- Normally normalize vectors to unit length.
- High-dimensional vector space:
  - Terms are axes
  - 10,000+ dimensions, or even 100,000+
  - Docs are vectors in this space

- How can we do classification in this space?

# Classification Using Vector Spaces

- In vector space classification, training set corresponds to a labeled set of points (equivalently, vectors)
- Premise 1: Documents in the same class form a contiguous region of space
- Premise 2: Documents from different classes don't overlap (much)
- Learning a classifier: build surfaces to delineate classes in the space

# Documents in a Vector Space

# Test Document of what class?

# Test Document = Government

Our focus: how to find good separators

# Definition of centroid

$$\vec{\mu}(c) = \frac{1}{|D_c|} \sum_{d \in D_c} \vec{v}(d)$$

- Where $D_c$ is the set of all documents that belong to class $c$ and $v(d)$ is the vector space representation of $d$.

- *Note that centroid will in general* not *be a unit vector even when the inputs are unit vectors.*

# Rocchio classification

- Rocchio forms a simple representative for each class: the centroid/prototype
- Classification: nearest prototype/centroid
- It does not guarantee that classifications are consistent with the given training data

Why not?

# Two-class Rocchio as a linear classifier

- Line or hyperplane defined by:

$$\sum_{i=1}^{M} w_i d_i = \theta$$

- For Rocchio, set:

$$\vec{w} = \vec{\mu}(c_1) - \vec{\mu}(c_2)$$

$$\theta = 0.5 \times (|\vec{\mu}(c_1)|^2 - |\vec{\mu}(c_2)|^2)$$

# Linear classifier: Example

- Class: "interest" (as in interest rate)
- Example features of a linear classifier

| $w_i$ | $t_i$ | $w_i$ | $t_i$ |
|---|---|---|---|
| 0.70 | prime | −0.71 | dlrs |
| 0.67 | rate | −0.35 | world |
| 0.63 | interest | −0.33 | sees |
| 0.60 | rates | −0.25 | year |
| 0.46 | discount | −0.24 | group |
| 0.43 | bundesbank | −0.24 | dlr |

- To classify, find dot product of feature vector and weights

# Rocchio classification

- Little used outside text classification
  - It has been used quite effectively for text classification
  - But in general worse than Naïve Bayes
- Again, cheap to train and test documents

# *k* Nearest Neighbor Classification

- kNN = *k* Nearest Neighbor

- To classify a document *d*:
- Define *k*-neighborhood as the *k* nearest neighbors of *d*
- Pick the majority class label in the *k*-neighborhood
- For larger *k* can roughly estimate P(c|*d*) as #(c)/k

# Test Document = Science

Voronoi diagram

# Nearest-Neighbor Learning

- Learning: just store the labeled training examples *D*
- Testing instance *x (under 1NN)*:
  - Compute similarity between *x* and all examples in *D*.
  - Assign *x* the category of the most similar example in *D*.
- Does not compute anything beyond storing the examples
- Also called:
  - Case-based learning
  - Memory-based learning
  - Lazy learning
- Rationale of kNN: contiguity hypothesis

# k Nearest Neighbor

- Using only the closest example (1NN) is subject to errors due to:
  - A single atypical example.
  - Noise (i.e., an error) in the category label of a single training example.
- More robust: find the *k* examples and return the majority category of these *k*
- *k* is typically odd to avoid ties; 3 and 5 are most common

# Nearest Neighbor with Inverted Index

- Naively finding nearest neighbors requires a linear search through $|D|$ documents in collection
- But determining $k$ nearest neighbors is the same as determining the $k$ best retrievals using the test document as a query to a database of training documents.
- Use standard vector space inverted index methods to find the $k$ nearest neighbors.
- Testing Time: $O(B|V_t|)$ where $B$ is the average number of training documents in which a test-document word appears.
  - Typically $B << |D|$

# kNN: Discussion

- No feature selection necessary
- No training necessary
- Scales well with large number of classes
  - Don't need to train *n* classifiers for *n* classes
- Classes can influence each other
  - Small changes to one class can have ripple effect
- Done naively, very expensive at test time
- In most cases it's more accurate than NB or Rocchio
  - As the amount of data goes to infinity, it has to be a great classifier! – it's "Bayes optimal"

# Rocchio Anomaly

- Prototype models have problems with polymorphic (disjunctive) categories.

# 3 Nearest Neighbor vs. Rocchio

- Nearest Neighbor tends to handle polymorphic categories better than Rocchio/NB.

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Christopher Manning and Pandu Nayak

## Lecture 13: Decision trees and machine learning on documents

Some slides borrowed from John Canny

# Text classification

- Last lecture: Basic algorithms for text classification
  - Naive Bayes classifier
    - Simple, cheap, high bias, linear
  - K Nearest Neighbor classification
    - Simple, expensive at test time, high variance, non-linear
  - Vector space classification: Rocchio
    - Simple linear discriminant classifier; perhaps too simple*
- Today
  - Decision trees
  - Some empirical evaluation and comparison
  - Decision tree ensembles
    - Will lead into using tree-based methods (GBRT) for ranking
  - Text-specific issues in classification

# Text Classification Evaluation: Classic Reuters-21578 Data Set

- Most (over)used data set
- 21578 documents
- 9603 training, 3299 test articles (ModApte/Lewis split)
- 118 categories
  - An article can be in more than one category
  - Learn 118 binary category distinctions
- Average document: about 90 types, 200 tokens
- Average number of classes assigned
  - 1.24 for docs with at least one category
- Only about 10 out of 118 categories are large

Common categories (#train, #test)

- Earn (2877, 1087)
- Acquisitions (1650, 179)
- Money-fx (538, 179)
- Grain (433, 149)
- Crude (389, 189)
- Trade (369,119)
- Interest (347, 131)
- Ship (197, 89)
- Wheat (212, 71)
- Corn (182, 56)

# Reuters Text Categorization data set (**Reuters-21578)** document


<REUTERS TOPICS="YES" LEWISSPLIT="TRAIN" CGISPLIT="TRAINING-SET" OLDID="12981" NEWID="798">

<DATE> 2-MAR-1987 16:51:43.42</DATE>

<TOPICS><D>livestock</D><D>hog</D></TOPICS>

<TITLE>AMERICAN PORK CONGRESS KICKS OFF TOMORROW</TITLE>

<DATELINE> CHICAGO, March 2 - </DATELINE><BODY>The American Pork Congress kicks off tomorrow, March 3, in Indianapolis with 160 of the nations pork producers from 44 member states determining industry positions on a number of issues, according to the National Pork Producers Council, NPPC.

