Faster TF-IDF

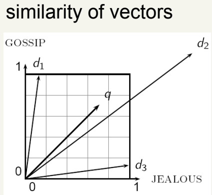
David Kauchak
cs458
Fall 2012
adapted from:
http://www.stanford.edu/class/cs276/handouts/lecture6-tfidf.ppt

Administrative

- Videos
- Homework 2
- Assignment 2
- CS lunch tomorrow

TF-IDF recap

- Represent the queries as vectors
- Represent the documents as vectors
- proximity = similarity of vectors



What do the entries in the vector represent in the tf-idf scheme?

TF-IDF recap: document vectors

	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

A document is represented by a vector of weights for each word

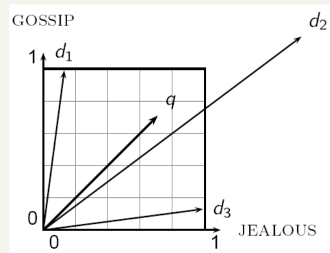
TF-IDF recap: document vectors

	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

One option for this weighting is TF-IDF:

$$w_{t,d} = \text{tf}_{t,d} \times \log(N/\text{df}_t)$$

TF-IDF recap: similarity



Given weight vectors, how do we determine similarity (i.e. ranking)?

TF-IDF recap: similarity

Dot product

Unit vectors

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

cos(q,d) is the cosine similarity of q and d … or, equivalently, the cosine of the angle between q and d.

Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

The basic idea

Index-time:

calculate weight (e.g. TF-IDF) vectors for all documents

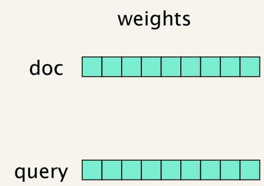
Query time:

calculate weight vector for query

calculate similarity (e.g. cosine) between query and all documents

sort by similarity and return top K

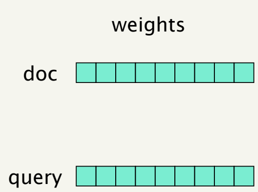
Calculating cosine similarity



How do we do this?

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

Calculating cosine similarity

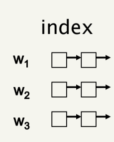


Traverse entries calculating the product

- Accumulate the vector lengths and divide at the end
- How can we do it faster if we have a sparse representation?

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

Calculating cosine tf-idf from index



What should we store in the index?

How do we construct the index?

How do we calculate the document ranking?

$$w_{t,d} = \text{tf}_{t,d} \times \log(N/\text{df}_t)$$

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

Index construction: collect docIDs

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

Index construction: sort dictionary

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

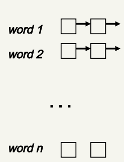
sort based on terms

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

Index construction: create postings list

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

create postings lists from identical entries



$$w_{t,d} = \text{tf}_{t,d} \times \log(N/\text{df}_t)$$

Do we have all the information we need?

Obtaining tf-idf weights

Store the *tf* initially in the index

In addition, store the number of documents the term occurs in in the index (length of the postings list)

How do we get the idfs?

- We can either compute these on the fly using the number of documents in each term
- We can make another pass through the index and update the weights for each entry

Pros and cons of each approach?

An aside: speed matters!

Urs Holzle, Google's chief engineer:

- When Google search queries slow down a mere 400 milliseconds, traffic drops 0.44%.
- 80% of people will click away from an Internet video if it stalls loading.
- When car comparison pricing site Edmunds.com reduced loading time from 9 to 1.4 seconds, pageviews per session went up 17% and ad revenue went up 3%.
- When Shopzilla dropped load times from 7 seconds to 2 seconds, pageviews went up 25% and revenue increased between 7% and 12%.

http://articles.businessinsider.com/2012-01-09/tech/30607322_1_super-fast-fiber-optic-network-google-services-loading

Do we have everything we need?

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

Still need the document lengths

- Store these in a separate data structure
- Make another pass through the data and update the weights

Benefits/drawbacks?

Computing cosine scores

Similar to the merge operation

Accumulate scores for each document

float *scores*[N] = 0

for each query term *t*
  calculate $w_{t,q}$
  for each entry in *t*'s postings list: *docID*, $w_{t,d}$
    scores[docID] += $w_{t,q}$ * $w_{t,d}$

return top *k* components of scores

Computing cosine scores

What are the inefficiencies here?

