

Information Retrieval

23252 characters in 3795 words on 620 lines

Florian Moser

August 15, 2018

1 introduction

1.1 information retrieval

target

find unstructured data satisfying an information need
from within large collections

history

1970 relational area ("throw data at computers")
1980 object area ("throw computers at computers")
2000 no sql area ("throw computers at data")
now machine learning ("throw data at data")

core models

list of words (words with exact positions)
bag of words / vector model (words with frequency)
set of words (words with boolean occurrence)

1.2 data

three V

Volume (how much data there is)
Variety (how well structured it is)
Velocity (how fast the analysis has to take place)

essential factors

capacity (how much data)
throughput (how fast transmitted)
latency (when data receival start)

data shapes

structured (text, graphs, trees, tables, cubes)
unstructured (more common, implicit features like grammar)

semantic gap

representation of data vs semantics of information

information in perspective

information = data + meaning
knowledge = information + application

1.3 definitions

document

the indexed element
column in adjacency matrix
entries in posting list

vocabulary

(term, posting list) dictionary

token

parsed from the raw text, chars grouped together for processing
specific occurrence of a term/type at some place in a document
positional posting in list of words model

type

a word, class of all tokens with same character sequence

term

a word, possibly normalized type with linguistic processing
entry in index dictionary, row in adjacency matrix

posting

records that term appears in document
needs to record at least docid, possibly containing location of term

2 boolean retrieval

grep

usage like `grep keyword file | grep -v non_keyword`
but cannot rank/scale, no proximity semantics

indices

everywhere, for example book glossary
incidence matrix with y, x are terms, documents
1 denotes a match (hence a very sparse matrix)

3 inverted index

dictionary of terms to postings
dictionary in memory, posting lists on disk

3.1 term-document model

document as a list of words
simplified to term-document bipartite graph (set of words)
generate postings & adjacency matrix

3.2 structure

entry \rightarrow pointer \rightarrow count of postings \rightarrow list of postings
"hi" \rightarrow [2] (doc2, 12), (doc4, 16)

3.3 index construction

3.3.1 collect document

process sequence of characters, character sets

UTF-8

maps character with unicode defined codepoint to binary
dynamically extend encoding size
0_ for 1 byte, 110_ 10_ for 2 bytes, 1110_ 10_ 10_ for 3 bytes

detect content

detect language, encoding, type
user-defined, annotated, machine learning, software-specific
data-specific, binary

detect scope

single content may be distributed over multiple files

3.3.2 tokenize

convert document in list of tokens
need to know language for proper processing
detect tokens like number, emails, urls to decide whether to remove

remove

punctuation (careful with york-based, isn't)
stop words (the, that; see the Reuters list)

split tokens

split compounds (useful for german, difficult with asian languages)
segment words (automatically split into subwords, for asians)

3.3.3 linguistic preprocessing

normalization of tokens with equivalence classes / list

remove punctuation

like U.S.A \rightarrow USA
but C.A.T. \rightarrow cat

remove diacritics

cliché \rightarrow cliche, zürich \rightarrow zuerich

case-folding

like USA \rightarrow usa
but Apple \rightarrow apple; need truecasing (maybe using ML)

language specific

english with color, colour
french with l', le, la, les
german with schütze, schuetze
japan with different writing systems
different spellings of the same Beijing, Peking

stemming

chop of end of word based on rules (like porter, lovins, paice)
return lemma as final result
analysis → alalysi, are → ar
reduces word to its word stem (wortstamm)

lemmatization

morphological (wortstamm) analysis of respective language
transfer semantic meaning, to syntactic words
english has little morphology, better for german, spanish
return lemma as final result
computer, compute, computes, ...

expansion

window → windows, window (inverted equivalence classes)
write directly into index or transform query (windows OR window)

3.3.4 build up

naiive

build list of (term, docid) for each document
sort list by term, then docid
remove duplicates, possibly saving info as term frequency
create posting lists of terms, get tuple (word, sorted_list_of_docids)
add document frequency to term (#docid of that term)

advanced

see chapter "advanced inverted index construction"