Delegates to the three day Congress will be considering 26 resolutions concerning various issues, including the future direction of farm policy and the tax law as it applies to the agriculture sector. The delegates will also debate whether to endorse concepts of a national PRV (pseudorabies virus) control and eradication program, the NPPC said.

A large trade show, in conjunction with the congress, will feature the latest in technology in all areas of the industry, the NPPC added. Reuter

&#3;</BODY></TEXT></REUTERS>

# Newer Reuters data: RCV1: 810,000 docs

- Top topics in Reuters RCV1

# Decision Trees for text classification

- A tree with internal nodes labeled by terms
- Branches are labeled by tests on the weight that the term has (or just presence/absence)
- Leaves are labeled by categories
- Classifier categorizes document by descending tree following tests to leaf
- The label of the leaf node is then assigned to the document
- Most decision trees are binary trees (never disadvantageous; may require extra internal nodes)

# Decision Tree Example

# Decision Tree Learning

- Learn a sequence of tests on features, typically using top-down, greedy search
  - At each stage choose unused feature with highest Information Gain
- At leaves, either categorical (yes/no) or continuous decisions

learn - [Learned Network]
Display Mode
Save
Table
Graph
Select
Animate
Zoom
Scale
Colors
Note that you can reuse the same attribute
Category: "interest"
rate=1
f_318436 = 0
f_334692 = 0
f_334692 != 0
f_318437 = 0
rate.t=1
lending=0
prime=0
discount=0
pct=1
year=0
year=1
f_318513 != 0
f_221503 != 0
f_318514 != 0
f_155282 != 0
f_152244 != 0
f_305798 != 0
f_229109 != 0
f_318514 != 0
f_322247 != 0
f_312895 != 0
f_108151 != 0
f_130778 = f_130778 != 0
f_130777 = f_130777 != 0
f_139624 != 0
f_139624 = 0
f_322469 = 0
f_322469 != 0
f_229064 = 0
f_229064 != 0
f_155282 = 0
f_155282 != 0
0 141 0.93
1 10 0.072
0 1700 0.99
1 12 0.0076
0 46 0.78
1 12 0.22
0 14 0.34
1 28 0.66
0 10 0.85
1 1 0.15
0 33 0.79
1 8 0.21
0 442 0.98
1 8 0.02
0 14 0.75
1 4 0.25
Ready

# Decision tree learning

- If there are $k$ features, a decision tree might have up to $2^k$ nodes. This is almost always much too big!
- We want to find "efficient" (small but effective) trees.
- We can do this in a **greedy manner by recursively choosing a best split feature at each node**.

# Choosing an attribute

- Idea: a good features splits the examples into subsets that are (ideally) "all positive" or "all negative"
  - This is binary case. Same idea and method works for n-ary case except for a bias towards many valued features.

A good feature

A bad feature: Wheat is still 60%

# Using Information Theory

**Entropy** is defined at each node based on the class breakdown:

- Let $p_i$ be the fraction of examples in class *i*.
- Let $p_i^f$ be the fraction of elements with feature *f* that lie in class *i*.
- Let $p_i^{\neg f}$ be the fraction of elements without feature *f* that lie in class *i*

Finally let $p^f$ and $p^{\neg f}$ be the fraction of nodes with (respectively without) feature *f*

# Information Gain

Before the split by *f*, entropy is

$$E = -\sum_{i=1}^{m} p_i \log p_i$$

After split by f, the entropy is

$$E_f = -p^f \sum_{i=1}^{m} p_i^f \log p_i^f - p^{\neg f} \sum_{i=1}^{m} p_i^{\neg f} \log p_i^{\neg f}$$

The information gain = $E - E_f$ (information = – entropy)

# Using Information Theory

**Entropy: amount of uncertainty in class distribution**

**Two class case:**

# Example

$$E = -\sum_{i=1}^{m} p_i \log p_i$$

$E = -\sum p_i \log p_i =$

0.5*1+0.25*2+0.25*2 = 1.5 bits

After:

$E_f$ = (0.5+0.5)*1.5= 1.5 bits

No gain!

Before: E = 1.5 bits

After:

$E_f$ = (0.5+0.5)*1 bits = 1 bits

Gain = $E - E_f$ = 0.5 bits

# Choosing best features

At each node, we choose the feature *f* which **maximizes the information gain**.

This tends to be produce mixtures of classes at each node that **are more and more "pure"** as you go down the tree.

If a node has examples all of one class c, we make it a leaf and output "c". Otherwise, we potentially continue to build

If a leaf still has a mixed distribution, we output **the most popular class** at that node.

# Numeric features (e.g., tf-idf, etc.)

- Commonly make a binary split (f < t), but where?
- Exhaustively: evaluate each split point between observed values for information gain.
  - Slow.
  - Can be made a bit more efficient by optimizing counting
- Discretize into bins
  - Divide all numeric values into *k* bins.
  - Feature is treated as if categorical
  - Binning can be based on statistics of the entire dataset
    - For instance one might use *k*-means clustering on values of feature

# When to stop?

- When all the examples at a node are of the same class
- When a fixed tree depth *d* is reached
- When there isn't an attribute you can split on where the split differentiates classes with statistical significance (e.g., with chi-square or Fisher's Exact)
- Commonest/best: Use separate validation data
  - Grow a big tree (perhaps with depth threshold)
  - Prune nodes bottom up that fail to (significantly) improve classification performance on the validation data

# Decision Tree Models

- As tree depth increases, bias decreases and variance generally increases. Why? (Hint: think about k-NN)

Bias decreases with tree depth

Variance increases with tree depth

- Some of the stuff fit deep in a tree is fairly random facts and associations in the training data

# Decision trees classify via rectangular regions

The (rectangular) regions are aligned with feature axes
The model cannot learn arbitrary linear decision boundaries
Overall, the result is a non-linear classifier

# Decision Tree Learning for Text

- Most people's intuitions are that text has many words and you have a lot of weak-evidence features
  - Hence, use of a small number of feature tests is potentially bad for text classification
  - But in fact the method can sometime do pretty well – such as for the Reuters dataset. Topics can have marker words.
- Decision trees are easily interpreted by humans – much more easily than methods like Naive Bayes
  - You can extract rules from decision trees, in fact.