- Only want the scores for the top *k* but are calculating all the scores
- Sort to obtain top k?

float *scores*[N] = 0

for each query term *t*
  calculate $w_{t,q}$
  for each entry in *t*'s postings list: *docID*, $w_{t,d}$
    scores[docID] += $w_{t,q}$ * $w_{t,d}$

return top *k* components of scores

Outline

Calculating tf-idf score

Faster ranking

Static quality scores

Impact ordering

Cluster pruning

Key challenges for speedup

Ranked search is more computationally expensive

float *scores*[N] = 0

for each query term *t*

calculate $w_{t,q}$

for each entry in *t*'s postings list: *docID*, $w_{t,d}$

scores[docID] += $w_{t,q}$ * $w_{t,d}$

return top *k* components of scores

Why is this more expensive than boolean?

Key challenges for speedup

Ranked search is more computationally expensive

float *scores*[N] = 0

for each query term *t*

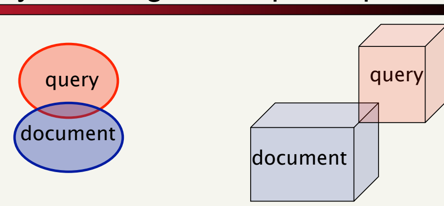
calculate $w_{t,q}$

for each entry in *t*'s postings list: *docID*, $w_{t,d}$

scores[docID] += $w_{t,q}$ * $w_{t,d}$ — more expensive

return top *k* components of scores — sort?

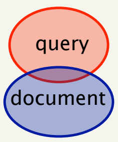
Key challenges for speedup



boolean — ranked

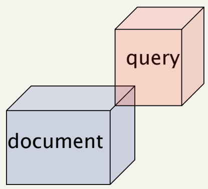
Intersection — strictly intersection?

Key challenges for speedup



boolean

Intersection



ranked

soft-intersection: only requires one or more words to overlap
Many, many more documents!

Speeding up the "merge"

float *scores*[N] = 0

for each query term *t*
    calculate $w_{t,q}$
    for each entry in *t*'s postings list: *docID*, $w_{t,d}$
        scores[docID] += $w_{t,q}$ * $w_{t,d}$

return top *k* components of scores

Any simplifying assumptions to make this faster?

Queries are short!

Assume query terms only occur once

Assume no weighting on query terms

Speeding up the "merge"

float *scores*[N] = 0

for each query term *t*
    calculate $w_{t,q}$
    for each entry in *t*'s postings list: *docID*, $w_{t,d}$
        scores[docID] += $w_{t,q}$ * $w_{t,d}$

return top *k* components of scores

Assume query terms only occur once

Assume no weighting on query terms

float *scores*[N] = 0

for each query term *t*
    for each entry in *t*'s postings list: *docID*, $w_{t,d}$
        scores[docID] += $w_{t,d}$

return top *k* components of scores

Selecting top K

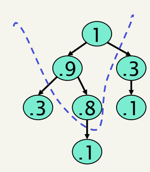
We could sort the scores and then pick the top K

What is the runtime of this approach?
    O(N log N)

Can we do better?

Use a heap (i.e. priority queue)

- Build a heap out of the scores
- Get the top K scores from the heap
- Running time?
    O(N + K log N)



For *N*=1M, *K*=100, this is about 10% of the cost of sorting

Inexact top K

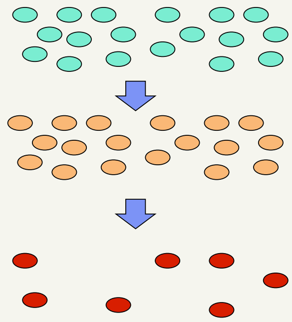
What if we don't return exactly the top K, but almost the top K (i.e. a mostly similar set)?

User has a task and a query formulation

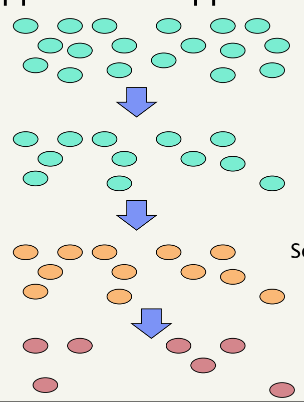
Cosine is a proxy for matching this task/query

If we get a list of *K* docs "close" to the top *K* by cosine measure, should still be ok

Current approach



Approximate approach

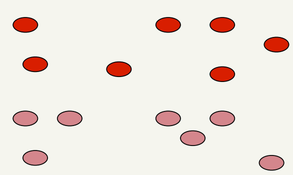


Exact vs. approximate

Depending on how **A** is selected and how large **A** is, can get different results

Can think of it as **pruning** the initial set of docs

How might we pick A?



Exact vs. approximate

How might we pick A (subset of all documents) so as to get as close as possible to the original ranking?

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

Documents with <u>more than one</u> query term

Terms with high IDF (prune postings lists to consider)

Documents with the highest weights

Docs must contain multiple query terms

Right now, we consider any document with at least one query term in it

For multi-term queries, only compute scores for docs containing several of the query terms

- Say, at least 3 out of 4 or 2 or more
- Imposes a "soft conjunction" on queries seen on web search engines (early Google)

Implementation?