3.4 core algorithms

intersection

two lists as input, start at first elements
if (same) add and advance
else advance where lower
repeat until both at end

union

two lists as input, start at first elements
if (lower) add & advance
else add once, advance both
repeat until both outside

3.5 optimizations

evaluation order

evaluate terms first with small #posting list
(min, max) OR is (max(A, B), A+B)
(min, max) AND is (0, min(A,B))

skip list

makes posting list faster with $\sqrt{\text{\#entries}}$ skip pointers
#comparisons vs #of irrelevant items which can be skipped

3.6 phrase queries

bi-word index

posting list with two words as term
"my long phrase" → "my long" AND "long phrase"
false positive for "my long long phrase"

phrase index

generalization of bi-word indexes
increase #words per term to reduce false positive rate
but vocabulary may get too big

positional index

save position of word
token/posting = (document id, #positions, positions)
match with position_before = position_after + 1

next word index

save the next word to the posting

combine schemes

use biword index for frequent queries
else use the positional index

4 tolerant retrieval

retrieve words from dictionaries as fast as possible
create additional indexes which resolve to a word
use this word to look up documents in inverted index

4.1 dictionary search structures

hash tables

for fast random access
but range queries not supported, collision avoidance needs space

binary tree

for fast ordered access, range queries
all nodes, or only leaves with values
but needs to be kept balanced

B+ tree

allows more children than binary tree
3-5 B+ requires 3-5 children (so 2-4 entries, except root)
#children always between d+1 and 2d+1 (3-5, 4-7, 5-11)
>1 child starts with entry of parent (enables range queries)
all leaves must be at same level
inserts if # > 2d entries, new & one more form upper level
delete removes if smaller than min, append to left part of tree

4.2 wildcards

word families, multiple spellings

trailing (ac*)

go to correct node, select all leaves
match leave entry with wildcard

leading (*ac)

reverse entries (cas → sac), sort & build new tree
match with reversed search term (ca*)

middle (ac*ca)

split to trailing AND leading, ac* AND *ca
possibly needs post filtering step

permuterm index

rotate word around itself (\$plant → t\$plan → nt\$pla)
easy wildcard treatment (pl*nt → nt\$pl*)
own index for single or collection of terms
permute words, sort & build up B+ tree
leaves of tree point to resulting word(s)

k-gram

split words in k-tuples, single index for all terms
build up like compute → \$compute\$ → \$com, comp, ..., ute\$
post filtering step (to remove terco result for co*ter)
leaves of tree point to resulting word(s)

4.3 spelling correction

correct using nearest (if tied, use more common) word

4.3.1 functionality variants

retrieve word & all spell corrected variants
retrieve spell corrections only if original word not in dictionary
retrieve spell corrections only if not enough results
present spell correction to user if not enough results

4.3.2 isolated term corrections

based on looking at single term

minimum editing distance

search space huge, use dynamic programming
permute, remove, add character
write word one x axis, word two y axis
min(diagonal +0 if letters match else +1, x/y +1)

k-gram overlap

2 words with similar k-grams have small editing distance
split search terms in k-grams, count matches
but lot of false positives

jaccard distance

between two words, #same k-grams / #total k-grams
small edit distance implies jaccard close to 1
but would need to compute union/intersection for all matches

speed up jaccard

#(k-grams in intersection) divided by
#(query grams) + #(found term grams) - #(intersection)

4.3.3 context-sensitive corrections

enumerate collections of certain amount of words
then correct to most likely with minimum editing distance

4.4 phonetic correction

every word to 4 character fingerprint
build B+ tree of these phonetic fingerprints to words

table

AEHIOUWY \rightarrow 0
BFPV \rightarrow 1
CGJKQSXZ \rightarrow 2
DT \rightarrow 3
L \rightarrow 4
MN \rightarrow 5
R \rightarrow 6

create fingerprint

keep first letter, translate the others to numbers
remove double numbers, remove 0's
keep only first four characters (or extend with 0's)
papper \rightarrow p01106 \rightarrow p0106 \rightarrow p16 \rightarrow p160