# Text classification
# Per class evaluation measures

- Recall: Fraction of docs in class *i* classified correctly:

$$\frac{c_{ii}}{\sum_j c_{ij}}$$

- Precision: Fraction of docs assigned class *i* that are actually about class *i*:

$$\frac{c_{ii}}{\sum_j c_{ji}}$$

- Accuracy: (1 - error rate) Fraction of docs classified correctly:

$$\frac{\sum_i c_{ii}}{\sum_j \sum_i c_{ij}}$$

# Micro- vs. Macro-Averaging

- If we have more than one class, how do we combine multiple performance measures into one quantity?
- Macroaveraging: Compute performance for each class, then average.
- Microaveraging: Collect decisions for all classes, compute contingency table, evaluate.

# Micro- vs. Macro-Averaging: Example

Class 1

| | Truth: yes | Truth: no |
|---|---|---|
| Classifier: yes | 10 | 10 |
| Classifier: no | 10 | 970 |

Class 2

| | Truth: yes | Truth: no |
|---|---|---|
| Classifier: yes | 90 | 10 |
| Classifier: no | 10 | 890 |

Micro Ave. Table

| | Truth: yes | Truth: no |
|---|---|---|
| Classifier: yes | 100 | 20 |
| Classifier: no | 20 | 1860 |

- Macroaveraged precision: (0.5 + 0.9)/2 = 0.7
- Microaveraged precision: 100/120 = .83

- Microaveraged score is dominated by score on common classes

# Dumais et al. 1998: Reuters – Break-even F1

| | Findsim | NBayes | BayesNets | Trees | LinearSVM |
|---|---|---|---|---|---|
| earn | 92.9% | 95.9% | 95.8% | 97.8% | 98.2% |
| acq | 64.7% | 87.8% | 88.3% | 89.7% | 92.8% |
| money-fx | 46.7% | 56.6% | 58.8% | 66.2% | 74.0% |
| grain | 67.5% | 78.8% | 81.4% | 85.0% | 92.4% |
| crude | 70.1% | 79.5% | 79.6% | 85.0% | 88.3% |
| trade | 65.1% | 63.9% | 69.0% | 72.5% | 73.5% |
| interest | 63.4% | 64.9% | 71.3% | 67.1% | 76.3% |
| ship | 49.2% | 85.4% | 84.4% | 74.2% | 78.0% |
| wheat | 68.9% | 69.7% | 82.7% | 92.5% | 89.7% |
| corn | 48.2% | 65.3% | 76.4% | 91.8% | 91.1% |
| Micro | | | | | |
| Avg Top 10 | 64.6% | 81.5% | 85.0% | 88.4% | 91.4% |
| Avg All Cat | 61.7% | 75.2% | 80.0% | na | 86.4% |

**Recall:** % labeled in category among those stories that are really in category

**Precision:** % really in category among those stories labeled in category

**Break Even:** When recall equals precision

# Reuters ROC - Category Grain

**Recall:** % labeled in category among those stories that are really in category

**Precision:** % really in category among those stories labeled in category

# ROC for Category - Earn

# ROC for Category - Crude

# ROC for Category - Ship

(a)

| | NB | Rocchio | kNN | SVM |
|---|---|---|---|---|
| micro-avg-L (90 classes) | 80 | 85 | 86 | 89 |
| macro-avg (90 classes) | 47 | 59 | 60 | 60 |

(b)

| | NB | Rocchio | kNN | trees | SVM |
|---|---|---|---|---|---|
| earn | 96 | 93 | 97 | 98 | 98 |
| acq | 88 | 65 | 92 | 90 | 94 |
| money-fx | 57 | 47 | 78 | 66 | 75 |
| grain | 79 | 68 | 82 | 85 | 95 |
| crude | 80 | 70 | 86 | 85 | 89 |
| trade | 64 | 65 | 77 | 73 | 76 |
| interest | 65 | 63 | 74 | 67 | 78 |
| ship | 85 | 49 | 79 | 74 | 86 |
| wheat | 70 | 69 | 77 | 93 | 92 |
| corn | 65 | 48 | 78 | 92 | 90 |
| micro-avg (top 10) | 82 | 65 | 82 | 88 | 92 |
| micro-avg-D (118 classes) | 75 | 62 | n/a | n/a | 87 |

Evaluation measure: $F_1$

# The discriminative alternative: Logistic Regression and Support vector machines

- Directly predict class conditional on words:
- (Binary) Logistic Regression:

$$\log \frac{P(C \mid d)}{P(\overline{C} \mid d)} = \alpha + \sum_{w \in d} \beta_w \times w$$

- Tune parameters $\beta_w$ to optimize conditional likelihood or "margin" (SVM) in predicting classes
- What a statistician would probably tell you to use if you said you had a categorical decision problem (like text categorization)

# LogR/SVM Performance

- Early results with LogR were disappointing, because people didn't understand the means to *regularize* (smooth) LogR to cope with **sparse textual features**
- Done right, LogR clearly outperforms NB in text categorization and batch filtering studies
- SVMs were seen as the best general text classification method in the period c. 1997– 2005
- LogR seems as good as SVMs (Tong & Oles 2001)
- But now challenged by:
  - Neural net methods (improve word similarity models)
  - Ensemble methods, e.g. **random forests, boosting**

# Ensemble Methods

Are like **Crowdsourced machine learning algorithms**:

- Take a collection of simple or *weak* learners
- Combine their results to make a single, better learner

Types:

- **Bagging:** train learners in parallel on different samples of the data, then combine by voting (discrete output) or by averaging (continuous output).
- **Stacking:** feed output of first-level model(s) as features into a second-stage learner like logistic regression.
- **Boosting:** train subsequent learners on the filtered/weighted output of earlier learners so they fix the stuff that the earlier learners got wrong

# Random Forests

Grow K trees on datasets sampled from the original dataset with replacement (bootstrap samples), p = number of features.