Just a slight modification of "merge" procedure

Multiple query terms

If we required all but 1 term be there, which docs would we keep?

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

Scores only computed for 8, 16 and 32.

Multiple query terms

How many documents have we "pruned" or ignored?

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

All the others! (1, 2, 3, 4, 5, 13, 21, 34, 64, 128)

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 don't alter rank-ordering much

Benefit:

- Postings of low-idf terms have many docs → these (many) docs get eliminated from *A*

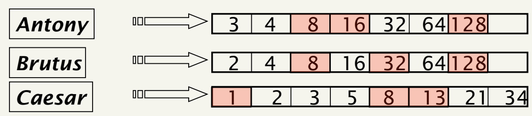
Can we calculate this efficiently from the index?

High scoring docs: champion lists

Precompute for each dictionary term the *r* docs of highest weight in the term's postings

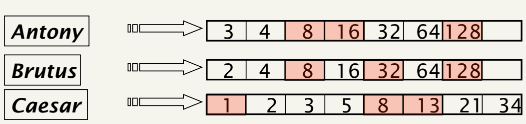
- Call this the champion list for a term
- (aka fancy list or top docs for a term)

Can we do this at query time?



Implementation details…

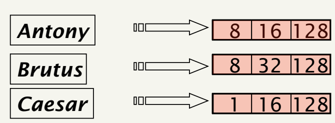
How can Champion Lists be implemented in an inverted index? How do we modify the data structure?



Champion lists

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



Are we guaranteed to always get K documents?

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

A way to segment the index into two <u>tiers</u>

Tiered indexes

Break postings up into a hierarchy of lists

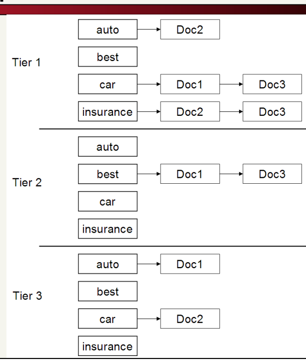
- Most important
- …
- Least important

Inverted index thus broken up into <u>tiers</u> 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



Quick review

Rather than selecting the best K scores from all N documents

- Initially filter the documents to a smaller set
- Select the K best scores from this smaller set

Methods for selecting this smaller set

- Documents with <u>more than one</u> query term
- Terms with high IDF
- Documents with the highest weights

Outline

Calculating tf-idf score

Faster ranking

Static quality scores

Impact ordering

Cluster pruning

Static quality scores

We want top-ranking documents to be both ***relevant*** and ***authoritative***

query: *dog*

Which will our current approach prefer?

my dog

Static quality scores

We want top-ranking documents to be both ***relevant*** and ***authoritative***

Cosine score models *relevance* but not authority

Authority is typically a query-independent property of a document

What are some examples of authority signals?

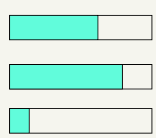
- Wikipedia among websites
- Articles in certain newspapers
- A paper with many citations
- Many diggs, Y!buzzes or del.icio.us marks
- Lots of inlinks
- Pagerank

Modeling authority

Assign to each document a *query-independent* quality score in [0,1] denoted *g(d)*

A quantity like the number of citations is scaled into [0,1]

Google PageRank



Net score

We want a total score that combines cosine relevance and authority

How can we do this?

addition: net-score(*q*,*d*) = *g(d)* + cosine(*q*,*d*)

can use some other linear combination than an equal weighting

Any function of the two “signals” of user happiness

Net score

Now we want the top *K* docs by net score

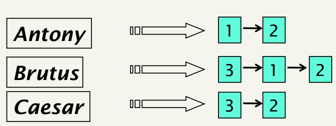
What does this change in our indexing and query algorithms?

Easy to implement:
similar to incorporating document length normalization

Top *K* by net score – fast methods

Order all postings by *g(d)*… does it change our merge/ traversal algorithms?

Key: this is still a common ordering for all postings

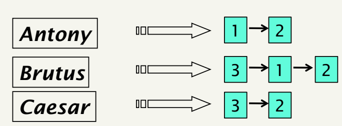


g(1) = 0.5, g(2) = .25, g(3) = 1

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



g(1) = 0.5, g(2) = .25, g(3) = 1

Champion lists in *g(d)*-ordering

We can still use the notion of champion lists…

Combine champion lists with *g(d)*-ordering

Maintain for each term a champion list of the *r* docs with highest $g(d)$ + tf-idf$_{td}$

Seek top-*K* results from only the docs in these champion lists

Discussion

- Who should be held responsible when a program generates undesirable data outside control of the programmer?
- Does removal from the autocomplete feature, but not the general search results, count as censorship?
- How much power should Google have to censor content?