5 advanced inverted index construction

5.1 hardware basics

generally fast & expensive vs slow & cheap
OS reads/stores in blocks, IO done by system bus
use compression because CPU can decompress parallel to IO
use caches for frequently used data
store read-together data continuously to reduce seek time

three factors

capacity (how much can we store)
latency (how much time till first byte)
throughput (how fast can we transmit)

devices

CPU (MB, ns, TB/s)
RAM (GB, 10ns, Gb/s)
HD (TB, 10ms, 500 Mb/s)
tape (PB, s/min/h, 200 Mb/s)

5.2 optimizations

terms as termid dict (first-pass or on-the-fly buildup)
compression of posting lists & terms

blocked binary search

leave contains multiple values (=block)
reduces memory seeks, values inside block are stored sequentially
faster for SPIMI, no effect for MapReduce, BSBI deal with raw postings

5.3 blocked sort based index (BSBI)

allows for big collection to be processed on single machine
processes pairs block-wise, merges the intermediate results

processing block

parse documents as (termid, docid) tuples till block full
sort tuples by termid and create posting lists with docid postings
save to disk and start next block

merge

open all block files simultaneously, process lowest termid
merge lists with same termid & save back to disk

complexity ($T = \#tuples$)

$O(T \log T)$ because of sorting in block process
behaves like $O(T)$ because of merge with huge drive constants

single pass in memory indexing (SPIMI)

does not use termid dictionary for scalability
creates terms with posting lists per block on the fly
appends docids to posting lists, no sorting required anymore
sorts terms before writing block to speed up final merge
in $O(T)$ because no sorting of tuples

5.4 distributed indexing

process on multiple machines, master node assigns tasks

MapReduce

data divided into splits (size vs communication overhead)
mappers parse doc to (termid, docid) tuples (termid mapping given)
reducers create posting lists from assigned range of termids

5.5 dynamic index

avoid complete rebuild with fast changing datasets
use auxilliary indexes, rebuild periodically
include auxilliary index in search

single file for posting list

very cheap inserts, but OS can't handle that many files

invalidation bit vector

mark bit for removed document
output filter checks in invalidation bit vector

multiple auxilliary

keep multiple auxilliary of different logarithmic sizes
put new entries in smallest one
if full, merge with next bigger (recursively)
space $O(\log(T/n))$, construction $O(T \log(T/n))$, query $O(\log(T/n))$
for T tokens & size 2, need $\log_2(T/2 - 1)$ space

6 index compression

reduces disk space usage
reduces transfer time from disk to memory
increases caching efficiency

6.1 statistics

term cleanup savings

type (% of #terms, % of postings, % of #positions per posting)
remove numbers (-2%, -8%, -9%)
case folding (-17%, -3%, -0%)
150 stopwords removal (-0%, -30%, -47%)
stemming (-17%, -4%, -0%)

heaps law

predicts vocabulary size M of collection with T token
motivated by observed straight line in log-log plot
 $M = kT^b$ for typically k in $[30, 100]$, $b = 0.5$

zipfs law

collection frequency cf of i 'th term proportional to $1/i$
 $cf_i = c \cdot i^{-k}$ for constant c & $k = -1$ (straight line in log-log plot)

6.2 dictionary compression

compress dictionary to keep in memory, posting lists on disk
compression vs search time tradeoff

fixed size (automan_)

fix length of dictionary entries, efficient lookup of n 'th entry
but waste of space if shorter & impossible if longer

string blob (automan)

create pointer table to position in blob where keyword starts
sort pointer table

string blob with blocking (4auto3man)

include keyword size into blob, reduce pointers (only after k terms)
but search time increases

front coding

common prefix elimination (8automat*a8_e9_ic10_ion)