- Draw K bootstrap samples of size N (size of original dataset)
- Grow each Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
  - Typically m might be e.g. sqrt(p)
- **Runtime:** Aggregate the predictions of the trees (most popular vote) to produce the final class.

# Random Forests

Principles: we want to take a vote between different learners so we don't want the models to be too similar. These two criteria ensure diversity in the individual trees:

- Data bagging: Draw K bootstrap samples of size N:
  - Each tree is trained on different data.
- Feature bagging: Grow a Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
  - Corresponding nodes in different trees (usually) can't use the same feature to split.

# Random Forests

- **Very popular in practice,** at one point the most popular classifier for dense data (<= a few thousand features)
- **Easy to implement** (train a lot of trees).
- **Parallelizes easily** (but not necessarily efficiently). Good match for MapReduce.
- **Now not quite state-of-the-art accuracy** – Gradient-boosted trees (less features) and Deep NNs (vision, speech, language, …) generally do better
- **Needs many passes over the data** – at least the max depth of the trees. (<< boosted trees though)
- **Easy to overfit** – need to balance accuracy/fit tradeoff.

# Boosted Decision Trees

- A more recent alternative to random Forests
- In contrast to RFs whose trees are trained **independently**, BDT trees are trained **sequentially** by **boosting**:
  - Each successive tree is trained on weighted data which emphasizes instances incorrectly labeled by the previous trees.
- Both methods can produce very high-quality models
- But boosted decision trees are now normally the method of choice for datasets with a medium number of features

# Random Forests vs Boosted Trees

- The "geometry" of the methods is very different:
- Random forest use 10's of deep, large trees:

# Random Forests vs Boosted Trees

- The "geometry" of the methods is very different:
- Boosted decision trees use 1000's of shallow, small trees:

Bias reduction through boosting – variance already low

# Random Forests vs Boosted Trees

- RF training embarrassingly parallel, can be very fast
- Evaluation of trees (runtime) also much faster for RFs

# The Real World

P. Jackson and I. Moulinier. 2002. *Natural Language Processing for Online Applications*

- "There is no question concerning the commercial value of being able to classify documents automatically by content. There are myriad potential applications of such a capability for corporate intranets, government departments, and Internet publishers"

- "Understanding the data is one of the keys to successful categorization, yet this is an area in which most categorization tool vendors are extremely weak. Many of the 'one size fits all' tools on the market have not been tested on a wide range of content types."

# The Real World

- Gee, I'm building a text classifier for real, now!
- What should I do?

- How much training data do you have?
  - None
  - Very little
  - Quite a lot
  - A huge amount and its growing

# Manually written rules

- No training data, adequate editorial staff?
- Never forget the hand-written rules solution!
  - If (wheat or grain) and not (whole or bread) then
    - Categorize as grain
- In practice, rules get a lot bigger than this
  - Can also be phrased using tf or tf.idf weights
- With careful crafting (human tuning on development data) performance is high:
  - Construe: 94% recall, 84% precision over 675 categories (Hayes and Weinstein IAAI 1990)
- Amount of work required is huge
  - Estimate 2 days per class … plus maintenance

# Very little data?

- If you're just doing supervised classification, you should stick to something high bias
  - There are theoretical results that Naïve Bayes should do well in such circumstances (Ng and Jordan 2002 NIPS)
- The interesting theoretical answer is to explore semi-supervised training methods:
  - Bootstrapping, EM over unlabeled documents, …
- The practical answer is to get more labeled data as soon as you can
  - How can you insert yourself into a process where humans will be willing to label data for you??

# A reasonable amount of data?

- Perfect!
- We can use all our clever classifiers
- Roll out logistic regression/SVMs/random forests!

- But if you are using an SVM/NB etc., you should probably be prepared with the “hybrid” solution where there is a Boolean overlay
  - Or else to use user-interpretable Boolean-like models like decision trees
  - Users like to hack, and management likes to be able to implement quick fixes immediately

# A huge amount of data?

- This is great in theory for doing accurate classification…
- But it could easily mean that expensive methods like SVMs (train time) or kNN (test time) are less practical

- Naïve Bayes can come back into its own again!
  - Or other methods with linear training/test complexity like **(regularized) logistic regression** (though much more expensive to train)

# Accuracy as a function of data size

- With enough data the choice of classifier may not matter much, and the best choice may be unclear
  - Data: Brill and Banko on context-sensitive spelling correction
- But the fact that you have to keep doubling your data to improve performance is a little unpleasant

# How many categories?

- A few (well separated ones)?
  - Easy!
- A zillion closely related ones?
  - Think: Yahoo! Directory, Library of Congress classification, legal applications
  - Quickly gets difficult!
    - Classifier combination is always a useful technique
      - Voting, bagging, or boosting multiple classifiers
    - Much literature on hierarchical classification
      - Mileage fairly unclear, but helps a bit (Tie-Yan Liu et al. 2005)
      - **Definitely helps for scalability**, even if not in accuracy
    - May need a hybrid automatic/manual solution

# Good practice department: Make a confusion matrix

This (*i*, *j*) entry means 53 of the docs actually in class *i* were put in class *j* by the classifier.

- In a perfect classification, only the diagonal has non-zero entries
- Look at common confusions and how they might be addressed

# Good practice department: N-Fold Cross-Validation

- Results can vary based on sampling error due to different training and test sets.
- Average results over multiple training and test sets (splits of the overall data) for the best results.
- Ideally, test and training sets are independent on each trial.
  - But this would require too much labeled data.
- Partition data into N equal-sized disjoint segments.
- Run N trials, each time using a different segment of the data for testing, and training on the remaining N–1 segments.
- This way, at least test-sets are independent.
- Report average classification accuracy over the N trials.
- Typically, N = 10.

# Good practice department: Learning Curves

- In practice, labeled data is usually rare and expensive.
- Would like to know how performance varies with the number of training instances.
- *Learning curves* plot classification accuracy on independent test data (*Y* axis) versus number of training examples (*X* axis).
- One can do both the above and produce learning curves averaged over multiple trials from cross-validation

# How can one tweak performance?

- Aim to exploit any domain-specific useful features that give special meanings or that zone the data
  - E.g., an author byline or mail headers
- Aim to collapse things that would be treated as different but shouldn't be.
  - E.g., part numbers, chemical formulas

- Does putting in "hacks" help?
  - You bet! Easiest way to improve practical systems
    - Feature design and non-linear weighting is *very* important in the performance of real-world systems

# Upweighting

- You can get a lot of value by differentially weighting contributions from different document zones:
- That is, you count as two instances of a word when you see the word in, say, the abstract
  - Upweighting title words helps (Cohen & Singer 1996)
    - Doubling the weighting on the title words is a good rule of thumb
    - Like what we talked about for BM25F
  - Upweighting the first sentence of each paragraph helps (Murata, 1999)
  - Upweighting sentences that contain title words helps (Ko *et al,* 2002)

# Two techniques for zones

1. Have a completely separate set of features/parameters for different zones like the title
2. Use the same features (pooling/tying their parameters) across zones, but upweight the contribution of different zones

- Commonly the second method is more successful: it costs you nothing in terms of sparsifying the data, but can give a very useful performance boost
  - Which is best is a contingent fact about the data

# Text Summarization techniques in text classification

- Text Summarization: Process of extracting key pieces from text, normally by features on sentences reflecting position and content
- Much of this work can be used to suggest weightings for terms in text categorization
  - See: Kolcz, Prabakarmurthi, and Kalita, CIKM 2001: Summarization as feature selection for text categorization
  - Categorizing with title,
  - Categorizing with first paragraph only
  - Categorizing with paragraph with most keywords
  - Categorizing with first and last paragraphs, etc.

# Does stemming/lowercasing/… help?

- As always, it's hard to tell, and empirical evaluation is normally the gold standard
- But note that the role of tools like stemming is rather different for TextCat vs. IR:
  - For IR, you often want to collapse forms of the verb *oxygenate* and *oxygenation*, since all of those documents will be relevant to a query for *oxygenation*
  - For TextCat, with sufficient training data, stemming *does no good*. It only helps in compensating for data sparseness (which can be severe in TextCat applications). *Overly aggressive stemming can easily degrade performance.*

# Measuring Classification Figures of Merit

- Accuracy of classification
  - Main evaluation criterion in academia
- Speed of training statistical classifier
  - Some methods are very cheap; some very costly
- Speed of classification (docs/hour)
  - No big differences for most algorithms
  - Exceptions: kNN, complex preprocessing requirements
- Effort in creating training set/hand-built classifier
  - human hours/topic

# Measuring Classification Figures of Merit

- Not just accuracy; in the real world, there are economic measures:
  - Your choices are:
    - Do no classification
      - That has a cost (hard to compute)
    - Do it all manually
      - Has an easy-to-compute cost if you're doing it like that now
    - Do it all with an automatic classifier
      - Mistakes have a cost
    - Do it with a combination of automatic classification and manual review of uncertain/difficult/"new" cases
  - Commonly the last method is cost efficient and is adopted
    - With more theory and Turkers: Werling, Chaganty, Liang, and Manning (2015). On-the-Job Learning with Bayesian Decision Theory. http://arxiv.org/abs/1506.03140

# A common problem: Concept Drift

- Categories change over time
- Example: "president of the united states"
  - 1998: clinton is great feature
  - 2018: clinton is bad feature
- One measure of a text classification system is how well it protects against concept drift.
  - Favors simpler models like Naïve Bayes
- Feature selection: can be bad in lessening protection against concept drift

# Summary

- Decision trees
  - Simple non-linear, discriminative classifier
  - Easy to interpret
  - Moderately effective for text classification
- Logistic regression and Support vector machines (SVM)
  - Linear discriminative classifiers
  - Close to state of art (except perhaps NNs) for a single classifier
  - We're not covering them in this year's class
- Classifier ensembles
  - Random forests (bagging)
  - Boosting
- Comparative evaluation of methods
- Real world: exploit domain specific structure!

# Resources for today's lecture


- S. T. Dumais. 1998. Using SVMs for text categorization, IEEE Intelligent Systems, 13(4)
- Yiming Yang, Xin Liu. 1999. A re-examination of text categorization methods. 22nd Annual International SIGIR
- Tong Zhang, Frank J. Oles. 2001. Text Categorization Based on Regularized Linear Classification Methods. *Information Retrieval* 4(1): 5-31
- Trevor Hastie, Robert Tibshirani and Jerome Friedman. *Elements of Statistical Learning: Data Mining, Inference and Prediction.* Springer-Verlag, New York.
- T. Joachims, *Learning to Classify Text using Support Vector Machines*. Kluwer, 2002.
- Fan Li, Yiming Yang. 2003. A Loss Function Analysis for Classification Methods in Text Categorization. ICML 2003: 472-479.
- Tie-Yan Liu, Yiming Yang, Hao Wan, et al. 2005. Support Vector Machines Classification with Very Large Scale Taxonomy, SIGKDD Explorations, 7(1): 36-43.
- 'Classic' Reuters-21578 data set: http://www.daviddlewis.com/resources/testcollections/reuters21578/

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Pandu Nayak and Prabhakar Raghavan

Lecture 12: Clustering

# Today's Topic: Clustering

- Document clustering
  - Motivations
  - Document representations
  - Success criteria
- Clustering algorithms
  - Partitional
  - Hierarchical

# What is clustering?

- Clustering: the process of grouping a set of objects into classes of similar objects
  - Documents within a cluster should be similar.
  - Documents from different clusters should be dissimilar.
- The commonest form of *unsupervised learning*
  - Unsupervised learning = learning from raw data, as opposed to supervised data where a classification of examples is given
  - A common and important task that finds many applications in IR and other places

# A data set with clear cluster structure

- How would you design an algorithm for finding the three clusters in this case?