6.3 posting file compression

observe gaps between postings are short for frequent terms
encode first number then store gaps to next number

fixed gap size

works only well if term frequent (then fixed size small)

variable byte encoding

first bit of block size indicates if another block needed (1=yes)
example for blocksize=4 is 010011 \rightarrow 010 011 \rightarrow 1010 0011
byte sized blocks usually used in practice

unary code (thermometer code)

4 \rightarrow 11110

gamma encoding

10011 \rightarrow chop off leading bit \rightarrow 0011
0011 \rightarrow encode length in unary \rightarrow 11110
11110 \rightarrow append rest \rightarrow 11110 0011
prefix-free (entry does not depend on earlier)
parameter-free (no parameter depending on content)
always has 0 in the middle by construction (1011 is wrong)

can't encode 0 (but docid difference can never be 0 anyways)
saves space because variable length possible
argue with zipfs law/entropy to show its nearly optimal

7 ranked retrieval

document as a bag of words (lose ordering)

7.1 parametric & zone indexes

index of metadata (consisting of fields like author)
allows for simple weighting schemes

parametric search

search by structured metadata instead of content
can use a conventional DB

zone search

differentiate free text zones like author, body, title
create own inverted index each zone
or shared index by appending .zone (like ETH.body, ETH.title)
or store zone in postings (like bob \Rightarrow 1.title \rightarrow 2.author, 2.title)

zone weights

sum_of (zone_weight g * exists_in_zone s), scalar product g*s
we want that g*s = r (for r perceived relevance)
use machine learning for argmin_g(r - g*s) to determine optimal weights
or use polynomial to calculate weights

7.2 weights

definitions

term frequency (tf) (term in single document)
collection frequency (cf) (term in collection, sum_of tf)
document frequency (df) (#documents which contain term)

simple

directly use tf, but not good for rare terms

tf-idf

inversed document frequency (idf) as $\log_2(\#total_documents / df)$
multiply with tf (therefore get unique score per term for each document)

7.3 overlap score measure

document vector d

component for each entry in dictionary
0 if not term not in document, tf-idf for the others

score

sum up all entries in d which also occur in query q

7.4 vector space model

assumption that angle of document/query determines relevance

vector operations

euclidian norm $\|x\| = \text{sum_of } x_i^2$
dot/inner product $x*y$ is $\text{sum_of } x_i*x_{i,j}$

cosine similarity

the higher the result the more similar
 $\text{sim}(x,y) = (x*y) / (\|x\| * \|y\|)$ (entries are tf-idf scores)
applies normalization because of different size documents

query as vector

convert query terms to entries in its vector (weight = $\sqrt{\# \text{ of entries}}$)
but large number of vectors (documents) with high dimensions (terms)

efficient evaluation

drop dimensions not needed for given query
 $\text{tf_tquery} * \text{idf_term}^2 * \text{tf_tdocument} / \|\text{query}\| * \|\text{document}\|$
matrix (x for documents, y for term), calculate term or document wise
create bucket per document to aggregate result (avoid matrix construction)
sort buckets with priority queue $O(2N)$
take top K results (repeated $O(\log n)$)

efficient storage

don't store weight within postings because kills index compression
store tf_tdocument in posting, $\|d\|$ & idf_term at dictionary key
for query precalculate tf_tquery & $\|q\|$

7.5 SMART weights

notation looks like atc.lnb
first group for document weights, second for query weights

each group specifies term frequency, document frequency, normalization

term frequency scalings

(n) natural ($wf = tf$)
(i) sublinear ($wf = 1 + \log(tf)$ for $tf > 0$, else 0)
(a) augmented ($a + (1-a)tf/tf_max$ for $a \sim 0.5$, $tf_max = \max(tf \text{ in document})$)
(b) boolean (1 for $tf > 0$, else 0)
(L) low-average ($(1 + \log(tf)) / 1 + \log(\text{average})$ for $t > 0$, else 0)