# Applications of clustering in IR

- Whole corpus analysis/navigation
  - Better user interface: search without typing
- For improving recall in search applications
  - Better search results (like pseudo RF)
- For better navigation of search results
  - Effective “user recall” will be higher
- For speeding up vector space retrieval
  - Cluster-based retrieval gives faster search

# Yahoo! Hierarchy *isn't* clustering but *is* the kind of output you want from clustering

# Google News: automatic clustering gives an effective news presentation metaphor

Google News

http://news.google.com/

**World »** edit

**Pirates Demand $25 Million Ransom for Hijacked Tanker (Update1)**
Bloomberg - **36 minutes ago**
By Caroline Alexander and Hamsa Omar Nov. 20 (Bloomberg) -- Somali pirates are demanding $25 million in ransom to release an oil-laden Saudi supertanker seized off the East African coast, and called on the ship's owners to pay up ``soon.
Somali pirates demand $25M for Saudi ship United Press International
African Union says Somali politicians fuel piracy Washington Post
BBC News - guardian.co.uk - Aljazeera.net - RIA Novosti
**all 4,015 news articles »**
BBC News

**Pakistan protests over US missile strikes**
Reuters - **2 hours ago**
By Simon Cameron-Moore ISLAMABAD (Reuters) - Pakistan summoned US ambassador Anne Patterson on Thursday to protest over missile strikes launched by pilotless drone aircraft against militant targets in Pakistan.
Pakistan protests US drone attacks, Taliban warns of reprisals AFP
Pakistan warns US over missile strike CNN International
Telegraph.co.uk - China Daily - Xinhua - PRESS TV
**all 560 news articles »**
Reuters

**Nighttime attack on Thai antigovernment protesters wounds at least 20**
Christian Science Monitor - **30 minutes ago**
The government denied attacking demonstrators, who have called for the ouster of the prime minister. By Huma Yusuf One person has been killed and 23 others wounded in a grenade attack Thursday against antigovernment protesters occupying the Thai prime ...
Blast Kills 1, Wounds 23 at Thai Prime Minister's Office Washington Post
Anti-government protestor in Thailand dies in grenade attack International Herald Tribune
Xinhua - United Press International - The Associated Press - AsiaOne
**all 688 news articles »**
WELT ONLINE

Show more stories | Show fewer stories

**U.S. »** edit

**Top Court in California Will Review Proposition 8**
New York Times - **1 hour ago**
By JESSE McKINLEY SAN FRANCISCO - Responding to pleas for legal clarity from those on both sides of the issue, the California Supreme Court said Wednesday that it would take up the case of whether a voter-approved ban on same-sex unions was ...
California Supreme Court to decide fate of Prop. 8 same-sex ... San Jose Mercury News
Prop. 8 gay marriage ban goes to Supreme Court Los Angeles Times
The Miami Herald - San Diego Union Tribune - Indiana Daily Student - San Francisco Chronicle
**all 1,241 news articles »**
Calgary Herald

**Drop That Cigarette, Today Is The Great American Smokeout**
dBTechno - **1 hour ago**
Washington (dbTechno) - Today marks the annual Great American Smokeout hosted by the American Cancer Society, and is trying to get people all across the US to drop their cigarettes for just one day.
Great American Smokeout: Time to kick the habit Capital Times
National Smoke Out Day is Thursday; be a quitter Las Cruces Sun-News
MPNnow.com - eMaxHealth.com - Times Tribune of Corbin - ABC15.com (KNXV-TV)
**all 338 news articles »**
eFluxMedia

**Perino: Bush would sign jobless benefits extension**
The Associated Press - **47 minutes ago**
WASHINGTON (AP) - With weekly jobless claims benefits at a 16-year high, the White House said Thursday that President George W. Bush would quickly sign legislation pending in Congress to provide further unemployment benefits.
Bush would sign measure to extend jobless benefits Houston Chronicle
Jobless claims show need for benefits extension: White House AFP
Washington Times - Wall Street Journal Blogs - WOI - Tampabay.com
**all 599 news articles »**
Seattle Post Intelligencer

Show more stories | Show fewer stories

http://www.google.com/hostednews/ap/article/ALeqM5hGjNbXl6O23C8QzqZMY0pGPAik-AD94INLTG1

# Scatter/Gather: Cutting, Karger, and Pedersen

# For visualizing a document collection and its themes

- Wise et al, “Visualizing the non-visual” PNNL
- ThemeScapes, Cartia
  - [Mountain height = cluster size]

# For improving search recall

- *Cluster hypothesis* - Documents in the same cluster behave similarly with respect to relevance to information needs
- Therefore, to improve search recall:
  - Cluster docs in corpus a priori
  - When a query matches a doc *D*, also return other docs in the cluster containing *D*
- Hope if we do this: The query "car" will also return docs containing *automobile*
  - Because clustering grouped together docs containing *car* with those containing *automobile.*

Why might this happen?

http://search.yippy.com/search?v%3aproject=clusty&v%3afile=viv_Xpo6AV&v%3arecluster=&

Most Visited | Getting Started | Latest Headlines | Fridge Filters

Yahoo! Search | SEARCH

web news images wikipedia jobs more »

clustering | Search | advanced preferences

clouds sources sites | remix

All Results (185)
- Analysis (23)
- Method (22)
- Computing (15)
- Search, Engine (13)
- Hierarchical (16)
- Definition (11)
- High availability (13)
- Linux (11)
- Windows, Microsoft (9)
- Papers (8)

more | all clouds

find in clouds: Find

Font size: A A A A

Yippy Approved
Shakespeare Searched

Top **179** results retrieved for the query **clustering** (definition) (details)

**Clustering**
Lower Latency In Your Data Center w/ Intel's **Cluster** Ready Solutions!
www.intel.com

Load Balancing 101
Learn the 'Nuts & Bolts' of Load Balancing with F5's White Paper
www.f5.com/load_balancing

Affordable Load Balancers
High Performance Load Balancing Solutions From KEMP- See Demo Today
kemptechnologies.com

Computer **cluster** - Wikipedia, the free encyclopedia
Middleware such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) permits compute **clustering** programs to be portable to a
/Computer_**cluster**
en.wikipedia.org/wiki/Computer_cluster - [cache] - Bing, Yahoo!

Writer's Web: Prewriting: **Clustering**
Prewriting: **Clustering** Melanie Dawson & Joe Essid (printable version here) **Clustering** is a type of prewriting that allows you to explore many idea
writing2.richmond.edu/writing/wweb/**cluster**.html
writing2.richmond.edu/writing/wweb/cluster.html - [cache] - Bing, Yahoo!

Getting Started: **Clustering** Ideas - CT Community Colleges
**Clustering. Clustering** is similar to another process called Brainstorming. **Clustering** is something that you can do on your own or with friends or
grammar.**ccc**.commnet.edu/grammar/composition/brainstorm_**clustering**.htm
grammar.ccc.commnet.edu/grammar/composition/brainstorm_clustering.htm - [cache] - Bing, Yahoo!