document frequency scaling

(n) natural ($wf = df$)
(t) inverted ($\log(N)$ for $tf > 0$, else 0; the standard)
(p) probability ($\log(N-1)$ for $tf > 0$, else 0; at $N/2$ already 0)

normalization

(n) no normalization
(c) cosine (=piv, euclidian norm)
(b) byte-size ($1/CharLength^a$; better than euclidian for large)
(p) pivoted normalization ($a * \|V_d\| + (1-a)piv$ for given $a < 1$)

long document weights

terms appear more often (verbose), more distinctive (multiple topics)
gives unfair advantage with (n)
use (a), but is unstable with stop words (if fail to detect or changing list)
use (p) to correct cosine (disadvantages long documents)

8 efficient scoring and ranking

8.1 optimized query

lose term weights (because all weights equal), set >0 entries to 1
drop the idf_term of the query because weights all 1
go through postings & sum up all results in document-wise buckets
drop/avoid empty buckets
divide by $\|d\|$ (division by $\|q\|$ unnecessary)
optimized formula is $\text{idf_term} * \text{tf_tdocument} / \|d\|$

8.2 pre-query optimizations

inexact top K retrieval; only look at documents which are likely to be top K

low idf drop

drop keywords which have high idf because irrelevant
long posting lists removed

no keywords drop

drop documents which contains too few keywords from the query

8.3 impact sorting

order posting lists based on some measure of impact
breaks compression & update mechanisms (need to keep original around)

champion lists

precompute r documents with highest score per term
for tf-idf, order by term frequency

static quality score

static score (independent of term & query) of relevance for each document
sort by static score, use different quality lists (high/lower)

impact ordering

sort inverted index by idf (high impact words first)
sort posting lists by term frequency
scan topleft \rightarrow bottomright, stop when score stops improving
stop traversing lists after constant for performance (bc no docid sorting)

clustering

pick \sqrt{n} leaders, compute nearest leader of other documents (=cluster)
for query, pick nearest leader, calculate scores only for cluster

tiered indexes

ordering by tf-ids & docids
create different tiers (each static number of documents)
go down tiers till enough results found

query term proximity

sort by decreasing (so closer together is better)

8.4 query interfaces

boolean queries (ETH AND Zürich, use standard inverted index)
phrase queries ("ETH Zürich", use biword index/positional index)
zone queries (place: Zürich, use database)
set of words (ETH Zürich, use the vector space model)
wildcard queries (Z*urich, use k-gram/permuterm index)

8.5 architecture

parse linguistics to create document cache (for previews) & indexes
parse user query, do spell correction (with k-gram index)
ask zone / top k / tiered / biwords / k-grams indexes ("evidence accumulation")
weight indexes & calculate score
output best documents, create preview from document cache

9 probabilistic information retrieval

9.1 basic notation

universe Ω , made from elementary events w
probabilistic distribution assigns $p(w)$ such that $\sum p(w) = 1$

event

subset of Ω , consisting of elementary events, $p(E) = \sum p(w)$
complement $1-p(E)$, odds $O(E) = p(E) / 1-p(E)$
partition rule $p(E \cup F) + p(E \cup F^c) = p(E)$
conditional probability $p(E|F) = p(E \cap F) / p(F)$
chain rule $p(E \cup F) = p(E|F) * p(F)$
independence $p(E \cup F) = p(E) * p(F)$
bayes rule $p(E|F) = p(F|E) * P(E) / p(F)$

random variable

X , maps Ω to other space
 $p_x(\text{event}) = \sum p(w)$ for all w which are part of X
can write $p_x(\text{event})$ or $P(X = \text{event})$, but not differently
join with $P(X = \text{event}, Y = \text{event2})$, treat like events

9.2 probability model for document retrieval

query as a random variable, probability ranking principle
 $P(R=1|D=d \wedge Q=q)$ for query q , document d , relevance indicator $R = 1$

probability ranking principle (PRP)

lose point for each false
return if $P(R=1|D=d \wedge Q=q) > P(R=0|D=d \wedge Q=q)$
sort results by probability