Advanced **Clustering** | Home

yippy.com – grouping search results

# Issues for clustering

- Representation for clustering
  - Document representation
    - Vector space? Normalization?
      - Centroids aren't length normalized
  - Need a notion of similarity/distance
- How many clusters?
  - Fixed a priori?
  - Completely data driven?
    - Avoid "trivial" clusters - too large or small
      - If a cluster's too large, then for navigation purposes you've wasted an extra user click without whittling down the set of documents much.

# Notion of similarity/distance

- Ideal: semantic similarity.
- Practical: term-statistical similarity
  - We will use cosine similarity.
  - Docs as vectors.
  - For many algorithms, easier to think in terms of a *distance* (rather than <u>similarity</u>) between docs.
  - We will mostly speak of Euclidean distance
    - <u>But real implementations use cosine similarity</u>

# Clustering Algorithms

- Flat algorithms
  - Usually start with a random (partial) partitioning
  - Refine it iteratively
    - *K* means clustering
    - (Model based clustering)
- Hierarchical algorithms
  - Bottom-up, agglomerative
  - (Top-down, divisive)

# Hard vs. soft clustering

- Hard clustering: Each document belongs to exactly one cluster
  - More common and easier to do
- Soft clustering: A document can belong to more than one cluster.
  - Makes more sense for applications like creating browsable hierarchies
  - You may want to put a pair of sneakers in two clusters: (i) sports apparel and (ii) shoes
  - You can only do that with a soft clustering approach.
- We won't do soft clustering today. See IIR 16.5, 18

# Partitioning Algorithms

- Partitioning method: Construct a partition of *n* documents into a set of *K* clusters
- Given: a set of documents and the number *K*
- Find: a partition of *K* clusters that optimizes the chosen partitioning criterion
  - Globally optimal
    - Intractable for many objective functions
    - Ergo, exhaustively enumerate all partitions
  - Effective heuristic methods: *K*-means and *K*-medoids algorithms

See also Kleinberg NIPS 2002 – impossibility for natural clustering

# *K*-Means

- Assumes documents are real-valued vectors.
- Clusters based on *centroids* (aka the *center of gravity* or mean) of points in a cluster, *c*:

$$\vec{\mu}(c) = \frac{1}{|c|}\sum_{\vec{x}\in c}\vec{x}$$

- Reassignment of instances to clusters is based on distance to the current cluster centroids.
  - (Or one can equivalently phrase it in terms of similarities)

# *K*-Means Algorithm

Select *K* random docs $\{s_1, s_2, \ldots s_K\}$ as seeds.

Until clustering *converges* (or other stopping criterion):

For each doc $d_i$:

Assign $d_i$ to the cluster $c_j$ such that $dist(x_i, s_j)$ is minimal.

(*Next, update the seeds to the centroid of each cluster*)

For each cluster $c_j$

$s_j = \mu(c_j)$

# *K* Means Example (*K*=2)

Pick seeds

Reassign clusters

Compute centroids

Reassign clusters

Compute centroids

Reassign clusters

Converged!

# Termination conditions

- Several possibilities, e.g.,
  - A fixed number of iterations.
  - Doc partition unchanged.
  - Centroid positions don't change.

Does this mean that the docs in a cluster are unchanged?

# Convergence

- Why should the *K*-means algorithm ever reach a *fixed point*?
  - A state in which clusters don't change.
- *K*-means is a special case of a general procedure known as the *Expectation Maximization (EM) algorithm*.
  - EM is known to converge.
  - Number of iterations could be large.
    - But in practice usually isn't

Lower case!

# Convergence of *K*-Means

- Define goodness measure of cluster *k* as sum of squared distances from cluster centroid:
  - $G_k = \Sigma_i (d_i - c_k)^2$ (sum over all $d_i$ in cluster *k*)
- $G = \Sigma_k G_k$
- Reassignment monotonically decreases G since each vector is assigned to the closest centroid.

# Convergence of *K*-Means

- Recomputation monotonically decreases each $G_k$ since ($m_k$ is number of members in cluster $k$):
  - $\Sigma (d_i - a)^2$ reaches minimum for:
  - $\Sigma -2(d_i - a) = 0$
  - $\Sigma d_i = \Sigma a$
  - $m_K a = \Sigma d_i$
  - $a = (1/ m_k) \Sigma d_i = c_k$
- *K*-means typically converges quickly

# Time Complexity

- Computing distance between two docs is O*(M)* where *M* is the dimensionality of the vectors.
- Reassigning clusters: O*(KN)* distance computations, or O*(KNM).*
- Computing centroids: Each doc gets added once to some centroid: O*(NM).*
- Assume these two steps are each done once for *I* iterations: O*(IKNM).*

# Seed Choice

- Results can vary based on random seed selection.
- Some seeds can result in poor convergence rate, or convergence to sub-optimal clusterings.
  - Select good seeds using a heuristic (e.g., doc least similar to any existing mean)
  - Try out multiple starting points
  - Initialize with the results of another method.

**Example showing sensitivity to seeds**

**In the above, if you start with B and E as centroids you converge to {A,B,C} and {D,E,F}**
**If you start with D and F you converge to {A,B,D,E} {C,F}**

# *K*-means issues, variations, etc.

- Recomputing the centroid after every assignment (rather than after all points are re-assigned) can improve speed of convergence of *K*-means
- Assumes clusters are spherical in vector space
  - Sensitive to coordinate changes, weighting etc.
- Disjoint and exhaustive
  - Doesn't have a notion of "outliers" by default
  - But can add outlier filtering

Dhillon et al. ICDM 2002 – variation to fix some issues with small document clusters

# How Many Clusters?

- Number of clusters *K* is given
  - Partition *n* docs into predetermined number of clusters
- Finding the "right" number of clusters is part of the problem
  - Given docs, partition into an "appropriate" number of subsets.
  - E.g., for query results - ideal value of *K* not known up front - though UI may impose limits.
- Can usually take an algorithm for one flavor and convert to the other.