PRP with retrieval costs

define C_0 for false positives, C_1 for false negatives
return d with lowest $C_0 * P(R=0) - C_1 * P(R=1)$

learning

take a lot of users running query q , which are given d as a result
how many of these users clicked on it ($R=1$)?
for future queries q return d with the highest percentage

9.3 binary independence model (BIM)

introduces assumptions to make P calculation practical
model documents & queries as vectors consisting of 1s, 0s
assume terms, documents are independent
is what we want to compute

computation

need to calculate $P(R=1 | D=d \wedge Q=q)$
need only to know ranking, can use odds $O(R|D=d \wedge Q=q)$
remove constant terms (only Q & R appearing)
can sum up for all elements x of vector D (independence assumption)
separate terms for $x=1$ and $x=0$, rename $P(x=1) = p_k$ & $P(x=0) = u_k$
limit the product to terms in q (set $u_k = p_k$ if not in q)
include $x=1$ in $x=0$ sum, adapt old $x=1$ to $p(1-u)/u(1-p)$
now $x=1 \& x=0$ sum constant for given query, can remove it
take the log of term to get RSV

retrieval status value (RSV)

for all k that occur in document & in query
 $\sum \log(\text{odds of } p_k) - \log(\text{odds of } u_k)$
left part big if high odds to appear in relevant document
right part small if low odds to appear in irrelevant document

practical estimations

estimate u_k to be df / N
estimate p_k to be tf in known relevant documents

relevance feedback

input query, execution, display & ask user if relevant
these marks are saved, and reused for the next query
but information need may be different for same query (python)

9.4 language model

finite state automaton (FSA)

start / accept state, arrows connecting state
if at the end of the word in accept state, FSA accepts
transfer function $\delta(q_1, a) = q_2$, Ω as language space (symbols)
can generate random words by assigning probability to arrows

unigram model

assumes term independence (term probability only depends on frequency)
single-state FSA per document needed

n-gram model

$P(T=t_1) * P(T=t_2|t_1)$ for ($n=2$, and two words t)

query

generate query likelihood model (LM) of document
estimate probability of query generated by each LM & sort

10 evaluation

check whether IR can answer information need with query
information need may depend on context (pet vs programming python)
test with datasets like cranfield, TREC

10.1 effectiveness

true are returned, positives are relevant & vice-versa
true positives = relevant & returned
false positives = not relevant & returned (type I error)
false negatives = relevant & not returned (type II error)
true negatives = not relevant & not returned

measures

precision = $\# \text{true positives} / \# \text{positives}$
high precision implies few false positives
recall = $\# \text{true positives} / \# \text{true}$
high recall implies few false negatives
specificity = $\# \text{true negatives} / \# \text{negatives}$
high specificity means no negatives in result
accuracy = $\# \text{true} / (\# \text{false} + \# \text{true})$
high accuracy means small type I and type II errors

combining precision P & recall R

more returned documents leads to less precision & higher recall
use arithmetic mean $(P + R)/2$, but 50% score when returning all
use harmonic mean $1/((1/P) + (1/R))$ for weighting factor a

10.2 evaluating ranked retrieval

use precision/recall measures to measure effectiveness

precision at k

only measure inside a specific range
makes sense if only top 10 are returned to user

r-precision

only compare with known subset r
for query with r known results, look at matches in actual top k

ROC

plots recall to sensitivity, want extremes (high/high or low/low to invert)
sensitivity defined as $(1 - \text{specificity})$
awful if recall / specificity same values

11 page rank

11.1 basics

pages divide impact between links (evenly)
probability of being on each page as random surfer
result is array of probabilities

11.2 stochastic approach

build weight matrix, column = document (sum of 1)
multiply with column array, look for fixpoint

find fixpoint

bruteforce
find eigenvector with eigenvalue 1 (use diagonalization)

damping

include 0.15 "randomness factor", and 0.85 with page rank