# *K* not specified in advance

- Say, the results of a query.
- Solve an optimization problem: penalize having lots of clusters
  - application dependent, e.g., compressed summary of search results list.
- Tradeoff between having more clusters (better focus within each cluster) and having too many clusters

# *K* not specified in advance

- Given a clustering, define the Benefit for a doc to be the cosine similarity to its centroid
- Define the Total Benefit to be the sum of the individual doc Benefits.

Why is there always a clustering of Total Benefit $n$?

# Penalize lots of clusters

- For each cluster, we have a Cost *C*.
- Thus for a clustering with *K* clusters, the Total Cost is *KC*.
- Define the Value of a clustering to be =

  Total Benefit - Total Cost.
- Find the clustering of highest value, over all choices of *K*.
  - Total benefit increases with increasing *K*. But can stop when it doesn't increase by "much". The Cost term enforces this.

# Hierarchical Clustering

- Build a tree-based hierarchical taxonomy (*dendrogram*) from a set of documents.

- One approach: recursive application of a partitional clustering algorithm.

# Dendrogram: Hierarchical Clustering

- Clustering obtained by cutting the dendrogram at a desired level: each connected component forms a cluster.

# Hierarchical Agglomerative Clustering (HAC)

- Starts with each doc in a separate cluster
  - then repeatedly joins the *closest pair* of clusters, until there is only one cluster.
- The history of merging forms a binary tree or hierarchy.

Note: the resulting clusters are still "hard" and induce a partition

# *Closest pair* of clusters

- Many variants to defining closest pair of clusters
- **Single-link**
  - Similarity of the *most* cosine-similar (single-link)
- **Complete-link**
  - Similarity of the "furthest" points, the *least* cosine-similar
- **Centroid**
  - Clusters whose centroids (centers of gravity) are the most cosine-similar
- **Average-link**
  - Average cosine between pairs of elements

# Single Link Agglomerative Clustering

- Use maximum similarity of pairs:

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$

- Can result in "straggly" (long and thin) clusters due to chaining effect.
- After merging $c_i$ and $c_j$, the similarity of the resulting cluster to another cluster, $c_k$, is:

$$sim((c_i \cup c_j), c_k) = \max(sim(c_i, c_k), sim(c_j, c_k))$$

# Single Link Example

# Complete Link

- Use minimum similarity of pairs:

$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$

- Makes "tighter," spherical clusters that are typically preferable.
- After merging $c_i$ and $c_j$, the similarity of the resulting cluster to another cluster, $c_k$, is:

$$sim((c_i \cup c_j), c_k) = \min(sim(c_i, c_k), sim(c_j, c_k))$$

# Complete Link Example

# Computational Complexity

- In the first iteration, all HAC methods need to compute similarity of all pairs of $N$ initial instances, which is $O(N^2)$.
- In each of the subsequent $N$–2 merging iterations, compute the distance between the most recently created cluster and all other existing clusters.
- In order to maintain an overall $O(N^2)$ performance, computing similarity to each other cluster must be done in constant time.
  - Often $O(N^3)$ if done naively or $O(N^2 \log N)$ if done more cleverly

# Group Average

- Similarity of two clusters = average similarity of all pairs within merged cluster.

$$sim(c_i, c_j) = \frac{1}{\left|c_i \cup c_j\right|\left(\left|c_i \cup c_j\right| - 1\right)} \sum_{\vec{x} \in (c_i \cup c_j)} \sum_{\vec{y} \in (c_i \cup c_j): \vec{y} \neq \vec{x}} sim(\vec{x}, \vec{y})$$

- Compromise between single and complete link.
- Two options:
  - Averaged across all ordered pairs in the merged cluster
  - Averaged over all pairs *between* the two original clusters
- No clear difference in efficacy

# Computing Group Average Similarity

- Always maintain sum of vectors in each cluster.

$$\vec{s}(c_j) = \sum_{\vec{x} \in c_j} \vec{x}$$

- Compute similarity of clusters in constant time:

$$sim(c_i, c_j) = \frac{(\vec{s}(c_i) + \vec{s}(c_j)) \bullet (\vec{s}(c_i) + \vec{s}(c_j)) - (|c_i| + |c_j|)}{(|c_i| + |c_j|)(|c_i| + |c_j| - 1)}$$

# What Is A Good Clustering?

- Internal criterion: A good clustering will produce high quality clusters in which:
  - the intra-class (that is, intra-cluster) similarity is high
  - the inter-class similarity is low
  - The measured quality of a clustering depends on both the document representation and the similarity measure used

# External criteria for clustering quality

- Quality measured by its ability to discover some or all of the hidden patterns or latent classes in gold standard data
- Assesses a clustering with respect to ground truth … requires *labeled data*
- Assume documents with $C$ gold standard classes, while our clustering algorithms produce $K$ clusters, $\omega_1, \omega_2, \ldots, \omega_K$ with $n_i$ members.

# External Evaluation of Cluster Quality

- Simple measure: purity, the ratio between the dominant class in the cluster $\pi_i$ and the size of cluster $\omega_i$

$$Purity(\omega_i) = \frac{1}{n_i} \max_j (n_{ij}) \quad j \in C$$

- Biased because having *n* clusters maximizes purity
- Others are entropy of classes in clusters (or mutual information between classes and clusters)

# Purity example

Cluster I Cluster II Cluster III

Cluster I: Purity = 1/6 (max(5, 1, 0)) = 5/6

Cluster II: Purity = 1/6 (max(1, 4, 1)) = 4/6

Cluster III: Purity = 1/5 (max(2, 0, 3)) = 3/5

# Rand Index measures between pair decisions. Here RI = 0.68

| **Number of points** | Same Cluster in clustering | Different Clusters in clustering |
|---|---|---|
| Same class in ground truth | 20 | 24 |
| Different classes in ground truth | 20 | 72 |

# Rand index and Cluster F-measure

$$RI = \frac{A + D}{A + B + C + D}$$

Compare with standard Precision and Recall:

$$P = \frac{A}{A + B} \qquad R = \frac{A}{A + C}$$

People also define and use a cluster F-measure, which is probably a better measure.

# Final word and resources

- In clustering, clusters are inferred from the data without human input (unsupervised learning)
- However, in practice, it's a bit less clear: there are many ways of influencing the outcome of clustering: number of clusters, similarity measure, representation of documents, . . .

- Resources
  - IIR 16 except 16.5
  - IIR 17.1–17.3